ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

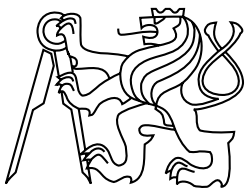
Fakulta elektrotechnická

Diplomová práce

Implementace simulátoru neuronových sítí

Tomáš Horyl

2005



Anotace

Úkolem této práce je prozkoumat simulační jazyk SiMoNNe pro neuronové sítě, prostudovat současnou implementaci javovského jádra simulátoru a navrhnout zdokonalení s přihlédnutím k nově vzniklým rysům jazyka v souvisejících pracích. Následně pak implementovat a na příkladech ověřit nové jádro simulátoru. Na základě analýzy současného jazyka SiMoNNe byl navržen nový jazyk, který odstraňuje nedostatky předešlé verze. Pro něj byl přepracován původní překladač a vytvořen nový interpret. Jeho funkčnost jsem ověřil na příkladech několika neuronových sítí.

Abstract

Goal of this work is to explore SiMoNNe language for neural networks simulation, study current simulator core written in Java and propose improvements with regard to related works. After new simulator core design and implementation, its usability will be tested on neural networks examples. Following current SiMoNNe language analysis, new language was designed and new simulator core was implemented. Its functionality was tested on several typical neural networks.

Poděkování

Chtěl bych poděkovat zejména vedoucímu práce Janu Koutníkovi za mnoho užitečných rad a spoustu hodin strávených při testování implementace, svým rodičům za podporu při studiu a všem, kteří mi pomohli při psaní této práce.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu. Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů.

V Praze dne _____ _____

podpis

Obsah

Seznam obrázků

Seznam tabulek

Přehled práce

Kapitola 1 stručně popisuje, co jsou to neuronové sítě, jaký vztah je mezi skutečnými a umělými neuronovými sítěmi, jak se dají sítě rozdělit a co jsou to vrstevnaté a modulární neuronové sítě.

Kapitola 2 shrnuje současnou situaci na poli simulátorů neuronových sítí. Rozděluje simulátory do několika skupin a u každého simulátoru krátce popisuje jeho funkce a vlastnosti.

Kapitola 3 popisuje předchozí implementaci jazyka SiMoNNe a nově vzniklý simulátor Simonnec. V kapilole jsou rozebírány jejich funkce a vlastnosti podrobněji než v případě kapitoly 2.

Kapitola 4 se zabývá úpravami původního jazyka SiMoNNe. V kapitole jsou nejdříve specifikovány požadavky na nový jazyk, potom jsou rozebrány jednotlivé změny v gramatice jazyka. Následuje popis požadavků kladených na simulátor a ve zbytku kapitoly je popsána architektura nového simulátoru a některých jeho částí.

Kapitola 5 rozebírá implementaci simulátoru a jeho vybraných částí. V kapitole jsou taky krátce popsány komponenty, které jsou v simulátoru použity a jsou tam vysvětleny některé termíny používané v simulátoru.

Kapitola 6 slouží jako uživatelská příručka. Provází uživatele konstrukcemi nového jazyka a na příkladech vysvětluje jejich použití. Je v ní vysvětlen postup při implementaci externích funkcí a možnosti konfigurace systémových proměnných simulátoru.

Kapitola 7 demonstruje použití simulátoru na příkladech typických neuronových sítí a u každého příkladu uvádí bohaté komentáře vysvětlující použité konstrukce.

Kapitola 8 shrnuje v několika větách celou práci.

Kapitola 9 rozebírá možnou budoucnost simulátoru a uvádí příklady, na které by se při případném rozvíjení simulátoru měl programátor zaměřit.

Příloha obsahuje popis spuštění simulátoru, gramatiku původního jazyka SiMoNNe a gramatiku nového jazyka a kompletní zdrojové kódy příkladů z textu.

Kapitola 1

Neuronové sítě

1.1 Úvod

Neuronové sítě spadají do oboru umělé inteligence. Svou předlohu mají v nervové soustavě živých organismů včetně lidské. Je to jedna z možností jak přistupovat k řešení problémů a právě proto, že v přírodě je tato metoda velice úspěšná, jedná se o velmi perspektivní oblast. Implementace neuronových sítí může být hardwarová i softwarová.

Základní stavební jednotkou v lidském mozku jsou nervové buňky (nazývané neurony), které jsou mezi sebou propojeny vazbami (axon a dendrity). Stejně tak je tomu i v případě umělých neuronových sítí. Skutečné neuronové sítě jsou ovšem mnohonásobně dokonalejší, protože dokáží pracovat paralelně. Takto může fungovat i hardwarová implementace, což přináší velké zrychlení výpočtu. Druhou výhodou, a toto je podstatně zásadnější rozdíl, je počet neuronů. To je zatím pro umělé neuronové sítě hlavní bariera, které je v současnosti nepřekonatelná pro obě implementace.

Jak již bylo zmíněno, hlavní výhodou hardwarové implementace je zejména rychlost díky paralelnímu zpracování. Na rozdíl od toho je však předností softwarové implementace možnost rychle a flexibilně měnit strukturu a parametry sítě a vytvářet sítě nové, což při přestavbě sítě nepřináší další dodatečné náklady. Jak je tedy vidět, obě implementace se dobře doplňují a před návrhem nějakého čipu realizujícího umělou neuronovou síť je lepší si softwarově funkci sítě nasimulovat. Z toho důvodu také vznikají simulátory neuronových sítí.

1.1.1 Umělé neuronové sítě

Základní stavební jednotkou umělé neuronové sítě je neuron. Každý neuron má jeden nebo několik vstupů a jeden výstup, který může být rozvětven. Vstupní data jsou podle určité nelineární přenosové funkce transformována na data výstupní. Touto funkcí je většinou sigmoida[1], nemusí to však být pravidlem.

Neuron reaguje na vstupní signály. Regulace síly vstupních signálů probíhá na tzv. synapsích, což jsou místa v nichž se spojují výstupy z předešlých neuronů a vstupy následujících neuronů. Jedná se vlastně o váhové koeficienty jednotlivých vstupů. Regulací lze tedy docílit změny výstupu neuronu. Toto chování je možné popsat jako paměť.

[1] Matematický zápis sigmoidy je $S(x) = 1/(1 + e^{-\gamma x})$, kde γ je koeficient určující strmost křivky.

Neuronové sítě živých organismů mají schopnost učit se, stejně tak tomu je i u umělých neuronových sítí. Při učení dochází k nastavování synapsí, příp. u některých, tzv. samoorganizujících, sítí ke změně struktury sítě. Učení může být dvojího druhu (učení s učitelem a bez učitele).

Učení s učitelem probíhá tak, že neuronová síť srovnává svůj výstup s již známými hodnotami a podle toho nastavuje váhy jednotlivých synapsí.

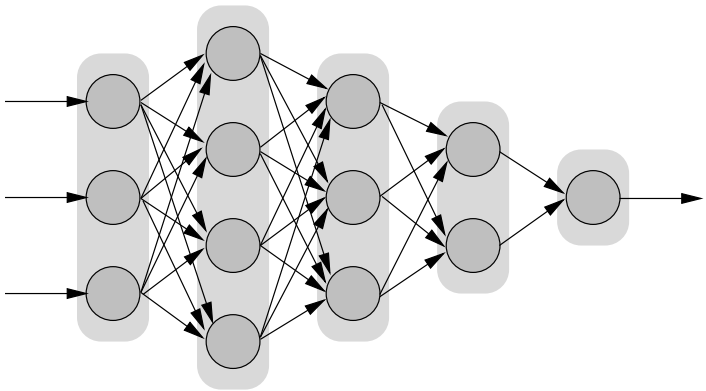
Naopak při učení bez učitele neexistují žádné srovnávací hodnoty a neuronová síť ve vstupních datech vyhledává určité závislosti.

Po fázi učení obvykle probíhá fáze vybavování, kdy neuronová síť odpovídá na předložené vzory podle toho, co se naučila. Může např. rozřazovat jednotlivé vzorky do různých skupin, což se nazývá klasifikace. Nebo na základě vstupních hodnot a průběhu v minulosti předpovídá výstupní hodnotu funkce v nějakém bodě (predikce). Možností aplikace je hodně. Toho všeho je neuronová síť schopna především díky svým vlastnostem generalizace[2].

Dalším z možných kritérií, podle kterých lze neuronové sítě rozdělovat, je struktura propojení mezi funkčními jednotkami. Nejobecnější by jistě mohlo být propojení mezi všemi neurony, které funkčně dokáže nahradit všechny ostatní typy propojení. To je však často zbytečné a i v rámci živých organismů to není běžné. Mnohem častěji se tento typ vyskytuje pouze lokálně v oblasti několika neuronů (obvykle s obdobnou funkcí). Proto vznikají vrstevnaté a modulární neuronové sítě.

1.1.2 Vrstevnaté a modulární neuronové sítě

V historii neuronových sítí vznikly z jednoduchého perceptronu [Haykin-1994], což byl původně jeden neuron s jednoduchým učicím algoritmem, sítě se složitější strukturou. Hlavní zájem je v současnosti obrácen k sítím vrstevnatým, ve kterých se uplatňuje paralelismus řešící i složitější problémy. Vrstevnaté sítě mají zpravidla pravidelnou strukturu složenou z vrstev, kde vrstva je množina neuronů podobné, či stejné funkce. Neurony jsou v rámci jedné vrstvy pravidelně propojeny, nebo jsou pravidelně (zpravidla úplně) připojeny k neuronům jiné vrstvy v síti, případně ke vstupu nebo výstupu celé sítě (viz obrázek 1.1).

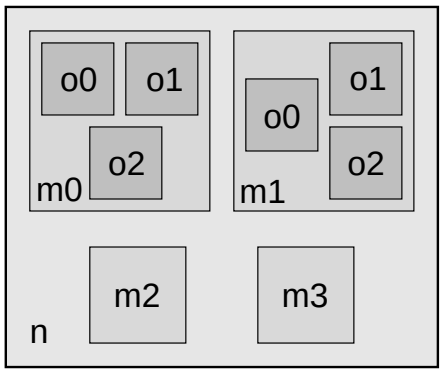


Obrázek 1.1: Vrstevnatá síť. Šedé sloupce znázorňují jednotlivé vrstvy neuronů, šipky propojení mezi vrstvami.

[2]Generalizace je schopnost neuronové sítě z konkrétních příkladů utvořit obecný popis, který zachovává společné rysy těchto podobných příkladů.

Poslední dobou se však začínají prosazovat tzv. *modulární neuronové sítě* [Ronco-1995]. Snaží se odstranit některé nedostatky klasických vrstevnatých sítí. Jejich hlavní nevýhodou je to, že jsou příliš rozsáhlé, ikdyž problém, který řeší nemusí být tak složitý. Je to tím, že ne vždy úplné propojení jednotek stejné funkce poskytne nejlepší řešení problému jen tím, že je masivní. Modulární neuronové sítě mají svou inspiraci v lidském mozku. Jeho šedá kůra je organizovaná do tzv. kortikálních sloupců, což jsou hustě propojené neuronové struktury. Neurony ve sloupcích jsou též propojeny s neurony v jiných sloupcích. Celé to pak funguje tak, že jednotlivé sloupce řeší podproblémy problému, který mozek zpracovává. Navzájem však spolu komunikují. Tento postup je dobře známý z reálného života, kdy jsme zvyklí při řešení nějakého problému jej dekomponovat na podproblémy. Modulární sítě se snaží kombinovat zpracování vstupu částečně lokálně působícími neurony a částečně globálně působícími neurony.

V literatuře [Koutník-2004] je pojem modularity zobecněn takto: modulární neuronová síť obsahuje základní funkční celky, kterým říkáme moduly. Moduly lze navzájem hierarchicky vnořovat, viz obrázek 1.2. Struktura vnořování je závislá na charakteru problému, který má síť řešit. Moduly na nejnižší úrovni vnoření jsou jednotlivé neurony, modul na nejvyšším stupni hierarchie představuje celou síť. Moduly na stejné úrovni lze navzájem propojovat a zapouzdřovat do modulů vyšší úrovně. Nejjednodušší moduly (neurony) bývají hustě propojeny, se stoupající hierarchií propojení může řídnout – tím se zjednodušují komunikační nároky sítě, ale v případě vhodného návrhu neklesá její výpočetní síla. Takové modulární sítě vhodně využívají hierarchické dekompozice problému na podproblémy a nejsou omezeny pouze lokalitou či globalitou práce jednotlivých částí sítě. Příkladem takové modulární sítě je síť-modul CALM (Categorizing and Learning Module) [Murre-1992] a z těchto modulů složená síť pro hierarchické zpracování obrazu – GOLOKO [Brunner-1997, KouBrunŠno-2002].



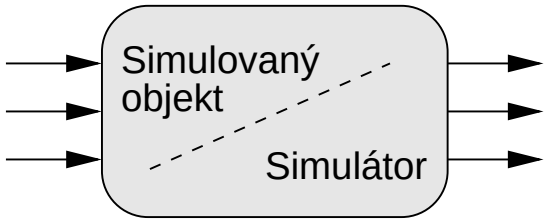
Obrázek 1.2: Modulární síť. Jednotlivé čtverce znázorňují moduly. Jak je vidět na obrázku, moduly různých druhů jsou zapouzdřeny do modulů na vyšším stupni hierarchie.

Učení takovýchto sítí však může být nesnadné a před stavbou složitějších modulárních sítí je nutné jednotlivé moduly řádně vyzkoušet. V dalších částech této práce je vidět proč nelze s takovými sítěmi jednoduše experimentovat pomocí standardních prostředků a jaká byla motivace při rozšiřování nového simulátoru pro modulární neuronové sítě [BrunKou-2002].

Kapitola 2

Rozdělení a rešerše simulátorů

Softwarový simulátor je program, který zastupuje nějaký objekt jehož činnost simuluje a přijímá vstupy určené tomuto objektu 2.1. Následně provádí transformaci dat na základě funkce simulovaného objektu a poskytuje výsledky simulace. Těmi jsou primárně výsledky transformace dat, ale i vnitřní průběhy v simulátoru.



Obrázek 2.1: Softwarový simulátor. Zastupuje objekt, jehož činnost simuluje. Přijímá jeho vstupy, simuluje funkci objektu a poskytuje výsledky simulace.

Simulátor neuronových sítí je speciálním případem obecného simulátoru. Na svých vstupech přijímá popis sítě (její strukturu a popis funkce jednotlivých prvků) a vstupní parametry. Výsledky jsou excitace výstupních neuronů sítě, příp. jak bylo uvedeno výše, vnitřní hodnoty v simulátoru.

Simulátory neuronových sítí (dále jen simulátory) lze rozdělit do dvou základních skupin. Obecné simulátory nepohlížejí na síť jako na nějaké její konkrétní paradigma a pracují tudíž pouze s *nějakou* neuronovou sítí. Jejich výhodou je, že dokáží simulovat jakoukoliv neuronovou síť.

Druhou skupinou jsou simulátory specializované. Jsou určeny pro simulaci jednoho, nebo několika typů sítí. Výhodu mají v tom, že díky specializaci je možné je optimalizovat na určené výpočty.

V praxi však takto striktně simulátory rozdělit nelze, protože u mnoha simulátorů také záleží na způsobu popisu sítě a jejich prvků. Někdy je možné sítě pouze upravovat pomocí parametrů, jindy se dá kompletně změnit funkce neuronů nebo celé sítě (včetně její struktury).

Lze však říci, že simulátor je tím obecnější, čím abstraktnější výrazový prostředek pro vytvoření neuronové sítě a řízení její simulace poskytuje. Tomu zcela odpovídá možnost popisu programovacím jazykem, protože ten má bohaté vyjadřovací schopnosti díky tomu, že skládáním elementárních operací teprve vznikají funkce simulátoru, zatímco v ostatních případech je uživatel omezen již naimplementovanými vlastnostmi.

Zvláštní kapitolou jsou simulátory s grafickým uživatelským rozhraním (dále GUI). Jejich hlavní výhodou je, že uživatel okamžitě vidí prováděné změny, strukturu sítě a v některých případech může snadno měnit její parametry. Další výhodou je interaktivita, která ve spojení s uživatelským rozhraním dovoluje v průběhu zasahovat do běhu simulace a sledovat výsledek provedených změn. Právě tyto výhody se však za jistých okolností mohou stát nevýhodami. Zejména pokud potřebuje uživatel některé věci raději řešit skriptováním[1], nebo pracovat dávkově (např. má mnoho souborů se vstupními daty a simulaci by pro každý soubor dat rád spustil bez nutnosti manuálně jej simulátoru předkládat).

Jedním z velmi častých handicapů GUI je jejich těsná či úplná svázanost s vlastním simulátorem. Simulátor je pak limitován pouze tím, co je naimplementováno v uživatelském rozhraní a jakákoliv rozšiřitelnost je velmi složitá, někdy dokonce nemožná. Autoři samozřejmě při návrhu nemohli implementovat všechny sítě a ani všechny neznali, protože obor umělé inteligence se rozvíjí a nových sítí přibývá velkým tempem [Haykin-1994].

Řešením je oddělení GUI a samotného simulátoru. Pro komunikaci mezi nimi je pak nutné najít dostatečně silný výrazový prostředek, pro který nebude problém komunikovat v obecné rovině s jádrem simulátoru. Jak bylo uvedeno dříve, takovým prostředkem může být programovací jazyk.

Zde je nutné se zamyslet nad tím, proč vytvářet nový jazyk, když těch současných existují stovky? [LangList] To je pravda, ale současné programovací jazyky nejsou pro vytváření neuronových sítí a popis simulací příliš vhodné. Nutí totiž uživatele soustředit se spíše na řešení algoritmů, které nesouvisí s prací návrháře neuronové sítě a měl by být od nich tudíž odstíněn[2]. V určitém bodě abstrakce je tedy pohodlnější přijmout již dané operace jako atomické, protože jejich další dělení by už nic nepřineslo (naopak, bylo by to spíše nevýhodou). A právě tyto operace tvoří nový jazyk s jehož pomocí se s neuronovými sítěmi pracuje lépe a jednodušeji.

Jedním z možných přístupů k simulaci neuronových sítí jsou také knihovny pro různé programovací jazyky. V daném jazyce jsou napsány kusy kódu, které zjednodušují některé typické operace prováděné s neuronovými sítěmi. Velkou výhodou těchto knihoven je přímý zápis v kódu daného programovacího jazyka, pro který většinou existují velmi kvalitní překladače a ladicí nástroje a simulace může být tudíž rychlá. Zápis kódu v klasickém programovacím jazyce, i přesto, že některé operace jsou již vyřešené v knihovně, však může být pro návrháře pořád nevýhodou, protože stále to musí být spíše programátor. Navíc když je síť popisována v jazyce, který na to není přímo určen, bývá její popis poměrně složitý a nepřehledný.

V následujícím textu jsou popsány některé současné simulátory.

2.1 Neuniverzální simulátory

2.1.1 Simulátor NeuralWorks

NeuralWorks [NeuralWorks] patří do skupiny GUI simulátorů. Umožňuje pracovat s jednotlivými neurony, spoji mezi neurony, vrstvami neuronů a spoji mezi těmito vrstvami. Chování neuronu je popsáno modelem. NeuralWorks obsahuje mnoho modelů pro běžné neuronové sítě. Aby bylo možné neurony přizpůsobit různým sítím, jsou modely parametrizovány. Vstupy je možné zadávat jednak z GUI (inter-

[1] Popis simulace je ve formě příkazů pro simulátor zapsaný v textovém souboru. Může jít například o popis konstrukce sítě nebo pouze hromadnou změnu některých početných parametrů.

[2] Toto se snaží částečně řešit různé knihovny pro klasické programovací jazyky.

aktivně) nebo ze souboru s hodnotami. Výstup je standardně na obrazovku, je možné však zvolit i výstup do souboru.

Jak již bylo zmíněno, simulátory s GUI trpí malou (nebo složitou) rozšiřitelností na sítě, pro které nebyly vytvořeny. Bohužel je to případ i tohoto simulátoru. V novějších verzích je možné rozšířit funkčnost simulátoru přes externí programovací jazyk. NeuralWorks umožňuje vyexportovat zdrojový kód sítě v jazyce C. Takto získaný zdrojový kód je možné si dále přizpůsobit, ovšem upravenou verzi již není možné zpět do simulátoru nahrát a pracovat s ní.

2.1.2 Simulátor SNNS a JavaNNS

SNNS (Stuttgart Neural Network Simulator) [SNNS-JNNS] je simulátor neuronových sítí a grafické uživatelské rozhraní. Jádro simulátoru je odděleno od GUI a komunikace mezi nimi probíhá podle dokumentovaných protokolů. Ty však nebyly navrženy pro přímé použití uživatelem. JavaNNS je novější a jednodušší verze GUI napsaná v jazyce Java. I v případě těchto simulátorů je podporován pouze omezený počet typů sítí.

GUI umí pracovat nejen s jednotlivými neurony, ale i s vrstvami (a to i dvojrozměrnými) – simulátor pracuje s tzv. jednotkami. Přidání složitějších objektů je však obtížné. Spojování vybraných neuronů je poloautomatické. Pro každou síť je možné nastavit způsob aktualizace hodnot v jednotkách (synchronně, náhodně, postupně).

Nespornou výhodou je podpora paralelního výpočtu a nástroje pro analýzu sítě a odstranění spojů, které nemají pro funkčnost sítě význam.

Protože se jedná o nekomerční a akademický simulátor, jsou dostupné i zdrojové kódy. I přesto je však rozšiřování simulátoru o nový typ sítě nesnadné.

2.2 Knihovny

2.2.1 Knihovna Joone

Java Objective Oriented Neural Engine [Joone] je framework pro vytváření, učení a testování neuronových sítí. Je rozdělen na samotné simulační jádro a GUI, které však není pro práci nutné a je možné pracovat pouze s jádrem. Výhodou GUI je snadná práce při vytváření neuronové sítě díky předem připraveným komponentám a učicím algoritmům.

Joone pracuje s několika základními komponentami. Jedná se o *vrstvy sítě* (obsahují jednotlivé neurony), *synapse* (spojení mezi vrstvami), *monitor* (centrální prvek kontrolující v jakém stavu se síť nachází) a *vstup a výstup* (příjem vstupních a zápis výstupních hodnot).

Jednotlivé vrstvy sítě jsou samostatné celky a každá vrstva běží jako zvláštní vlákno. Vlákna jsou potom řízená semafory. To přináší možnost spouštět výpočet distribuovaně, přičemž proces distribuovaného výpočtu je řízen a umožňuje dynamické přidávání a ubírání výpočetních uzlů.

Nevýhodou Joone však je nemožnost měnit způsob aktualizace výstupů – vlákna počítají pokud mají data k výpočtu. Některé sítě tímto způsobem není možné simulovat (např. CALM[3]). Dalším

[3] Categorizing and Learning Module

omezením je nemožnost pracovat interaktivně.

2.2.2 Knihovna Jane

Java Artificial Neural Engine je knihovna v jazyce Java, která se snaží implementovat vlastnosti společné všem neuronovým sítím, tj. automatizovat některé operace s neuronovými sítěmi.

Jane pracuje se čtyřmi základními prvky – *neuron* (tradiční prvek s více vstupy a jedním výstupem), *modul* (reprezentuje samostatné rozsáhlejší celky sítě), *synapse* (spojuje neurony a moduly) a *network* (neuronová síť jako celek). Při propojování neuronů a modulů jsou některé časté typy spojení zautomatizovány – Jane nabízí *jednoduché spojení* (spoj mezi dvěma objekty), *lineární spojení* (např. při propojení dvou vrstev jsou spojeny navzájem si odpovídající prvky), *úplné spojení* (propojení každého prvku se všemi ostatními).

Je možné si zvolit z několika nabízených simulačních strategií. Dostupná je synchronní simulace, datově řízená simulace, simulace v předem daném pořadí a simulace na základě analýzy topologie sítě. Poslední zmíněná strategie si s některými druhy sítí neporadí, ale Jane je mladý projekt a v době psaní tohoto textu probíhal na něm usilovný vývoj.

Přidanou hodnotou je možnost vykreslování sítí ve formátu svg. Každý objekt si řídí vykreslování sám a pozice jsou přiděleny algoritmem pro analýzu topologie nebo uživatelem.

Pro Jane existuje ještě jednoduchý klient, který dovoluje manipulovat s vytvořenou sítí. Změny v síti se automaticky zahrnou do jejího modelu. Tímto způsobem je zajištěna interaktivita.

2.3 Simulační jazyky, systémy

2.3.1 Simulátor Neural Network Toolbox

Neural Network Toolbox [NNToolbox] je nadstavbou programu MATLAB. Je to nástroj nejen pro simulaci, ale i vizualizaci. Dovede taky spolupracovat s jinými komponentami pro MATLAB, např. Simulinkem. Jedná se o univerzální nástroj neboť jeho funkčnost může být rozšířena pomocí programovacího jazyka zabudovaného v MATLABu (znalost tohoto jazyka je však potřebná i pro běžné použití).

Simulátor umí pracovat s vyššími funkčními celky – moduly. Ty zajišťují znovupoužitelnost, tj. moduly je možné používat opakovaně a není nutné text popisující moduly stále kopírovat.

Výhodou jsou taky „knihovny“ běžných sítí, které jsou již naimplementované a není tedy nutné je znova psát.

2.3.2 Simulátory pro VHDL

VHDL[4] je jazyk určený pro simulaci číslicových obvodů. Tomu je taky uzpůsoben a ikdyž je možné jej použít pro jakoukoliv simulaci a tedy i simulaci neuronových sítí, není pro nasazení v této oblasti příliš vhodný. Zejména malá podpora pro práci s čísly v plovoucí řádové čárce je v případě neuronových sítí poměrně omezující. Na druhou stranu výhodou je podpora hierarchických struktur a z toho plynoucí

[4]VHSIC (Very High Speed Integrated Circuits) Hardware Description Language

vhodnost pro simulaci modulárních neuronových sítí. Jistým omezením je, že VHDL umožňuje propojovat moduly pouze jednoduchým způsobem běžným u číslicových obvodů, možnost vyjádřit složitější propojení chybí (právě díky tomu, že jazyk VHDL není primárně určen k simulaci neuronových sítí).

Vizualizace průběhů je opět přizpůsobená číslicovým obvodům a její použití pro neuronové sítě je tedy omezené. Nespornou výhodou je však možnost syntetizovat hardware z vytvořeného návrhu.

2.3.3 Simulátory SiMoNNe a Simonnec

SiMoNNe a Simonnec patří také mezi univerzální simulátory. Jejich rozbor a bližší popis je v oddíle 3 na straně 11.

2.3.4 Simulátor VeSNNA

VeSNNA je simulátor, který využívá prostředky pro paralelní programování na platformě PC v podobě instrukcí typu SIMD [5]. Používá jazyk, který je i přes rozdílnou syntaxi a sémantiku v mnohém podobný SiMoNNe.

V případě neuronových sítí lze využít SIMD zejména při vektorových operacích [Navrátil-2002]. VeSNNA používá instrukce ze sady SSE (Streaming SIMD Extensions).

Simulátor má pouze jeden číselný typ, což je číslo v plovoucí řádové čárce tak, jak je realizováno v jazyce C++. Přidána je možnost násobit pole hodnot konstantou. Pole jsou však značně omezená. Dovolena jsou pouze jedno a dvojrozměrná pole a prvkem může být jen číslo.

Při použití vektorových operací je výpočet oproti SiMoNNe mnohem rychlejší (zlášť u velkých vektorů). Dále simulátor umožňuje analyzovat vrstvy spojů – VeSNNA umí najít závislosti mezi neurony, seřadit je do vrstev a vrstvy použít pro efektivnější simulaci.

Velkou nevýhodou simulátoru je jeho neinteraktivita. Po předložení vstupního textu je tento interpretován a simulátor je ukončen.

2.4 Jednoúčelové

Jednoúčelových simulátoru existuje mnoho [Murre-1995], zde jsou zmíněny pouze tři vybrané příklady.

2.4.1 GAME

Group of Adaptive Models Evolution [GAME] je simulátor induktivních sítí typu GMDH. Umožňuje tvorbu sítí pro modelování, klasifikaci a predikci. Podporuje tvorbu sítí MIA GMDH, ModGMDH a GAME. Sítě jsou učeny s učitelem a proto je potřeba mít k dispozici pro každý učený vzor odpovídající výstup. Obsahuje několik typů vizualizací a umožňuje tak studovat chování sítí pro různé hodnoty na vstupech, sledovat jak síť aproximuje závislosti v datech, klasifikuje vzory do tříd a další.

[5] SIMD – Single Instruction Multiple Data – jediná instrukce může pracovat s více nezávislými datovými buňkami

2.4.2 SOM_PAK

The Self-Organizing Map Program Package [SOM_PAK] je nástroj pro simulaci Kohonenových sítí, který pochází přímo z jeho laboratoře na HUT[6] v Helsinkách. V balíku se nachází několik programů pro inicializaci a trénování sítě, vyhodnocení chyby a vizualizaci výsledků. Je to poměrně starý nástroj, ale lze jej považovat za referenční. K dispozici jsou i zdrojové kódy.

2.4.3 RSOM&TKM

Jedná se o aplikaci pro simulaci neuronové sítě RSOM (Recurrent Self-Organizing Map) a TKM (Temporal Kohonen Map). S aplikací se pracuje přes grafické uživatelské rozhraní. Umožňuje načítat vstupní data ze souboru a výstup zobrazuje do grafické mřížky. Textovou podobu výstupu je možné uložit do souboru. Síť, její nastavení i nastavení simulátoru se dá také uložit do souboru a při dalším startu pokračovat v místě přerušení.

[6] `http://www.cis.hut.fi/`

Kapitola 3

Předchozí implementace

3.1 Simulátor SiMoNNe

SiMoNNe (Simulator of Modular Neural Networks) je simulátor modulárních neuronových sítí. Jedná se o univerzální simulátor řízený jazykem SiMoNNe Language. Tento jazyk slouží ke komunikaci jak s uživatelem, tak s případnou vyšší vrstvou, typicky GUI nebo textová konzole.

Jazyk SiMoNNe byl původně vytvořen pro simulaci neuronové sítě GOLOKO, protože v té době (rok 2001) ji nebylo možné simulovat v žádném známém simulátoru. Ještě během jeho implementace byl rozšířen obecně pro simulaci modulárních neuronových sítí.

Simulátor sám o sobě je pouze jádro, ke kterému je možné připojit např. GUI nebo konzoli. Komunikace s ním probíhá interaktivně přes textový vstup. Díky tomu, že jazyk SiMoNNe je interpretovaný a se simulátorem je možné pracovat interaktivně, je možné za běhu simulátoru zjišťovat parametry sítě a případně je měnit bez nutnosti restartu simulátoru.

Základním stavebním kamenem pro tvorbu neuronových sítí v jazyce SiMoNNe je modul. Ten může reprezentovat jak jednotlivé neurony, tak seskupení několika modulů, případně celou síť 1.2. Moduly jsou mezi sebou propojeny spoji, přes které se přenášejí hodnoty z výstupů modulů na vstupy následujících modulů (přičemž následníkem může být i původní modul, tj. výstup z modulu je přiveden na vstup téhož modulu).

Simulace probíhá synchronně. To znamená, že přenos informace po spojích probíhá v explicitně určených časových okamžicích. Výhoda je v tom, že je možné sledovat stav simulátoru v průběhu simulace mezi jednotlivými kroky a zároveň simulovat rekurentní sítě.

SiMoNNe pracuje s několika základními datovými typy:

- *celé číslo*,
- *číslo v plovoucí řádové čárce*,
- *řetězec*,
- *pole*,
- *instance modulu*.

Určení typu proměnných probíhá při prvním přiřazení do proměnné podle typu přiřazované hodnoty. V případě přiřazení hodnoty jiného typu se vypíše varovná hláška informující o jeho změně.

Původní verze jazyka poskytuje základní aritmetické operace sčítání, odčítání, násobení a dělení. Operandy mohou být pouze čísla, pokud aspoň jedno z čísel je v plovoucí řádové čárce pak je na stejný typ automaticky konvertován i výsledek operace. Gramatika jazyka dovoluje konstrukci polí jejichž všechny prvky jsou výhradně čísla, nebo výhradně vnořená pole. Pole jsou navíc omezena pouze na tři dimenze. K prvkům pole se přistupuje indexací a uživatel se tak může na hodnotu dotazovat, nebo ji měnit. Změnou hodnoty prvku je možné vytvořit i nehomogenní pole a zároveň použít jako prvky pole jiné typy než pouze čísla nebo vnořená pole.

Moduly mohou být různého typu:

- Moduly psané v přímo v jazyce SiMoNNe.
- Moduly psané v jazyce Java. Umožňují využití výhod jazyka Java (případně jeho knihoven), zejména tedy rozšíření o funkcionalitu, kterou SiMoNNe samotné neposkytuje.
- Moduly konfigurující již definované moduly. Jedná se o možnost změnit chování již nadefinovaných modulů při zachování původního typu modulu.

Při vytvoření nového modulu dojde zároveň k vytvoření nového kontextu, který slouží pouze danému modulu a vyhodnocují se v něm operace prováděné uvnitř modulu. Následně se provede inicializační část definice modulu (část před klíčovým slovem `stepcode`). Výkonná část modulu (za `stepcode`) se provádí při vykonání simulačního kroku (volání `step`). Popisuje transformaci hodnot ze vstupů na výstupy, tj. realizaci přenosové funkce neuronu.

V simulátoru není podpora pro práci s vrstvami, tu však lze nahradit vytvořením požadovaného počtu modulů a jejich ručním propojením.

Simulátor obsahuje interní parser, který převádí vstupní text do vnitřní stromové reprezentace a ta je následně vyhodnocen. Protože, jak již bylo uvedeno, simulátor pracuje interaktivně, parser je volán opakovaně na dílčí vstupy. Při každé změně je nová část vyhodnocena a výsledek je zahrnut do stavu simulátoru.

SiMoNNe poskytuje běžné řídicí struktury: `if`, `if..else`, `for`, `while`, `do..while`. O výpočtu se rozhoduje podle podmínek, které využívají operátorů: `=` (je rovno), `!=` (není rovno), `>` (je větší), `=>` (je větší nebo rovno), `<` (je menší), `<=` (je menší nebo rovno). Výsledky těchto operací se používají pouze pro řízení podmínek, protože jazyk neobsahuje logický datový typ *boolean* a nedá se tudíž uložit do nějaké proměnné a později s ní pracovat.

Zvláštností je operátor `len`, který spíš připomíná funkci v jiných programovacích jazycích. Slouží ke zjištění velikosti pole.

Protože velmi častou činností je dotazování se na hodnotu nějaké proměnné, poskytuje pro tuto činnost SiMoNNe jednoduchý prostředek . Stačí zapsat výraz následovaný středníkem. Díky tomu funguje dotaz na hodnotu proměnné, ale i dotaz na výsledek operace prováděné nad operandy, kterými mohou být přímo číselné hodnoty nebo hodnoty proměnných.

3.2 Simulátor Simonnec

Simonnec [Trávník-2004] je následníkem simulátoru SiMoNNe. Je to implementace nového jazyka SiMoNNe 2, který byl vytvořen s ohledem na nedostatky původního jazyka a simulátor, jak sám jeho autor uvádí, řeší problémy původního SiMoNNe a zároveň simulátoru VeSNNA.

Před implementací simulátoru došlo k poměrně velké změně gramatiky jazyka SiMoNNe a nová verze jazyka byla tedy pojmenována SiMoNNe 2. Jazyk tak doznal poměrně velkých rozšíření. [1] Následuje souhrn podstatných změn proti původnímu simulátoru SiMoNNe.

- Změna některých příkazů na výrazy[2], jedná se zejména o podmíněný příkaz `if`, ale i `while`, `for`, `do..while`.
- Sloučení jednoduchého a „strukturovaného“ odkazu na paměťové místo.
- Podpora pro definici a použití funkcí s využitím libovolného pevně daného počtu argumentů.
- Změna nutnosti zadávat číselnou konstantu u příkazu `connect` při udávání výstupního portu na libovolný výraz.
- Povinnost psát velké počáteční písmeno v názvu modulu a malé první písmeno u názvu proměnných a identifikátorů funkcí.
- Možnost využití výrazů v konstruktoru pole místo nutnosti inicializovat pole číselnými konstantami
- Odstranění možnosti vypsat zpětně kód modulů.

Protože Simonnec měl být zejména akcelerovanou verzí SiMoNNe, byla provedena změna simulace. Zatímco v původní implementaci docházelo k interpretaci objektové reprezentace odpovídající AST[3], Simonnec provádí překlad do kódu virtuálního stroje a jeho následnou interpretaci (překladač je tedy oddělen od výkonné části a pro jejich vzájemnou komunikaci se používá interního jazyka).

V simulátoru Simonnec schází proti původní implementaci znovupoužitelnost kódu, tj. že jakákoliv komunikace s uživatelem může sloužit jako validní vstup pro simulátor.

[1] Protože gramatika nové verze simulátoru SiMoNNe je téměř shodná, její bližší popis je uveden v oddíle 4.2

[2] Výraz narozdíl od příkazu vrací hodnotu.

[3] Abstract Syntax Tree (abstraktní syntaktický strom)

Kapitola 4

Úpravy jazyka

Původním úkolem této práce byla analýza původního simulátoru SiMoNNe, navržení vylepšení a jejich implementace. Protože inovace simulátoru vyžadovala úpravy v gramatice, byla tato kompletně přepsána. Na předchozím jádře simulátoru je vidět, že bylo původně navrženo pro konkrétní typ sítě a až později upraveno na obecnou simulaci více sítí. Proto bylo nutné kompletně přepsat i jádro celého simulátoru. V následujících oddílech budou sepsány požadavky kladené na nový vylepšený jazyk a simulátor a bude popsána gramatika.

4.1 Požadavky kladené na jazyk

1. *Podpora práce s moduly*
   Základním stavebním prvkem sítě je modul. V jazyce jsou tudíž zabudovány pravidla pro definici, vytváření a práci s moduly.

2. *Jednoduchá práce se spoji*
   Přenos hodnot mezi moduly je realizován přes spoje. Jazyk musí podporovat jednoduchou práci se spoji (jejich vytváření a rušení). Syntax operací se spoji nesmí být příliš složitá, pokud možno maximálně unifikovaná.

3. *Jednoduchost jazyka*
   Jedná se zejména o syntaxi a sémantiku, které by měly být snadno pochopitelné.

4. *Správná míra abstrakce*
   Jazyk nesmí omezovat uživatele ve vyjádření myšlenek, ale zároveň musí nabízet jistou míru abstrakce, takže uživatel se nemusí zabývat programováním zbytečných částí kódu, které nesouvisí s implementací algoritmu pro simulaci neuronových sítí.

5. *Podpora synchronní simulace*
   V jazyce musí být prostředky pro umožnění synchronní simulace simulátoru.

6. *Možnost definice funkcí*
   Protože často je potřeba opakovaně volat části kódu, měl by jazyk umožňovat definici a volání funkcí.

4.2 Úpravy gramatiky

Jak již bylo zmíněno, v gramatice došlo k několika zásadním změnám. Tyto změny vedly většinou k větší volnosti zápisu a vyšší expresivitě jazyka. Obě gramatiky jsou na syntaktických diagramech v příloze B, původní na obr. B.1 a obr. B.2, nová na obr. B.3 a obr. B.4. Následuje popis některých změn.

V původní gramatice existují dvě formy zápisu odkazu na paměťové místo[1]. V návaznosti na to jsou tam pak taky dvě možnosti zápisu přiřazení[2]. To je zbytečně komplikované a navíc to s sebou přináší některá omezení. V nové gramatice byla tedy pravidla sloučena. Navíc došlo ke sloučení neterminálů `pointer` (zajišťoval indexaci do pole) a `locator` (sloužil pro přístup k prvkům modulů). Vznikl tak neterminál `pointer`, který zajišťuje jak přístup k prvkům modulů, tak indexaci do pole.

Jak je možné vidět u neterminálu `statement` v původní gramatice, jazyk poměrně striktně odděloval všechny příkazy a výrazy. Díky tomu nebylo možné zapsat například vícenásobné přiřazení. To je však poměrně užitečná vlastnost a nová gramatika by ji měla dovolit. V gramatice SiMoNNe2 [Trávník-2004] bylo dosaženo toho, že mezi výrazy se dostal i podmíněný výraz (dříve podmíněný příkaz) a zároveň s ním i všechny typy cyklů. Občas je tato vlastnost vhodná, ovšem v některých případech může být výsledný zápis značně matoucí a nepřehledný. Z toho důvodu byl podmíněný příkaz i všechny cykly zachovány jako příkazy, ale narozdíl od původní gramatiky bylo umožněno vícenásobné přiřazení.

Objevil se však ještě jeden poměrně častý problém – konflikt přesun/redukce u podmíněného příkazu. V souladu s klasickým přístupem k tomuto konfliktu, byl vyřešen přesunem (stejný problém se vyskytoval i v původní gramatice a jeho řešení bylo identické).

Poměrně zbytečným omezením v původní gramatice byla nutnost použít při konstrukci pole nejdříve číselné konstanty. V nové gramatice byly v konstruktoru pole dovoleny výrazy při inicializaci prvků a práce s poli se tak značně zjednodušila.

V nové gramatice došlo také k vylepšení u příkazu `connect`. Původní verze příkazu dovolovala ke specifikaci výstupního portu připojovaného modulu použít pouze číselnou konstantu. Toto omezení bylo zrušeno a konstanta byla nahrazena výrazem.

Do nového jazyka přibyla podpora pro funkce. Pro definici funkce bylo přidáno nové klíčové slovo `function`. Je možné použít libovolný avšak pevně daný počet argumentů a typ návratové hodnoty není specifikován. Syntax volání funkce byla zvolena stejná jako v jazyce C.

S funkcemi přibyly další možné identifikátory, společně s funkcemi ještě proměnné a názvy modulů. Protože zápis by se mohl stát nepřehledným byl identifikátor rozdělen na `ident` a `identmodule`. Přičemž u `identmodule` bylo zavedeno pravidlo, že začíná velkým písmenem, podobně jako je tomu například u názvů tříd v některých jiných programovacích jazycích. Pro ostatní identifikátory to znamená, že naopak velkým písmenem začínat nesmí.

Během konstrukce nové gramatiky se objevil problém se středníkem za klíčovým slovem `end`[3]. Simulátor používá parser CUP [Hudson-1999], což je LALR parser. U takového parseru lze problém řešit vygenerováním středníku za každým nalezeným klíčovým slovem `end` při lexikální analýze. Lexikální analyzátor JFlex [Klein-2004] toto umí a problém byl tímto způsobem vyřešen.

[1] `pointersimple` a `pointerstructured`
[2] `assignmentsimple` a `assignmentstructured`
[3] Výjimka oproti ostatním konstrukcím v jazyce – nepíše se za ním středník.

Do gramatiky byl dále přidán prázdný příkaz a v definici modulu byla umožněna prázdná inicializační i výkonná část.

4.3 Požadavky na simulátor

Simulátor by měl samozřejmě implementovat jazyk se všemi jeho možnostmi bez omezení. Zároveň lze na něj ještě klást rozšiřující požadavky.

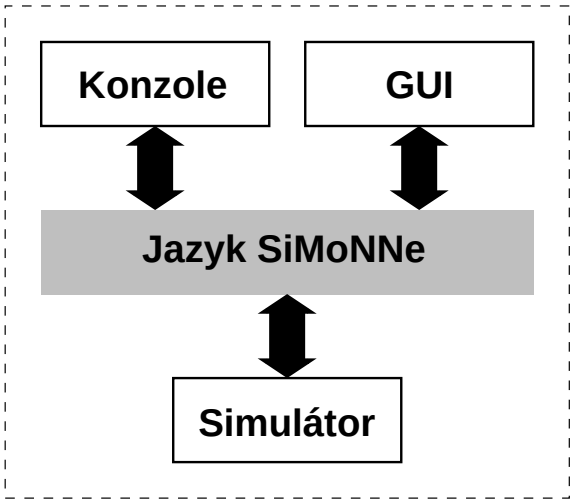
1. *Synchronní simulace*
   Tato vlastnost již byla zmíněna v požadavcích na jazyk, je to však především záležitost samotného simulátoru s tím, že má v jazyce pro ni podporu.

2. *Interaktivita*
   To je velmi důležitá vlastnost a umožňuje uživateli zasahovat do běhu simulace, měnit její parametry a dotazovat se na stav simulace bez nutnosti restartovat simulátor.

3. *Jednoduché rozhranní pro vyšší vrstvy*
   Simulátor by měl být skutečně jenom jádro celého simulačního systému a poskytovat jednoduché rozhraní pro připojení k ostatním vrstvám (konzole, GUI).

4.4 Architektura simulátoru

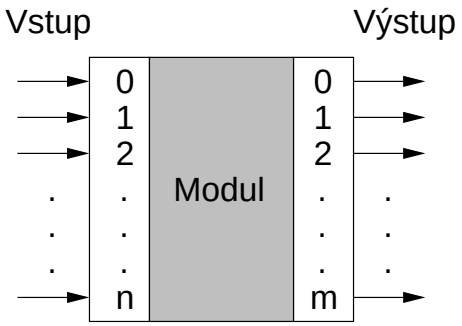
Simulátor SiMoNNe je navržen tak, aby byl absolutně nezávislý na grafickém či jiném rozhraní. Jak je patrné z obrázku 4.1, komunikuje každé rozhraní se simulátorem pouze prostřednictvím jazyka SiMoNNe. To má výhodu, že simulátor funguje pouze jako jádro celého systému, který může být reprezentován pouze jednoduchou konzolí, grafickým uživatelským rozhraním (GUI), či síťovým rozhraním pro připojení dalších klientů.



Obrázek 4.1: Architektura simulátoru. Jádro simulátoru komunikuje s okolním světem pouze přes jazyk SiMoNNe.

4.5 Moduly

Modul je základní stavební jednotkou v simulátoru. Jeho strukturu je možné vidět na obrázku 4.2.



Obrázek 4.2: Struktura modulu. Modul je tvořen svým tělem, vstupem a výstupem.

Modul se skládá z jednoho či několika vstupů a výstupů (pinů) a těla, které transformuje vstupní data na výstupní (v případě neuronu se jedná o přenosovou funkci). Počet vstupů a výstupů může být samozřejmě rozdílný a dynamicky se mění nově přidávanými spoji nebo při přiřazení hodnoty výrazu do daného pinu. Moduly mohou být do sebe i vnořené.

4.6 Spoje

Další důležitou částí SiMoNNe jsou spoje. Jsou to datové cesty, po kterých se přenášejí hodnoty z výstupů na vstupy modulů.

Do jednoho vstupu modulu může vést pouze jeden spoj, viz obrázek 6.2 v oddíle 6.5.1. Důvodem je, že by nebylo možné rozlišit, který spoj bude poskytovat hodnotu pro výpočet v modulu. Simulátor nedovolí dostat se do takového stavu (více o spojích v oddíle 6.5 na straně 30).

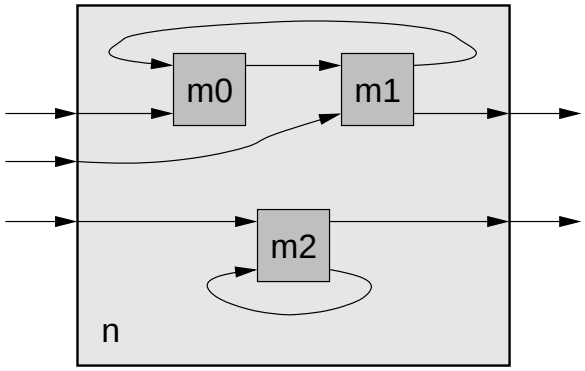
Jak je vidět na obrázku 4.3 spoje mohou být mezi jednotlivými moduly, ale zároveň i v rámci jednoho modulu (výstup je přiveden na vstup téhož modulu). Spoje nemusí být pouze dopředné, ale mohou vést i do předchozích modulů, čímž je umožněno simulovat i rekurentní sítě.

4.7 Funkce

SiMoNNe poskytuje možnost použití dvou druhů funkcí. Jedná se o *vlastní funkce* (zapsané v jazyce SiMoNNe) a *externí funkce* (zapsané v jazyce Java).

V původním jazyce SiMoNNe funkce neexistovaly a uživatel tak byl nucen neustále opisovat opakující se části kódu. *Vlastní funkce* tuto nevýhodu odstraňují a usnadňují tak uživateli práci.

Pokud prostředky jazyka SiMoNNe neposkytují potřebnou funkcionalitu (SiMoNNe například neposkytuje generátor náhodných čísel), je možné rozšířit vlastnosti simulátoru *externími funkcemi*, které zpřístupní funkce a možnosti jazyka Java, ve kterém je SiMoNNe implementováno. Externí funkce se do simulátoru přidávají jako přeložené třídy. Jejich výhodou je, že je lze přidávat do simulátoru za běhu. Pokud tedy uživatel během simulace zjistí, že mu nějaká funkce chybí, stačí, když si pouze překopíruje



Obrázek 4.3: Spoje a možnosti propojení uvnitř modulu. Moduly mohou být vnořené (moduly m v modulu n) a mohou být mezi sebou různě propojené. S vnějším světem komunikují přes vstupy a výstupy obalujícího modulu.

její přeloženou implementaci do balíku `simonne.simulator.functions` a bez restartu simulátoru ji může začít ihned používat.

Kapitola 5

Popis implementace

V kapitole je uvedena implementace některých části simulátoru a nabízí tudíž případnému zájemci její osvětlení.

5.1 Základní struktura simulátoru

Simulátor je napsán kompletně v jazyce Java. Pro svou činnost využívá několika komponent. První je lexikální analyzátor JFlex (The Fast Scanner Generator for Java) [Klein-2004], který slouží ke generování lexikálních symbolů ze vstupního textu. Symboly jsou následně zpracovány parserem CUP (LALR Parser Generator for Java) [Hudson-1999], který na základě gramatiky vytváří *abstraktní syntaktický strom* (AST). Ten je posléze interpretován jádrem simulátoru. Popis lexikálních symbolů je v souboru `simonne.flex` a pravidla gramatiky pro CUP jsou uvedena v souboru `simonne.cup`. Poslední komponentou je commons logging [CLogging] pro logování chybových hlášek.

Jádro simulátoru je složeno z několika balíků (viz tabulka 5.1).

Tabulka 5.1: Stručný popis obsahu jednotlivých balíků simulátoru SiMoNNe.

`simonne.simulator.config`	Obsahuje třídu pro konfiguraci simulátoru.
`simonne.simulator.connections`	Třídy pro spoje mezi moduly (datové cesty).
`simonne.simulator.exceptions`	Výjimky simulátoru.
`simonne.simulator.functions`	Implementace externích funkcí.
`simonne.simulator.nodes`	Třídy s implementací uzlů AST.
`simonne.simulator.runtime`	Třídy s implementací vnitřních prvků simulátoru.
`simonne.simulator.values`	Implementace datových typů.

Jakmile je vytvořen AST, simulaci je možné ihned spustit voláním metody `eval()`, kterou mají všechny uzly AST, a která zajišťuje vyhodnocení uzlu. Parametrem metody je tzv. *kontext*.

5.2 Kontext

Kontext je klíčový objekt celého simulátoru. Je to paměťové místo, ve kterém se uchovávají hodnoty jednotlivých proměnných a určuje též jejich viditelnost. Je reprezentován třídou Context, která je potomkem třídy SymbolTable. Ta uchovává, jak již její název naznačuje, symboly. Jsou to dvojice *jméno-hodnota*, ve kterých jsou uloženy všechny proměnné simulátoru. Protože je velmi důležité mít možnost vidět v hierarchicky níže postavených jednotkách na proměnné z vyšších kontextů (kontexty hierarchicky vyšších jednotek), má každý kontext i svůj tzv. *super kontext*, což je kontext hierarchicky výše postaveného prvku. Vyhledávání potom probíhá tím způsobem, že se nejdříve prochází kontext přiřazený aktuálnímu prvku a až v případě, že proměnná (symbol v tabulce symbolů) není nalezena, prohledávají se nadřazené kontexty podle totožného schématu. Pokud proměnná není nalezena ani v těchto kontextech je vytvořena a uložena do svého kontextu.

Struktura propojení kontextů ještě však ve zvláštních případech obsahuje tzv. *paralelní kontexty*. To jsou kontexty, ve kterých se ukládají a vyhodnocují řídicí proměnné cyklů. Ty jsou zvláštní tím, že platí pouze v době provádění cyklu a po jeho ukončení zanikají. Při jejich volání se tedy vytváří paralelní kontext a vyhledávání proměnné probíhá tak, že nejdříve se hledá v paralelním kontextu a teprve poté v aktuálním kontextu a jeho „nadkontextech“.

5.3 Datové typy

Jádro simulátoru má své vlastní datové typy a aritmetiku. Implementace vlastní aritmetiky přináší výhody zejména velké flexibility jazyka díky tomu, že při ní bylo využito techniky *multiple dispatchingu* [HebJohn-1990]. Datové typy (kromě modulů, u kterých to však není potřebné) tak umí pracovat se všemi operacemi, které SiMoNNe aritmetika nabízí.

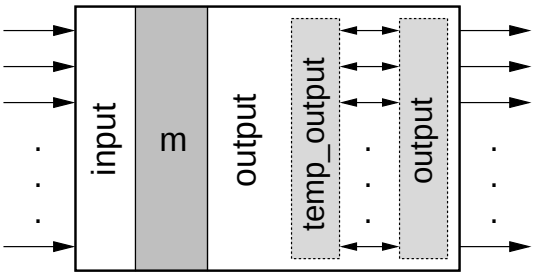
Datové typy jsou v balíku `simonne.simulator.values` a všechny implementují rozhraní IValue. To zaručuje, že všechny třídy, které jej budou implementovat, budou poskytovat metody pro sčítání, odčítání, násobení, dělení, porovnávání a další. Datové typy byly uzpůsobeny potřebám simulátoru a umožňují tak jednodušeji provádět operace, které jsou při simulaci často potřebné. Velké nároky byly kladeny zejména na datový typ pole, který je v původní implementaci omezen pouze na dvě dimenze vnoření. Pole mají často klíčovou úlohu nejen při samotném výpočtu, ale i při propojování, a proto byl vytvořen nový typ pole, který dovoluje neomezené vnořování a díky *multiple dispatchingu* může být použit i při operacích s jednoduchými datovými typy (IntConst, DoubleConst, StringConst).

Datový typ pole též implementuje rozhraní IValue a je realizován pomocí třídy java.util.ArrayList. Jeho prvky mohou být potom opět objekty jejichž třídy implementují rozhraní IValue, což znamená, že i pole samotné může být svým prvkem – tím je dosaženo vnořování. Navenek potom poskytuje metody pro aritmetické operace a ostatní metody podle rozhraní. Při implementaci metod sčítání, odčítání, násobení a dělení je využito skutečnosti, že v poli se mohou vyskytovat pouze objekty se společným nadtypem IValue a *multiple dispatching* zařídí, že všechny operace budou provedeny korektně.

5.4 Moduly

Modul je případem strukturovaného datového typu. Je však natolik odlišný od datových typů implementujících rozhraní IValue, že v této kapitole bude popsán samostatně.

Jak již bylo popsáno v oddíle 4.1, SiMoNNe pracuje synchronně. Tento typ simulace musí být v modulech podporován, protože jinak by se výstupy modulů měnily automaticky a při simulačním kroku, ve kterém probíhá výpočet u více modulů, by nové hodnoty na výstupech některých z nich nahradily ty původní, které měly být použity jako vstup pro jiné moduly. Problém je vyřešen *dočasným výstupem* (viz obrázek 5.1).



Obrázek 5.1: Struktura modulu z hlediska implementace. Modul obsahuje své tělo (tmavě šedá část), vstup a výstup. Výstup je tvořen dvěmi částmi – dočasným výstupem (temp_output) a skutečným výstupem (output), které slouží pro realizaci synchronní simulace.

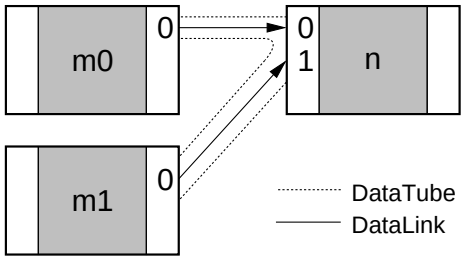
Výpočet při volání simulačního kroku probíhá dvojfázově. Nejdříve se přepočítají hodnoty výstupů pro každý volaný modul do dočasného výstupu (`temp_output` v obrázku) a potom se najednou u všech modulů nahradí jejich výstup (`output` v obrázku) dočasným výstupem s novými hodnotami. Navenek se však modul tváří, jako by měl pouze jeden výstup. Tím je zajištěno, že k výpočtu se nebudou používat nesprávné hodnoty (v aktuálním čase neexistující), ale hodnoty původní.

5.5 Spoje

Spoje společně s moduly tvoří nejdůležitější části jazyka. Proto jim také byla věnována velká pozornost a velký důraz byl kladen zejména na práci se spoji. Náhled na implementaci spojů je na obrázku 5.2.

Každý spoj musí mít vždy určeny položky: konzument (modul, do nějž spoj vstupuje), vstupní pin, producent (modul, z nějž spoj vystupuje), výstupní pin. Konzument je určen objektem třídy DataTube, který u každého modulu reprezentuje vstupní pole (uvnitř modulu přístupné přes proměnnou `input`). V rámci tohoto pole má svou pozicí v něm každý spoj přiřazen i vstupní pin. Položky pole jsou objekty třídy DataLink, které určují svými proměnnými jak výstupní modul (proměnná `output`), tak i výstupní pin. Tím je spoj popsán jednoznačně.

Protože v SiMoNNe jsou dovoleny i vnořené moduly a tím i jejich propojení na vstupy či výstupy obalujících modulů, nastává problém, že *input* a *output* nejsou moduly a není možné k nim tedy přistupovat jako k ostatním modulům. Transparentnost přístupu byla jedním z hlavních požadavků při návrhu simulátoru a tuto nesrovnalost bylo tedy nutné vyřešit. Řešení je poměrně nasnadě. Protože *input* i *output* se mají chovat jako moduly, je nejlepší když moduly také budou. V implementaci to znamená,



Obrázek 5.2: Implementace spojů. Tečkovanou čárou je znázorněn DataTube, který seskupuje všechny spoje vedoucí do jednoho modulu. Šipkama jsou nakresleny DataLinky (prvky objektu DataTube) – objekty nesoucí hodnotu z výstupu do vstupu modulu.

že třída DataTube dědí od třídy Module, překrývá některé metody a nechává si svou funkčnost vstupního či výstupního pole.

5.6 Funkce

Funkce jsou dvojího typu – vlastní funkce a externí funkce.

Vlastní funkce jsou součástí jazyka SiMoNNe a zapisují se v něm. Jejich definice se ukládá do zvláštního kontextu pouze pro funkce. Při jejich vyhodnocování se vytváří nový vlastní kontext, který dostává jako parametr ještě starý kontext pro funkce (to je z důvodu možnosti volat jiné funkce, případně sama sebe v případě rekurzivních funkcí). Zároveň se při volání funkce předává ještě seznam parametrů, které se ukládají do nového kontextu funkce.

V SiMoNNe neexistuje příkaz na okamžité ukončení provádění kódu (např. v cyklech), je však nutné zařídit návrat hodnoty z funkce. K tomu slouží klíčové slovo `return`, které vrací výraz uvedený za ním. Výraz za tímto klíčovým slovem se netiskne na konzoli. Pokud ve funkci uvedené není, pak se vrací hodnota posledního příkazu pokud tento vrací hodnotu. Když nedojde ani k jednomu z uvedených případů, vrátí se hodnota 0 (nula).

Externí funkce jsou funkce zapisované v jazyce Java a simulátor si je připojuje za běhu pomocí mechanizmu reflexí[1] (angl. *reflections*). Díky tomu je možné nové externí funkce přidávat za běhu. Stačí pouze splnit několik podmínek (více viz 6.4.2). Po zavolání externí funkce se nejdříve zjistí jméno třídy, ve které je kód s implementací (první písmeno volané funkce se změní na velké, přidá se předpona Function a přípona .java) a zavolá se metoda eval, kterou musí všechny třídy implementující externí funkce obsahovat. Návratová hodnota z metody se vrátí jako výsledek. Aby byla zaručena kompatibilita mezi vnitřními hodnotami v simulátoru a hodnotami s nimiž pracují externí funkce, musí se parametry předávat pomocí datových typů SiMoNNe a rovněž návratová hodnota musí být některého z těchto typů.

Vzhledem k tomu, že existují dva druhy funkcí, je nutné rozhodnout, které se budou volat dříve v případě, že existují stejné názvy funkcí u obou typů. Simulátor tedy nejdříve prohledává vlastní funkce a teprve pokud neuspěje, pokusí se implementaci funkce najít mezi externími funkcemi.

[1] `http://java.sun.com/developer/technicalArticles/ALT/Reflection/`

5.7 Systémové proměnné simulátoru

Simulátor má tzv. systémové proměnné, které slouží k jeho parametrizaci. K uchování a poskytování těchto proměnných slouží třída `Config` z balíku `simonne.simulator.config`. Proměnné nejsou ukládány do kontextu, ale do zvláštní třídy, aby bylo možné se na ně dotazovat kdekoliv v simulátoru bez nutností znát kontext. Třída si udržuje privátní statickou proměnnou, ve které má instanci sama sebe a navenek ji nelze vytvořit protože má privátní konstruktor. Tím je zaručeno, že bude v simulátoru pouze jedna instance této třídy. Pro přístup ke konfiguračním proměnným jsou ve třídě statické metody. Nastavení systémových proměnných probíhá přes jazyk SiMoNNe, neboť jsou to běžné proměnné jako kterékoliv jiné.

5.8 Logování chyb

Velmi důležitým prostředkem pro hledání chyb v kódu jsou chybové hlášky simulátoru. Pro tzv. logování byla použita již hotová komponenta *commons logging* [CLogging] z *The Jakarta Project*. Naprosto postačuje pro činnosti, které jsou požadovány.

Začlenění logování do simulátoru je velice jednoduché. Protože objekt, nad kterým se volají metody pro logování je rozpoznáván přes jméno, udržuje se i při více voláních konstruktoru jeho třídy pouze jedna jeho instance. Během výstavby AST je tedy jednoduše při volání konstruktorů jednotlivých uzlů zavolán i konstruktor třídy logovacího objektu a každý uzel tak má přístup k jedné (stejné) instanci logovacího objektu. Při výskytu chyby pak stačí vždy jen zavolat příslušnou metodu.

Pro logování je možné zvolit několik úrovní, od jemného logování (hlášky pro odstranění chyb) až po logování pouze kritických chyb. Změna úrovně se provádí v souboru `logging.properties` v adresáři `conf` v kořenovém adresáři projektu.

5.9 Dotazy do simulátoru

Při implementaci byl rovněž velký důraz kladen na dotazy na hodnoty proměnných do simulátoru. Tuto funkci zastává třída `NodePointer`. Klíčové jsou zejména tři její metody – `getContext`, `getLast` a `eval`. Při procházení a interpretaci AST se vždy nejdříve volá metoda eval. Proměnné však mohou být vnořené v modulech nebo polích a všechny tyto objekty se mohou vnořovat do sebe navzájem a mohou tak vytvářet složitou hierarchii vnoření. Proto je nejdříve důležité získat správný kontext. K tomu slouží metoda getContext, která projde vnořené objekty až do předposlední úrovně a tu vrátí jako kontext, ve kterém se bude vyhledávat požadovaná proměnná. Metoda getLast pouze ze složeného jména získá jméno platné v tomto kontextu. V metodě eval už potom stačí pouze najít symbol odpovídající jménu proměnné v získaném kontextu, případně zahlásit chybu pokud se tam nevyskytuje.

S dotazy do simulátoru je taky těsně spjat jejich případný výpis do konzole. Ten je realizován ve třídě `NodeCommandSemi`, která podle zpracovávaného uzlu a výsledku pozná zda šlo o dotaz a případný výsledek vytiskne na konzoli. K tisku slouží třída `VirtualConsole`, která tvoří obal nad klasickými prostředky výpisu jazyka Java. Výhoda je v tom, že uživatel není nucen používat jednu konkrétní konzoli, ale bez zásahu do simulátoru ji může vyměnit za jinou, která například místo výpisu na obrazovku bude text ukládat na disk do souboru.

Kapitola 6

Uživatelská příručka – manuál

Tato kapitola pohlíží na simulátor z pohledu návrháře neuronové sítě a experimentátora. V následujícím textu budou vysvětleny všechny konstrukce jazyka a na příkladech probráno jejich použití. Příklady jsou sázeny strojovým typem písma a vstupní text je odlišen od výstupu simulátoru tak, že je sázen tučně.

6.1 Datové typy

6.1.1 Číselné datové typy a typ řetězec

V jazyce SiMoNNe je možné pracovat s třemi základními datovými typy známými z klasických programovacích jazyků.

- IntConst – celočíselný datový typ (*integer*),
- DoubleConst – číslo s pohyblivou řádovou čárkou (*double*),
- StringConst – datový typ řetězec (*string*).

Typ proměnných není třeba předem deklarovat, je určen v momentě přiřazení do proměnné.

`x = 1;`	*IntConst*
`y = 1.0;`	*DoubleConst*
`z = ”retezec”;`	*StringConst*

Více o implementaci viz oddíl 5.3.

6.1.2 Datový typ pole

SiMoNNe má svůj vlastní datový typ *pole*. Počet dimenzí pole není nijak omezen (pouze velikostí paměti). Není však dovoleno vytváření tzv. *řídkých polí*[1]. Pole mohou být *heterogenní*, je tedy možné přiřadit do pole prvky různých datových typů.

[1] Řídké pole je takové, v němž je dovoleno mít prvek bez hodnoty, resp. prvky polí na některých indexech neexistují. V krajních případech pak může docházet k tomu, že např. v poli o velikosti 100 na prvních 99 pozicích není žádný prvek a poslední stá pozice má prvek s nějakou hodnotou.

Při konstrukci pole se používají složené závorky, pokud chceme přistupovat k prvkům v poli, používají se klasicky hranaté závorky pro vyjádření indexu do pole.

Možností konstrukce pole je několik:

Inicializace heterogenního pole přiřazením

```
array1 = {1.0, 3, "retezec"};
```

Na rozdíl od původní implementace je možné již při konstrukci pole přiřazovat do něj různé datové typy.

Inicializace vícerozměrného pole

```
array2 = {{1.0, 1.1, 1.2},
          {2.0, {2.1}}};
```

Dimenze ani ortogonalita pole není nijak omezena.

Inicializace pole s použitím proměnných

```
array3 = {array1, array2};
```

Při inicializaci pole je možné použít proměnné. Pokud se jedná o jednoduché proměnné (čísla a řetězec), změna hodnoty v původní proměnné nemá na hodnotu v poli žádný vliv. V případě, že inicializační proměnnou je pole nebo modul, pak pokud se změní hodnota v původní proměnné, promítne se změna i do pole.

Inicializace pole postupným rozšiřováním

```
array4[0] = 1;
array4[1] = 2;
array4[2] = 3;
```

Pro jednoduchost není potřeba před vkládáním prvků pole vytvářet, inicializovat či jinak specifikovat jeho velikost. Podmínkou však je, že nový index je první volný (nevyužitý) index v poli. Pro nově vytvářená pole to znamená, že index musí být 0 (ve všech nově přidávaných dimenzích).

Indexace do pole je obdobná jako v klasických programovacích jazycích – zapisuje se pomocí hranatých závorek. Příklad přístupu do vícerozměrného pole:

```
array = {{{0, 1}, 2, {4, 8}}};
array[0];
{{0, 1}, 2, {4, 8}};

array[0][2][1];
8;
```

6.1.3 Aritmetické operace

Výsledky operací sčítání, odčítání, násobení, dělení jsou definovány podle tabulky 6.1. Pro přehlednost byla v tabulce jména typů zkrácena a tedy IntConst = IC, DoubleConst = DC a StringConst = SC.

Typ výsledku aritmetické operace se vždy řídí podle *síly* typů operandů. Přičemž platí, že StringConst $>$ DoubleConst $>$ IntConst. Tak například výsledkem součtu IntConst $+$ DoubleConst je DoubleConst. Jediná výjimka je v případě dělení u celých čísel (IntConst), protože se při používání ukázalo,

Tabulka 6.1: Výsledné typy hodnot aritmetických operací. Jména typů jsou v tabulce zkrácena: IntConst = IC, DoubleConst = DC, StringConst = SC.

Sčítání	*IntConst*	*DoubleConst*	*StringConst*
IntConst	IC	DC	SC
DoubleConst	DC	DC	SC
StringConst	SC	SC	SC
Odčítání	*IntConst*	*DoubleConst*	*StringConst*
IntConst	IC	DC	—
DoubleConst	DC	DC	—
StringConst	—	—	—
Násobení	*IntConst*	*DoubleConst*	*StringConst*
IntConst	IC	DC	SC
DoubleConst	DC	DC	—
StringConst	SC	—	—
Dělení	*IntConst*	*DoubleConst*	*StringConst*
IntConst	DC *	DC	—
DoubleConst	DC	DC	—
StringConst	—	—	—

** Výsledek dělení dvou celých čísel je možné nastavit pomocí systémových proměnných (viz 6.11). Pokud je typ výsledku nastaven na DoubleConst a výsledek po dělení je celé číslo beze zbytku, pak je mu ponechán typ IntConst a není konvertováno.*

že mnohem užitečnější je aby výsledný typ operace celočíselného dělení byl DoubleConst (výsledek operace je možné nastavit konfigurací systémových proměnných – více v oddíle 6.11).

Všechny operace lze rovněž provádět s poli, přičemž s prvky polí je nakládáno podle předchozí tabulky 6.1.

Následuje několik příkladů operací a jejich výstup (opět ve formě, která je zpracovatelná simulátorem). Na příkladech jsou zároveň vidět typy výsledků operací a asociativita.

```
1 + 2;
3;

3 / 2;
1.5;

”16 deleno 2 je ” + (16 / 2);
”16 deleno 2 je 8.0”;

2 + 3 * 4;
14;

(2 + 3) * 4;
20;
```

Další aritmetickou operací je unární mínus. Výsledný typ je vždy dán podle typu operandu a unární mínus nelze aplikovat na StringConst.

6.1.4 Logické operace

V SiMoNNe je možné použít logické operátory podle tabulky 6.2.

Tabulka 6.2: Logické operátory v SiMoNNe.

Operátor	Význam
$<$	je menší
$<=$	je menší nebo rovno
$==$	rovná se
$>=$	je větší nebo rovno
$>$	je větší
$!=$	nerovná se

Výsledky operací jsou použity v podmínkách (více viz oddíl 6.6). SiMoNNe nezná datový typ boolean a není tedy možné uložit výsledek do nějaké proměnné a později ho použít k dalšímu výpočtu.

6.2 Moduly

Velmi důležitou součástí jazyka jsou moduly. V následujícím příkladu je uvedena základní forma zápisu modulu.

```
moduledef M begin
  coef = 0.5;
stepcode
  output[0] = output[0] * coef;
end
```

Zápis modulu je rozdělen na dvě části. V první části (před klíčovým slovem `stepcode`), neboli taky *inicializační části*, je uveden kód, který se vykoná pouze při vytvoření modulu (je to obdoba konstruktoru). Druhá část (*výkonná*) popisuje činnost modulu v simulačním kroku a provádí se vždy při zavolání příkazu `step` na danou instanci modulu.

Protože simulace probíhá synchronním způsobem, je vždy zajištěno, že výstupní hodnoty modulů se změní až po přepočítání všech modulů volaných v daném simulačním kroku.

Předchozí zápis slouží pouze k nadefinování nového typu modulu, k vytvoření jeho instance je potřeba ještě užít klíčového slova `new`. Instance modulů je možné přiřazovat nejen do proměnných, ale i do polí. Ukázky instanciace modulů jsou v následujícím příkladu.

```
m = new M;
n = new M;
array = {m, n};
array[2] = new M;
```

Oproti dřívější verzi SiMoNNe je možné v současné verzi do modulu při jeho vytváření předat parametry. Ty jsou v modulu přístupné přes proměnnou `arg`. Parametry předávané do modulu se zapisují při vytváření instance za jméno typu modulu. Jako parametr je možné předat jakýkoliv datový typ. Příklad použití je uveden v následujícím kódu.

```
moduledef A begin
  input = arg;
stepcode
  output[0] = input[0] * 10;
end

a = new A {0.5, 2.3};
```

Předávané parametry se kopírují, jedná se tedy o obdobu předávání parametrů hodnotou u funkcí z klasických programovacích jazyků. To znamená, že pokud provedeme změny v proměnné, která sloužila jako parametr při konstrukci modulu, změny se v modulu neprojeví.

6.3 Dotazy do simulátoru

Jednoduchou, ale důležitou, funkcí jsou dotazy na hodnoty proměnných do simulátoru. Je tak možno zjistit hodnoty proměnných v aktuálním oboru viditelnosti, ale díky vnořování je možné zjišťovat i proměnné v polích, modulech, či kombinaci obojího při vzájemném vnoření. Způsob zápisu je nejjednodušší ilustrovat na příkladech.

```
moduledef A begin                    vytvoření typu modulu
  input = arg[0];                    přiřazení parametru modulu
  x = arg[1];
stepcode end

a = new A {{2, 4}, 8};               vytvoření instance modulu
b[0] = new A {{16, 32}, 64};         vytvoření pole modulů
b[1] = new A {{128, 256}, 512};
c = 1024;                            přiřazení hodnoty do proměnné

c;                                   jednoduchý dotaz
1024;

a.x;                                 dotaz do modulu
8;

a.input[0];                          dotaz do pole v modulu
2;

b[0].input[1];                       složitější dotaz do pole v modulu v poli
32;
```

6.4 Funkce

SiMoNNe má dva druhy funkcí. Prvním typem jsou tzv. *vlastní funkce*, jejichž kód je zapsán v jazyce SiMoNNe. Druhým typem jsou *externí funkce*, které jsou zapsány v jazyce Java a slouží k rozšíření možností simulátoru o složitější konstrukce, které by v SiMoNNe nebylo možné jednoduše vyjádřit (zejména proto, že SiMoNNe je především jazyk pro simulaci, zatímco složité matematické výpočty je jednodušší vyjádřit v jazyce Java).

6.4.1 Vlastní funkce

```
function max(a, b) begin
  if (a > b)
    return a;
  else
    return b;
end
```

Hlavička funkce je uvozena klíčovým slovem `function`, následuje jméno funkce a kulaté závorky v nichž mohou ale nemusejí být uvedeny parametry funkce, kterých může být libovolný, ale pevně daný počet. Za hlavičkou je pak tělo funkce, které se vykoná při každém zavolání funkce. Výstup hodnoty z funkce je realizován přes klíčové slovo `return`. Návratová hodnota funkce se nespecifikuje, může tedy být jakéhokoli typu.

Předávání parametrů probíhá stejným způsobem jako v případě modulů. Pokud je předáván jednoduchý datový typ (IntConst, DoubleConst, StringConst), pak se jeho hodnota kopíruje, v případě, že parametrem je pole, nebo modul, změna předávaného parametru provedená ve funkci zůstane platná i po jejím ukončení.

Vlastní funkce se dají při simulaci využít zejména při učení neuronových sítí.

6.4.2 Externí funkce

Pokud prostředky jazyka SiMoNNe neposkytují potřebnou funkcionalitu, je možné ji rozšířit tzv. *externími funkcemi*. Ty jsou implementovány v jazyce Java a do simulátoru se přidávají jako přeložené třídy. Je to jednoduchá možnost jak rychle rozšiřovat schopnosti simulátoru. Při jejich implementaci je nutné dodržet pouze několik podmínek.

1. Třída s implementací dané funkce je umístěna v balíku `simonne.simulator.functions`,
2. název třídy se skládá z prefixu `Function` a jména funkce s velkým prvním písmenem, například třída implementující funkci `exp` bude mít název `FunctionExp.java`,
3. třída musí obsahovat metodu `eval()`, která má stejný počet a typ parametrů jako funkce volaná ze simulátoru a návratová hodnota je typu `Object`,
4. komunikace mezi simulátorem a *externími funkcemi* probíhá v datových typech jazyka SiMoNNe.

Výhodou těchto funkcí je, že se dají přidávat do simulátoru za běhu. Pokud tedy uživatel zjistí během simulace, že mu nějaká funkce chybí, stačí pouze když překopíruje její přeloženou implementaci

do balíku `simonne.simulator.functions` a bez restartu simulátoru ji ihned může začít používat.

Následující kód uvádí příklad implementace funkce `sigmoid`, což je jedna ze základních přenosových funkcí.

```
package simonne.simulator.functions;

import simonne.simulator.values.DoubleConst;
import simonne.simulator.exceptions.SimonneIllegalMethodException;

public class FuncionSigmoid {
  public Object eval(DoubleConst x, DoubleConst gamma) throws
    SimonneIllegalMethodException {

    double d = x.getDouble().doubleValue();
    double e = gamma.getDouble().doubleValue();

    return new DoubleConst(1 / (1 + Math.exp(-e * d)));
  }
}
```

Po přeložení třídy s implementací funkce a překopírování výsledného souboru do patřičného balíku s funkcemi stačí v kódu SiMoNNe zavolat funkci `sigmoid` naprosto stejně jako se volají *vlastní funkce*.

```
a = 2.1;
sigmoid(3, a);
0.99817;
```

6.5 Spoje

```
m = new M;
n = new N;
o = new O;
connect n to m:0;
connect n:1 to m:3;
connect o to {n:0, m:2};
```

Vedle modulů jsou další klíčovou částí jazyka pro popis simulace neuronových sítí spoje. Při vytváření jazyka SiMoNNe byl kladen důraz na jednoduchost propojování modulů. Díky tomu je zápis propojování různých typů objektů mezi sebou pro uživatele naprosto transparentní a existují pouze tři možnosti jak propojení zapsat. Všechny možnosti jsou uvedeny v předešlém příkladu. V prvním případě je vstup modulu *n* spojen s nultým výstupním pinem modulu m. V druhém případě je specifikován i vstupní pin modulu *n*. Rozdíl mezi nimi je v tom, že zatímco první propojení nespecifikuje vstupní pin modulu *n*, a spoj je tudíž připojen na první nepoužitý pin vstupu[2], druhý zápis přesně udává, který pin[3] je propojen z výstupním pinem modulu *m*. Poslední zápis je určen pro případy, kdy je potřeba propojit vstup modulu s různými výstupními piny modulů v poli.

[2] Vstupní pole nemůže být řídké, to znamená, že neexistují „díry“ mezi zapojenými vstupy.

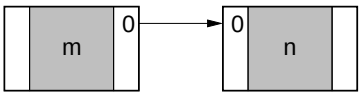
[3] Uvedený pin musí existovat a nebo to musí být první nezapojený pin.

Jak již bylo uvedeno, tři výše popsané způsoby jsou jediné tři možnosti jak propojení modulů zapsat. Transparentnost spočívá v tom, že simulátor na základě znalosti propojovaných objektů sám rozhodne jak je spojit. Pokud se jedná pouze o moduly, je situace jasná. Za proměnnou *n* či *m* se však může skrývat například pole modulů. Potom je propojení realizováno na každý modul daného pole. Možné situace jsou znázorněny na následujících příkladech.

6.5.1 Connect – vytvoření spoje

Příklad: Propojení modul – modul

```
# n ještě nemá žádné vstupní spoje
connect n to m:0;
# nebo jinak
connect n:0 to m:0;
```

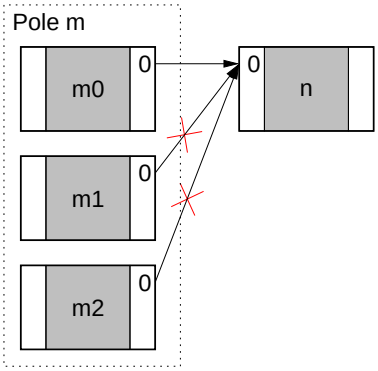


Obrázek 6.1: Spojení modul – modul. Šipka znázorňuje spoj vedoucí z modulu m do modulu n.

Nejjednodušší spojení. Oba dva objekty v příkazu `connect` jsou moduly. V případě, že modul *n* ještě nemá žádný spoj jsou oba příkazy pro spojení ekvivalentní. Oba příkazy tedy říkají: „*spoj nultý pin modulu n s nultým výstupním pinem modulu m*“.

Příklad: Propojení modul – pole modulů (zakázané spojení)

```
connect n:0 to m:0;
```



Obrázek 6.2: Zakázané spojení modul – pole modulů. Více spojů vedoucích do jediného vstupu není povoleno.

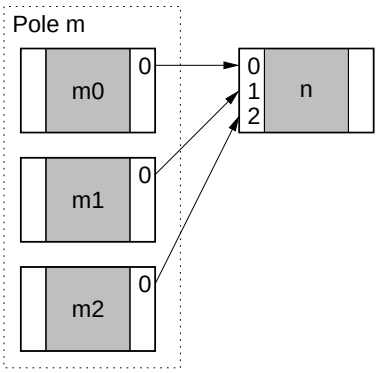
Jedná se o propojení kdy objekt *n* je modul a objekt *m* je pole modulů. Takto zapsaný typ propojení je zakázaný, protože jak již je vidět z obrázku 6.2, není možné v pinu, kde se všechny spoje scházejí rozlišit, která hodnota z několika možných bude použita jako vstup.

Pokud uživatel takový typ spojení zapíše, pak je informován chybovým hlášením.

```
# m je pole modulů
connect n:0 to m:0;
# Forbidden connection
```

Příklad: Propojení modul – pole modulů

```
connect n to m:0;
```



Obrázek 6.3: Spojení modul – pole modulů. Při připojení pole se každý nový spoj připojí na první volný vstup koncového modulu.

V tomto příkladu jde o stejný typ propojení jako v příkladu předchozím. Rozdíl je v tom, že není specifikován přesně vstupní pin modulu *n*. Zápis z předchozího příkladu není možný, protože z připojovaného pole modulů se vygeneruje několik spojů, které jsou po řadě připojeny na nejbližší volné vstupní piny modulu *n*. Výsledkem tedy bude vstup modulu, kde každý pin bude svázaný s modulem z připojovaného pole na indexu, který odpovídá číslu pinu.

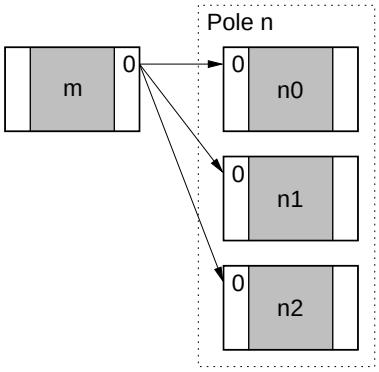
```
connect n to {m:0, m:1, n:0};
```

Alternativou je ještě příkaz connect s přesnou specifikací pole modulů, kdy uživatel do pole explicitně vyjmenuje zdrojové moduly a uvede i jejich výstupní piny. Tento příkaz je imunní vůči zakázanému spojení, protože již v gramatice není dovoleno specifikovat vstupní pin cílového modulu.

Příklad: Propojení pole modulů – modul

```
connect n to m:0;
connect n:0 to m:0;
```

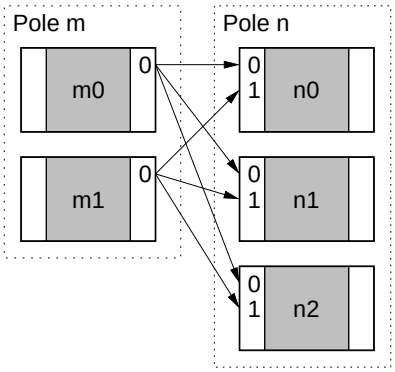
Na rozdíl od příkladu propojení modul – pole modulů kdy více spojů směřuje do jediného vstupního pinu, je v tomto případě situace obrácená. Tedy z jednoho výstupního pinu vychází několik spojů. To je však naprosto korektní chování. Možné jsou oba způsoby zápisu s tím, že se vytvoří spoje z nějakého výstupního pinu jednoho modulu a ty směřují do každého modulu z pole modulů *n* (případně je ještě možné specifikovat vstupní pin těchto modulů).



Obrázek 6.4: Spojení pole modulů – modul. Rozvětvení jediného výstupu na vstup každého modulu z pole.

Příklad: Propojení pole modulů – pole modulů

```
connect n to m:0;
```



Obrázek 6.5: Spojení pole modulů – pole modulů (úplné spojení). Připojované pole m rozvětví svůj výstup a každý spoj vede do jednoho modulu z pole n.

Jedná o propojení *všech se všemi*. Jak je patrné z obrázku 6.5 opět není možné specifikovat vstupní pin, protože by do něj bylo koncentrováno hned několik spojů. Propojení mezi vrstvami modulů se vytvoří tak, že z každého modulu v poli *m* a určitého výstupního pinu je veden spoj do všech modulů v poli *n* (postupně se připojují nové neobsazené piny vstupu).

Jak je z předchozích příkladů patrné, tak vždy lze propojení zapsat jen dvěma způsoby přičemž o správné vytvoření spojů se postará simulátor na základě znalosti typu propojovaných objektů.

6.5.2 Disconnect – zrušení spojů

```
disconnect n;
```

Důležitým příkazem je ještě `disconnect`. Jeho zápis je jednoduchý. Stačí pouze specifikovat modul, příp. moduly oddělené čárkou a u těchto modulů jsou všechny vstupní piny odpojeny, takže modul

nemá zapojený žádný vstup.

6.6 Cykly

Cykly patří k základním kamenům každého programovacího jazyka. SiMoNNe nabízí ty tři nejčastější.

6.6.1 For cyklus

```
for (i = 0; i < 10; i = i + 1) begin
  j = i * 10;
end
```

For cyklus je rozdělen do čtyř částí. Jednak je to *inicializační část* (`i = 0`), ve které se inicializuje řídicí proměnná. V druhé části, *ukončovací podmínka* (`i < 10`), se rozhoduje o ukončení cyklu. Třetí část, *iterační* (`i = i + 1`), slouží nejčastěji k inkrementaci, či dekrementaci řídicí proměnné (to však není podmínkou). Poslední, *výkonná část* (`begin j = i * 10; end`) obsahuje samostatný nebo složený příkaz (blok).

6.6.2 While cyklus

```
while (i < 10) begin
  i = i + 1;
end
```

While cyklus (neboli cyklus s testem na začátku) obsahuje pouze *ukončovací podmínku* (`i < 10`) a *výkonnou část* (`begin i = i + 1; end`). Díky tomu, že ukončovací podmínka se testuje ještě před provedením výkonné části, je možné v případě neplatnosti podmínky tělo cyklu nevykonat ani jednou.

6.6.3 Do-while cyklus

```
do begin
  i = i + 1;
end while (i < 10)
```

Do..while cyklus (cyklus s testem na konci), stejně jako while cyklus, obsahuje *výkonnou část* (`begin i = i + 1; end`) a *ukončovací podmínku* (`i < 10`). Jediný rozdíl oproti while cyklu je v tom, že ukončovací podmínka se otestuje až po provedení výkonné části. Do..while cyklus tak vždy proběhne alespoň jednou.

6.7 Podmínka if

```
if (i < 10) begin
  i = i * 10;
end
else begin
  i = i + 10;
end
```

If podmínka slouží k větvení programu. Nejdříve se otestuje podmínka (`i < 10`), podle které se následně rozhodne zda se vykoná *true blok* (`begin i = i * 10; end`) nebo *false blok* (`i = i + 10`), podle toho zda je pravdivá či ne. Oba bloky mohou být zastoupeny pouze samostatnými příkazy a else část může chybět.

6.8 Komentáře

V každém programovacím jazyce je velmi důležité mít možnost zapsat do kódu komentáře. V SiMoNNe se pro uvození začátku komentáře používá znak mříž (#). Za komentář se poté považuje vše od tohoto znaku až do konce řádky.

```
# Samostatný komentář
x = 1; # Komentář za přiřazením
# Samostatný komentář
# Komentář za přiřazením
```

Komentáře se v kódu pamatují aby bylo možné rekonstruovat původní zápis. Výsledkem zápisu komentáře je tedy opět jeho výstup v nezměněné formě.

6.9 Příkaz pro provedení simulačního kroku

```
step n, m;
```

Příkaz pro provedení simulačního kroku neboli příkaz `step` v SiMoNNe. Argumenty tohoto příkazu mohou být samostatné moduly nebo pole modulů. U všech modulů uvedených v argumentech je provedena *výkonná část* modulu (část za klíčovým slovem `stepcode`). Protože simulace je synchronní, hodnoty výstupů všech modulů se změní až po přepočítání všech modulů uvedených jako argumenty příkazu `step`. V uvedeném příkladu se provede paralelně step n a m.

6.10 Prázdný příkaz

Stejně jako například jazyk C, obsahuje i SiMoNNe prázdný příkaz, který se zapisuje pomocí středníku (`;`). Použití takového příkazu není nijak široké, dovoluje však třeba eliminaci přebytečných středníků v kódu. Například za klíčovým slovem `end`.

6.11 Konfigurace systémových proměnných

Simulátor dovoluje konfigurovat systémové proměnné, kterými je možné ovlivňovat jeho chování. Přiřazení do proměnných se zapisuje jako normální přiřazení. V současné době jsou v simulátoru implementovány proměnné podle tabulky 6.3.

Tabulka 6.3: Popis systémových proměnných simulátoru.

Jmnéno proměnné	*Význam*
`__sys_precision`	nastavení počtu desetinných míst u DoubleConst
`__sys_array_formating`	nastavení formátovaného výpisu polí
`__sys_intdiv_int`	nastavení typu výsledku dělení celých čísel

Proměnná `__sys_precision` má počáteční hodnotu -1 a to znamená, že počet desetinných míst není nijak omezen. Jakákoliv jiná kladná hodnota specifikuje na kolik desetinných míst se bude DoubleConst vypisovat.

Pomocí proměnné `__sys_array_formating` se dá určit jestli bude výstup polí na konzoli formátován podle vnitřní struktury, nebo bude pole vypsáno na jeden řádek bez dodatečného formátování. Počáteční hodnota je nenulová a to znamená, že výstup bude formátován, nulová hodnota proměnné znamená opak.

Proměnnou `__sys_intdiv_int` se provádí změna mezi typem výsledku dělení celých čísel. Pokud je proměnná nastavena na hodnotu nula (a to je počáteční nastavení), pak výsledek dělení dvou celých čísel, je číslo s plovoucí řádovou čárkou, ve všech ostatních případech je tomu naopak.

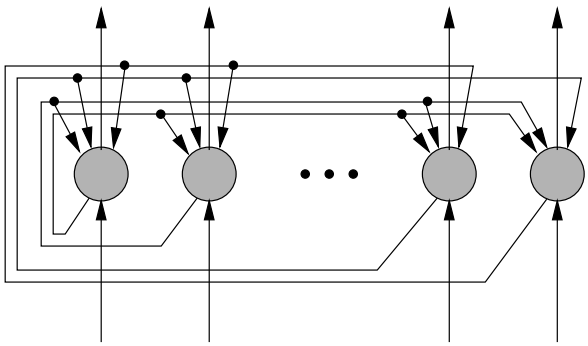
Kapitola 7

Příklady použití

V této kapitole bude na příkladech krok po kroku vysvětlena konstrukce sítě, její učení a vybavování.

7.1 Hopfieldova síť

Nejdříve vyzkoušíme funkci simulátoru na příkladu klasické Hopfieldovy sítě 7.1.



Obrázek 7.1: Hopfieldova síť. Výstup každého neuronu vede do vstupu všech ostatních.

Základním stavebním kamenem sítě je neuron. V následující části kódu je nadefinování typu neuronu, který poslouží jako šablona pro vytváření instancí jednotlivých neuronů.

```
1   moduledef HopfieldNeuron begin
2     output[0] = 0;
3   stepcode
4     sum = 0;
5     for (i=0; i<length(input); i=i+1)
6       sum = sum + input[i] * weight[i];
7     if (sum >= 0)
8       output[0] = 1;
9     else
10      output[0] = -1;
11  end
```

První řádek oznamuje, že se jedná o definici typu neuronu, který se bude jmenovat HopfieldNeuron. Ve druhém řádku přiřazujeme do výstupu neuronu hodnotu nula, aby byl výstup inicializovaný pro pozdější propojení. Tím končí *inicializační část* definice modulu a na třetím řádku začíná *část výkonná*, tedy kód, který se bude provádět v simulačních krocích a který popisuje přenosovou funkci neuronu.

Na pátém řádku je for cyklus, v jehož ukončovací podmínce je použitá funkce length. Jejím argumentem je pole vstupů a výsledkem je délka tohoto pole. Díky této konstrukci se nemůže stát, že bychom indexovali mimo meze pole, což je velice častá chyba a používání funkce length je tudíž velmi užitečné.

Šablonu pro instance modulů už máme vytvořenou a následuje definice dvou funkcí, které budeme potřebovat při inicializaci a učení neuronů.

```
12  function init(layer) begin
13    for (i=0; i<length(layer); i=i+1)
14      for (j=0; j<length(layer[i].input); j=j+1)
15        layer[i].weight[j] = 0;
16  end
```

Funkce init inicializuje váhové vektory všem neuronům v předávaném poli. Pro každý neuron z pole layer jsou nastaveny prvky jeho váhového vektoru na hodnotu nula. Zde se ukazuje užitečnost funkce length, protože ve funkci nevíme jak velké je vstupní pole (input) každého neuronu a nebyli bychom tudíž schopni správně vytvořit vektor weight, jehož každá hodnota je přiřazena jednomu vstupu. Díky funkci length však je problém vyřešen.

```
17  function learn(layer, pattern) begin
18    l = length(pattern);
19    for (i=0; i<l; i=i+1)
20      for (j=0; j<l; j=j+1)
21        if (i != j)
22          layer[i].weight[j] = layer[i].weight[j] + pat[i] * pat[j];
23  end
```

Ve funkci learn dochází k učení neuronů v poli layer podle parametru pattern. Jsou tedy upraveny váhy vstupů každého neuronu podle předloženého vzoru. Ještě je potřeba upozornit, že se nenastavují váhy na diagonále pomyslné čtvercové matice vzniklé z pole neuronů a váhových vektorů jednotlivých neuronů. Zůstávají tedy na hodnotě nula. Jak uvidíme dále, využili jsme možnosti úplného propojení a spoje, které spojují výstup neuronu se vstupem téhož neuronu a v Hopfieldově síti nejsou, jsme tímto eliminovali.

```
24  for (i=0; i<16; i=i+1)
25    hop[i] = new HopfieldNeuron;
```

Teď už můžeme vytvořit pole neuronů, které se konstruují podle typu HopfieldNeuron. Jak bylo zmíněno v oddíle 6.1.2, při vytváření nového pole není nutné ho nijak inicializovat a pokud indexujeme od prvního prvku (s indexem nula), inicializaci zařídí sám simulátor.

```
26  connect hop to hop:0;
```

Na tomto příkladu je dobře vidět kolektivní operace se spoji. Proměnná `hop` je totiž pole a zápis tedy říká: „*Připoj vstup každého neuronu z pole hop k nultému výstupnímu pinu každého neuronu z pole*

hop“. Tím jsme jediným příkazem dosáhli toho, že výstupy všech neuronů budou připojeny na vstupy všech neuronů (úplné spojení). Přebytečné spoje (ty které spojují výstup se vstupem téhož neuronu) jsou potlačeny nulovou váhou, která se zachovává i při učení – upravená učicí funkce learn.

Síť je již kompletně připravená a můžeme ji tedy inicializovat (řádek 27). Následně předložíme několik vzorů a síť na ně naučíme.

```
27  init(hop);
28  learn(hop, {-1,-1,1,1,-1,-1,1,1,1,1,1,1,1,1,1,1});
29  learn(hop, {1,-1,-1,-1,1,1,-1,-1,1,1,1,1,1,1,1,-1});
30  learn(hop, {1,1,1,1,-1,1,1,1,-1,-1,1,1,-1,-1,-1,1});
31  learn(hop, {-1,1,1,1,1,1,1,1,1,1,-1,-1,1,1,-1,-1});
```

Abychom mohli jednoduše sledovat váhové matice jednotlivých neuronů, vytvoříme si funkci vypisující váhové matice všech neuronů v poli, které je jejím parametrem.

```
32  function printWeightMatrix(layer) begin
33    for (i=0; i<length(layer); i=i+1)
34      layer[i].weight;
35  end

36  printWeightMatrix(hop);
```

Zjistíme, jak vypadají jednotlivé váhové matice a předložíme síti připravený vzor k vybavování.

```
37  init_vector={1,1,1,1,1,1,1,1,1,1,-1,-1,1,1,1,1};

38  for (i=0; i<16; i=i+1)
39    hop[i].output[0] = init_vector[i];
```

Pro výpis hodnot výstupů celého pole se hodí vytvořit si funkci, která posbírá výstupy z jednotlivých neuronů a vytvoří z nich pole, které vrátí jako svou výstupní hodnotu.

```
40  function getOutput(hop) begin
41    for (i=0; i<length(hop); i=i+1)
42      result[i] = hop[i].output[0];
43    return result;
44  end
```

Nakonec v cyklu zavoláme výkonnou část modulů v poli (řádek 46). Proměnná `hop` reprezentuje pole, takže simulační krok probíhá u všech prvků pole najednou a díky synchronní simulaci počítá každý neuron s korektními hodnotami. Po proběhnutí simulačního kroku se podíváme na to, jak dopadly výstupy jednotlivých neuronů (řádek 47). Funkce `getOutput` vrací pole hodnot, protože výsledek volání funkce je identický se zjištěním hodnoty nějaké proměnné, simulátor vrácenou hodnotu vytiskne.

```
45  for (i=0; i<6; i=i+1) begin
46    step hop;
47    getOutput(hop);
48  end
```

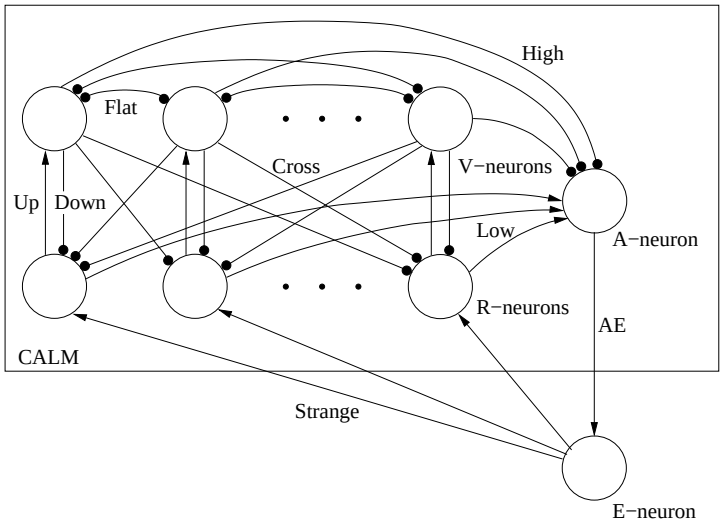
Podle výstupů je vidět, že Hopfieldova síť pracuje správně. Výstupy jsou zkrácené, protože síť rozpoznala předložený vzor hned v první iteraci.

```
{-1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, 1, 1, -1, -1};
```

Kompletní příklad je uveden v příloze C.1.

7.2 Síť CALM

CALM (Categorizing and Learning Module) je poměrně složitá síť (viz obrázek 7.2). Popis tedy bude dost rozsáhlý a kompletní kód je opět v příloze C.2. Je však na něm dobře vidět zapouzdřování modulů v modulárních sítích a je možné na něm ukázat hodně vlastností simulátoru. Popis funkcí jednotlivých neuronů a učičích algoritmů je uveden v literatuře [Murre-1992].



Obrázek 7.2: Síť CALM. Šipky označují excitační a kuličky inhibiční spoje.

Nejdříve si nadefinujeme několik funkcí, které budeme dále při výpočtu používat. Následující funkce slouží v jednotlivých modulech k počítání hodnoty výstupu – podle zadaných parametrů vrátí hodnotu, která se použije pro nastavení výstupní hodnoty.

```
1   function activation(exc, k, oldact) begin
2     if (exc >= 0)
3       return (1-k) * oldact + exc / (1+exc) * (1-(1-k) * oldact);
4     else
5       return (1-k) * oldact + exc / (1-exc) * (1-k) * oldact;
6   end
```

Funkce `makeArray` slouží, jak už název napovídá, k vytvoření pole o velikosti length a inicializaci jeho prvků hodnotami value.

```
7   function makeArray(length, value) begin
8     for (i=0; i<length; i=i+1)
9       array[i] = value;
10    return array;
11  end
```

Funkce `max` a `min` vrátí větší/menší z předávaných parametrů.

```
12  function max(x, y) begin
13    if (x > y) return x; else return y;
14  end

15  function min(x, y) begin
16    if (x < y) return x; else return y;
17  end
```

Následující funkce se používá k přenastavení hodnot výstupů na nulu (tzv. reset). Do funkce se jako parametr předá celá síť a uvnitř funkce se postupně cyklem projde přes všechny neurony a vynulují se hodnoty jejich výstupů.

```
18  function calmReset(calm) begin
19    for (i=0; i<calm.degree; i=i+1) begin
20      calm.r[i].output[0] = 0.0;
21      calm.v[i].output[0] = 0.0;
22      calm.a.output[0] = 0.0;
23      calm.e.output[0] = 0.0;
24    end
25  end
```

Potřebné funkce máme hotové, můžeme začít definovat moduly. V následující definici je možné vidět proměnnou arg, která však není nikde definována. Jedná se o speciální proměnnou, přes kterou jsou přístupné parametry předávané do modulu při vytváření jeho instance (obdoba parametrů konstruktoru v mnoha programovacích jazycích). Jako parametr lze předat jakýkoliv datový typ, v našem případě se předávají pouze celá čísla.

```
26  # arg == {stupeň modulu, počet vstupů}
27  moduledef R begin
28    output = {0.0};
29    weights = makeArray(arg[1], 0.6);
30    strange = 0.5;
31    cross = makeArray(arg[0], -10.0);
32    k = 0.05;
```

Ve výkonné části modulu jsou vidět dva zajímavé řádky. Na řádku 42 je použita funkce `random`. Jedná se o tzv. externí funkci a je to názorná demonstrace užitečnosti externích funkcí, protože dokáží rozšiřovat funkcionalitu simulátoru, kterou sám nenabízí. Na řádku 44 je pak použito funkce `activation`, kterou jsme si zkonstruovali dříve a jak uvidíme dále, protože ji budeme ještě několikrát potřebovat, ušetří nám spoustu řádků kódu, který bychom museli zbytečně opisovat, tak jak tomu bylo v předešlé verzi simulátoru.

```
33  stepcode
34    exc = 0;
35    l_input = length(weights);
36    for (i=0; i<l_input; i=i+1)
37      exc = exc + weights[i] * input[i];
38    l_cross = l_input + length(cross);
39    for (i=l_input; i<l_cross; i=i+1)
40      exc = exc + cross[i - l_input] * input[i];
41    if (input[l_cross] <= 0.0) tmp = 0.0;
42    else tmp = random() * input[l_cross];
43    exc = exc + strange * tmp;
44    output[0] = activation(exc, k, output[0]);
45  end
```

V následujících neuronech již nejsou žádné neznámé konstrukce a proto jejich popis přeskočíme.

```
46  moduledef V begin
47    output = {0.0};
48    up = 0.5;
49    flat = -1.0;
50    k = 0.05;
51  stepcode
52    exc = input[0] * up;
53    for (i=0; i<length(input); i=i+1)
54      exc = exc + flat * input[i];
55    output[0] = activation(exc, k, output[0]);
56  end

57  moduledef A begin
58    output = {0.0};
59    low = 0.4;
60    high = -0.6;
61    k = 0.05;
62  stepcode
63    exc = 0;
64    for (i=0; i<length(input); i=i+2)
65      exc = exc + input[i] * low + input[i+1] * high;
66    output[0] = activation(exc, k, output[0]);
67  end

68  moduledef E begin
69    output = {0.0};
70    aE = 0.5;
71    k = 0.05;
72  stepcode
73    exc = aE * input[0];
74    output[0] = activation(exc, k, output[0]);
75  end
```

A dostáváme se konečně k hlavnímu modulu, který zapouzdřuje jednotlivé další moduly přesně podle obrázku 7.2. Opět je vidět použití proměnné arg, která specifikuje v tomto případě stupeň – počet

R-neuronů. Následně se v cyklu vytvoří pole R-neuronů a stejně velké pole V-neuronů.

```
76   moduledef CALM begin
77     degree = arg;
78     for (q=0; q<degree; q=q+1) begin
79       r[q] = new R {arg, 2};
80       r[q].cross[q] = -1.2;
81       v[q] = new V;
82     end
83     a = new A;
84     e = new E;
85     d = 0.005;
86     w_mu_e = 0.05;
87     k = 1.0;
88     l = 1.0;
```

Nadefinovali jsme všechny potřebné moduly a vytvořili jejich instance, v následující části je můžeme mezi sebou propojit. Řádky 94 a 95 ukazují, že k indexaci do výstupů modulu lze použít jakýkoliv výraz, tedy ne pouze konstantní výraz nebo proměnnou, ale třeba i výsledek nějaké aritmetické operace. Na řádcích 92 a 110 je propojení vnořených modulů na obalující modul. Při propojování uvnitř modulu se pracuje se vstupy a výstupy obalujícího modulu naprosto stejně jako s ostatními moduly.

```
89     no_of_inputs = length(r[0].weights);
90     for (i=0; i<degree; i=i+1) begin
91       for (j=0; j<no_of_inputs; j=j+1)
92         connect r[i]:j to input:j;
93       for (j=0; j<degree; j=j+1)
94         connect r[i]:j+no_of_inputs to v[j]:0;
95       connect r[i]:degree+no_of_inputs to e:0;
96     end
97     for (i=0; i<degree; i=i+1) begin
98       connect v[i] to r[i]:0;
99       for (j=0; j<i; j=j+1)
100        connect v[i] to v[j]:0;
101      for (j=i+1; j<degree; j=j+1)
102        connect v[i] to v[j]:0;
103    end

104    for (j=0; j<degree; j=j+1) begin
105      connect a:j*2 to r[j]:0;
106      connect a:j*2+1 to v[j]:0;
107    end
108    connect e to {a:0};
109    for (j=0; j<degree; j=j+1)
110      connect output:j to r[j]:0;
```

Po nadefinování a inicializaci máme vytvořenou a propojenou celou síť a potřebujeme pouze specifikovat, jak bude vypadat simulační krok. Na řádku 112 je vidět použití příkazu step, který u všech modulů, které má jako své argumenty spustí jejich výkonnou část. Díky synchronní simulaci bude každý modul pracovat s korektními hodnotami. Poté dochází k výpočtu, jehož výsledkem je upravení vah u R-neuronů. Důležité je všimnout si, že díky zapouzdření specifikujeme přímo v modulu kdy se mají

spouštět simulační kroky jeho prvků a neřešíme tuto věc na jedné globální úrovni, jako by tomu bylo v případě, kdyby moduly nebyly do sebe vnořeny a byly by pouze pospojovány.

```
111   stepcode
112     step r, v, a, e;
113     mu = d + w_mu_e * e.output[0];
114     for (i=0; i<degree; i=i+1) begin
115       sumf = 0;
116       for (f=0; f<length(r[i].weights); f=f+1)
117         sumf = sumf + r[i].weights[f] * input[f];
118       for (j=0; j<length(r[i].weights); j=j+1) begin
119         w = r[i].weights[j];
120         delta_w = mu * r[i].output[0] * ((k - w) * input[j] -
121           l * w * (sumf - w * input[j]));
122         r[i].weights[j] = max(min(w + delta_w, k), 0);
123       end
124     end
125   end
```

Celou síť už máme kompletní, všechny moduly už mají popsané své chování, můžeme tedy vytvořit instanci modulu CALM. Díky zapouzdření se při jeho inicializaci vytvoří všechny ostatní vnořené moduly. Poté předložíme na vstup několik vzorů a v iteracích se je pokusíme CALM naučit. Následně pak předložíme nový vektor a zkusíme jak síť vybavuje.

```
126   c = new CALM 4;

127   for (i=0; i<10; i=i+1) begin
128     c.input = {0,1};
129     for (j=0; j<30; j=j+1) step c;
130     calmReset(c);
131     c.input = {1,1};
132     for (j=0; j<30; j=j+1) step c;
133     calmReset(c);
134   end

135   c.input = {0, 1};
136   for (i=0; i<20; i=i+1) begin
137     step c;
138     c.output;
139   end
```

Výstup ze sítě ukazuje, že experiment se nám povedl.

```
{3.9572444558772535E-9, 4.7079223831564785E-9,
 3.804666820117138E-9, 0.8250666964617092};
{4.522148335053646E-10, 5.445798049879844E-10,
 4.524462338840319E-10, 0.7868757047053916};
{4.999576593241607E-11, 6.129234692410953E-11,
 5.096789941445408E-11, 0.7751698272792837};
```

Kapitola 8

Závěr

V této práci jsem prostudoval původní jazyk SiMoNNe Language a jeho implementaci. Na základě analýzy jazyka a s použitím nově vzniklé gramatiky pro akcelerovanou implementaci Simonnec jsem vytvořil gramatiku pro nové jádro simulátoru. Při návrhu gramatiky byl kladen důraz zejména na rozšíření funkčnosti simulátoru a snadnou a intuitivní práci při návrhu neuronových sítí.

V nově vzniklém simulátoru jsem odstranil nedostatky předchozí verze a přidal některé další užitečné vlastnosti (možnost definice a použití vlastních i externích funkcí, vlastní vícedimenzionální pole, kolektivní práce se spoji, konfigurace simulátoru). Při tvorbě nové implementace jsem se zaměřil především na jednoduchost práce se simulátorem a odstranění některých původních omezení. Novou verzi simulátoru jsem otestoval na několika typech neuronových sítí a ověřil jsem tak její funkčnost.

Přínosem nového simulátoru je hlavně jednoduchost a intuitivnost při návrhu neuronových sítí a odstranění omezení původní verze. Díky tomu bude po implementaci běžných neuronových sítí formou jednoduchých skriptů v jazyce SiMoNNe sloužit jako pomůcka při výuce předmětu Neuronové sítě a neuropočítače (36NAN).

Kapitola 9

Budoucnost

Nová verze simulátoru přinesla mnoho vylepšení oproti původní implementaci. I přes tento fakt je však možné simulátor dále rozvíjet a stále je zde široký prostor pro vylepšení uživatelské přívětivosti. Hlavní směry vývoje vidím především v přístupu k proměnným v polích a vnořených modulech, což znamená rozšíření možností simulátoru o kolektivní operace při přístupu k proměnným. Jedná se o podobnou funkčnost jakou nyní poskytují spoje. Mějme pole modulů n, situaci znázorňuje následující příklad:

```
# v současné implementaci
for (i=0; i<length(n); i=i+1)
  n[i].input;
# rozšíření
n.input;
```

Dalšími oblastmi, ve kterých by se dalo mnohé zlepšovat, jsou výpisy ze simulátoru, zejména rekonstrukce stavu simulátoru při volání příkazu dump a znovupoužitelnost výstupu (uložení stavu simulátoru a jeho pozdější načtení).

Rozšířena by také mohla být maticová aritmetika nebo větší parametrizace simulátoru. Při implementaci byl vznesen požadavek na rozšíření definice modulu o učicí část – tedy kód, kterým se instance modulů učí a nyní je zapisován ve funkcích. V tomto případě se sice jedná už o zásah do gramatiky, ale je to pouze jednoduchá úprava. Jiným možným řešením je implementace funkcí v modulech (obdoba metod v objektových programovacích jazycích), které by se daly využít i jinak než pouze na učení.

Jednou ze zajímavých oblastí by mohla být implementace ladicího programu. Vzhledem k charakteru simulátoru – nezávislé simulační jádro, ke kterému lze připojovat různá rozhraní – se otevírají oblasti vytvoření grafického uživatelského rozhraní nebo síťového rozhraní pro možnost vzdálené simulace. V neposlední řadě by se dalo zaměřit na optimalizaci a zvýšení rychlosti samotného simulátoru.

Příloha A

Spuštění simulátoru

Pro překlad a běh simulátoru je potřeba mít na počítači dvě komponenty. Před spuštěním je tedy potřeba se ujistit, zda je na počítači nainstalováno JRE (Java Runtime Environment). V případě jeho nepřítomnosti je možné jej zdarma stáhnout ze stránek `http://java.sun.com/` a dále postupovat podle příslušného návodu na instalaci od výrobce.

Pro překlad simulátoru je potřeba ještě mít nainstalovaný Ant z *The Apache Ant Project*. Distribuční balík lze opět stáhnout zdarma z `http://ant.apache.org/` a opět se řídit návodem na instalaci.

Pokud jsou obě komponenty na počítači přítomny, lze simulátor přeložit nebo jej přímo spustit.

Pro překlad se přepneme do kořenového adresáře projektu `simonne-m` a spustíme skript `make.sh`, který zkontroluje nastavení proměnných (`JAVA_HOME` a `ANT_HOME`) a zavolá program Ant. V případě OS Windows můžeme zavolat Ant okamžitě (bez parametrů).

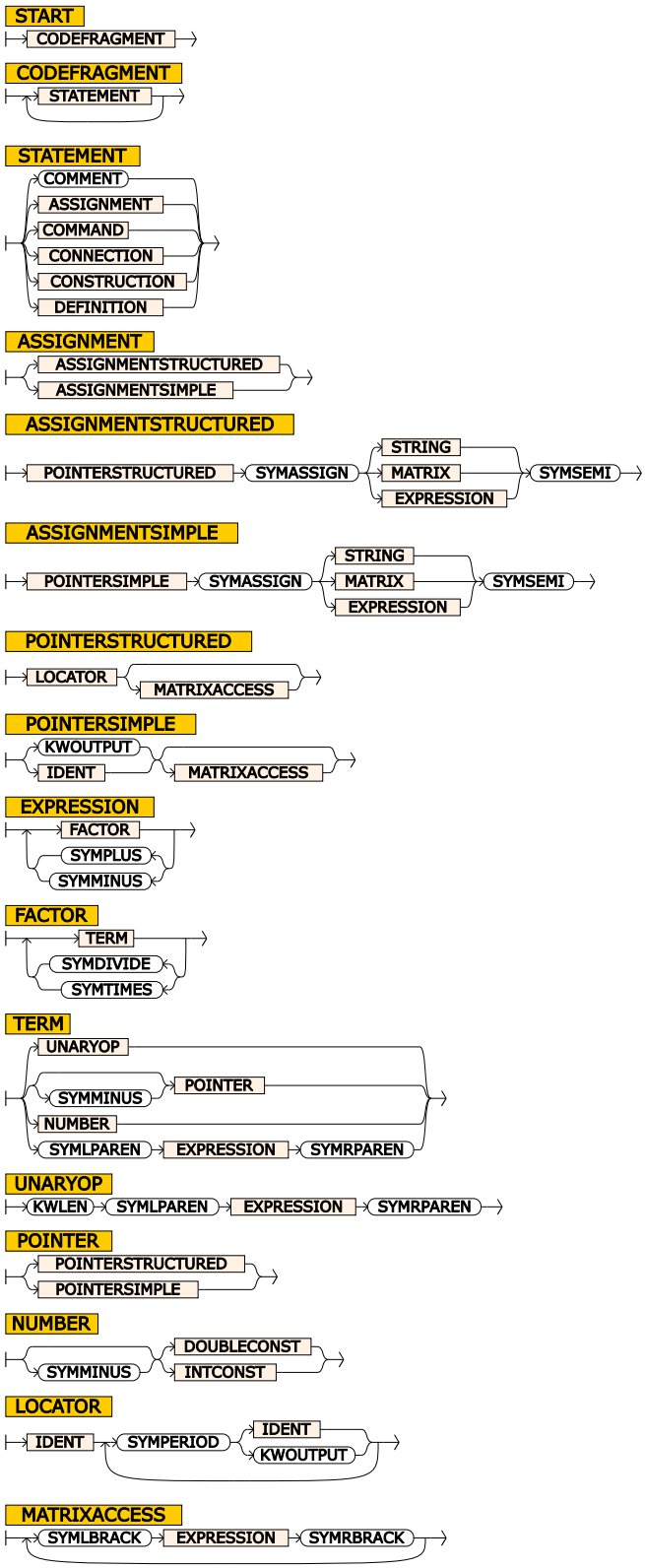
Po překladu budou přeložené třídy uloženy v adresáři `build`. Simulátor není potřeba nijak instalovat a můžeme ho začít okamžitě používat. Jednoduchou textovou konzoli spustíme stejným skriptem s parametrem `rc` (run console). Případně opět přímo zavoláním programu Ant s parametrem `rc`.

Příloha B

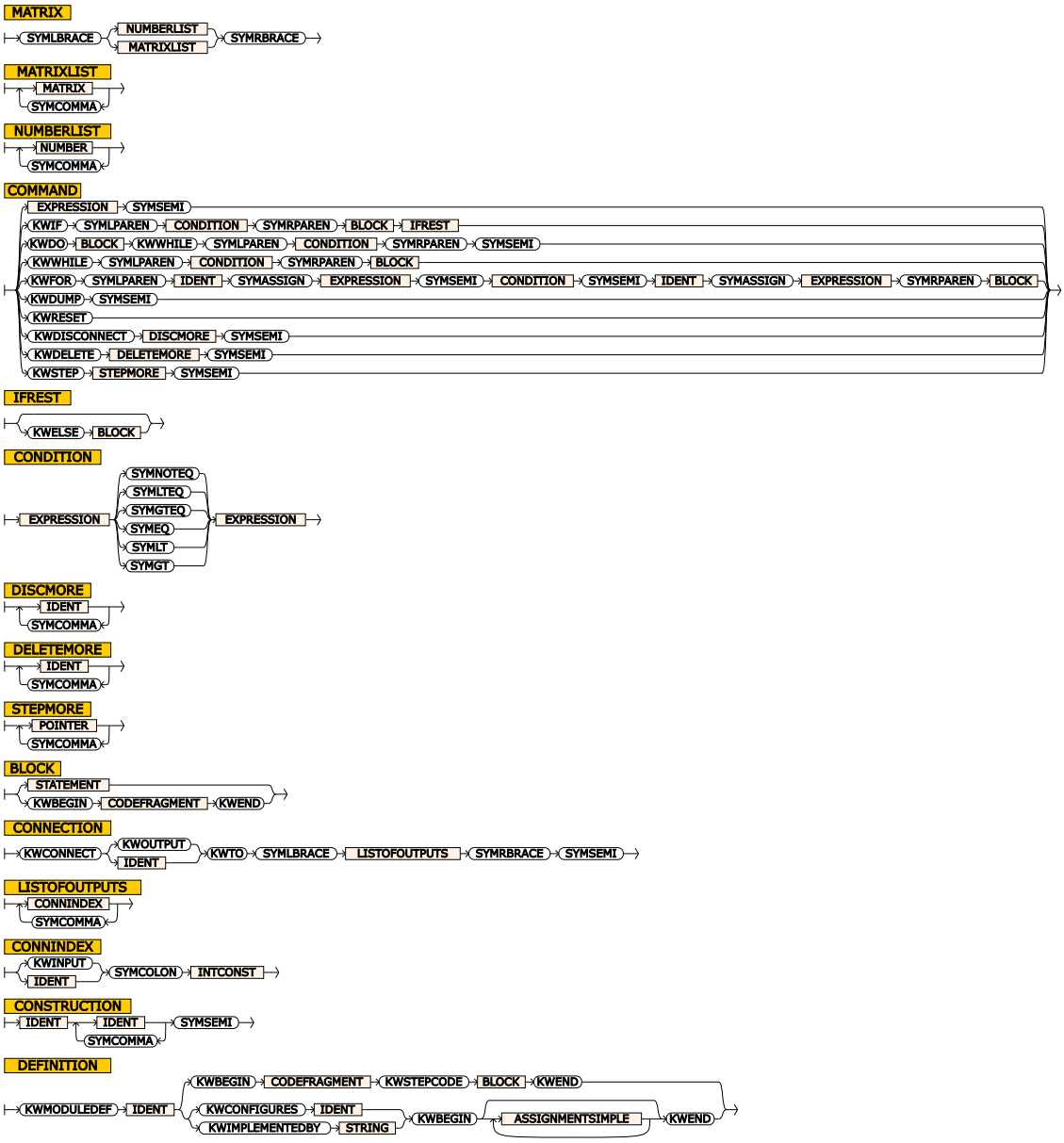
Gramatika

Na následujících obrácích jsou gramatiky původní a nové verze jazyka. Ke tvorbě diagramů byl použit nástroj *syndiagrapher* od Jakuba Trávníka.

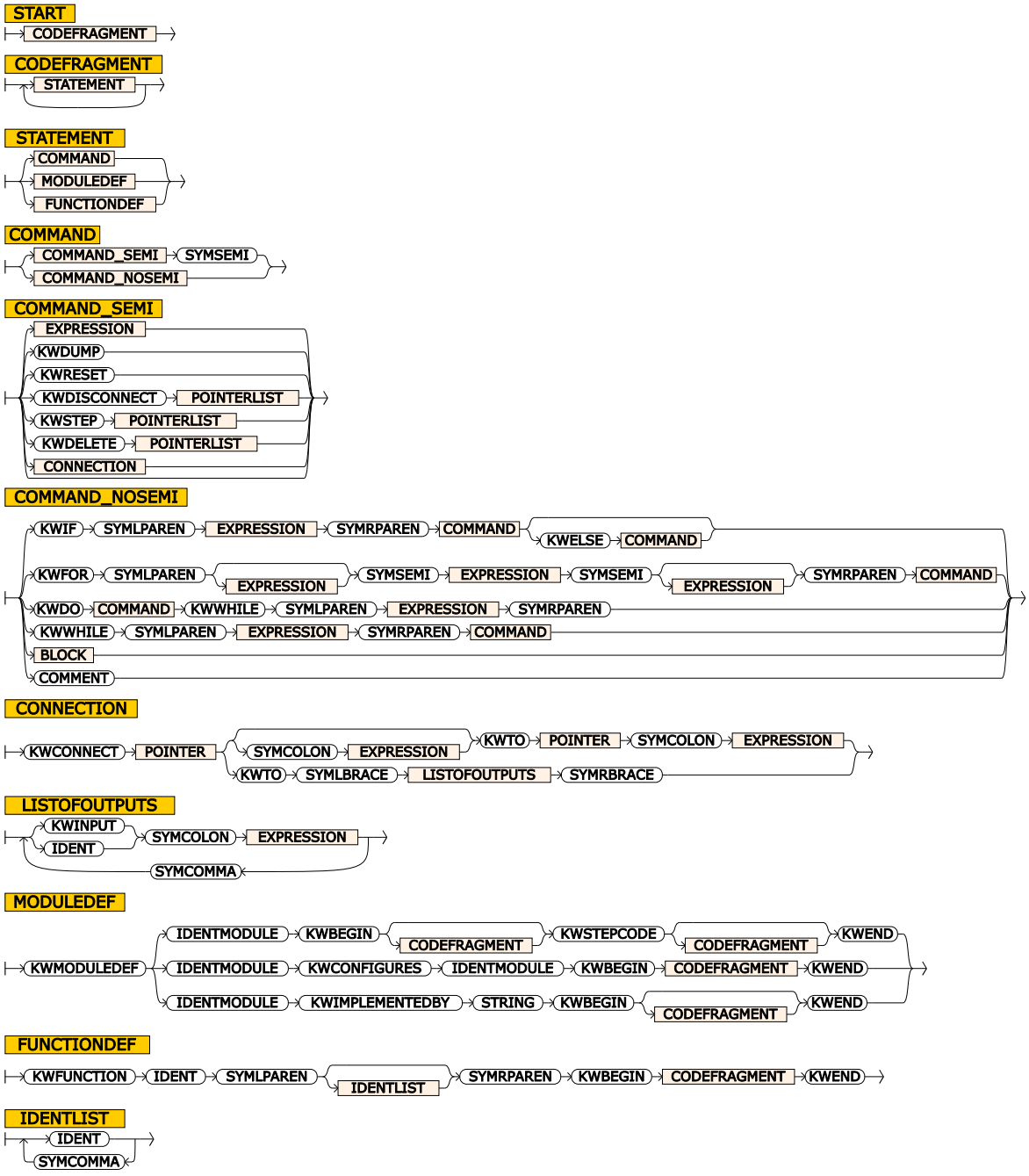
Diagramy B.1 a B.2 znázorňují původní gramatiku SiMoNNe. Na diagramech B.3 a B.4 je pak uvedená nová gramatika.



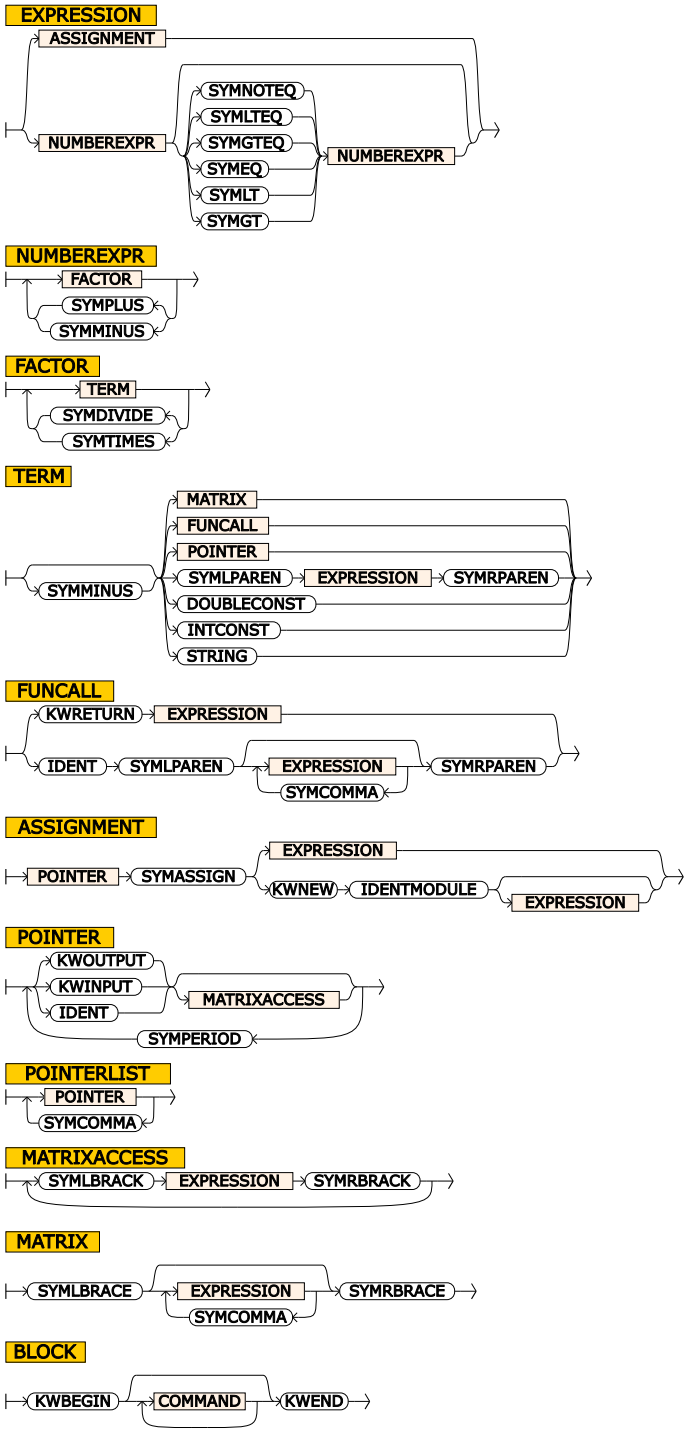
Obrázek B.1: Gramatika původního SiMoNNe, část 1



Obrázek B.2: Gramatika původního SiMoNNe, část 2



Obrázek B.3: Gramatika nového SiMoNNe, část 1



Obrázek B.4: Gramatika nového SiMoNNe, část 2

Příloha C

Zdrojové kódy příkladů

C.1 Hopfieldova síť

```
moduledef HopfieldNeuron begin
  output[0] = 0;
stepcode
  sum = 0;
  for (i=0; i<length(input); i=i+1)
    sum = sum + input[i] * weight[i];
  if (sum >= 0)
    output[0] = 1;
  else
    output[0] = -1;
end

function init(layer) begin
  for (i=0; i<length(layer); i=i+1)
    for (j=0; j<length(layer[i].input); j=j+1)
      layer[i].weight[j] = 0;
end

function learn(layer, pattern) begin
  l = length(pattern);
  for (i=0; i<l; i=i+1)
    for (j=0; j<l; j=j+1)
      if (i != j)
        layer[i].weight[j] = layer[i].weight[j] + pat[i] * pat[j];
end

for (i=0; i<16; i=i+1)
  hop[i] = new HopfieldNeuron;

connect hop to hop:0;
```

```
27  init(hop);
28  learn(hop, {-1,-1,1,1,-1,-1,1,1,1,1,1,1,1,1,1,1});
29  learn(hop, {1,-1,-1,-1,1,1,-1,-1,1,1,1,1,1,1,1,-1});
30  learn(hop, {1,1,1,1,-1,1,1,1,-1,-1,1,1,-1,-1,-1,1});
31  learn(hop, {-1,1,1,1,1,1,1,1,1,1,-1,-1,1,1,-1,-1});

32  function printWeightMatrix(layer) begin
33    for (i=0; i<length(layer); i=i+1)
34      layer[i].weight;
35  end

36  printWeightMatrix(hop);

37  init_vector={1,1,1,1,1,1,1,1,1,1,-1,-1,1,1,1,1};

38  for (i=0; i<16; i=i+1)
39    hop[i].output[0] = init_vector[i];

40  function getOutput(hop) begin
41    for (i=0; i<length(hop); i=i+1)
42      result[i] = hop[i].output[0];
43    return result;
44  end

45  for (i=0; i<6; i=i+1) begin
46    step hop;
47    getOutput(hop);
48  end
```

C.2 Síť CALM

```
 1  function activation(exc, k, oldact) begin
 2    if (exc >= 0)
 3      result = (1-k) * oldact + exc / (1+exc) * (1-(1-k) * oldact);
 4    else
 5      result = (1-k) * oldact + exc / (1-exc) * (1-k) * oldact;
 6    return result;
 7  end

 8  function makeArray(length, value) begin
 9    for (i=0; i<length; i=i+1)
10      array[i] = value;
11    return array;
12  end

13  function max(x, y) begin
14    if (x > y) return x; else return y;
15  end

16  function min(x, y) begin
17    if (x < y) return x; else return y;
18  end

19  function calmReset(calm) begin
20    for (i=0; i<calm.degree; i=i+1) begin
21      calm.r[i].output[0] = 0.0;
22      calm.v[i].output[0] = 0.0;
23      calm.a.output[0] = 0.0;
24      calm.e.output[0] = 0.0;
25    end
26  end

27  # arg == {stupeň modulu, počet vstupů}
28  moduledef R begin
29    output = {0.0};
31    weights = makeArray(arg[1], 0.6);
32    strange = 0.5;
33    cross = makeArray(arg[0], -10.0);
34    k = 0.05;
35  stepcode
36    exc = 0;
37    l_input = length(weights);
38    for (i=0; i<l_input; i=i+1)
39      exc = exc + weights[i] * input[i];
40    l_cross = l_input + length(cross);
41    for (i=l_input; i<l_cross; i=i+1)
42      exc = exc + cross[i - l_input] * input[i];
```

```
43      if (input[l_cross] <= 0.0) tmp = 0.0;
44      else tmp = random() * input[l_cross];
45      exc = exc + strange * tmp;
46      output[0] = activation(exc, k, output[0]);
47    end

48    moduledef V begin
49      output = {0.0};
50      up = 0.5;
51      flat = -1.0;
52      k = 0.05;
53    stepcode
54      exc = input[0] * up;
55      for (i=0; i<length(input); i=i+1)
56        exc = exc + flat * input[i];
57      output[0] = activation(exc, k, output[0]);
58    end

59    moduledef A begin
60      output = {0.0};
61      low = 0.4;
62      high = -0.6;
63      k = 0.05;
64    stepcode
65      exc = 0;
66      for (i=0; i<length(input); i=i+2)
67        exc = exc + input[i] * low + input[i+1] * high;
68      output[0] = activation(exc, k, output[0]);
69    end

70    moduledef E begin
71      output = {0.0};
72      aE = 0.5;
73      k = 0.05;
74    stepcode
75      exc = aE * input[0];
76      output[0] = activation(exc, k, output[0]);
77    end
```

```
78   moduledef CALM begin
79     degree = arg;
80     for (q=0; q<degree; q=q+1) begin
81       r[q] = new R {arg, 2};
82       r[q].cross[q] = -1.2;
83       v[q] = new V;
84     end
85     a = new A;
86     e = new E;
87     d = 0.005;
88     w_mu_e = 0.05;
89     k = 1.0;
90     l = 1.0;

91     no_of_inputs = length(r[0].weights);
92     for (i=0; i<degree; i=i+1) begin
93       for (j=0; j<no_of_inputs; j=j+1)
94         connect r[i]:j to input:j;
95       for (j=0; j<degree; j=j+1)
96         connect r[i]:j+no_of_inputs to v[j]:0;
97       connect r[i]:degree+no_of_inputs to e:0;
98     end
99     for (i=0; i<degree; i=i+1) begin
100       connect v[i] to r[i]:0;
101       for (j=0; j<i; j=j+1)
102         connect v[i] to v[j]:0;
103       for (j=i+1; j<degree; j=j+1)
104         connect v[i] to v[j]:0;
105     end

106     for (j=0; j<degree; j=j+1) begin
107       connect a:j*2 to r[j]:0;
108       connect a:j*2+1 to v[j]:0;
109     end
110     connect e to {a:0};
111     for (j=0; j<degree; j=j+1)
112       connect output:j to r[j]:0;
113   stepcode
114     step r, v, a, e;
115     mu = d + w_mu_e * e.output[0];
116     for (i=0; i<degree; i=i+1) begin
117       sumf = 0;
118       for (f=0; f<length(r[i].weights); f=f+1)
119         sumf = sumf + r[i].weights[f] * input[f];
120       for (j=0; j<length(r[i].weights); j=j+1) begin
121         w = r[i].weights[j];
122         delta_w = mu * r[i].output[0] * ((k - w) * input[j] -
123           l * w * (sumf - w * input[j]));
124         r[i].weights[j] = max(min(w + delta_w, k), 0);
125       end
126     end
127   end
```

```
128   c = new CALM 4;

129   for (i=0; i<10; i=i+1) begin
130     c.input = {0,1};
131     for (j=0; j<30; j=j+1) step c;
132     calmReset(c);
133     c.input = {1,1};
134     for (j=0; j<30; j=j+1) step c;
135     calmReset(c);
136   end
137   c.input = {0, 1};
138   for (i=0; i<20; i=i+1) begin
139     step c;
140     c.output;
141   end
```

Literatura

[Brunner-1997] Brunner J., *Neuronová síť pro rozpoznávání objektu v obraze*, Diplomová práce na ČVUT FEL, Praha 1997

[BrunKou-2002] Brunner J., Koutník J., SiMoNNe – Simulator of Modular Neural Networks, *Neural Network World.*, 2002, vol. 12, no. 3, p. 267-278. ISSN 1210-0552.

[GoJoStBr-2000] Gosling J., Joy B., Steele G., Bracha G., *The Java Language Specification (Second Edition).* 2000 Sun Microsystems, Inc.

[Haykin-1994] Haykin S., *Neural Networks, A Comprehensive Foundation*, IEEE Press, 1994, ISBN 0-02-352761-7

[HebJohn-1990] Hebel K. J., Johnson R. E., *Arithmetic and double dispatching in Smalltalk* 1990. Journal Object Oriented Program. ISSN 0896-8438, SIGS Publications.

[Hudson-1999] Hudson S., *LALR Parser Generator for Java*, User's Manual, Version 0.10k, 1999
URL: `http://www.cs.princeton.edu/~appel/modern/java/CUP/`

[Klein-2004] Klein G., *The Fast Scanner Generator for Java*, User's Manual, Version 1.4.1, 2004
URL: `http://jflex.de/`

[Koutník-2000] Koutník J., *Implementation of the GOLOKO Neural Network.* Praha, 2000. Diplomová práce na ČVUT FEL, katedra počítačů.

[KouBrunŠno-2002] Koutník J., Brunner J., Šnorek M., The GOLOKO Neural Network for Vision – Analysis of Behavior, ICCVG, Gliwice: Silesian Technical University, 2002, vol. 2, p. 437-442. ISBN 8-39176-831-7.

[Koutník-2004] Koutník J., Šnorek M., Efficient Simulation of Modular Neural Networks, Eurosim 2004, Paris

[Murre-1995] Murre J.M.J, *Neurosimulators, The Handbook of Brain Theory and Neural Networks*, Cambridge, 1995

[Murre-1992] Murre J.M.J., Phaf R.H., Wolters G., CALM: Categorizing and Learning Module, *Neural Networks*, Vol. 5, pp. 55-82, Pergamon Press 1992

[Navrátil-2002] Navrátil M., *Simulace neuronové sítě s využitím SIMD instrukcí.* Praha, 2002. Diplomová práce na ČVUT FEL, katedra počítačů.

[Ronco-1995] Ronco E., Gawthrop P., *Modular Neural Networks: State of the Art*, Technical Report CSC-95026, Center for System and Control, University at Glasgow, UK, 1995

[Šnorek-1996] Šnorek M., *Neuronové sítě a neuropočítače.* Praha, 1996. Skriptum. Vydavatelství ČVUT.

[Trávník-2004] Trávník J., *Kompilátor pro neuronové sítě.* Praha, 2004. Diplomová práce na ČVUT FEL, katedra počítačů.

[CLogging] *Commons logging*
URL: `http://jakarta.apache.org/commons/logging/`

[GAME] *Group of Adaptive Models Evolution*
URL: `http://neuron.felk.cvut.cz/game/index.html`

[Joone] *Java Object Oriented Neural Engine*
URL: `http://www.jooneworld.com/`

[LangList] *The Language List*
URL: `http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm`

[NNToolbox] *Neural Network Toolbox*
URL: `http://www.mathworks.com/products/neuralnet/`

[NeuralWorks] *NeuralWorks*
URL: `http://www.neuralware.com/products_pro2.jsp`

[SNNS-JNNS] *Stuttgart Neural Network Simulator, Java Neural Network Simulator*
URL: `http://http://www-ra.informatik.uni-tuebingen.de/SNNS/`

[SOM_PAK] *The Self-Organizing Map Program Package*
URL: `http://www.cis.hut.fi/research/som-research/nnrc-programs.shtml`

[VHDL-1998] ANSI/IEEE Std 1076-1987, *IEEE Standard VHDL Language Reference Manual*, New York, USA, 1998