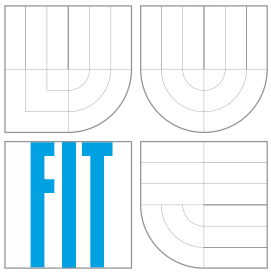
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NEURONOVÉ SÍTĚ A GENETICKÉ ALGORITMY

NEURAL NETWORKS AND GENETIC ALGORITHM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ŠTĚPÁN KARÁSEK

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. FRANTIŠEK ZBOŘIL, CSc.

BRNO 2016

Zadání diplomové práce/7525/2015/xkaras24

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Karásek Štěpán, Bc.**
Obor: Inteligentní systémy
Téma: **Neuronové sítě a genetické algoritmy**
Neural Networks and Genetic Algorithm
Kategorie: Umělá inteligence

Pokyny:

1. Prostudujte zadanou literaturu.
2. Proveďte přehled možné spolupráce genetických algoritmů a neuronových sítí.
3. Navrhněte demonstrační program pro některou ze zjištěných možností.
4. Navržený program implementujte.
5. Proveďte potřebné experimenty s cílem porovnání rychlosti učení a kvality odezvy pro klasickou neuronovou síť (například backpropagation) a neuronovou síť optimalizovanou genetickým algoritmem.
6. Zhodnoťte výsledky.

Literatura:

- David, O. E., Greental, I.: Genetic algorithms for evolving deep neural networks, in Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary
- Gowda, C. Ch., Mayya, S. G.: Comparison of Back Propagation Neural Network and Genetic Algorithm Neural Network for Stream Flow Prediction, Journal of Computational Environmental Sciences, 2014
- Gupta, P., Kaur, B.: Accuracy Enhancement of Artificial Neural Network using Genetic Algorithm, International Journal of Computer Applications, Volume 103 - No 13, October 2014

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání

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

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

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

Vedoucí: **Zbořil František V., doc. Ing., CSc.**, UITS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

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

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato práce se zabývá evolučními a genetickými algoritmy a jejich možnou spoluprací při tvorbě a učení neuronových sítí. V teoretické části jsou popsány genetické algoritmy a neuronové sítě. Také jsou popsány možnosti jejich kombinace a je proveden přehled existujících algoritmů. V praktické části je popsána implementace algoritmu NEAT. Dále jsou s algoritmem NEAT provedeny experimenty a na základě jejich výsledků je navrhnuta kombinace algoritmu s diferenciální evolucí. Výsledky kombinace algoritmů jsou zhodnoceny. V závěru je algoritmus NEAT porovnán s klasickými učícími metodami backpropagation (pro dopředné neuronové sítě) a backpropagation through time (pro rekurentní neuronové sítě) a to z hledika rychlosti učení, kvality odezvy sítě i jejich závislosti na velikosti sítě.

Abstract

This thesis deals with evolutionary and genetic algorithms and the possible ways of combining them. The theoretical part of the thesis describes genetic algorithms and neural networks. In addition, the possible combinations and existing algorithms are presented. The practical part of this thesis describes the implementation of the algorithm NEAT and the experiments performed. A combination with differential evolution is proposed and tested. Lastly, NEAT is compared to the algorithms backpropagation (for feed-forward neural networks) and backpropagation through time (for recurrent neural networks), which are used for learning neural networks. Comparison is aimed at learning speed, network response quality and their dependence on network size.

Klíčová slova

Evoluční algoritmy, genetické algoritmy, neuronové sítě, neuroevoluce, NEAT, diferenciální evoluce.

Keywords

Evolutionary algorithms, genetic algorithms, neural networks, neuroevolution, NEAT, differential evolution.

Citace

KARÁSEK, Štěpán. *Neuronové sítě a genetické algoritmy.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Zbořil František.

Neuronové sítě a genetické algoritmy

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Doc. Ing. Františka Zbořila, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Štěpán Karásek
23. května 2016

Poděkování

Chtěl bych poděkovat vedoucímu práce Doc. Ing. Františku Zbořilovi, CSc. za jeho odborné rady, vedení práce a neustálý optimismus.

Obsah

Kapitola 1

Úvod

Již od počátku vzniku počítačů se objevuje velký zájem o umělou inteligenci. Ten dal vzniknout řadě různých softcomputingových metod inspirovaných přírodními a biologickými procesy. V 40. letech 20. století vznikaly první modely umělého neuronu a v následujícím desetiletí se přidávají i evolucí inspirované algoritmy.

Vznik umělých neuronových sítí byl podpořen faktem, že fungující neuronové sítě jsou v mnoha živých organismech a jsou schopné zpracovávat komplexní informace o svém okolí. Tyto informace jsou mnohdy nekompletní či nepřesné, přesto si s nimi neuronové sítě dokáží poradit. Oblast vývoje umělých neuronových sítí však dodnes není schopna dosáhnout komplexity, která je k vidění u vyspělejších živých organismů.

Evoluční algoritmy, potažmo jejich konkrétnější podskupina – genetické algoritmy, jsou také založené na principech fungujících v přírodě. Procesy selekce nejúspěšnějších a jejich křížením lze získat nové jedince s vlastnostmi svých rodičů. Při použití jisté míry abstrakce lze řešení úlohy vyjádřit jako jedince a simulovat nad nimi proces evoluce. Tímto způsobem je možné optimalizovat složité problémy bez nutné znalosti přesného postupu pro výpočet řešení.

Kombinací výše uvedených metod lze snadno přijít na myšlenku vývoje neuronových sítí pomocí evolučních algoritmů. Jejich spojení však není zdaleka tak snadné, jak by se mohlo zdát. Vývoj se komplikuje mohutným prostorem různých řešení, možnostmi jejich permutací při zachování funkčnosti či proměnnou velikostí různých řešení.

Prvním cílem této práce je ukázat možnosti použití evolučních algoritmů pro vývoj neuronových sítí, s tím související problémy a jejich možná řešení. Kombinace genetických algoritmů a neuronových sítí bude předvedena na implementaci algoritmu NEAT. Na základě dosažených výsledků budou navrženy a vyhodnoceny další úpravy.

Druhým cílem této práce je srovnání implementovaného algoritmu s klasickými přístupy k učení neuronových sítí. Za tímto účelem budou provedeny experimenty, které mají za cíl porovnání algoritmů z různých hledisek. Především se jedná o kvalitu odezvy sítě, rychlost učení a stabilitu výsledků při různých kombinacích parametrů.

Kapitola 2

Evoluční a genetické algoritmy

Genetické algoritmy spadají do třídy heuristických optimalizačních algoritmů. Jedná se o výpočetní model, který využívá jako svou předlohu proces evoluce. Genetické algoritmy spadají do širší třídy evolučních algoritmů a typické jsou svým zaměřením na operátor křížení. Ten je hlavním prvkem při prohledávání stavového prostoru. Informace v této kapitole byly čerpány ze zdrojů [3, 20] a [15].

Řešení jsou zakódována a představena jako jednotlivé organismy. Množina těchto jedinců tvoří populaci, nad kterou probíhá simulovaný proces evoluce za použití selekce silnějších jedinců (kvalitnějších řešení), jejich křížení (angl. „*crossover*“) a mutace. Tento proces se opakuje až do chvíle, kdy je splněna ukončující podmínka (je nalezeno dostatečně kvalitní řešení, nebo je vyčerpán přidělený čas). Jednotlivé genetické algoritmy se mohou lišit různým provedením některé z fází, či kódováním jedinců. Obecný algoritmus však lze popsat následovně:

Algoritmus 1: Obecný genetický algoritmus

```
G = 0;
Pop[G] = náhodnáPopulace(velikostPopulace);
pokud není splněna ukončující podmínka :
    Ohodnoť(Pop[G]);
    Rodiče = selekce(Pop[G]);
    Pop[G+1] = kříženíAMutace(Rodiče);
    G = G+1;
konec
```

2.1 Názvosloví v genetických algoritmech

Přestože jsou genetické algoritmy inspirovány biologií a používají stejné termíny, nejsou tyto významy zcela totožné. Základem algoritmu jsou jednotlivá zakódovaná řešení, která jsou označována jako *chromozomy*, *genomy*, či *jedinci*. Celý genom se skládá z jednotlivých atributů, které lze optimalizovat, a které dohromady popisují řešení. Těmto jednotlivým atributům se říká *geny*. Jedna konkrétní hodnota genu se pak nazývá *allela*.

Z každého genomu lze získat řešení, které je v něm zakódováno. Může se jednat například o pořadí prací, rozvrh učeben či popis tvaru antény [10]. Tomuto získanému řešení se říká *fenotyp* a jeho získání záleží na způsobu, jakým daný genom rozkódujeme. Pro každý problém pak musí existovat funkce, pomocí které lze z daného genom získat řešení nebo ho

naopak do něj zakódovat.

K ohodnocení jedince slouží *fitness funkce*, která nejprve rozkóduje daný genom na fenotyp, zhodnotí jeho kvalitu a přidělí mu jeho ohodnocení. Tato hodnota se pak nazývá *fitness* daného jedince. Čím vyšší fitness, tím lépe je jedinec hodnocen. V některých úlohách je však využívána *inverzní fitness funkce*, kdy lepší jedinci mají nižší fitness a genetický algoritmus se poté snaží fitness minimalizovat. Tomu musí být uzpůsobeny i jiné části algoritmu, především selekce.

Při navrhování fitness funkce je potřeba mít na vědomí, že fitness funkce musí umět co nejlépe rozlišit různou kvalitu řešení gradientem směrem k řešení. To umožňuje směřovat prohledávání do prostoru kvalitnějších řešení.

2.2 Selekce

Proces selekce vybírá z populace všech jedinců ty s lepším ohodnocením. Ti se poté zůčastňují křížení a mohou část svého řešení předávat dál, zatímco jedinci se špatným ohodnocením mají malou či žádnou šanci k předání svého genetického materiálu.

Přestože selekce upřednostňuje lépe ohodnocené jedince, nemusí řešení konvergovat k optimálnímu řešení, ale pouze k lokálnímu maximu. Jedinci, kteří obsahují část optimálního řešení, mohou být hůře ohodnoceni a při procesu selekce ztraceni. Jedná se o matoucí či deceptivní problémy a touto tématikou se více zaobírá *teorie schémat*.

Důraz, který je kladen na výběr pouze nejlepších řešení je nazýván **selekční tlak**. Pomocí selekčního tlaku lze ovlivňovat poměr mezi výběrem pouze nejlépe hodnocených řešení a výběrem i slabších jedinců.

2.2.1 Elitismus

Jedním ze základních přístupů k selekci je takzvaný elitismus. Celá populace je nejdříve seřazena sestupně podle hodnocení jednotlivých řešení. Poté je zachováno určité procento nejlepších jedinců, kteří se dále použijí jako rodiče pro další generaci.

Velikost elity (vybraných jedinců) se může značně lišit podle řešeného problému. Obecně se však jedná o desítky procent, přičemž platí, že snížením velikosti elity je možné algoritmus donutit k rychlejší konvergenci, ovšem za cenu snížení rozmanitosti genomů a zvýšeného rizika uváznutí v lokálním extrému. Naopak zvětšením elity se ke křížení dostanou i jedinci, kteří nemusí obsahovat kvalitní řešení a tím zpomalují celý proces. V celé populaci je však větší diverzita a mohou se objevit řešení, která by byla při přísnějším elitismu ztracena.

2.2.2 K-turnaje

Další paralelu z přírody při selekci využívá k-turnaj (angl. „*tournament selection*“). Oproti elitismu nejsou všechna slabší řešení ztracena, ale také se účastní výběru rodičů. Každý nový rodič se zvolí soubojem mezi k náhodně vybranými jedinci. Pouze nejlepší z nich je poté zvolen jako rodič.

Při *k-turnaji* mají tedy jistou šanci i slabší jedinci, zatímco ti silnější vyjdou z turnaje téměř vždy vítězně. Velikostí turnaje lze navíc upravovat selekční tlak. Zvětšením jeho velikosti mají slabší jedinci menší šanci, že ostatní jedince porazí. Čím více jedinců se turnaje zúčastní, tím spíš někdo porazí slabé řešení.

2.2.3 Ruleta

Ruleta (angl. „*Roullete-wheel selection*“), je algoritmus umožňující výběr rodičů proporcionálně dle jejich ohodnocení. Šance na výběr jedince je dána poměrem fitness daného jedince vůči celkové fitness populace:

$$p_i = \frac{fitness_i}{\sum_{j=1}^{n} fitness_j} \tag{2.1}$$

Tato selekce sice nejlépe zohledňuje kvalitu jedince v porovnání s populací, je však silně závislá na použité fitness funkci a rozložení hodnot fitness jednotlivých jedinců. Pokud jsou jejich ohodnocení příliš málo odlišná, výrazně klesá selekční tlak. Naopak pokud jedno z řešení výrazně převyšuje všechny ostatní, může je naprosto utlačit a podílet se na vzniku většiny nových jedinců.

2.3 Genetické operátory

Jako genetické operátory se označují takové, které operují nad jedinci v populaci a slouží k vytváření nových jedinců a zvyšování diverzity populace. Jejich použitím může fungovat celý proces simulované evoluce v genetických algoritmech. Genetické operátory jsou velice specifické zadanému problému a použitému kódovaní. Tomu může odpovídat i jejich komplikovanost, sahající od základních a velmi jednoduchých až po komplikované operátory vytvořené speciálně pro konkrétní úlohu. Konkrétní ukázky některých operátorů lze nalézt v kapitole 2.4.

2.3.1 Křížení

Standardně se jedná o *binární* genetický operátor, který kombinuje genetickou informaci dvou různých jedinců a vytváří nového jedince. Jedná se o paralelu k biologickému pohlavnímu rozmnožování. Potomek (nové řešení) je složen z genů obou rodičů.

2.3.2 Mutace

Genetický operátor mutace je narozdíl od křížení operátorem *unárním*, který umožňuje měnit geny jednoho jedince. Operátor mutace zachovává diverzitu populace a je schopen vytvoření genomů, které by pouhým křížením byly obtížně dostupné či zcela nedostupné. Tím také snižuje možnost uváznutí GA v lokálním suboptimálním řešení. Jeho použitím dochází k lokálnímu prohledávání prostoru řešení.

Celá třída algoritmů nazývaná *Evoluční Strategie*, ES, je založena na myšlence použití pouze operátoru mutace v kombinaci se selekcí.

2.4 Kódování v evolučních a genetických algoritmech

Aby mohl algoritmus správně pracovat, je potřeba řešení nejdříve zakódovat do genomu. Kódování může být prakticky libovolné, pokud se mu přizpůsobí ostatní části algoritmu. Ovšem jeho volba může mít výrazný vliv na rychlost algoritmu a kvalitu jeho výsledků. Existuje však několik základních přístupů, kterými lze řešení zakódovat. S výběrem kódování zároveň úzce souvisí i genetické operátory křížení a mutace, které budou pro každé kódování popsány zvlášť.

2.4.1 Bitové kódování

Jedním ze základních a nejjednodušších kódování je bitové kódování. Genom se skládá z řetězce genů, které mohou nabývat pouze dvou hodnot, typicky 1 a 0. Přestože lze na binární kódování převést množství problémů, ne vždy je toto kódování vhodné. Lze na něm však velmi jednodušše aplikovat genetické operátory bez nutnosti znát řešený problém.

Jednobodové křížení (angl. „*one point crossover*“) – genomy obou rodičů jsou rozděleny na dvě části ve stejném místě. Vzniknout může buď jeden potomek obsahující jednu část z prvního rodiče a druhou část z druhého rodiče, nebo dva různí potomci, kteří vzniknou záměnou jedné z částí mezi rodiči.

Dvoubodové křížení (angl. „*two point crossover*“) – vychází z jednobodového křížení. Místo rozdělení na dvě části jsou genomy rozděleny na tři části. Potomek získává prostřední část z jednoho rodiče a krajní části z druhého.

Uniformní křížení – Zatímco výše zmíněné operátory křížení zaručují překopírování celých úseků genomu, tak uniformní křížení pracuje na úrovni genů. Genomy se postupně procházejí po jednotlivých genech a se stejnou pravděpodobností je do potomka vybrán gen z prvního nebo druhého rodiče.

Mutace – V binárním kódování je mutace přímočará a jedná se o převrácení hodnoty náhodného genu (bitu) v genomu. Může se jednat o jednorázovou mutaci jednoho bitu, kdy je s pravděpodobností p_{om} vybrán náhodný gen, jehož hodnota je obrácena. Druhou možností je kontinuální procházení genomem a hodnota každého z genů je s pravděpodobností p_{cm} obrácena.

2.4.2 Realné kódování

Přestože lze reálná čísla převést do binárního kódování, nemusí být tento postup výhodnější z důvodů numerické přesnosti při bitovém zápisu či nevyrovnaném vlivu bitových změn na zakódovanou hodnotu reálného čísla. Reálné kódovaní zavádí hodnotu každého genu jako reálné číslo.

Křížení reálných čísel existuje více druhů. Nejjednodušší se prakticky neliší od křížení v binárním kódování. Jelikož operátor křížení pracuje nad geny a nemusí znát jejich hodnotu (datový typ), lze využít všechny tři druhy křížení použité v binárním kódování.

Složitější variantu pak představuje aritmetické, či heuristické křížení. Tyto způsoby křížení již využívají informace v genech. **Aritmetické křížení** kombinuje hodnotu obou genů podle rovnice 2.2:

$$g_i^O = g_i^{P_1} a_1 + g_i^{P_2}(1 - a_1) \qquad a_i \in [-0,25; 1,25] \tag{2.2}$$

Kde g_i^O je gen i v potomkovi, $g_i^{P_1}$ a $g_i^{P_2}$ jsou geny i v prvním a druhém rodiči a a_i je náhodná hodnota.

Heuristické křížení využívá hodnoty genů v lepším jedinci a přičítá k nim váhovanou diferenci oproti genům druhého rodiče podle rovnice 2.3.

$$O = P_{lep} + r \cdot (P_{lep} - P_{hor}) \qquad r \in (0; 1] \tag{2.3}$$

Kde O značí potomka, P značí rodiče a r je předem zvolený koeficient.

2.4.3 Permutační kódování

Příkladem, kdy lze kódováním problému výrazně ovlivnit prostor řešení je permutační kódování. O permutační kódování se jedná ve chvíli, kdy jsou předem dané hodnoty, které mají být obsažené v genomu a genetický algoritmus se snaží nalézt jejich optimální pořadí. Typickým zástupcem je například *problém obchodního cestujícího* nebo *problém osmi dam*.

Problém osmi dam spočívá v rozmístění osmi dam na šachovnici takovým způsobem, aby se navzájem podle pravidel šachů neohrožovaly. Jednoduchým zakódováním by mohlo být 8 genů obsahující celá čísla s rozsahem hodnot 1 až 8. Každá z dam se by pohybovala na jednom řádku a odpovídal by jí jeden gen s její pozicí. V tomto kódování však může nastat situace, kdy je více dam na stejné pozici, čímž se navzájem ohrožují. Zavedením permutačního kódování, kdy je v genomu každá pozice pouze jednou, jsou všechny tyto situace eliminovány a prohledávaný prostor je výrazně zmenšen.

Toto omezení stavového prostoru je však za cenu nemožnosti použít klasické křížení z reálného kódování, neboť umožňuje vznik nevalidních jedinců. Pro permutační problémy jsou proto používány speciální genetické operátory, které pouze permutují existující geny.

2.4.4 Speciální kódování

Předchozí kódování a genetické operátory ukazují základní principy, které lze využít. Při složitějších problémech je však často využíváno specializovaných kódování. Zároveň jsou těmto kódováním uzpůsobeny i genetické operátory. Kódování i genetické operátory jsou často specifické řešenému problému a snaží se zabránit situacím, které nevedou k řešení (obdobně jako při permutačním kódování).

Příkladem takového kódování může být algoritmus NEAT popsaný v této práci (kap. 5). Kódování v NEATu může obsahovat nestejně dlouhé genomy s geny na náhodných pozicích. Jedním ze specifik kódování tohoto algoritmu je použití identifikátoru určujícího původ genu. Jednotlivé geny poté obsahují tento identifikátor, který je při křížení využit k zarovnání shodných genů. Tím je docíleno efektu, kdy se při křížení proti sobě postaví shodné geny (geny se shodným původem). Bez této vlastnosti by neexistoval způsob, jak rozhodnout, které geny si odpovídají a v potomkovi by se mohl objevit stejný gen dvakrát (z každého rodiče jeden).

Kapitola 3

Neuronové sítě

Lidský mozek umožňuje zpracování obrovského množství informací. S příchodem počítačů zároveň vznikla snaha napodobit způsob zpracování informací, který mozek využívá. S prvním modelem umělého neuronu přišel v roce 1943 W. McCulloch a W. Pitts [18]. Vzájemným propojením umělých neuronů lze poté vytvořit umělé neuronové sítě schopné řešit komplexní úlohy. V dnešní době se neuronové sítě využívají pro řešení řady úloh mezi které patří například zpracování vizuálních informací, zpracování řeči, predikce, rozhodování či optimalizace.

Během let vznikla celá řada neuronových sítí, které se mezi sebou liší v různých aspektech, ať už je to zpracování dat neurony, topologie sítí, způsob učení či rozdílné výpočetní jednotky (LSTM - Long Short-Term Memory).

Neuronové sítě jsou navíc **Turingovsky kompletní**, což vede k důležitému závěru, že jsou schopné jakéhokoliv výpočtu jako klasický počítač. To se podařilo dokázat pánům Franklinu a Garzonovi, kteří vytvořili implementaci Turingova stroje pomocí neuronové sítě [13].

Model umělého neuronu a potažmo i neuronových sítí je založen na biologické předloze mozku. Z jeho funkčnosti se pokouší abstrahovat obecné principy a využít je k výpočtům.

3.1 Model neuronu

Model biologického i umělého neuronu je popsán na základě informací z článků [9] a [18]. Biologický neuron je buňka, která je schopná zpracovávat a přenášet elektrické (elektrochemické) signály. Přestože funkčnost biologického neuronu je založena na komplexním systému elektrochemických vazeb, lze od nich při popisu funkčnosti abstrahovat. Základem buňky je její jádro, které je schopné udržovat na svém povrchu elektrický potenciál. Pokud potenciál překročí jistou prahovou hodnotu, je vyslán elektrický signál do dalších neuronů, kde příchozí signál opět ovlivňuje celkový potenciál jádra. Jednotlivá spojení mohou být *inhibitována* nebo *exhibitována*.

Základním stavebním prvkem (výpočetní jednotkou) umělých neuronových sítí je umělý neuron (dále již pouze neuronová síť a neuron). Neuron přebírá základní principy z funkčnosti biologického neuronu. Může být napojen na jiné neurony a jejich příchozí hodnota je modifikována váhou spojení w (inhibiční i exhibiční). Každý neuron má svou výstupní hodnotu. Výpočet výstupní hodnoty neuronu je prováděn pomocí dvou funkcí, bázové a aktivační.

$$y = f_a(f_b(\vec{w}, \vec{x})) \tag{3.1}$$

Bázová funkce f_b slouží ke kombinaci všech vstupních signálů do jedné hodnoty. Na základě této hodnoty je poté rozhodnuto o výstupu neuronu pomocí **aktivační funkce** f_a – angl. „*transfer function*“. Ta mívá charakter schodové funkce se změnou hodnoty v bodě θ, což odpovídá hraniční hodnotě potenciálu v biologickém neuronu. Příklad jednoduché aktivační funkce je rovnice 3.2.

$$f_a(x) = \begin{cases} 0 & pro \quad x < \theta \\ 1 & pro \quad x \geq \theta \end{cases} \tag{3.2}$$

Kromě hodnoty 0 a 1 bývají obvykle používány i hodnoty -1 a 1. Zvolit však lze teoreticky libovolné. Aby nemusel být práh vyjádřen jako atribut neuronu a bylo možné jej jednodušše měnit, je používána častá úprava, kdy je **práh** dodán jako vstup neuronu s indexem 0 (viz rovnice 3.3).

Skládání potenciálů v biologickém neuronu odpovídá *lineární bázová funkce* (LBF), která provádí sumu všech váhovaných vstupů neuronu (rovnice 3.3). Kromě LBF existují i jiné bázové funkce (např. RBF – Radial Basis Function), ve zbytku práce se však bude pracovat s LBF.

$$f_b(\vec{w}, \vec{x}) = \sum_{i=0}^{n} w_i \cdot x_i \qquad\qquad w_0 = \theta, \quad x_0 = -1 \tag{3.3}$$

Jak již bylo řečeno, výstup neuronu je určen aktivační funkcí. Zatímco základní aktivační funkcí byla schodová funkce, lze použít řadu jiných. Další možností je například po částech spojitá lineární funkce. Mezi nejpoužívanější pak patří **logistická funkce** (rovnice 3.4), kterou je možné derivovat, což je potřebné pro některé učící algoritmy. Existují však i neuronové sítě, které používají pro aktivační funkci periodické nebo gaussovské funkce [22].

$$\frac{1}{1 + e^{-kx}} \tag{3.4}$$

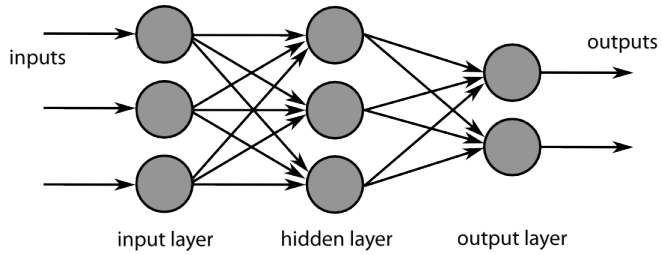
3.2 Umělé neuronové sítě

Neuronové sítě jsou definovány jako množina *propojených* jednoduchých výpočetních jednotek (neuronů), které jsou založeny na podobnosti s biologickým neuronem. Výpočetní schopnosti sítě jsou určeny váhami jednotlivých spojení. Proces nastavování vah sítě je označován jako *učení* [9].

Množina neuronů a způsob jejich vzájemného propojení je označován jako **topologie sítě**. Podle vlastností topologií se rozlišují různé **architektury sítí**. Jednotícím prvkem jsou vstupní a výstupní neurony (vstupní a výstupní vrstva). Zatímco výstupní vrstva neuronů je zřejmou nutností, vstupní vrstva je volitelná. Vstupní hodnoty mohou být vloženy přímo na požadovaná spojení, nicméně používanějším přístupem je pro každý vstup vytvořit **vstupní neuron**, který má konstantní hodnotu danou vstupem sítě. Na tento vstupní neuron(y) pak může být napojen zbytek sítě. Všechny ostatní neurony jsou označovány jako skryté – angl. „*hidden*“.

3.2.1 Dopředné sítě

Jednou z nejpoužívanějších architektur jsou dopředné (angl. „*Feed Forward*“) sítě. Ty se skládají z několika vrstev neuronů a základním pravidlem je, že výstupy neuronů jedné vrstvy jsou napojeny pouze na neurony následující vrstvy (lze vidět na obrázku 3.1). V sítí se tedy vyskytují pouze dopředná spojení (odtud anglické „Feed Forward“). Běžnou variantou jsou **plně propojené** Feed Forward sítě, kdy na vstup každého neuronu jsou připojeny výstupy všech neuronů z předešlé vrstvy. Vyhodnocení sítě lze jednoduše provést postupným propagováním výsledků z vrstvy do vrstvy s počátkem ve vstupní vrstvě. Mezi běžné využití patří zejména klasifikace, řízení, či predikce.



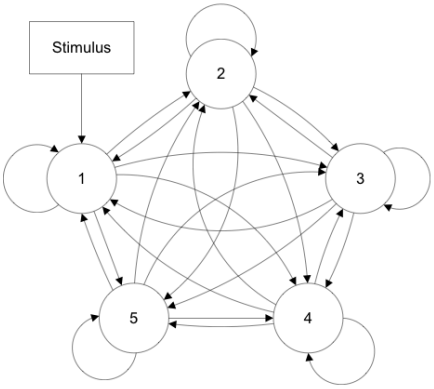
Obrázek 3.1: Dopředná neuronová síť. Síť je rozdělena do vrstev. Výstupní hodnota neuronů je vždy propagována pouze do následující vrstvy. [29]

Pro volbu přesného počtu skrytých vrstev a počtu neuronů neexistuje obecný návod a tyto parametry jsou často voleny podle zkušeností. Z tohoto důvodu vznikají algoritmy, které nejenom optimalizují váhy spojení, ale také budují topologii sítě. Výhodou dopředných sítí je jejich prozkoumané matematické pozadí a snazší způsoby jejich učení. Mezi nejznámější učící algoritmy patří „*Backpropagation*“ algoritmus.

3.2.2 Rekurentní sítě

Narozdíl od dopředných sítí mohou mít rekurentní sítě libovolné propojení mezi neurony (viz obr. 3.2). Vzhledem k neuspořádanému propojení neuronů již nemusí existovat rozdělení do vrstev. Předností rekurentních sítí je jejich možnost vzít v potaz i předchozí výstupy z různých neuronů sítě, čímž mohou vznikat rekurentní vazby a cyklické závislosti. Síť má tedy možnost vzít v úvahu i **předchozí stavy**.

Vzhledem ke zpětným vazbám již nelze síť vyhodnocovat stejně jako dopředné sítě. Místo toho se používá přístup podobný spojité simulaci. V prvním kroku se výstup každého z neuronů propaguje do všech ostatních na něj napojených neuronů. V druhém kroku se z těchto aktualizovaných vstupů vypočítá nová výstupní hodnota neuronu. To lze opakovat dokud se výstupní hodnota sítě neustálí. Jako zjednodušení však lze použít fixní počet iterací.



Obrázek 3.2: Rekurentní neuronová síť (zde konkrétně plně propojená). V síti se objevují i zpětné vazby. Výstupní hodnoty neuronů mohou být propagovány do libovolného neuronu. [29]

Rekurentní sítě umožňují vytvářet vztahy mezi předchozími a aktuálními vstupy a využívají se proto k úlohám, ve kterých se vyskytují sekvenční vztahy. Mezi takové úlohy patří například rozpoznávání řeči, syntaktická analýza, predikce časových řad či kontinuální řízení dynamických systémů. Pro mnohé problémy však lze použít oba dva typy sítí.

Nevýhodou rekurentních sítí je jejich **náročnější učení**. Dvěma známými problémy jsou „*vanishing gradient*“ a „*exploding gradient*“ u metod založených na propagování chyby sítí. Ty jsou problematické, neboť závisí na derivacích aktivačních funkcí neuronů a hodnotě vah jejich spojení. Mnohonásobným průchodem rekurentními spojeními je chyba v závislosti na derivacích aktivačních funkcí neuronů buď utlumena (pro aktivační funkce s derivací < 1) – „*vanishing gradient*“ – nebo nepřiměřeně zesílena (v případě derivací aktivačních funkcí > 1) – „*exploding gradient*“. Tento problém řeší například *LSTM* (Long Short-term Memory) sítě [4]. Existují také upravené algoritmy pro rekurentní sítě jako jsou *BPTT* (Backpropagation Through Time) [8] nebo *Hebbian learning* [26].

Kapitola 4

Spojení evolučních algoritmů a neuronových sítí

Během 80. let došlo k intezivnímu výzkumu v oblasti propojení evolučních algoritmů a neuronových sítí [12]. Zatímco evoluční algoritmy jsou schopné optimalizace složitých problémů, neuronové sítě naopak potřebují množství parametrů nastavit. Navíc pro výběr topologie neuronových sítí neexistuje přesný postup a často se musí zkoušet různé varianty. Spojení evolučních algoritmů a neuronových sítí se zdá být tedy přirozené. Existuje však řada výzev, které je potřeba vyřešit. Velký vliv má v tomto ohledu především způsob kódování neuronové sítě.

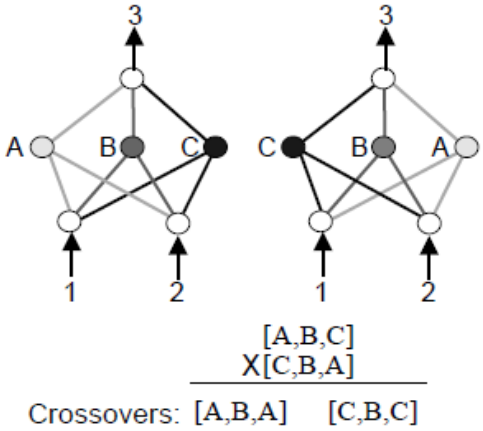
Kódování lze rozdělit na dvě odlišné skupiny – **přímé** a **nepřímé** kódování. Zatímco u **přímého kódování** jednotlivé geny odpovídají konkrétním neuronům a spojením, **nepřímé kódování** zpravidla určuje, jakým způsobem danou síť vytvořit. Nepřímé kódování umožňuje vznik kompaktních genomů, které dokáží generovat mnohem větší sítě, či opakovaný výskyt stejných podsítí či vzorů.

Jednou z největších výzev je **Competing Convetion** problém. Jedná se o situaci, kdy existuje pro neuronovou síť více různých genomů, které se mohou lišit pouze permutací neuronů v genomu. Při křížení takovýchto genomů pak vznikají nepoužitelní jedinci. Čím větší je síť, tím více takových permutací existuje a tím více poškozených jedinců vzniká [24] (viz obr. 4.1).

Další výzvou je nestejná **délka genomů**. Standardní genetické algoritmy pracují na stejně dlouhých genomech. Při tvorbě rozrůstající se topologie ale délka genomu nemůže zůstat konstantní. S tímto je potřeba počítat při definici operátoru křížení. V kombinaci s *competing convetion* problémem je navíc zapotřebí mít způsob, kterým lze u dvou různých topologií určit, které části si navzájem odpovídají. Z tohoto důvodu mnohé neuroevoluční algoritmy nepoužívají operátor křížení [1, 6, 11].

4.1 Existující algoritmy

V dnešní době existuje více neuroevolučních algoritmů, z nichž se každý snaží jít jinou cestou a jsou v nich použity mnohdy značně rozdílné způsoby kódování i evoluce. Většina algoritmů se kvůli *competing convention* problému vyhýbá operátoru křížení a tudíž se jedná o obecnější evoluční algoritmy.



Obrázek 4.1: Příklad *Competing Convetion* problému. Obě sítě reprezentují stejné řešení. Skryté neurony jsou však permutované a při křížení dochází ke ztrátě funkčnosti. [25]

4.1.1 GNARL

Algoritmus GNARL (GeNeralized Acquisition of Recurrent Links) vychází z předpokladu, že operátor křížení v genetických algoritmech neumožňuje vznik smyslupných potomků [1] a dává přednost pouze operátoru mutace. V další kapitole však bude ukázáno, že při volbě správného kódování a využití komplexnějších operátorů křížení je schopný vyvíjet neuronovou síť i genetický algoritmus.

Na počátku vývoje algoritmus GNARL nepoužívá sítě s minimální topologií. Kromě vstupních a výstupních neuronů je vygenerováno i určité množství skrytých neuronů a náhodných spojení. Množství skrytých neuronů i spojení je dáno uživatelem určenými hodnotami. Pro spojení navíc platí několik pravidel:

- Spojení nesmí vést do vstupního neuronu
- Spojení nesmí vést z výstupního neuronu
- Nesmí existovat více spojení stejného směru mezi dvěma neurony

V každé generaci jsou sítě ohodnoceny a proběhne elitistická selekce 50% nejlepších jedinců. Nad těmito jedinci dále probíhají mutace. Ty lze rozdělit na dvě skupiny, parametrické a strukturální mutace.

Parametrické mutace upravují váhy jednotlivých spojení. Velikost mutací je určována podobně jako u algoritmu simulovaného žíhání pomocí hodnoty teploty sítě [2]. Ta je vypočítána na základě blízkosti řešení sítě k optimálnímu řešení. Změnu váhy w pro síť η lze poté popsat vztahem:

$$w = w + N(0, \alpha\hat{T}(\eta)) \tag{4.1}$$

Kde w je hodnota váhy, α značí konstantu ovlivňující velikost mutací, $N(\mu, \sigma^2)$ je gaussovské rozložení a $\hat{T}(\eta)$ je náhodná hodnota z rozsahu $[0, T(\eta)]$ daná vzorcem:

$$T(\eta) = 1 - \frac{f(\eta)}{f_{max}} \tag{4.2}$$

$$\hat{T}(\eta) = N(0, T(\eta)) \tag{4.3}$$

Kde $f(\eta)$ značí fitness sítě η a f_{max} značí maximální možnou hodnotu fitness. Z výše uvedených vzorců je vidět, že teplota T klesá s tím, jak se řešení blíží k optimu. Nevýhodou však je potřeba znát fitness optimálního řešení dopředu.

Strukturální mutace umožňují změnu topologie sitě. GNARL umožňuje přidávat i mazat jak skryté neurony, tak spojení. Při vkládání nových spojení a neuronů je hlavní snaha co nejméně zasáhnout do funkčnosti sítě. Nová spojení mají nulovou váhu, která musí být poté optimalizována dalšimi mutacemi. Nový neuron naopak nemá žádná spojení. Ta se musí také teprve vyvinout. Rušení neuronu či spojení již nelze provést s nižším dopadem na funkčnost sítě a aplikace této mutace sebou nese pravděpodobnost narušení této funkčnosti. Každá strukturální mutace má dány limity četnosti výskytu této mutace hodnotami $[\Delta_{min}, \Delta_{max}]$.

Algoritmus GNARL byl použit pro simulaci konečných automatů (Trigger Problem), indukci regulárních jazyků či problém mravence (The Ant Problem) [1].

4.1.2 EANT

Jedním ze zástupců novějších přístupů ke kódování a k optimalizaci neuronových sítí je algoritmus EANT (angl. „*Evolutionary Acquisition of Neural Topologies*“) [11]. Jeho největší zajímavostí je nepřímé kódovaní, které umožňuje vyhodnocení sítě i bez nutnosti jejího sestavení (získání fenotypu).

Genomy v algoritmu EANT jsou lineární a skládají se z několika základních typů genů:

- Vstupní gen – vstup sítě
- Gen neuronu – skrytý či výstupní, umožňující výpočet hodnoty
- „Jumper gen“ – umožňuje přidat další spojení

Kódování je velice úsporné a to i z toho důvodu, že topologie sítě je částečně zakódována v pořadí jednotlivých genů. Fenotyp lze z genomu získat tím způsobem, že si neurony představíme jako funkce (funkční symboly) a vstupy s jumper geny jako terminály (terminální symboly). Každý neuron má implicitně jeden výstup. Tento výstup je použit jako vstup předchozího neuronu. Pokud je potřeba více spojení vedoucích z jednoho neuronu, musí být zakódována jumper geny.

Genom si lze také představit jako zápis v postfixové formě. Operátory jsou představovány jednotlivými neurony a jsou následovaná svými vstupy – operandy. Tímto způsobem je umožněna zvláštní schopnost tohoto kódování – vyhodnocení genomu bez nutnosti dekódování na síť. Genom je procházen zprava doleva a všechny vstupní hodnoty jsou přidávány na zásobník. Pokud je na řadě neuron, je ze zásobníku vybráno tolik hodnot, kolik má neuron vstupů a výsledná hodnota neuronu je vrácena zpět na zásobník. Při výpočtu rekurentního jumper genu je získána hodnota přímo z neuronu, který je již vypočítaný. Při zpracování dopředného jumper genu (který získává hodnotu z neuronu, který ještě vyhodnocen nebyl) je nejdříve zkopírován subgenom patřící příslušnému neuronu. Ten je nezávisle vyhodnocen a jeho výsledná hodnota je předána jako hodnota dopředného jumper genu.

Úvodní inicializovaná populace je tvořena minimálními jedinci, kteří mají pouze vstupy a výstupy. Každý z výstupů je připojen k cca 50% vstupů. Celá smyčka algoritmu EANT se pak skládá ze tří kroků. Prvním je tzv. „**structural exploitation**“, což lze volně přeložit

jako využití struktury. Tento krok optimalizuje váhy sítě (struktura zůstává) za pomoci CMA-ES algoritmu (Covariance Matrix Adaptation Evolution Strategy).

Po optimalizaci vah všech sítí je na řadě **selekce**. Ta je založena především na hodnotě fitness, ovšem při více jedincích s podobnou fitness mají při selekci přednost jedinci s menším počtem genů v genomu. Taktéž se hodnotí strukturální podobnost jedinců a příliš podobní jedinci jsou penalizováni.

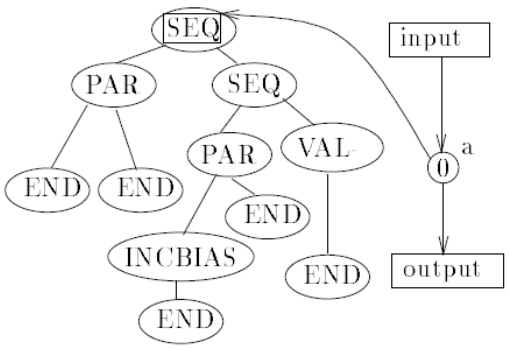
Posledním krokem ve smyččce algoritmu je tzv. „**structural exploration**“. Volně přeloženo se tentokrát jedná o průzkum možných topologií. V tomto kroku je aplikován operátor mutace, který umožňuje přidávání neuronů a jejich spojení. Podle [11] ubírání neuronů příliš zhorší funkčnost sítě a je proto vypuštěno. Taktéž operátor křížení je vynechán, neboť stejných genomů lze dosáhnout i pouhou aplikací strukturálních mutací.

Nově vytvořené struktury jsou poté opět podrobeny prvnímu kroku (*structural exploitation*), kde jsou optimalizovány jejich váhy a je možné zjistit, které strukturální mutace byly prospěšné.

Algoritmus je schopný tvořit minimální struktury, což bylo ukázáno na generování sítí řešících funkci XOR. Úspěšně byl použit k vyřešení úlohy „*double pole balance*“ a to i složitější verze bez informace o rychlostech. Silnou stránkou algoritmu je výrazně nižší počet vyhodnocení fitness funkce, než u jiných neuroevolučních algoritmů (porovnáváno s CE, ESP a NEAT) [11].

4.1.3 CE

Zajímavý způsob kódování neuronových sítí je pomocí tzv. *Cellular Encoding* – CE. Ten je založen na stromové architektuře reprezentující gramatiku pro vznik sítě a počáteční buňce, která je touto gramatikou rozvíjena (viz obr. 4.2). Každá buňka obsahuje čtecí hlavu vedoucí do stromu s gramatikou. Čtecí hlava tedy ukazuje na operaci, která se má provést. Jednotlivé uzly stromu obsahují symboly určující další vývoj aktuální buňky.



Obrázek 4.2: Ukázka celulárního kódování – *Cellular Encoding*. Vlevo je vidět strom reprezentující gramatiku pro vznik sítě. Vpravo je počáteční buňka označená písmenem *a*. Šipka vedoucí do uzlu *SEQ* ukazuje, kam směřuje čtecí hlava buňky *a*. [6]

Proces vzniku sítě pak začíná v počáteční buňce, která je napojená na vstupy a výstupy. Zároveň je čtecí hlava v kořenu stromu a provádí se instrukce, která je v něm obsažena. Pokud dojde k duplikaci buněk, tak se duplikuje i čtecí hlava (viz obr. 4.3). Nová pozice čtecí hlavy je popsána v každé instrukci. Základními instrukcemi jsou:

- SEQ – sekvenční rozdělení buňky. Buňka je rozdělena na dvě. První z nich získá všechny vstupní spojení z původní buňky, zatím co druhá získá výstupní spojení. Z první buňky do druhé je přidáno spojení s váhou 1. Vytváření nové buňky odpovídá v gramatickém stromu jeho rozdvojení na dva podstromy a duplikaci čtecí hlavy do obou podstromů. První buňka je dále řízena čtecí hlavou z levého podstromu, zatímco pravá buňka čtecí hlavou z pravého podstromu.

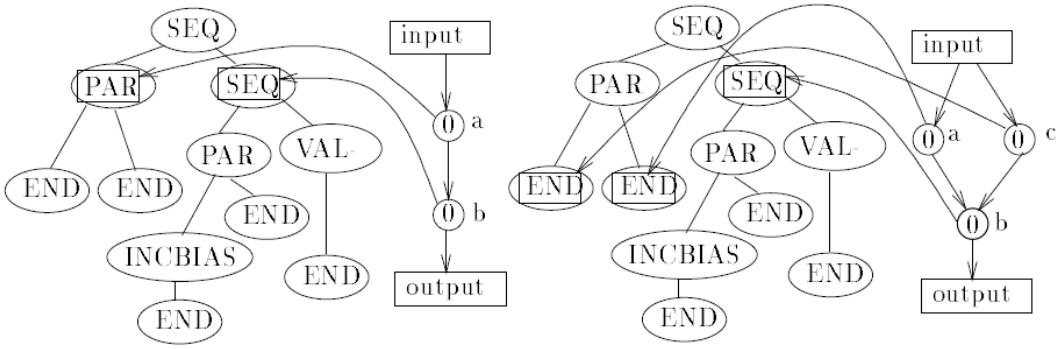
- PAR – paralelní rozdělení buňky. Buňka je namísto rozdělení zduplikována. Obě buňky získají stejné vstupní i výstupní spojení. Duplikaci buňky opět odpovídá rozdvojení gramatického stromu na dva podstromy, včetně rozdvojení čtecí hlavy. Každá z buněk pak získá vlastní čtecí hlavu ve svém podstromě.

- END – vznik neuronu. Tento symbol označuje, že buňka už se dále nebude měnit a vzniká z ní neuron. Uzel s tímto symbolem nemá žádné potomky a ukončuje celý gramatický strom (jedná se vždy o listy stromu).

- INCBIAS/DECBIAS – Úprava prahu dané buňky. INCBIAS hodnotu zvedne o 1, zatímco DECBIAS hodnotu sníží o 1. Tento uzel má jednoho potomka, na kterého se posune čtecí hlava.

- INCLR/DECLR a VAL+/VAL- – Úprava vah spojení buňky. Pomocí symbolů INCLR a DECLR je možné měnit spojení v registru buňky (Link Register). Spojení v registru buňky je možné měnit pomocí symbolu VAL+ a VAL-, které zvýší, či sníží váhu spojení o 1.

- CUT – zrušení spojení. Použitím tohoto symbolu je spojení uložené v registru buňky zrušeno.



Obrázek 4.3: Krok při generování sítě. Na levém obrázku byla nad buňkou *a* provedena operace *SEQ* (viz níže) a vznikla buňka *b*. Čtecí hlava buňky *a* nyní směřuje do kořene levého podstromu a čtecí hlava buňky *b* do kořene pravého podstromu. Na pravém obrázku byla nad buňkou *a* v dalším kroku provedena operace *PAR*. Čtecí hlava buňky *a* opět směřuje do kořene levého podstromu, zatímco čtecí hlava nově vzniklé buňky *c* směřuje do kořene pravého podstromu. [6]

Každá buňka obsahuje hodnotu BIAS, ze které je při vzniku neuronu určen jeho práh a registr obsahující odkaz na některé ze spojení vedoucí do této buňky. Změny vah jsou

prováděny pouze nad spojením, které je aktuálně v registru buňky. Existují však operace umožňující měnit aktuální spojení v registru buňky.

K získání gramatického stromu je možné s úspěchem použít genetické programování [6], což vedlo k úspěšnému vyřešení složitější verze úlohy „double pole balance“ bez informace o rychlostech [7]. CE kódování navíc umožňuje modularitu a znovupoužití subsítí, kdy je umožněno vyvíjet více gramatických stromů zároveň a odkazovat se na ně. Jedná se tak o jistou paralelu k podprogramům.

Kapitola 5

NEAT

V roce 2002 navrhnul Kenneth O. Stanley a Risto Miikkulainen novou metodu neuroevoluce nazvanou NEAT – NeuroEvolution of Augmenting Topologies [25]. V roce 2004 následovala disertační práce K. Stanleyho na téma efektivní evoluce neuronových sítí pomocí komplexifikace, kde popisuje NEAT algoritmus [24]. Tato metoda je zaměřena nejen na optimalizaci vah neuronové sítě, ale také na tvorbu její topologie. Oproti jiným metodám (viz kap. 4.1) NEAT používá přímé kódování včetně operátoru křížení.

Algoritmus je založen na několika základních bodech. Jedním z hlavních je řešení *Competing Convention* problému (viz kap. 4) a křížení různých topologií s nestejně dlouhými genomy. Z tohoto důvodu bylo zavedeno kódování umožňující **efektivní porovnání topologií** sítí.

Dalším bodem je **komplexifikace** řešení z minimální topologie. Všechny vyvíjené sítě začínají bez skrytých neuronů, které se teprve musí vyvinout.

Posledním bodem je pak **ochrana inovací** pomocí rozdělení jedinců na jednotlivé „druhy“. Nová strukturální mutace většinou fitness sítě zpočátku zhorší a je nutné tuto mutaci nejdříve zoptimalizovat. Aby nedošlo k zahození těchto jedinců při procesu selekce, jsou podobné genomy umístěny do jedné skupiny (druhu) a evoluce probíhá v každém druhu zvlášť.

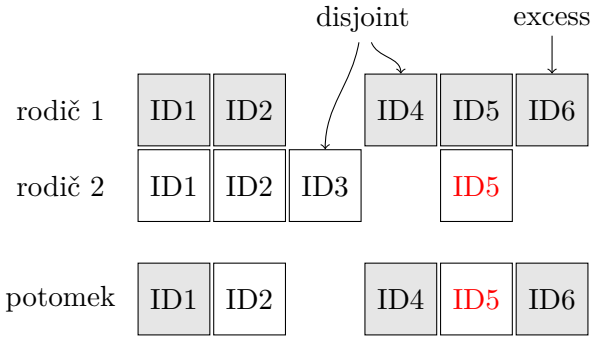
5.1 Kódování

Jak bylo již zmíněno, NEAT používá **přímé kódování** vyjádřené v lineárním genomu. Každému neuronu či spojení odpovídá jeden gen („*Node Gene*“ a „*Connection Gene*“). Pro úspěšné použití operátoru křížení musí zvolené kódování umožnit řešit *Competing Convention* problém. Také musí vhodně kódovat různé topologie, aby bylo možné křížit nestejně dlouhé genomy. NEAT je v tomto ohledu inspirován přírodou, kdy jsou při křížení párovány geny vyjadřující stejnou vlastnost. V algoritmu je každému nově vzniklému genu přiřazeno **konkrétní ID**, které se již nemění. Při tvorbě potomka jsou v genech děděny i hodnoty ID a je tak možné u každého genu jednoznačně určit jeho předchůdce, ze kterého vznikl. Díky tom lze také zjistit, který gen odpovídá kterému genu v druhém genomu. Při křížení jsou pak jednotlivé geny **sjednoceny** („*lined up*“) podle jejich ID a soupeří spolu geny vyjadřující stejnou vlastnost (mají shodné ID). Tím se lze vyhnout Competing Convention problému, jelikož geny jsou označeny a nelze je zaměnit. Zároveň je umožněno porovnání podobnosti dvou topologií. Pokud je většina genů shodná v obou genomech, jedná se o podobné jedince, kteří jsou vhodní ke křížení.

V NEAT je možné neurony přidávat a spojení přidávat i ubírat. Ubírání spojení se ale neřeší odebráním genu, ale jeho vypnutím. Gen v genomu nadále zůstává, jeho hodnota se však ve fenotypu neprojeví. Jednou vypnutý gen může být během následujících mutací opět zapnut.

5.2 Křížení

Díky zvolenému kódování je možné vytvořit operátor křížení, který je schopný pracovat nad nestejně dlouhými genomy a díky párování genů se stejným ID nevytváří poškozené potomky.



Obrázek 5.1: Ukázka křížení v algoritmu NEAT. Rodič 1 má lepší hodnotu fitness. Gen ID5 je v rodiči 2 vypnutý a má proto větší šanci být zděděn.

Celá operace začne seřazením genů podle jejich ID. Poté je z každého páru genů se shodným ID vložen do potomka gen z prvního či druhého rodiče, a to se stejnou pravděpodobností. Pokud je gen v některém z rodičů vypnutý, existuje 75% šance (hodnota udávaná v článku [25]), že bude vypnutý tento gen i v potomkovi. Geny, které nemají v druhém rodiči odpovídající pár, jsou převzaty pouze z rodiče s lepším fitness. Proces křížení lze vidět na obrázku 5.1.

5.3 Mutace

Stejně jako u ostatních neuroevolučních algoritmů lze mutace v NEAT rozdělit na **strukturální** a **parametrické**. Strukturálními mutacemi jsou přidání spojení („*add connection*“) a vložení neuronu („*add node*“). Mezi parametrické mutace patří mutace vah („*weight mutation*“) a znovuzapnutí genu („*toggle link*“).

Při strukturálních mutacích dochází k rozšiřování genomu. Nově vzniklé geny získávají své ID z globálního čítače nazvaného „*innovation number*“. NEAT registruje různé strukturální mutace během jedné generace a pokud se vyskytne stejná mutace ve více genomech, nově vzniklé geny sdílí stejné ID. Cílem je omezit množství shodných genů s různými hodnotami ID a zabránit přílišnému nárustu různých druhů (viz kap. 5.4).

5.3.1 Přídání spojení

Při mutaci přidání spojení jsou nalezeny dva neurony, mezi kterými ještě neexistuje spojení a je mezi ně přidáno. Spojení nemůže vést zpět do vstupních neuronů, ale může vytvářet

rekurentní vazby. Váha spojení je volena náhodně v rozmezí daném uživatelem. Jelikož se jedná o strukturální mutaci při které vznikne nový gen, je mu přiřazeno jeho ID podle „*innovation*“ čítače a ten je inkrementován.

5.3.2 Vložení neuronu

Mutace *vložení neuronu* nalezne již existující spojení a do něj „vloží“ nový neuron. Toho dosáhne tím, že původní spojení vypne (gen nadále zůstane v genomu) a na jeho místo vloží nový neuron. Ten je napojen na neurony z původního spojení. Aby byl vliv mutace co nejméně destruktivní, je váha spojení vedoucí do nového neuronu nastavena na 1 a váha výchozího spojení na váhu shodnou s váhou původního spojení. Nově vzniklé geny samozřejmě získají nové ID.

5.3.3 Mutace vah

Způsob provedení mutace vah se liší podle různých publikací i konkrétních implementací. K. Stanley v původním článku pouze poznamenává, že každá váha je s určitou pravděpodobností pozměněna („*perturbed*“) [25]. Ve své disertační práci [24] pak upřesňuje, že je k aktuální váze přidána reálná hodnota ze zadaného rozsahu s uniformním rozložením.

Algoritmus 2: Mutace vah genomu

```
konec = velikost genomu * 0.9;
výrazně = nepravda;
pokud s pravděpodobností výrazné mutace :
    výrazně = pravda
// slabá mutace upravuje starou váhu
// silná mutace přiřazuje novou váhu
pro každý gen spojení :
    // při výrazné mutaci je generováno více nových vah
    pokud výrazně :
        // 70% šance na slabou mutaci
        bod slabé mutace = 0.3;
        // pokud nedojde ke slabé mutaci, má ještě šanci silná
        bod silné mutace = 0.1;
    // nejnovější geny podstupují nejvíce mutací
    jinak pokud pozice genu > konec :
        // 20% šance na slabou mutaci
        bod slabé mutace = 0.8;
        // jinak dojde vždy k silné
        bod silné mutace = 0.0;
    // šance na slabou mutaci je dána parametry algoritmu
    // existuje 50% šance že se může objevit i silná mutace
    jinak :
        pokud s pravděpodobností 50% :
            bod slabé mutace = 1 - pravděpodobnost mutace;
            bod silné mutace = 1 - pravděpodobnost mutace - 0.1;
        jinak :
            // bez silné mutace
            bod slabé mutace = 1 - pravděpodobnost mutace;
            bod silné mutace = 1 - pravděpodobnost mutace;
    velikost mutace = náhodné číslo s uniformním rozložením [-1,1];
    volba = náhodné číslo s uniformním rozložením [0,1];
    pokud volba > bod slabé mutace :
        // váha spojení je pouze upravena
        váha spojení += velikost mutace * síla mutací;
    jinak volba > bod silné mutace :
        // váha spojení je vygenerována nová
        váha spojení = velikost mutace * síla mutace nové váhy;
konec
```

Další přístup k mutaci vah je pak představen v samotné implementaci algoritmu NEAT K. Stanleym [23], kterou lze vidět na algoritmu 2. V té je použit sofistikovanější přístup k úpravám vah založený na myšlence, že starší spojení jsou již z větší části optimálizována a není je tedy potřeba příliš měnit. Velikost mutací se pak volí podle stáří genu (respektive jeho ID, které je větší pro mladší geny), kdy určitá část nejmladších genů má zvýšenou šanci na výraznější změnu váhy.

5.3.4 Znovuzapnutí genu

Jedná se o velice přímočarou metodu mutace, kdy je v genomu vyhledán dříve vypnutý gen a je opět zapnut. Vypnuté geny vznikají při mutaci přidání neuronu a mohou se do genomu dostat i při křížení. V některých implementacích existuje i varianta, která může také vypnout některý gen.

5.4 Speciation – rozdělení do druhů

Při strukturální mutaci vede změna v genomu často k momentálnímu zhoršení odezvy celé neuronové sítě a snížení fitness jedince. Jedince, jejichž genom je menší, je snazší optimalizovat a dosahují dříve vyšší fitness (méně hodnot k optimalizaci). Aby nebyly změny ztraceny při procesu selekce, jsou v NEATu jedinci rozděleni do několika druhů („*species*“) na základě jejich podobnosti. Evoluce probíhá v každém z druhů zvlášť, což umožňuje souběžnou evoluci genomů s rozdílnými topologiemi. Pokud je některý z jedinců mutací a křížením příliš pozměněn a příliš se liší od ostatních existujících druhů, je pro něj vytvořen nový druh. V kontextu evolučních algoritmů se jedná o *multimodální optimalizaci*.

Metriku pro měření rozdílnosti genomů poskytuje samo kódování NEATu. Díky faktu, že geny obsahují své ID, lze určit, které geny jsou shodné a které rozdílné. NEAT využívá tři faktory pro zjištění rozdílnosti genomů. Prvními dvěma je počet rozdílných genů. Ty se rozdělují na tzv. „*disjoint*“ a „*excess*“ geny. „*Excess*“ geny jsou všechny geny jednoho z jedinců, které mají větší ID, než je největší ID v druhém genomu. Všechny ostatní geny bez páru v druhém jedinci jsou označovány jako „*disjoint*“ geny. Třetím faktorem je rozdílnost vah mezi jedinci.

NEAT měří příslušnost genomu do druhu pomocí hranice kompatibility δ_t („*compatibility treshold*“) a vzdálenosti dvou jedinců δ. Jedná se o lineární kombinaci výše zmíněných faktorů, kterou lze zapsat jako:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{5.1}$$

Kde N je délka většího z genomů použitá k normalizaci, E je počet „*excess*“ genů, D je počet „*disjoint*“ genů a $\overline{W}$ je průměrný absolutní rozdíl mezi váhami shodných genů spojení, které se vyskytují v obou genomech. Dále jsou použity tři koeficienty, c_1, c_2 a c_3, kterými je možné měnit vliv jednotlivých faktorů. Pokud jsou genomy příliš krátké ([25] zmiňuje hodnotu 20 genů), je možné N ponechat na hodnotě 1, čímž se podpoří rozdílnost genomů v počátečních fázích algoritmu.

Využití koeficientů lze demonstrovat na příkladě řešení úloh citlivých na změnu výstupu (např. „double pole balance“). Pro tyto případy může být výhodnější zvýšit koeficient c_3, čímž se začnou druhy více odlišovat na základě rozdílných vah. Při křížení se pak setkají jedinci s podobnými váhami, což umožní preciznější evoluci vah.

Pro určení, jestli je jedinec kompatibilní s druhem, lze využít výpočetně náročnější přístup, kdy je použita průměrná vzdálenost jedince od všech ostatních jedinců v druhu. Druhou možností je jednodušší přístup, kdy je měřena vzdálenost pouze od jednoho jedince – reprezentanta druhu.

5.4.1 Multimodalita a fitness sharing

Pro dosažení vyrovnaných subpopulací – druhů, je v NEATu využit přístup z *multimodální optimalizace* – tzv. *nichingu*. Ten používá upravenou fitness funkci pro multimodální úlohy „*explicit fitness sharing*“. Původní fitness jedince je značena f_i a upravená fitness $f_i^{'}$.

$$f_i^{'} = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{5.2}$$

Funkce sdílení $sh()$ je rovna 0 pokud platí $\delta(i,j) > \delta_t$, jinak je 1. V důsledku to znamená, že fitness jedince je podělena množstvím jedinců v jeho druhu. Každému z druhů je pak vypočítána jeho celková fitness na základě sumy fitness všech jedinců v daném druhu. Na jejím základě je určeno, kolik potomků z kterého druhu vznikne. Pomocí sdílení fitness jedinců v rámci druhu je zaručeno, že celková fitness druhu, a tím pádem i množství potomků, je závislá na kvalitě jedinců a nikoliv na jejich množství.

5.5 Nová generace

Na začátku každé nové generace dojde nejdříve k **ohodnocení nových potomků** a jejich **přidělení do druhů** („*speciation*“). K udržení druhů slouží jejich seřazený seznam. Jednotliví potomci jsou postupně testováni na příslušnost do druhů. Jedinec je přiřazen do prvního kompatibilního druhu (viz kap. 5.4).

Před vytvořením nových potomků dojde k **selekci**. Ta probíhá v rámci každého druhu zvlášť. Použita je elitistická selekce (viz kap. 2.2.1). **Počet potomků** z rodičů jednoho druhu je vypočítán podle poměru fitness druhu vůči celkové fitness všech druhů dohromady:

$$N_{off_i} = \frac{f_{spcs_i}}{\sum_{j=1}^{n} f_{spcs_j}} \cdot PopSize \tag{5.3}$$

Kde N_{off_i} je počet potomků druhu i, f_{spcs_i} je fitness druhu i a $PopSize$ je velikost celé populace. Nová generace je tvořena pouze potomky. Před rozřazením potomků do druhů jsou z nich nejdříve odstraněni rodiče.

Při studiu implementace algoritmu K. Stanleym bylo v kódech [23] objeveno množství dalších nezdokumentovaných úprav. Taktéž existuje celá řada různých implementací algoritmu, které upravují některé části a funkcionality původního algoritmu. Mezi nalezené nezdokumentované úpravy patří:

- delta-coding – Pokud nedojde po určitou dobu ke zlepšení fitness, dojde k tzv. delta kódování, kdy se reprodukují pouze dva nejlepší druhy. Oba produkují 50% nových potomků.
- ukradené děti – Nejlepší druh získává bonus k množství potomků a to na úkor nejslabšího druhu
- dynamická hranice δ_t – V průběhu evoluce může být zvolená hranice nevhodná, což vede k příliš malému nebo velkému množství druhů. Hranici δ_t lze v těchto případech posouvat.
- Zachování nejlepšího – Z každého druhu je překopírován nejlepší jedinec beze změny.
- Omezení maximální váhy spojení – Při mutaci vah existuje limit pro absolutní váhu spojení.

- Křížení průměrováním – Při křížení je místo výběru spojovacího genu jednoho z rodičů (a tím i jeho váhy) použita nová váha vzniklá zprůměrováním vah z obou spojení.

5.6 Vlastnosti

NEAT je komplexní systém pro neuroevoluci. Za použití přímého kódování a identifikace genů byl, narozdíl od jiných algoritmů (viz 4.1), schopný použít operátor křížení. Taktéž byl úspěšně použit pro řešení „*double pole balance*“ problému a to s i bez informace o rychlostech. NEAT byl v množství vyhodnocení fitness funkce potřebných k vyřešení této úlohy poražen až s příchodem metody EANT [11].

Na druhou stranu NEAT obsahuje množství různých parametrů, které potřebují být optimálně nastaveny. Taktéž použití přímého kódování limituje algoritmus pouze na menší sítě, neboť musí optimalizovat každé spojení zvlášť. Řešení omezení přímého kódování dalo vzniknout nové metodě využívající nepřímého kódování v kombinaci s algoritmem NEAT – HyperNEAT.

Kapitola 6

Implementace

Pro implementaci algoritmu NEAT byl zvolen programovací jazyk Java verze 8. Program je rozdělen na několik základních celků. Hlavní částí je balík `neat`, který obsahuje vše potřebné pro běh algoritmu a experimentů. Balík `neat` obsahuje:

1) balík `core` obsahující všechny kódy související s algoritmem NEAT.
2) balík `neuralNetwork` implementující neuronové sítě.
3) balík `visualizers` obsahující moduly určené k zobrazování generovaných sítí, či rozdělení druhů.
4) balík `problems` obsahující experimenty a řešené problémy.

Centrálním bodem celého programu je třída `neat.core.NEAT`, která umožňuje spuštění experimentů (nazývány také jako *problém*), změny v nastavení parametrů a obsahuje základní funkčnosti algoritmu NEAT. Po nastavení všech počátečních hodnot z ní lze spustit evoluci pomocí funkce `evolve()`. V konstruktoru třídy NEAT lze zvolit, jestli má mít algoritmus zapnuté vizualizace sítě a druhů.

6.1 Evoluce

Po spuštění evoluce je vygenerována **počáteční populace**. K tomu slouží počáteční genom, který poskytuje konkrétní implementace *problému* (viz kap. 6.3). Počátečním genomům jsou náhodně vygenerovány váhy v rozsahu daném parametrem `initialWeightRange`. Jednotlivá ID genů jsou pro všechny spojení i neurony u počátečních genomů shodná.

Následuje samotná evoluce populace, kdy je po nastavený počet generací, či do vyřešení problému volána funkce `nextGeneration()`, která obstarává vznik nové generace pomocí mechanismů algoritmu NEAT a genetických operátorů.

Na začátku je **vyhodnocena** celá populace pomocí metody `evaluatePopulation()`. Ta kromě evaluace jednotlivých genomů také udržuje hodnotu maximálního fitness a kopii nejlepšího jedince. Z důvodů speciálních úprav algoritmu umožňujících vyhodnocování genomů i na jiných místech obsahuje každý genom informaci o tom, jestli již byl vyhodnocen. Tím se lze vyhnout opakovanému vyhodnocování genomů.

Po vyhodnocení populace je celá populace seřazena sestupně podle hodnoty fitness. Cílem je, aby nové druhy vytvářely vždy jedinci s co nejvyšším fitness. Seřazení jedinci jsou následně **rozděleni do druhů** (viz kap. 5.4). Rozdělení do druhů podle článku [25] je implementováno v metodě `neatSpeciation()`.

Následně dochází v každém z druhů zvlášť k **selekci**, kdy v seznamu aktivních jedinců zůstane pouze část jedinců druhu. Jejich množství je určeno parametrem *selectionPressure* udávajícím velikost části populace k přežití (hodnota 0,2 značí, že přežívá 20% jedinců). Také druhy, které příliš dlouhou dobu nezlepšili svou fitness (dáno parametrem *stagnateSpecLimit*) nebo neobsahují žádné jedince jsou vymazány.

Po provedení selekce, rozdělení do druhů a jejich pročištění přechází algoritmus do fáze křížení a mutací. Nejdříve je spočítána celková fitness všech druhů. K tomu slouží funkce `getFitness()` každého druhu, která vrací již upravenou verzi fitness (viz kap. 5.4 a rovnice 5.1). Poté je pro každý z druhů spočítán cílový počet potomků a dochází ke vzniků nových jedinců. U druhů, jejichž velikost přesahuje pět jedinců nebo obsahují šampiona populace, je kopírován nejlepší jedinec do další generace. Pokud druh obsahuje pouze jednoho jedince, je do další generace pouze kopírován. Při více jedincích jsou vybráni dva různí a ti podstoupí křížení, ke kterému slouží metoda `crossOver(Genome g1, Genome g2)`.

Při **křížení** existuje šance na vytvoření asexuálního potomka, který vznikne pouze z jednoho rodiče. Tato šance je dána doplňkem k parametru pravděpodobnosti křížení *xRate*. Tento potomek má poté 100% šanci, že bude mutován. Snížením parametru *xRate* proto nedochází ke snížení šance na mutaci, ale právě naopak k navýšení.

Při sexuálním křížení jsou geny nejdříve seřazeny a procházeny od konce. Také je vytvořen prázdný návratový genom. K přidávání genu slouží metoda trídy Genome `addGene()`. Ta vytvoří kopii genu a provede všechny potřebné úpravy v genomu (viz kap. 6.2.3). Pokud mají dva geny shodné ID a některý z genů je vypnutý, je zvýšena šance na jeho přidání o 25% (hodnota použitá v článku [25]). Dále je s pravděpodobností danou parametrem *avgMutRate* provedeno průměrování vah v případě genů spojení.

Každý potomek má navíc šanci na mutace danou doplňkem k parametru *nonMutatingMating*. Asexuální potomci mají šanci 100%. K oddělení a lepší správě mutací byla vytvořena třída `neat.core.Mutations`, která obsahuje všechny rozdílné metody mutace. Ty jsou tedy odděleny od samotného evolučního cyklu. Ve třídě `neat.core.NEAT` je pak metoda `mutate(Genome g)`. Ta obsahuje provedení jednotlivých druhů mutací podle přidělených pravděpodobností a také umožňuje lepší kontrolu nad jejich provedením. **Strukturální mutace** („*add node*“ a „*add connection*“) mají výlučné provedení (jedna strukturální mutace naruší funkčnost sítě dostatečně) s předností pro přidání neuronu. **Parametrických mutací** může proběhnout více naráz.

Ve třídě `neat.core.NEAT` existují dva seznamy jedinců. První udržuje aktivní jedince a druhý udržuje nově vytvořené potomky. Po ukončení fáze křížení a mutací je reference na seznam s potomky přiřazena seznamu aktivních jedinců a pro potomky je vytvořen nový prázdný seznam. Nakonec jsou aktualizovány všechny druhy (pomocí jejich metody `reset()`) a jsou vymazány druhy, které překročí maximální povolený věk bez zlepšení daný parametrem *stagnateSpecLimit*.

6.2 Použité struktury

6.2.1 Druhy

Druh je implementován jako třída `neat.core.Species`. Ta obsahuje dva seznamy genomů. Jeden obsahuje aktivní jedince a druhý odumřelé (ty, které neprošli selekcí). Uchovávat odumřelé jedince je potřeba kvůli výpočtu fitness, ve kterém jsou zohledněni všechni jedinci. Přesunout genom do seznamu odumřelých lze metodou `kill()`. Kromě toho také udržuje informace o druhu, jako jsou věk, počet generací od posledního zlepšení, reprezentanta druhu,

či fitness druhu. Pomocí funkce `reset()` se druh na konci generace pročistí a aktualizuje své statistiky.

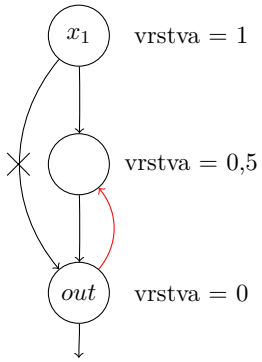
6.2.2 Geny

V algoritmu NEAT existují dva druhy genů – spojovací geny a geny neuronů. V programu je vytvořena třída `neat.core.Gene`, která zahrnuje společné vlastnosti všech genů, včetně statického čítače inovací (*static int innovation*), podle kterého je genům přidělováno ID. Součástí je také výčtový typ určující typ genu (`GeneType`), řetězec, který se vypisuje při zobrazení sítě či proměnná určující, jestli je gen aktivní (`boolean enabled`). Třída dále obsahuje implementaci rozhraní `Comparable<Gene>` pro porovnání genů podle jejich ID.

Spojovací gen implementovaný v třídě `neat.core.ConnectionGene` rozšiřuje třídu `neat.core.Gene`. Spojovací gen navíc obsahuje informaci o výchozím a cílovém neuronu a o váze spojení.

Gen neuronu je implementovaný ve třídě `neat.core.NodeGene` a také rozšiřuje třídu `neat.core.Gene`. Gen neuronu je dále určen svým typem (`NodeType` – udává, jestli se jedná o vstupní, výstupní, či skrytý neuron), aktivační funkcí a hodnotou udávající vrstvu při vynucení dopřednosti sítě.

Aktivační funkce je implemntována jako rozhraní, které deklaruje metodu `float evaluate(float in)`, která vypočítá výstupní hodnotu neuronu. Tímto způsobem je možné dynamicky měnit aktivační funkce jednotlivých neuronů a rozšiřuje to možnost dalších pokusů s programem.



Obrázek 6.1: Ukázka vynucení dopřednosti sítě při vkládání neuronu do spojení mezi x_1 a *out*. Nový neuron získá hodnotu vrstvy jako průměr vrstev původně spojených neuronů. Vrstva nového neuronu je tedy $(1+0)/2 = 0,5$. Červené spojení nelze přidat, neboť nevede z vrstvy s vyšší hodnotou do vrstvy s nižší hodnotou.

Dopřednost sítě lze zaručit přístupem, kdy je vstupním neuronům určena hodnota vrstvy 1 a výstupním 0. Při vložení neuronu do některého spojení je jeho vrstva nastavena na průměr z hodnot vrstev neuronů, které byly spojeny. Při vytváření nového spojení jsou poté přijata pouze ta spojení, která vedou z vyšší vrstvy do nižší, viz obrázek 6.1.

6.2.3 Genom

Genom, reprezentovaný třídou `neat.core.Genome`, obsahuje všechny informace o jedinci. Používá tři seznamy s geny. Prvním jsou spojovací geny, druhým jsou geny neuronů a

třetím jsou všechny geny dohromady. Přestože dochází k redundanci dat, je to výhodné při manipulacích s genomem, kdy je potřeba pracovat pouze s určitým druhem genů.

Genom také obsahuje **informace** o tom, jestli byl vyhodnocen (čímž lze zabránit vícenásobnému vyhodnocení jedince), informaci o vzniku (sexuální / asexuální) a historii úprav genomu. Historie úprav je vyjádřena řetězcem, do kterého lze libovolně zapisovat. Při provedení některé úpravy (křížení, mutace) jsou do řetězce přidány informace o úpravě a případně jejích detailech.

K **manipulaci** s genomem slouží několik funkcí. Pro vkládání genů při křížení slouží funkce `addGene(Gene g)`, která rozpozná druh, udělá jeho kopii a aktualizuje seznamy s geny. Při mutaci vložení neuronu je naopak použita funkce `addNodeBias(NodeGene g)`, jejímž cílem je kromě vložení neuronu i vytvoření spojovacího genu se vstupním bias neuronem.

Pro ulehčení vytváření **počátečního genomu** existují ještě funkce `randomConnect()` a `fullyConnect()`, které vygenerují buď náhodné spojení mezi existujícími neurony nebo vytvoří plné propojení všech vstupů se všemi výstupy.

6.3 Vkládání řešeného problému

Ke **vkládání řešeného problému** do algoritmu NEAT slouží rozhraní `Problem` v balíku `problems`. Toto rozhraní zajišťuje základní funkce potřebné pro běh algoritmu NEAT. Mezi ty patří především metoda `void evaluate(Genome g)`, která vloží do genomu g jeho hodnotu fitness. Dále rozhraní obsahuje řadu dalších metod, jako jsou například metody k dotazům na vyřešenost problému (`boolean solved()`), na možnost resetovat experiment (`void reset()`), pro získání počátečního genomu k řešení (`Genome startingGenome()`), získání nastavení specifických pro experiment (`NEATSettings getSettings(NEATSettings s)`) nebo metodu ke zjištění množství provedených evaluací (`int getEvaluations()`).

Poté, co je vytvořena instance třídy implementující toto rozhraní, je možné ji vložit do algoritmu NEAT pomocí metody `setProblem()`. Tato metoda si uloží zadaný problém a zároveň z něj získá pomocí metody `getSettings(...)` parametry experimentu. Podle získaných parametrů je navíc vytvořena instance třídy s mutacemi danými získaným nastavením.

Při programování problému je potřeba si ručně vytvářet počáteční genom. Ten by bylo možné vytvářet i automaticky podle počtu vstupů a výstupů, nicméně tento přístup umožňuje mnohem větší volnost v plánování počáteční topologie (lze vytvořit libovolnou topologii). Třída `neat.core.Genome` navíc obsahuje některé funkce k automatickému dokončování topologie.

6.4 Neuronové sítě

V rámci projektu byla vytvořena také implementace neuronových sítí ve třídě `neat.neural-Network.NN`. Tato třída má implementovány konstruktory kompatibilitní s třídou `Genome`. Ty umožňují vytvoření neuronové sítě přímo předáním genomu.

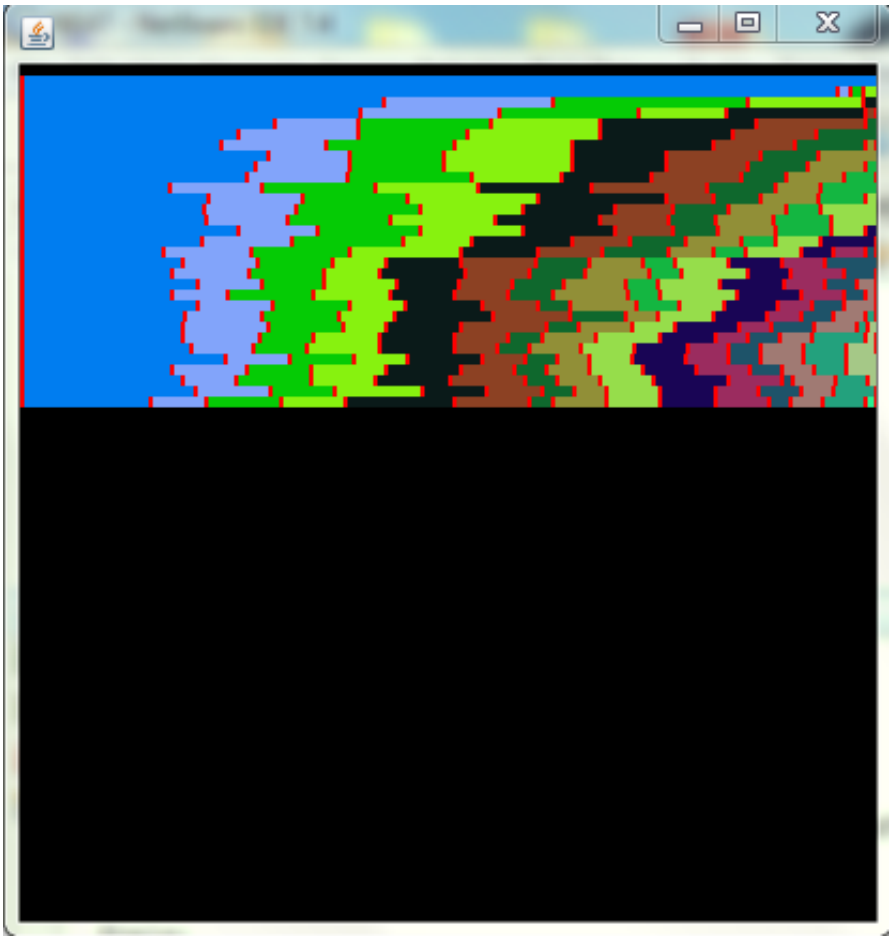
Neuronová síť je vyhodnocována po jednotlivých krocích. V každém kroku jsou nejdříve vypočítány výstupy každého z neuronů. Poté co jsou nové výstupy vypočítány, jsou propagovány dále přes spojení do další neuronů. Při dalším opakování aktivace sítě jsou použity nově vypočítané výstupy. K jedné aktivaci sítě lze použít funkce `ArrayList<Float> nextOutpusts(ArrayList<Float> inputs)`, která má parametrem seznam vstupních hod-

not (které jsou namapovány na vstupní neurony) a výstupem je seznam hodnot výstupních neuronů.

Pokud je potřeba jednu síť vyhodnotit pro více různých vstupních hodnot, je možné ji pouze resetovat pomocí funkce `reset()` do výchozího stavu. Tím lze ušetřit čas potřebný k novému vytváření síťe z genomu.

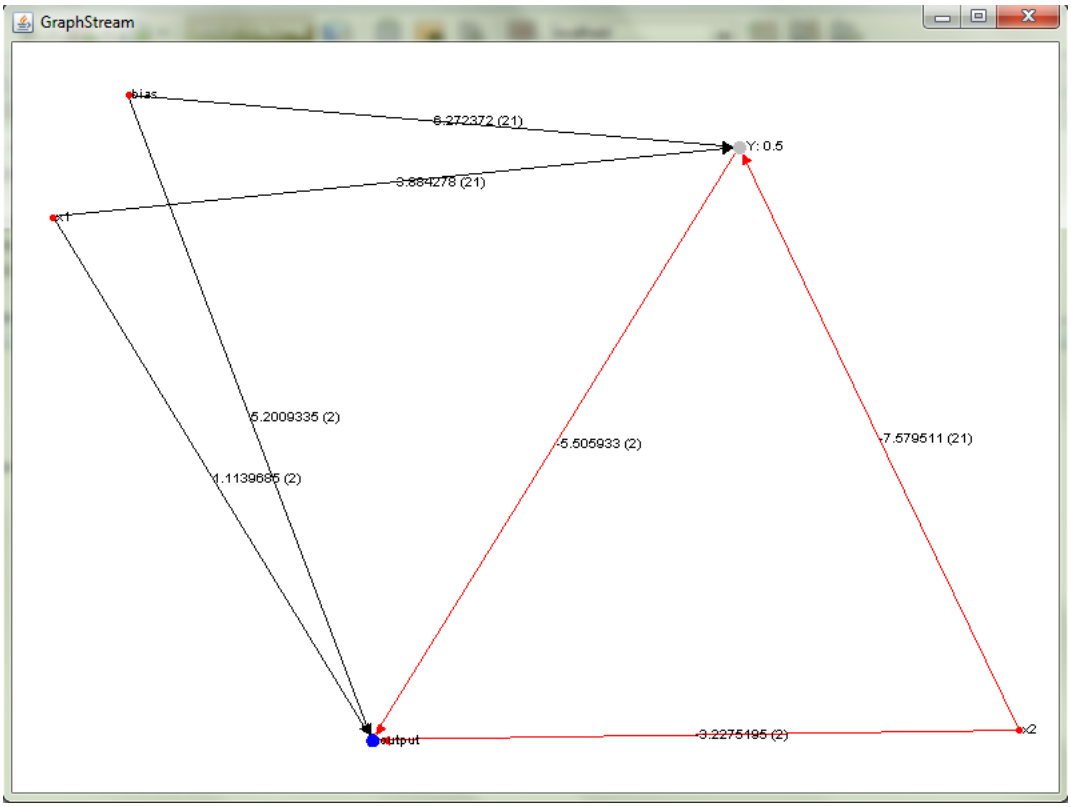
6.5 Grafické výstupy

Pro snažší ověření funkčnosti algoritmu a jeho průběh byly vytvořeny dvě pomůcky s grafickým výstupem. První pomůckou je okno, ve kterém jsou zobrazeny druhy a poměr jejich velikostí v populaci (obr. 6.2). Každý druh je zobrazen v odlišné barvě přiřazené na základě ID druhu. Jeden řádek odpovídá jedné generaci.



Obrázek 6.2: Vizualizér druhů. V druhé generaci vzniklo několik nových druhů. V dalších generacích se tyto druhy již rozrostly a navíc vznikly i jiné.

Druhou pomůckou je okno zobrazující neuronovou síť (obr. 6.3) vytvořenou z aktuálně nejlépe ohodnoceného jedince. K zobrazení sítě je použita knihovna GraphStream[1]. Knihovna není primárně určena k vizualizaci neuronových sítí. Vizualizér neuronové sítě slouží pouze jako pomůcka k lehčímu představení si vygenerované sítě.



Obrázek 6.3: Vizualizér neuronové sítě. Zobrazuje aktuálně nejlépe ohodnoceného jedince. Body značí jednotlivé neurony, kde červená barva značí vstup, modrá barva značí výstup, šedá barva značí skrytý neuron a černá barva značí rekurentní spojení. Vstupy a výstupy jsou opatřeny popisky. Spojení jsou opatřena číslem značícím váhu spojení.

[1]Knihovna GraphStream je dostupná na adrese http://graphstream-project.org/.

Kapitola 7

Experimenty s algoritmem NEAT

7.1 XOR

Pro počáteční ověření funkčnosti algoritmu NEAT lze využít logickou funkci XOR. Ta je díky své vlastnosti lineární neseparovatelnosti vhodná k experimentům. Důvodem je, že si síť musí umět vypočítat mezistav. V závislosti na vyhodnocení sítě musí vzniknout alespoň jeden skrytý neuron nebo rekurentní vazba.

Experimenty jsou rozděleny do dvou skupin. V první je vynucená dopřednost sítě (bez rekurentních spojení) a v druhém jsou povoleny i rekurentní spojení. Parametry použité k experimentu jsou zvoleny stejné, jako v článku [25]:

- Velikost populace N: 150
- Max. počet generací: 300
- Max. váha spojení: [-8,8]
- Aktivační funkce: $f(x) = \frac{1}{1+e^{-4,9 \cdot x}}$
- Počáteční síť: plně propojená bez skrytých neuronů
- Počáteční rozsah vah: [-0,1;0,1]
- Pst. (pravděpodobnost) křížení: 50%
- Pst. křížení bez mutací: 20%
- Pst. přidání neuronu: 3%
- Pst. přidání spojení: 30%
- Pst. křížení průměrem: 40%
- Síla mutací: 3,4
- Rozsah nově generovaných vah: [-2,2]
- Mezidruhová hranice δ_t: 2,0
- Procento rodičů: 40%
- Dynamická hranice δ_t: aktivní

Při aktivní dynamické hranici byla každou pátou generaci upravena mezidruhová hranice δ_t o 1%. Pokud existovalo příliš mnoho (10 a více) nebo příliš málo (5 a méně) druhů. Dále bylo při dopředné variantě experimentu použito šest aktivací sítě, což by měl být dostačující

počet i pro největší vzniklé sítě. Pro rekurentní sítě byly použity tři aktivace, které umožní snažší využití rekurentních spojení. Fitness funkce byla počítána podle rovnice níže:

$$f(x) = 4 - \sum_{i=1}^{4} |Y_i - y_i| \tag{7.1}$$

Kde Y_i značí očekávaný výstup a y_i reálný výstup sítě v případě vstupních hodnot vzorku i. Používá se i umocněná verze rovnice. Ta se však neukázala výhodnější. Výsledná síť byla považována za naučenou, když se žádný z výstupů nelišil od očekávané hodnoty o více než $0,3$. Každý z experimentů sestával ze 100 běhů. Výsledky lze vidět v tabulce 7.1 a ukázky vygenerovaných sítí na obrázku 7.1.

Typ topologie	Generace		Evaluace		Spojení	
	Průměr	Směr. odchylka	Průměr	S. odchylka	Průměr	S. odchylka
rekurentní	33,46	14,87	5260,60	2278,66	10,37	2,75
dopředná	40,73	19,28	6383,55	2960,62	9,86	3,00

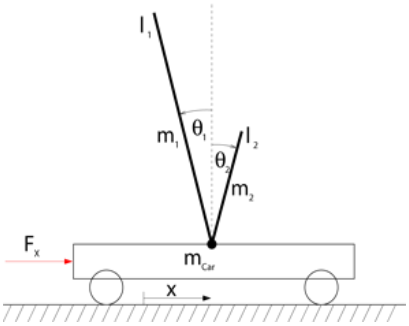
Tabulka 7.1: Výsledky experimentů s XOR pro rekurentní i dopředné topologie. Hodnoty v tabulce vychází ze 100 pokusů pro každý typ topologie.

Obrázek 7.1: Vygenerované sítě řešící XOR. Vlevo je rekurentní síť řešící XOR a vpravo je dopředná síť řešící XOR. Obě sítě využívají jeden skrytý neuron.

Ze souhrnných 200 běhů bylo řešení nalezeno vždy. Průměrná generace nalezení řešení je o něco vyšší, než je udávána v článku [25] (32 generací). Zajímavým faktem je, že při povolení rekurentních spojení byl vývoj rychlejší.

7.2 Pole balancing – balancování tyčí

Klasickým benchmarkovým testem pro rekurentní neuronové sítě vytvořené pomocí neuroevolučních algoritmů je takzvané balancování tyčí („*pole balancing*“) definované v článku [5]. Tato úloha má za cíl balancovat dvě tyče s různou délkou (existuje i jednodušší varianta pouze s jednou tyčí), které jsou připevněny k pohyblivému vozíku. Vozík se smí hýbat pouze v jedné ose a navíc nesmí překročit předem dané hranice (musí zvládnout balancování na omezeném prostoru). Neuronová síť rozhoduje o síle, která má být na vozík aplikována. Tyč je považována za vybalancovanou, pokud je její úhel náklonu v rozmezí [-36°,36°].



Obrázek 7.2: Ukázka problému balancování dvou tyčí.

Vstupy neuronové sítě jsou:

1. pozice vozíku - normalizovaná na rozsah [-1,1] mezi krajními pozicemi
2. rychlost vozíku v m/s
3. úhel θ_1 první tyče - normalizovaný na rozsah [-1,1] mezi krajnímy úhly
4. rychlost změny úhlu θ_1 v rad/s
5. úhel θ_2 druhé tyče - normalizovaný na rozsah [-1,1] mezi krajnímy úhly
6. rychlost změny úhlu θ_2 v rad/s

Existují dvě verze problému. První je jednodušší a obsahuje všechny výše uvedené vstupy. Druhá, složitější, neobsahuje informaci o rychlostech (body 2, 4 a 6). Aby bylo možné tyče vyrovnat, musí být síť schopná rychlosti sama odvozovat. Pro tuto verzi se používá zkratky *DPNV* (double pole no velocities).

Výjimkou může být situace, kdy se síť naučí vyrovnávat tyče pomocí rychlého kývavého pohybu ze strany na stranu. Aby byla síť donucena používat k vyvažování i rychlosti, byla vytvořena speciální fitness funkce penalizující oscilace pohybu (rovnice 7.4).

$$f_1 = t/1000 \tag{7.2}$$

$$f_2 = \begin{cases} 0 & pro \quad t < 100 \\ \frac{0.75}{\sum_{i=t-100}^{t}(|x^i|+|\dot{x}^i|+|\theta_1^i|+|\dot{\theta}_1^i|)} & jinak \end{cases} \tag{7.3}$$

$$f = 0.1 \cdot f_1 + 0.9 \cdot f_2 \tag{7.4}$$

Kde t odpovídá kroku simulace, x^i vzdálenosti vozíku od středu, $\dot{x}^i$ rychlosti vozíku, θ_1^i velikosti úhlu velké tyče a $\dot{\theta}_1^i$ rychlosti náklonu velké tyče. Tato rovnice je definována vždy pro 1000 kroků. Do výsledné hodnoty se započítávají i odchylky od středu a rychlosti. Jejich minimalizací lze zlepšit hodnotu fitness a tím pádem penalizovat vyrovnávání kmitavým pohybem.

Počáteční podmínky simulace byly použity shodně s [24]. Délka trasy, na které lze balancovat je 4.8m. Kratší tyč měří 0.1m a míří kolmo vzhůru. Delší z tyčí měří 1.0m a je nakloněná pod úhlem 1°. Délka kroku simulace je 0.01s a cílová doba simulace je 100 000 kroků. Síť je použita při každém kroku simulace a je provedena vždy jedna aktivace. Výstup sítě je upraven do rozsahu [-10,10] a odpovídá síle aplikované na vozík v Newtonech. K výpočtu simulace je použit algoritmus Runge-Kutta 4. řádu.

7.2.1 Experimenty

Parametry algoritmu jsou – obvzláště u algoritmu NEAT – zásadní pro výsledné chování a výsledky. Parametry použité pro experiment jsou shodné s parametry použitými v článku [25]. Oproti článku však bylo zjištěno zlepšení výsledků při snížení síly mutací. Hodnot síly mutací bylo testováno více:

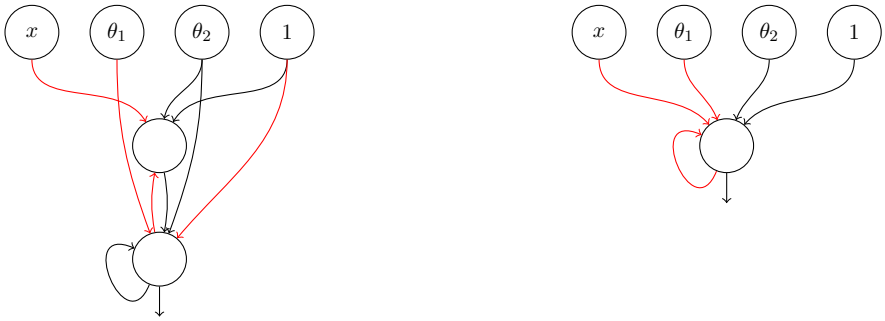
- Velikost populace N: 1000
- Max. počet generací: 300
- Max. váha spojení: [-8,8]
- Aktivační funkce: $f(x) = \frac{1}{1+e^{-4,9 \cdot x}}$
- Počáteční síť: plně propojená bez skrytých neuronů
- Počáteční rozsah vah: [-0,01;0,01]
- Pst. (pravděpodobnost) křížení: 75%
- Pst. křížení bez mutací: 80%
- Pst. přidání neuronu: 3%
- Pst. přidání spojení: 30%
- Pst. křížení průměrem: 40%
- Síla mutací: 0,75 / 1,0 / 1,3 / 1,8
- Rozsah nově generovaných vah: [-2,2]
- Mezidruhová hranice δ: 4,0
- $c_1 = 1, \quad c_2 = 1, \quad c_3 = 3$
- Procento rodičů: 20%

Experiment byl pro každou rozdílnou sílu mutací spuštěn padesátkrát. Výjimečně nastala situace, kdy nebylo nalezeno řešení (z 200 běhů nebylo nalezeno řešení ve 3 případech). V případech, kdy řešení nebylo nalezeno, došlo během prvních generací ke slibnému nárustu fitness. Ten se ovšem zastavil v lokálním extrému, ze kterého se algoritmus nedokázal dostat. Výsledky ze všech experimentů lze vidět níže:

Síla mutací	Generace		Evaluace		Spojení	
	Průměr	Směr. odchylka	Průměr	S. odchylka	Průměr	S. odchylka
0,75	24,693	22,858	25321,55	22424,62	5,285	1,195
1,00	22,000	16,972	22392,46	15710,51	5,160	0,880
1,30	26,244	19,181	25729,75	17573,92	5,408	1,442
1,80	30,551	37,437	28397,00	31735,94	5,571	1,917

Tabulka 7.2: Výsledky experimentů s DPNV pro různé hodnoty síly mutací. Hodnoty vychází z 50 pokusů pro každou z hodnot mutací.

Zajímavým faktem, který lze pozorovat v tabulce 7.2, je, že průměrný počet generací (a tedy i evaluací fitness funkce) je výhodnější pro slabší sílu mutací, než jaká je udávána v článku [25]. To může být způsobeno implementačními rozdíly v nezdokumentovaných úpravách algoritmu.



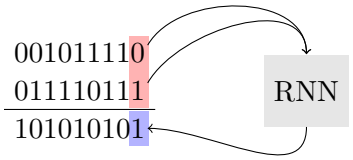
Obrázek 7.3: Vygenerované sítě řešící DPNV. Síť vlevo řeší DPNV s jedním skrytým neuronem. Síť vpravo představuje minimální topologii schopnou DPNV řešit.

Nejčastěji vyvýjenou topologií byla minimální možná topologie pro vyřešení DPNV obsahující jedno rekurentní spojení výstupního neuronu se sebou samým (viz obr. 7.3 vpravo). Toto řešení ukazuje snahu algoritmu NEAT hledat minimální řešení. Na obrázku 7.3 vlevo pak lze vidět komplikovanější síť využívající skrytý neuron a zpětnou vazbu do něj vedoucí z výstupního neuronu.

7.3 Binární sčítačka

Dalším z testovacích problémů byla binární sčítačka. K vytvoření binární sčítačky lze použít i dopředné sítě, které přijímají binární čísla k sečtení jako vstupní vektor a výsledkem je výstupní vektor. Takové sítě však vedou na některá omezení. Především se jedná o pevně daný počet vstupních bitů (velikost vstupního vektoru).

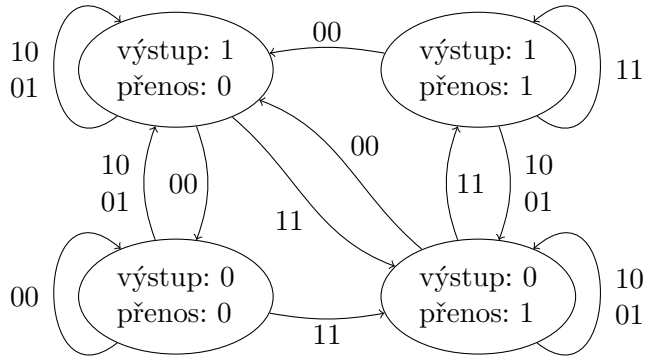
Oproti tomu řeší rekurentní binární sčítačky tento problém způsobem, kdy na vstupu jsou pouze dva bity určené k sečtení a výstupem je výsledek jejich součtu (viz obr. 7.4). Síť je aktivována opakovaně pro každou dvojici bitů. Ty vstupují do sítě postupně a síť si musí mezi jednotlivými vstupy pamatovat, jestli nedošlo k přenosu a pokud došlo, tak jej musí umět započítat. Proud vstupních bitů může být prakticky nekonečný a správně naučená síť tudíž zvládá sčítat libovolně dlouhá binární čísla.



Obrázek 7.4: Rekurentní binární sčítačka. Síť má na vstupu dvojici bitů k sečtení. Výstupem je výsledný bit. Síť musí být schopná uchovávat informace o přenosech (carry) a umět je správně přičítat k mezivýsledkům.

Problém balancování tyčí k vyřešení potřebuje správné vyladění vah a schopnost si spočítat diference mezi následujícími vstupními hodnotami. Binární sčítačka však ověřuje některé jiné vlastnosti rekurentní neuronové sítě. Ta musí být schopná nejen rozlišit čtyři stavy v odpovídajícím konečném automatu binární sčítačky (viz obr. 7.5) a správně mezi

nimi přecházet, ale zároveň musí být schopná si pamatovat přenos po teoreticky neomezený počet kroků.



Obrázek 7.5: Graf přechodů mezi stavy binární sčítačky. Neuronová síť musí být schopná se tyto vztahy naučit.

Dalším potenciálním problémem je nespojitost fitness funkce, kdy jsou zlepšení často skoková s tím, jak se síť naučí některou z dalších zákonitostí při sčítání dvou binárních čísel.

7.3.1 Experimenty

Před začátkem experimentu jsou sčítaná čísla nejdříve upravena na shodnou délku, kdy jsou v případě kratšího čísla doplněny na jeho začátek nuly. Neuronová síť počítající součet poté dostane dvojici bitů (z každého čísla jeden) počínaje nejméně významným bitem. Síť je poté dvakrát aktivována a zaokrouhlená výstupní hodnota je použit jako výstup. Tyto číslice jsou poté zkonkatenovány do binárního řetězce a porovnány s očekávanou hodnotou. Za každá shodný bit je fitness zvětšena o pozici, na které došlo ke shodě (shoda na významějším bitu je ohodnocena více než shoda na méně významném bitu). Sčítání je testováno na kombinaci součtů deseti menších čísel (0_2 – 1001_2) a součtu jednoho velkého čísla (11110001001000000_2). Výsledná hodnota fitness funkce je navíc umocněna.

Parametry použité k experimentům byly do jisté míry podobné, jako u DPNV, kdy je dáván větší důraz na rozlišnost druhů na základě vah. Jako zcela nevýhodné se ukázalo použití křížení průměrováním a bylo proto sníženo na 5%. Také byla snížena pravděpodobnost přidání spojení s cílem soustředit mutace na úpravy vah. Síla mutací vah byla podobně úspěšná v rozsahu 1,5 až 2,0. Počet generací byl zvýšen, aby byl schopen pokrýt větší výkyvy v rychlosti řešení. Souhrn zvolených parametrů je možné vidět níže:

- Velikost populace N: 1000
- Max. počet generací: 500
- Max. váha spojení: [-8,8]
- Aktivační funkce: $f(x) = \frac{1}{1+e^{-4,9 \cdot x}}$
- Počáteční síť: plně propojená bez skrytých neuronů
- Počáteční rozsah vah: [-2;2]

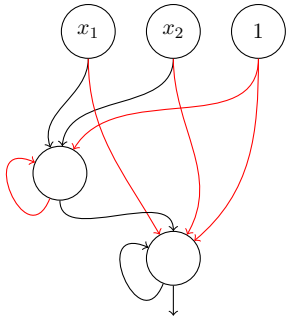
- Pst. (pravděpodobnost) křížení: 75%
- Pst. křížení bez mutací: 40%
- Pst. přidání neuronu: 3%
- Pst. přidání spojení: 20%
- Pst. křížení průměrem: 5%
- Síla mutací: 1,7
- Rozložení náhodných čísel: uniformní
- Rozsah nově generovaných vah: [-2,2]
- Mezidruhová hranice δ: dynamická
- $c_1 = 1, \quad c_2 = 1, \quad c_3 = 3$
- Procento rodičů: 20%

K experimentu bylo použito 50 běhů. Výsledky lze vidět v tabulce níže:

Populace	Generace		Evaluace		Úspěšnost
	Průměr	Směr. odchylka	Průměr	Směr. odchylka	
1000	209,5	100,4	211926,17	101033,57	80%
	Neurony		Spojení		
	Průměr	Stř. odchylka	Průměr	Stř. odchylka	
	6,42	0,9	16,75	5,01	

Tabulka 7.3: Výsledky experimentu s binární sčítačkou. Lze si povšimnout výrazně náročnější evoluce, než je u problému DPNV. Výrazným prvkem je také střední odchylka generací a evaluací, která dosahuje hodnot okolo 50% průměru. Také se již nepodařilo dosáhnout 100% úspěšnosti, která dosáhla hodnoty 80%.

Jak je z tabulky 7.3 vidět, je problém binární sčítačky výrazně náročnější, než DPNV. Především zajímavým faktem je výrazná hodnota střední odchylky generací i evaluací dosahující hodnot okolo 50% z průměru. Tento fakt by mohl souviset s obtížností vyváznout z lokálních extrémů.



Obrázek 7.6: Nejmenší vygenerovaná síť pro binární sčítačku. Skrytý neuron slouží k „ukládání“ přenosu. Dynamika přechodu mezi stavy je silně založena na dvou krocích – síť efektivně využívá dvě aktivace mezi jednotlivými vstupy.

Vygenerovaná síť při přechodu využívá **dvě úrovně vyladění vah** zároveň. Na **první úrovni** není důležité váhy precizně vyladit, jako spíše mít správné poměry. Například, dvě 1 na vstupu zaktivují neuron ukládající přenos tím, že mají společné větší váhu, než má záporné spojení z biasu. Zatímco při vstupu pouze jedné 1 převáží negativní hodnota z biasu a neuron zůstává neaktivní.

Druhou úrovní je pak přesné vyladění vah používané pro rozlišení stavů. Příklad takovéhoto vyladění je náročnější a postrádá názornosti, neboť zahrnuje dynamiku přechodů mezi stavy přes více aktivací sítě. Zjednodušeně řečeno se však jedná například o situace, kdy jedna kombinace vstupů neuronu musí mít v různých situacích hodnotu v mezích mezi druhou a třetí kombinací vstupů. Tyto kombinace musí být funkční na první úrovni a poté ještě správně vyladěny na druhé. Kombinace těchto dvou úrovní může také stát za častější stagnací v lokálních extrémech.

Kromě nejmenší topologie (viz obr. 7.6) algoritmus často vytvářel i sítě se dvěma, či třemi skrytými neurony. To lze dobře vidět v tabulce 7.3 v položkách *Neurony* a *Spojení*.

Kapitola 8

Diferenciální evoluce

8.1 Úvod

Diferenciální evoluce (dále také DE) byla představena v roce 1996 Rainerem Stornem a Kennethem Pricem [27]. Jedná se o populační algoritmus, který pracuje s množinou řešení podobně jako tomu je u genetických algoritmů. Algoritmus umožňuje optimalizaci nelineárního spojitého prostoru. V článku je navíc poukázáno na vyšší úspěšnost i odolnost oproti ostatním optimalizačním metodám.

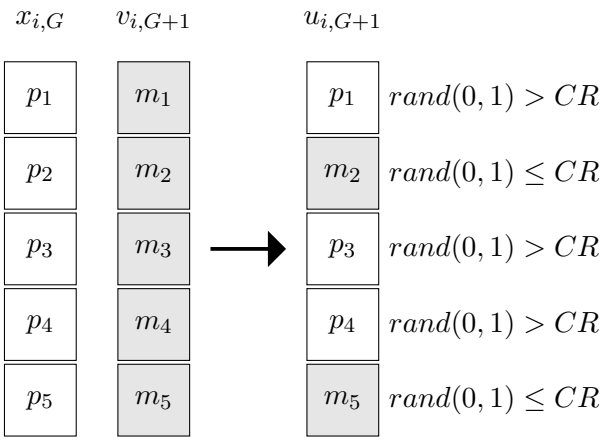
Diferenciální evoluce je založena na principu změn parametrů podle rozdílnosti ostatních jedinců v populaci. Velikost změn je tak závislá na rozdílnosti jedinců v populaci. Tento přístup je rozdílný oproti genetickým algoritmům či simulovanému žíhání. Každý z jedinců v populaci je definován D-dimenzionálním vektorem (kde D je počet parametrů). Populace je tvořena NP jedinci $x_{i,G}$ pro $i = 1, 2..NP$ a kde G značí aktuální generaci.

8.2 Průběh

Na začátku algoritmu je vygenerováno NP jedinců. Hodnoty parametrů by měly být co nejlépe rozprostřeny přes stavový prostor. Algoritmus používá ke změně parametrů takzvaný mutující vektor „*mutant vector*“. Ten vzniká jako váhovaná diference mezi dvěma vektory přičtená ke třetímu vektoru:

$$v_{i,G+1} = x_{r1,G} + F \cdot (x_{r2,G} - x_{r3,G}) \tag{8.1}$$

Kde $r1 \neq r2 \neq r3$ jsou indexy tří rozdílných jedinců, $v_{i,G+1}$ je mutující vektor pro jedince i v generaci G a F je faktor váhování rozdílu a patří mezi volené parametry algoritmu. Jedná se konstatní hodnotu v rozsahu $(0, 2]$.



Obrázek 8.1: Křížení mezi mutujícím vektorem a jedním z vektorů populace. S pravděpodobností CR je použita hodnota z mutujícího vektoru. Vzniklý vektor nahradí vektor $x_{i,G}$ pouze v případě, že má lepší fitness.

Pomocí mutujícího vektoru poté vznikají nová řešení křížením s některým z existujících jedinců. Ke křížení se vztahuje parametr CR, který udává s jakou pravděpodobností bude parametr kříženého jedince nahrazen parametr z mutujícího vektoru (viz obr. 8.1).

Vznik nové generace probíhá vždy tím způsobem, že každý jedinec v populaci podstoupí křížení s mutujícím vektorem. Pokud je nově vzniklý jedinec ohodnocen lépe, nahrazuje svého rodiče. V opačném případě zůstává rodič v populaci.

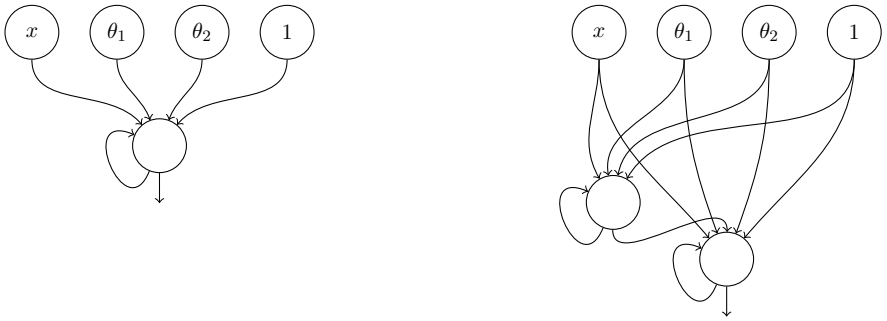
Existuje ovšem více různých variant tohoto algoritmu. Varianty se popisují kódovým označením $DE/x/y/z$, kde x značí vektor, který má být mutován ($x_{r1,G}$ v rovnici 8.1), y značí počet diferencí vektorů použitých k výpočtu mutujícího vektoru a z značí použité schéma křížení.

Výše popisovaná verze je označována jako $DE/rand/1/bin$. Vektor $x_{r1,G}$ je vybírán náhodně a je k němu přičítána váhovaná diference jedné dvojice jedinců. Varianta „*bin*“ popisuje dříve popsané schéma křížení. V článku [27] je zmíněna ještě druhá úspěšná varianta – $DE/best/2/bin$. Vektor $x_{r1,G}$ je vždy nejlépe ohodnocený jedinec populace a přičítá se k němu váhovaná diference ze dvou dvojic vektorů, jak je vidět v rovnici 8.2. V článku [27] dodávají, že využitím diferencí ze dvou dvojic se zdá být zvýšena diverzita populace.

$$v_{i,G+1} = x_{best,G} + F \cdot (x_{r1,G} + x_{r2,G} - x_{r3,G} - x_{r4,G}) \tag{8.2}$$

8.3 Ověření

Nejdříve je ověřena možnost použít diferenciální evoluci při vývoji neuronové sítě. Toho je docíleno pokusem, kdy je nejdříve testována evoluce samotných vah. Použití diferenciální evoluce na vývoj vah dopředných neuronových sítí byl již testován například v článku [21] a dokonce byly navrženy úpravy pro zefektivnění diferenciální evoluce při trénování neuronových sítí [17]. Tyto úpravy se však neukázaly být efektivnější než původní algoritmus.



Obrázek 8.2: Sítě, na kterých byla differencíální evoluce testována. Na levé straně je vidět minimální topologie schopná řešit DPNV. Na pravé straně je vidět složitější verzi sítě s 11 spojeními.

Pro otestování vývoje vah byl použit experiment s balancováním vah bez informace o rychlostech (DPNV). Nejdříve je algoritmus vyzkoušen na minimální topologii schopné řešit problém. Poté byl algoritmus otestován i na větším genomu obsahujícím 11 spojení k optimalizaci, z čehož dvě byly rekurentní vazby. Parametry pro experimenty byly zvoleny empiricky a to na hodnotách *F=0.7* a *CR=0.8*. Bylo provedeno více experimentů pro rozdílné velikosti populací. Pro každý experiment bylo provedeno 100 běhů.

Populace	Generace		Evaluace		Úspěšnost
	Průměr	Směr. odchylka	Průměr	Směr. odchylka	
25	22,57	7,85	634,5	212,19	54%
50	21,60	6,12	1173,41	318,65	96%
100	20,29	4,43	2169,58	452,79	100%

Tabulka 8.1: Výsledky experimentů s differenciální evolucí pro menší topologii a experiment DPNV.

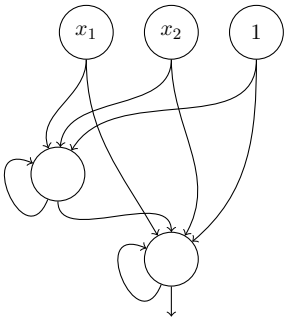
Populace	Generace		Evaluace		Úspěšnost
	Průměr	Směr. odchylka	Průměr	Směr. odchylka	
50	27,13	7,06	1461,17	367,37	87%
100	29,52	7,00	3111,57	714,71	99%
150	28,68	6,76	4509,36	1028,64	100%

Tabulka 8.2: Výsledky experimentů s differenciální evolucí pro větší topologii a experiment DPNV.

Z výsledků uvedených v tabulkách výše je zřejmé, že použití diferenciální evoluce na optimalizaci vah je při vhodných podmínkách bez problému použitelné. Důležitým parametrem se ukázala velikost populace. Při její menší velikosti se stávalo, že populace uvázla v lokálním extrému. Z toho se populace nedokázala dostat, protože schopnost „prozkoumávat“ okolní prostor je závislá na diferencích mezi jednotlivými jedinci. Pokud se celá populace nachází v lokálním extrému, jsou mezi jedinci malé rozdíly a populace nemá schopnost se vymanit z lokálního extrému. Větší populace lépe pokryla stavový prostor a měla menší šanci na stagnaci.

Přestože větší populace má vyšší pravděpodobnost úspěchu, je to vyváženo výpočetními nároky. Průměrná generace nalezení řešení je podobná pro všechny velikosti populací, větší populace k tomu však potřebuje vyhodnotit více jedinců. Pokud se však menší populaci podaří neuváznout v lokálním extrému, je počet evalucí potřebných k nalezení řešení několikanásobně menší (viz tabulky 8.1 a 8.2).

Další pokusy s diferenciální evolucí byly provedeny na optimalizaci vah pro binární sčítačku. Jak bylo ukázáno v kapitole 7.3, jedná se o značně obtížnější problém, než je DPNV. Obzvláště náročné jsou pak lokální extrémy spojené s komplikovanějším laděním vah (viz kapitola 7.3.1). Cílem je ověřit, že diferenciální evoluce zvládá řešit i problémy se silným sklonem k lokálním extrémům.



Obrázek 8.3: Topologie rekurentní sítě použité v experimentu s binární sčítačkou. Váhy jsou optimalizovány pomocí diferenciální evoluce.

K experimentům byla použita topologie vyvinutá algoritmem NEAT (ověřeno, že je dostatečná k funkčnosti binární sčítačky). Váhy pro počáteční generaci byly generovány náhodně. Nejdříve byly vyzkoušeny hodnoty podobné hodnotám pro DPNV.

Populace	Generace		Evaluace		Úspěšnost
	Průměr	Směr. odchylka	Průměr	Směr. odchylka	
100	64,25	19,10	6653,5	1949,20	4/10
200	66,10	10,65	13552,2	2151,94	10/10
400	88,40	27,76	35936,8	11159,68	10/10

Tabulka 8.3: Výsledky experimentů s differenciální evolucí pro binární sčítačku a $CR = 0,7$ a $F = 0,8$.

Populace	Generace		Evaluace		Úspěšnost
	Průměr	Směr. odchylka	Průměr	Směr. odchylka	
100	84,0	72,99	8668,0	7445,76	6/10
200	62,4	16,17	12804,8	3267,40	10/10
400	74,7	39,66	30429,4	15943,80	10/10

Tabulka 8.4: Výsledky experimentů s differenciální evolucí pro binární sčítačku a $CR = 0,8$ a $F = 0,7$.

Výsledky pokusů (tabulky 8.3 a 8.4) ukazují obdobné tendence jako výsledky pokusů s DPNV. Vyšší náročnost se projevila potřebou větší populace pro spolehlivější vyřešení.

Menší populace byla opět schopná nalézt řešení velice rychle, ale zároveň byla znatelně nižší úspěšnost. Kromě těchto hodnot bylo experimentováno i s jinými parametry diferenciální evoluce. Populace byla nastavena na 400 jedinců.

CR/F	Generace		Evaluace		Úspěšnost
	Průměr	Směr. odchylka	Průměr	Směr. odchylka	
0,0/1,5	142,0	66,7	57484,0	26827,3	4/10
0,2/1,5	93,4	31,7	37946,8	12778,5	10/10
0,2/1,0	89,6	26,7	36419,2	10740,5	10/10
0,5/1,5	105,6	26,8	42878,0	10783,4	6/10

Tabulka 8.5: Výsledky experimentů s differenciální evolucí pro binární sčítačku a některé další hodnoty CR/F.

V tabulce 8.5 si lze povšimnout, že srovnatelných výsledků dosahují i některé jiné kombinace parametrů. Zajímavou kombinaci představuje nízká hodnota CR (nejlépe se osvědčila hodnota okolo 0,2) a vyšší hodnota F (nejlépe přesahující hodnotu 1, výhodnou se zdá být hodnota 1,5). Výsledkem křížení je poté z větší části rodičovský vektor s menším počtem parametrů převzatých z mutujícího vektoru. Mutující vektor je však schopen dosáhnout většího rozsahu hodnot díky zvýšené hodnotě F. Při zvýšení parametru CR úspěšnost klesá. Stejně tak jeho další snížení výsledky nezlepší. Je vhodné připomenout, že $CR = 0$ znamená z definice křížení v diferenciální evoluci jeden převzatý parametr z mutujícího vektoru.

8.4 Hypotéza

Typickým problémem genetických algoritmů je jejich tendence uchýlit se k lokálním extrémům. Možným způsobem, jak se z lokálních extrémů vymanit, je diverzifikace způsobů prohledávání prostoru. Předpokladem je, že rozdílné metody se mohou navzájem doplnit.

Na podobném principu jsou založeny takzvané **memetické algoritmy**. Jedná se o větev či rozšíření genetických algoritmů založené na myšlence, že jedinci mohou „vylepšit“ svůj genom o zkušenosti získané během života. V praxi se jedná o zařazení optimalizačních kroků pro jednotlivé jedince. Genom se tedy nemění pouze genetickými operátory, ale podstupuje i krok učení, který se snaží jedince dále optimalizovat. V biologických paralelách se jedná zkušenosti získané jedincem během života (nazývané memy) [16].

Na myšlence memetických algoritmů je navržena i úprava algoritmu NEAT v kombinaci s diferenciální evolucí. Z výsledků pokusů (tabulky 8.1 a 8.2) přichází v úvahu dva způsoby kombinace NEATu s diferenciální evolucí:

1. Použít diferenciální evoluci v případě detekce lokálního extrému (stagnace populace) k vyváznutí a nalezení nového řešení.

2. Použít diferenciální evoluci jako operátor lokální optimalizace vah nově provedené strukturální mutace.

Myšlenkou za tvrzením číslo 1 je použití větší populace, která má teoreticky lepší schopnost vymanit se z lokálního extrému. Populace pro diferenciální evoluci by byla vygenerována nová za použití topologie existujících jedinců, případně i s podobnými váhami. V úvahu připadá větší příděl vypočetních zdrojů zaměřených na malou skupinu nejlépe ohodnocených jedinců. Místo očekávání vhodné mutace, či křížení, kterým se dostane některý jedinec

z lokálního extrému, může být výhodnější soustředit výpočetní zdroje do jiné metody, která bude cílit na nalezení nového řešení.

Tvrzení číslo 2 vychází z do jisté míry opačné myšlenky. Použít diferenciální evoluci k optimalizaci vah po provedení strukturální evoluce. Optimalizace by mohla upravovat pouze nové struktury v síti. Případné úpravy starších struktur by byly pouze omezené a prohledávání stavového prosotru by tak bylo soustředěno na nejnovější úpravy. K tomu by byla použita malá populace, která má sice nižší úspěšnost, ale disponuje nejrychlejší konvergencí v poměru k poču evaluací.

8.5 Testování a výsledky

K ověření je použita diferenciální evoluce při detekci stagnace populace v lokálním extrému. Cílem je nalezení nového řešení. Za tímto účelem byla vytvořena speciální mutace, která má na vstupu jedince k optimalizaci.

Nejdříve je vytvořena počáteční populace pro diferenciální evoluci. Při jejím vytváření přichází v úvahu dva způsoby generování nových vah. Pro případy, kdy je jedinec poblíž řešení, ale je potřeba přesného vyladění vah, lze ke generování populace využít vstupního jedince k optimalizaci. Populace je poté tvořena jeho kopiemi, kde **každá z vah je náhodně pozměněna**. Diferenciální evoluce by poté měla být schopná najít nejlepší řešení v okolí prostoru původních vah jedince.

Druhým způsobem je **generování nových vah**. Pokud je řešení uvázlé v lokálním extrému, je také velmi pravděpodobné, že není možné dosáhnout příliš lepšího výsledku v okolním prostoru vah jedince určeného ke generování. Snahou tedy je vyzkoušet zcela nové váhy za použití topologie jedince určeného k optimalizaci.

8.5.1 Optimalizace nejlepšího

Optimalizace pomocí diferenciální evoluce byla spuštěna po detekci stagnace po určený počet generací. Zároveň byla ponechána i příležitost na vyváznutí z extrému na algoritmu NEAT. Optimalizace byly spuštěny každou 20. generaci beze změny fitness.

Váhy populace	Generace		Evaluace		Úspěšnost
	Průměr	S. odchylka	Průměr	S. odchylka	
pozměněné	174,2	71,3	273587,5	135353,6	39/50
náhodné	186,11	89,3	295218,6	173905,2	42/50

Tabulka 8.6: Výsledky experimentů s optimalizací differenciální evolucí pro binární sčítačku. Do počáteční populace byl vložen i optimalizovaný jedinec. Průměrná generace nalezení řešení je nižší, než bez diferenciální evoluce, ovšem počet evaluací byl výrazně vyšší.

Prvním z pokusů byla optimalizace nejlepšího jedince z populace. Tento jedinec byl vložen do počáteční populace DE. Po optimalizaci byl nejlepší jedinec vložen do populace v NEATu. Díky funkčnosti DE byl do NEATu nejhůře vložen nezměněný jedinec. Výsledky experimentů lze vidět v tabulce 8.6. Z těch lze vyčíst, že optimalizace vah, které jsou pouze pozměněné, je ve srovnání s novými váhami rychlejší (méně evaluací) a stabilnější (nižší směrodatná odchylka). Nižší je oproti standardnímu NEATu i průměrná generace vyřešení a její směrodatná odchylka. Vypočetní náročnost diferenciální evoluce však převyšuje tato vylepšení – počet evaluací je značně vyšší, než u standardního NEATu (viz tabulka 7.3).

I přes nižší průměrnou generaci, ve které je nalezeno řešení, je směrodatná odchylka stále vysoká. Z toho lze usoudit, že řešení, která mají vhodnou topologii jsou pomocí DE optimalizována rychleji (čímž je snížena průměrná generace). Na druhou stranu, pokud má řešení nevhodnou topologii, DE nepomůže a navíc jejím opakovaným použitím výrazně roste počet evaluací.

8.5.2 Optimalizace s přidáním spojení

Z předchozích zkušeností, kdy pro vhodnou optimalizaci řešení nebyla dostupná vhodná topologie, vychází další úprava. Optimalizace se liší v závislosti na tom, kolikrát byla spuštěna beze změny fitness nejlepšího jedince. Při prvním spuštení je pro DE použit jedinec stejně, jako v předchozích experimentech. Při druhém spuštění beze změny fitness je však navíc použita **mutace přidání spojení**. Tento přístup umožňuje lehké rozšíření topologie, které je rovnou optimalizováno pomocí DE. Tento jedinec je vložen do populace a slouží k rozšíření diverzity. Přidán je i v případě, kdy je výsledná fitness menší než nejlepší fitness v populaci, neboť se jedná o jedince s upravenou topologií přinášející diverzitu do stagnující populace.

Váhy populace	Generace		Evaluace		Úspěšnost
	Průměr	Směr. odchylka	Průměr	Směr. odchylka	
pozměněné	172,3	69,9	269749,4	135325,5	43/50
náhodné	174,5	81,1	281955,7	159994,3	43/50

Tabulka 8.7: Výsledky experimentů s optimalizací differenciální evolucí pro binární sčítačku. Optimalizace nejlepšího jedince. Při druhém spuštění bez zlepšení je přidáno spojení.

Výsledky této úpravy lze vidět v tabulce 8.7. Při použití pozměnění vah byly výsledky velice podobné předchozím výsledkům (tabulka 8.6). Větší změna se však projevila u generování nových vah, kdy se průměrná generace nalezení řešení snížila na úroveň pozměněných vah. V obou případech generování vah se lehce zvýšila úspěšnost algoritmu.

8.5.3 Optimalizace s vícenásobným přidáním spojení

Snížení počtu evaluací při použití nových spojení při delší stagnaci bylo dále sledováno. Dalším experimentem bylo navýšení přidaných spojení s dobou, po kterou dochází ke stagnaci. První spuštění je opět samotná optimalizace jedince. Při druhém (a dalším) spuštění je přidáno jedno spojení. Při čtvrtém (a dalším) spuštění dochází k několikanásobným mutacím. Je přidáno další spojení a další dvě mohou být přidána s 50% šancí. Při dlouhodobé stagnaci tedy může jedinec před optimalizací získat 2-4 nová spojení. Tyto silnější zásahy do topologie mají za cíl výrazněji pozměnit topologii a vytvořit nové jedince schopné konkurovat stagnující populaci.

Váhy populace	Generace		Evaluace		Úspěšnost
	Průměr	Směr. odchylka	Průměr	Směr. odchylka	
pozměněné	183,9	91,9	306942,7	178520,4	43/50
náhodné $[-8, 8]$	151,6	69,5	222542,6	132963,6	41/50
náhodné $[-2, 2]$	172,3	79,4	273654,7	149581,5	42/50

Tabulka 8.8: Výsledky experimentů s optimalizací differenciální evolucí pro binární sčítačku. Optimalizace nejlepšího jedince. Při druhém (a dalším) spuštění bez zlepšení je přidáno spojení. Při čtvrtém (a dalším) je přidáno další spojení a další dvě s 50% šancí.

Při použití několikanásobných mutací s novými spojeními je vidět (viz tabulka 8.8), že vzrůstá množství evaluací pro pozměněné počáteční váhy. Naopak výsledky při použití náhodných počátečních vah se nadále zlepšují. Průměrný počet generací klesnul až ke zhruba 150 generacím se směr. odchylkou okolo 70. I přes tyto výsledky se však počet evaluací snížil pouze do blízkosti samostatného NEATu. Tyto hodnoty nebyly vylepšeny ani snížením počátečního rozsahu vah a naopak to vedlo k opačnému efektu.

Z dosavadních výsledků se zdá, že váhy jsou často již kvalitně optimalizovány algoritmem NEAT. To znamená, že často se při stagnaci jedná spíše o nevhodnou topologii, než o potřebu přesného vyladění vah. Parametry algoritmu jsou však nastaveny tak, aby byla pokládán větší důraz na rozdílnost vah a jejich evoluci. K tomu slouží především silnější parametr rozdílu vah c_3 při rozdělení do druhů v kombinaci s vyšší populací, která umožňuje udržet více jedinců s větší diverzitou.

8.5.4 Vliv velikosti populace

Jelikož DE je cílená na optimalizaci vah, její funkčnost se do značné míry překrývá s parametry NEATu volenými pro zlepšení optimalizace vah. Způsobem, jak snížit toto překrytí je snížení velikosti populace algoritmu NEAT. Ten bude poté sloužit jako způsob, jak najít slibné topologie, zatímco DE bude použita k přesnějšímu vyladění vah.

Populace / DE	Generace		Evaluace		Úspěšnost
	Průměr	S. odchylka	Průměr	S. odchylka	
100 / ne	581,7	214,4	55226,8	20679,0	18/50
100 / ano	316,4	177,8	304714,7	188969,4	45/50
250 / ne	414,5	206,4	104507,3	51693,6	37/50
250 / ano	210,3	127,4	204292,9	151434,2	50/50
500 / ne	289,1	168,1	146473,2	85272,1	45/50
500 / ano	175,8	89,1	192123,0	118127,4	50/50

Tabulka 8.9: Výsledky experimentů pro různé velikosti populací s použitím násobných mutací.

Ve výsledcích experimentů s různou velikostí populace (tabulka 8.9) jsou kromě výsledků s optimalizací pomocí DE také přidány výsledky samotného algoritmu NEAT bez DE. Z těch se ukazuje, že jisté zmenšení populace je oproti očekávání prospěšné. Pro velikost populace 500 je úspěšnost lehce zvýšená a množství evaluací klesá pod 200000. Při velikosti populace 250 klesá množství evaluací téměř ke 100000, ale již se začíná snižovat úspěšnost. To ovšem poukazuje na fakt, že volba parametrů algoritmu NEAT může značně ovlivnit výsledky. Jejich správné nastavení se však odvíjí od specifických problémů a může se velmi lišit.

Z výsledků experimentů, které DE používají, je vidět silný nárůst úspěšnosti při velikosti populací 250 a 500. Tyto populace zaznamenaly 100% úspěšnost při řešení. Počet evaluací je také nižší než při použití standardního algoritmu NEAT s velikostí populace 1000. Ten dosahuje nižšího počtu evaluací pouze u menších populací. Žádná z nich však nedosahuje 100% úspěšnosti. Vlastností, která není na první pohled zřejmá, je počet evaluací, pokud není řešení nalezeno. Při použití DE je při stejném počtu generací počet evaluací mnohem vyšší. Řešením by mohlo být použít jako kritérium zastavení evoluce maximální počet evaluací místo maximálního počtu generací.

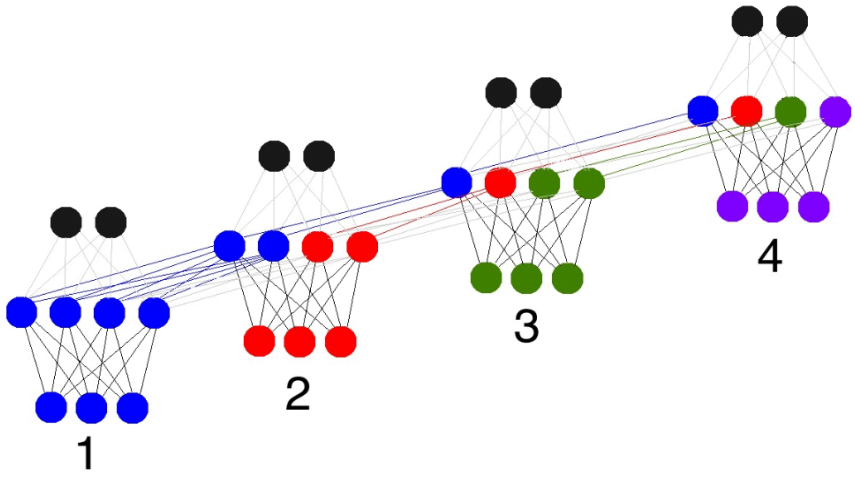
Kapitola 9

Srovnání s klasickými algoritmy

V této kapitole je algoritmus NEAT porovnán s klasickými algoritmy určenými k učení neuronových sítí. Jmenovitě s algoritmem Backpropagation pro dopředné neuronové sítě a algoritmem Backpropagation Through Time pro rekurentní neuronové sítě.

Na úvod jsou stručně představeny použité algoritmy. Algoritmus Backpropagation (dále jen BP) a Backpropagation Through Time (dále jen BPTT) patří do třídy gradientních metod. Oba algoritmy používají k optimalizaci vah chybovou funkci učené sítě.

Hodnota chyby mezi očekávaným výstupem a skutečným výstupem se zpětně šíří celou sítí a upravují se podle ní váhy. Algoritmus BP funguje pouze na dopředné sítě. Pro rekurentní sítě je potřeba použít k tomu uzpůsobenou verzi – algoritmus BPTT. Ten se v mnohém podobá BP, ale bere v potaz i vliv zpětných spojení při předchozích aktivacích sítě. Toho algoritmus dosahuje tak, že vytvoří pro každou aktivaci sítě jednu její kopii bez rekurentních spojení. Místo rekurentního spojení v jedné kopii sítě se vytvoří dopředné spojení spojující síť v aktivaci t se sítí v aktivaci $t-1$. To si lze lépe představit podle obrázku 9.1.



Obrázek 9.1: Zobrazení sítě se 4 skrytými neurony rozložené přes 4 časové kroky. Neurony ve skryté vrstvě jsou plně propojené a tvoří rekurentní spojení. Při učení algoritmem BPTT jsou však výstupy neuronů v předchozí aktivaci propojeny se skrytou vrstvou reprezentující následující aktivaci. Algoritmus BPTT propaguje chybu přes všechny časové kroky tím, že učí celou síť propojených kopií v čase naráz. [28]

Algoritmy BP a BPTT mají dva parametry, kterými lze ovlivnit průběh učení. Prvním parametrem je rychlost učení (angl. learning rate), který ovlivňuje velikost změn vah při učení. Vyší hodnota může urychlit učení, ale zároveň ztížit nalezení optima. Druhým parametrem je setrvačnost (angl. momentum). Tento parametr umožňuje započítat i část předchozích změn vah a urychlit tím postup učení.

Ke všem pokusům byla použita knihovna PyBrain dostupná na [19]. Tato knihovna má naimplementovány algoritmy BP i BPTT. Vzhledem k naprosto rozdílným přístupům k učení neuronových sítí mezi těmito algoritmy a evolučními přístupy je jejich porovnání náročnější.

Jako hrubý ukazatel rychlosti učení byl zvolen počet učících iterací nad celou datovou sadou. Algoritmy BP a BPTT se však učí již při každém vstupním vzorku a nejedná se tedy o hodnotu přesně odpovídající počtu evaluací použitého v experimentech s algoritmem NEAT. Při větším množství spojení musí algoritmy BP i BPTT zároveň učit více vah, což je teoreticky výpočetně náročnější. Mnohé implementace algoritmů však používají paralelizaci výpočtů.

K měření kvality odezvy sítě je použita funkce MSE (Mean Squared Error), viz rovnice 9.1. Měření kvality odezvy je porovnáno především na klasifikační úloze nad datasetem Iris (kap. 9.2).

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - y_i)^2 \tag{9.1}$$

Kde n je počet výstupů, Y_i je očekávaný výstup neuronu i a y_i je pozorovaný výstup neuronu i.

Dále bylo také prozkoumáno chování a úspěšnost algoritmů BP a BPTT pro různé hodnoty jejich parametrů při řešených úlohách.

9.1 Experimenty XOR

K porovnání dopředné sítě (vrstvené dopředné sítě) je použita logická funkce XOR. Přestože lze vytvořit síť řešící funkci XOR s jediným skrytým neuronem, je k tomu potřeba spojení vedoucí ze vstupu přímo do výstupu. Standardní vrstvené dopředné sítě mohou tento problém vyřešit s minimálním počtem dvou neuronů ve skryté vrstvě.

Experiment byl proveden jak pro různé kombinace obou parametrů algoritmu BP, tak pro různé množství neuronů ve skryté vrstvě. Učení bylo ukončeno a považováno za neúspěšné po 4000 učících iterací přes celou trénovací sadu.

Rychlost učení	Setrvačnost	S. neuronů	Iterací	Úspěšnost
0,01	0,0	2	-	0/10
0,01	0,0	5	654,5	2/10
0,01	0,0	10	2916,0	2/10
0,01	0,1	10	3027,0	6/10
0,01	0,9	10	548,3	10/10
0,2	0,9	5	75,6	10/10
0,2	0,9	10	42,3	10/10
0,9	0,0	10	64,7	10/10
0,9	0,9	5	30,5	10/10
0,9	0,9	10	27,9	10/10

Tabulka 9.1: Vybrané výsledky experimentů s logickou funkcí XOR pro různé parametry algoritmu BP.

V tabulce 9.1 lze nalézt výběr z výsledků experimentu. Kompletní výsledky jsou přiloženy v tabulce B.1 v příloze. Z výsledků je možné vyčíst, že počet potřebných iterací k vyřešení této úlohy se snižuje se zvyšujícími se parametry rychlosti učení i setrvačnosti. Při nevhodně zvolených parametrech je však rozdíl v rychlosti vyřešení markantní (3027,0 iterací pro 0,01/0,1/10 a 27,9 pro 0,9/0,9/10, kde čísla odpovídají parametrům rychlost učení / setrvačnosti / počtu skrytých neuronů). Zásadním činitelem se však také ukazuje počet neuronů ve skryté vrstvě. Algoritmus BP těží z větší velikosti sítě, kterou zvládá naučit v menším počtu iterací. Při učení sítí se dvěma skrytými neurony měl algoritmus BP dokonce problémy s uváznutím v lokálních extrémech, ze kterých v některých případech nebyl schopen vyváznout. Je však nutné zmínit, že při větší velikosti sítě narůstá výpočetní složitost každého učícího kroku, protože se zároveň učí více vah. Při přílišném navýšení počtu skrytých neuronů (až na 50) již klesá i schopnost algoritmu úlohu včas vyřešit.

Rozdílem oproti algoritmu NEAT a evolučním přístupům je, že větší množství neuronů pomáhá při rychlosti řešení. Algoritmus NEAT v porovnání s BP používá k řešení úlohy v průměru 9,86 spojení, zatímco algoritmus BP s použitím nejryhleji konvergujících parametrů používá 22 spojení a 10 skrytých neuronů. K nalezní správného řešení však potřebuje v průměru pouhých 27,9 iterací nad datovou sadou a 111,6 učících kroků (pro každý vzorek). Algoritmus NEAT potřebuje v průměru 40,7 generací. Během každé z nich však vyhodnotí 150 sítí.

V této úloze je rychlost učení algoritmem BP výrazně rychlejší než přístupem algoritmu NEAT. To je ovšem zcela očekávané, neboť algoritmus BP je cílen pouze na učení, zatímco NEAT zároveň vyvíjí i celou topologii sítě. NEAT však často zvládá vyvynout síť s minimální, či téměř minimální velikostí. Oproti tomu, aby se urychlilo učení algoritmem BP, potřebuje síť mnohem větší počet skrytých neuronů.

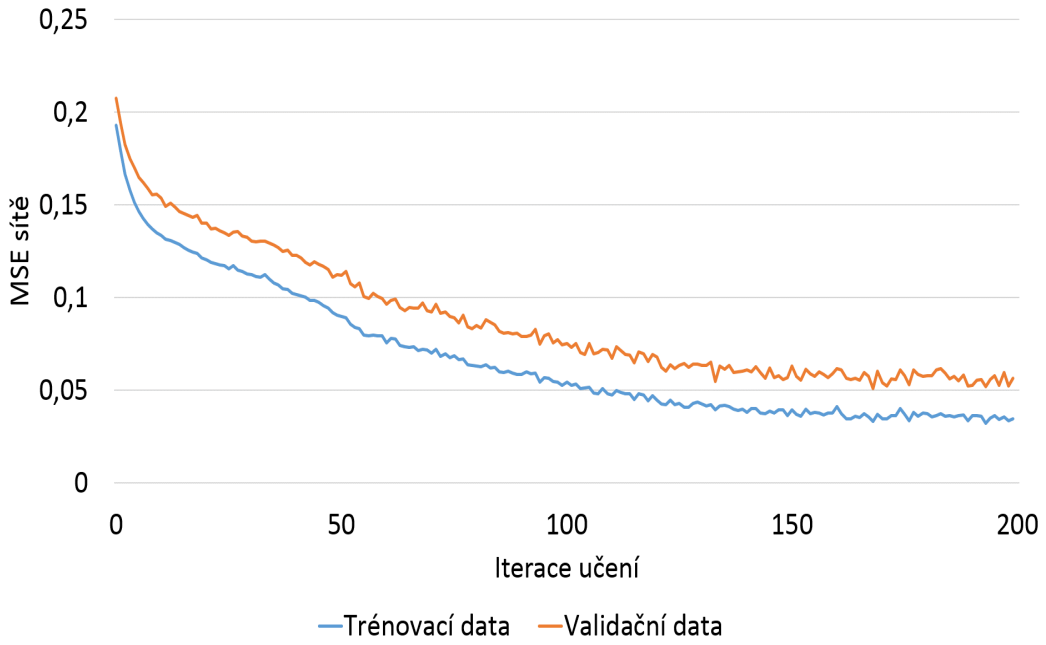
9.2 Klasifikace datasetu Iris

Jedním z častých typů úloh, pro které jsou dopředné neuronové sítě využívány, je problém klasifikace. Na základě vstupního vektoru musí síť umět přiřadit jednu ze tříd.

Ke klasifikační úloze se často volí topologie sítě taková, že má stejný počet výstupních neuronů, jako je počet tříd. Každému výstupnímu neuronu odpovídá jedna třída a jako predikovaná třída se považuje výstupní neuron s nejsilnější aktivací.

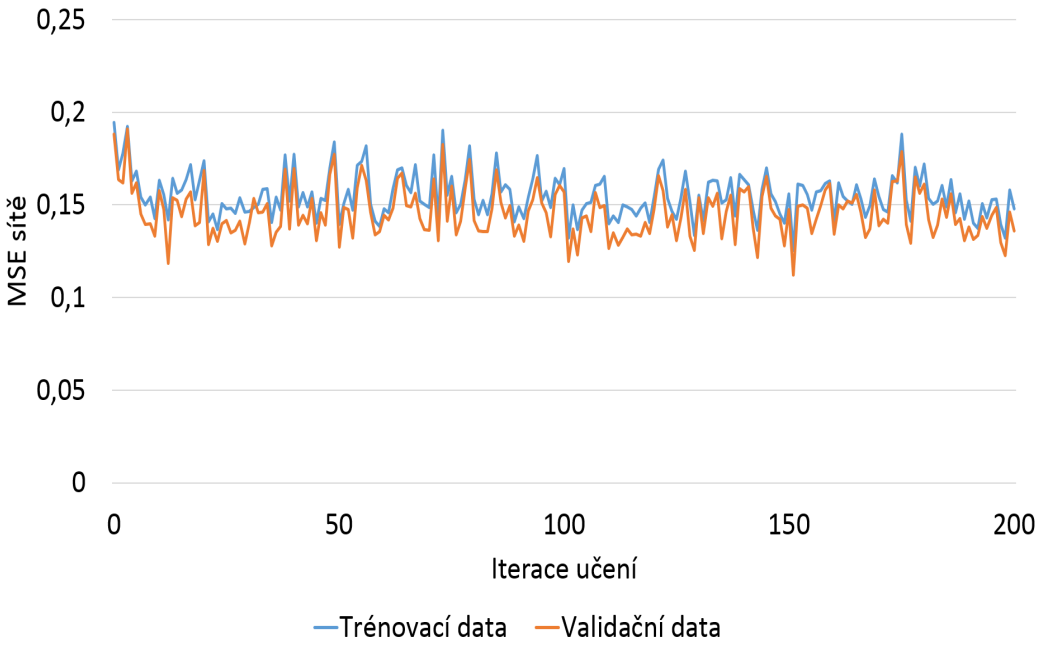
Klasifikace v tomto experimentu je prováděna na datasetu Iris [14]. Ten klasifikuje kosatce (druh květiny - lat. Iris) na základě čtyř vstupních hodnot do jedné ze tří tříd – Iris setosa, Iris versicolor, nebo Iris virginica. Parametry, podle kterých se klasifikuje, jsou délka a šířka okvětních lístků a délka a šířka kalichu květiny. Dataset obsahuje 150 vzorků.

Zvolené parametry algoritmu BP pro tuto úlohu jsou: rychlost učení 0,02 a setrvačnost 0,1. Vhodné nastavení parametrů bylo nalezeno experimentálně. Vzorky byly rozděleny na dvě skupiny. První skupina obsahovala 70% vzorků a byla použita k trénování. Druhá skupina obsahovala 30% vzorků a sloužila k validaci. Výsledky jsou průměrem z 20 experimentů, které se učily po 200 iterací nad trénovací datovou sadou.



Obrázek 9.2: Graf zobrazující chybu sítě (MSE) se 3 skrytými neurony v závislosti na počtu učících iterací. Parametry algoritmu BP jsou zvoleny 0,02 pro rychlost učení a 0,1 pro setrvačnost. Lze si všimnout stabilnějšího vývoje chyby sítě než na obrázku 9.3.

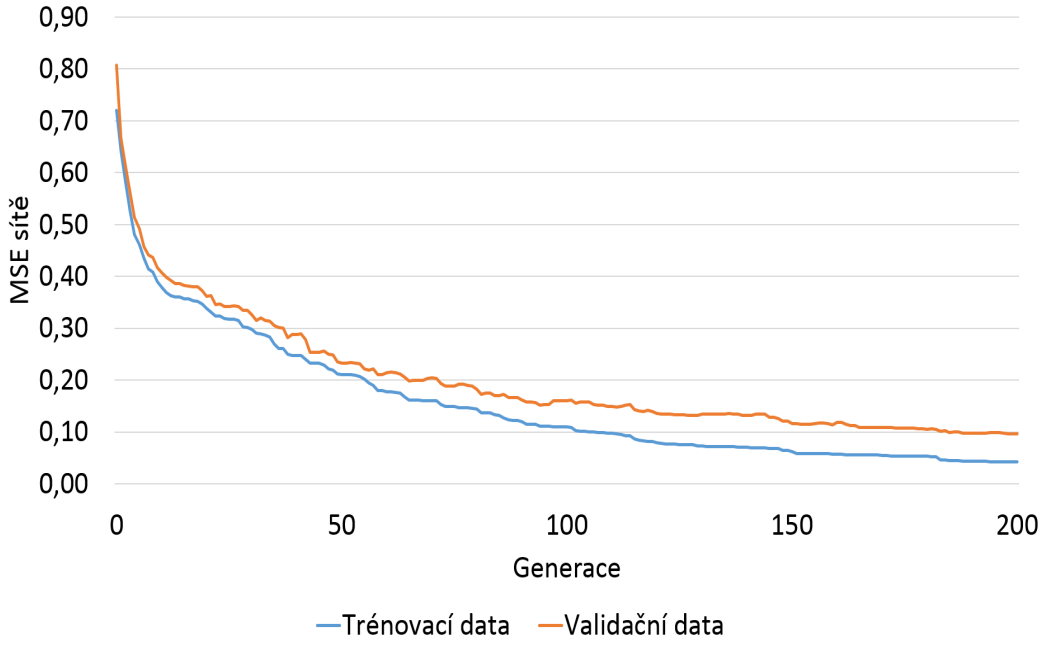
Výsledný graf s hodnotou MSE sítě lze vidět na obrázku 9.2. Síť dosáhla úspěšnosti při klasifikaci okolo 95% nad trénovací datovou sadou a okolo 90% nad validační datovou sadou. Průběh chyby učení klesá bez větších výkyvů a konverguje lehce nad hodnotou 0,03. Validační data vykazují lehce vyšší hodnoty MSE, což odpovídá i nižší úspěšnosti nad touto datovou sadou.



Obrázek 9.3: Graf zobrazující chybu sítě (MSE) s 10 skrytými neurony v závislosti na počtu učících iterací. Parametry algoritmu BP jsou zvoleny 0,1 pro rychlost učení a 0,9 pro setrvačnost. Při těchto hodnotách měl algoritmus problém s konvergencí a kvalita odezvy sítě byla nestabilní (k porovnání obrázek 9.2).

Pří zvýšení parametru rychlosti učení, narozdíl od exprimentů s logickou funkcí XOR, se rychlost snižování chyby sítě ani konečná kvalita odezvy nezlepšily (narozdíl od experimentů s logickou funkcí XOR). Také přidání neuronů do skryté vrstvy, či zvýšení parametru setrvačnosti při učení nepomohlo. Naopak, při zvýšených hodnotách parametrů měla síť problémy s konvergencí a MSE sítě bylo značně nestabilní. To lze vidět obzvláště dobře na obrázku 9.3, ve kterém je použito 10 skrytých neuronů, rychlost učení 0,1 a parametr setrvačnosti 0,9. Při těchto hodnotách klesla úspěšnost na trénovací sadě k 70% a na validační sadě k 65%.

K pokusům s algoritmem NEAT byly použity podobné hodnoty parametů jako v kapitole 7.2.1. Rozdílné parametry jsou: max. počet generací = 200, velikost populace = 200, počáteční rozsah vah = [-2,2], síla mutací = 1,7, náhodné rozložení hodnot vah = uniformní, pravděpodobnost přidání spojení = 20%. Fitness funkce byla počítána podle funkce $f(N) = 1/\text{MSE sítě}(N)$. Výsledky jsou průměrem z 20 běhů algoritmu.



Obrázek 9.4: Graf zobrazující chybu sítě (MSE). K učení byl použit algoritmus NEAT. Kvalita odezvy sítě je nižší než u algoritmu BP a to především pro validační datovou sadu.

Z grafu na obrázku 9.4 je vidět, že oproti algoritmu BP zůstává chyba sítě vyšší. Algoritmus BP konvergoval k hodnotě MSE okolo 0,03, zatímco NEAT má průměrně hodnotu MSE zhruba 0,05 pro trénovací datovou sadu a 0,1 pro validační datovou sadu. Zajímavým zjištěním je, že průměrná hodnota úpěšnosti klasifikace se u NEATu pohybuje okolo 97% pro trénovací datovou sadu a 95% pro validační datovou sadu. Tyto výsledky jsou lepší, než výsledky algoritmu BP. Jedna ze sítí s hodnotou MSE 0,19 (vysoko nad průměrnou hodnotou MSE po učení algoritmem BP) zaznamenala dokonce 100% úspěšnost při klasifikaci.

Při srovnání obou algoritmů v této klasifikační úloze je algoritmus NEAT schopný dosáhnout o něco lepší úspěšnosti klasifikace, než algoritmus BP. Algoritmus BP však dosahuje lepší odezvy sítě (nižší hodnoty MSE) než algoritmus NEAT. K nastavení optimálních parametrů algoritmu BP bylo zapotřebí naprosto odlišných hodnot, než byly použity pro řešení logické funkce XOR (např. rychlost učení: Iris – 0,02, XOR – 0,9). To poukazuje na možné problémy při volbě vhodných parametrů algoritmu BP.

9.3 Rekurentní binární sčítačka

S učením rekurentních neuronových sítí algoritmem BPTT je experimentováno na problému rekurentní binární sčítačky popsané v kapitole 7.3. Síť s minimální možnou topologií, která je schopná řešit tento problém, obsahuje jeden skrytý neuron. Sítě použité pro učení algoritmem BPTT obsahují jednu skrytou vrstvu plně propojenou se vstupem i výstupy. Rekurentním spojením je plně propojená skrytá vrstva. Experimentováno bylo i se spojením každého ze skrytých neuronů pouze se sebou samým. Tuto topologii se však nepodařilo na daný problém naučit.

Experiment byl proveden pro různé kombinace parametrů algoritmu BPTT a různé množství skrytých neuronů. Datová sada sestávala z kombinací součtů dvou čísel od 0 do 20. Čísla v učící sadě byla zarovnána na 8 bitů. Maximální počet iterací přes celou učící sadu byl zvolen 2000.

Rychlost učení	Setrvačnost	S. neuronů	Úspěšnost	Iterací
0,1	0,0	2	0/10	-
0,1	0,0	5	8/10	884,75
0,1	0,0	10	9/10	715,44
0,1	0,0	25	0/10	-
0,5	0,0	5	10/10	591,0
0,5	0,0	10	10/10	733,0
0,5	0,1	5	10/10	626,0
0,1	0,9	5	10/10	388,0
0,1	0,9	10	9/10	636,55
0,1	0,9	25	4/10	838,5
0,9	0,0	5	10/10	518,0
0,9	0,0	10	10/10	448,0

Tabulka 9.2: Výběr z výsledků učení rekurentní binární sčítačky algoritmem BPTT pro různé parametry.

V tabulce 9.3 je vidět výběr některých ze zajímavějších výsledků. Kompletní výsledky lze nalézt v příloze C.1. Tuto úlohu nebyl algoritmus BPTT schopný vyřešit, pokud měl k dispozici pouze 2 skryté neurony. Oproti logické funkci XOR také uplně neplatí, že více skrytých neuronů zlepšuje rychlost učení. Nejlepších výsledků bylo dosahováno při 5 skrytých neuronech. Nárustem na 10 až 25 skrytých neuronů již klesala schopnost řešit problém a narůstal i počet iterací potřebný k naučení.

Nejlepších výsledků dosahoval algoritmus BPTT s parametry rychlosti učení 0,1 a setrvačnosti 0,9. Další vhodnou kombinací pak byla rychlost učení 0,9 a setrvačnost 0,0. Kombinace vyšších hodnot obou parametrů na druhou stranu již zhoršila úspěšnost řešení.

V porovnání s algoritmem NEAT je pro optimální rychlost učení potřeba zhruba dvakrát více skrytých neuronů (NEAT potřebuje v průměru 2,42 skrytých neuronů, viz kapitola 7.3). Počet evaluací potřebný algoritmem NEAT je více než 100000. Algoritmus BPTT potřeboval při vhodných parametrech méně než 400 iterací nad celou datovou sadou. Jedná se však o téměř 160000 učících kroků po jednom vzorku. Navíc způsob, kterým algoritmus BPTT učí síť včetně rekurentních spojení, vyžaduje mnohem více výpočetních prostředků (lze vidět na obrázku 9.1). Při zarovnání čísel na 8 bitů je potřeba 8 aktivací sítě. Algoritmus BPTT učí síť pro každou její aktivaci, což odpovídá téměř osminásobnému zvýšení vypočetních nároků oproti učení stejné, avšak pouze dopředné sítě.

Kapitola 10

Závěr

V této práci byly popsány dvě velké skupiny softcomputingových metod - *genetické* (*evoluční*) *algoritmy* a *neuronové sítě*. Byla ukázána možná spolupráce při optimalizaci vah i topologie neuronové sítě pomocí evolučních algoritmů a problémy, které to obnáší. Dále byly popsány existující algoritmy a z nich byl podrobněji popsán algoritmus NEAT.

Algoritmus NEAT byl naimplementován a byly s ním provedeny experimenty. Na problému balancování dvou tyčí byla ukázána funkčnost algoritmu a jeho chování. Generované sítě byly často tvořeny kompaktní topologií a algoritmus neměl problémy s vyřešením. Těžším problémem byla evoluce binární rekurentní sčítačky. Algoritmus NEAT byl schopný problém ve většině případů vyřešit, často však vývoj stagnoval v lokálních extrémech.

Stagnace populace byla podnětem pro úpravu algoritmu. Algoritmus NEAT byl rozšířen o diferenciální evoluci a rozšířené mutace při stagnaci populace. Ty mají za cíl zvýšit diverzitu populace a pomoct s vyváznutím z lokálního extrému. Při použití upraveného algoritmu bylo dosaženo 100% úspěšnosti při řešení rekurentní binární sčítačky. Výpočetní náročnost se však oproti algoritmu NEAT zvýšila.

Algoritmus NEAT byl také srovnán s algoritmy Backpropagation (BP) pro dopředné neuronové sítě a Backpropagation through time (BPTT) pro rekurentní neuronové sítě. Porovnání bylo provedeno jak z hledika kvality odezvy učení, tak z hlediska chování pro různé parametry algoritmů i sítě. Také byla ukázána rychlost učení algoritmů a stabilita jejich výsledků.

Při experimentech se podle očekávání ukázálo, že algoritmus NEAT má vyšší časovou náronočnost, a to už jenom díky faktu, že zároveň vyvíjí topologii sítě. Algoritmy BP a BPTT se však ukázaly velmi citlivé na různé hodnoty a kombinace učících parametrů. Navíc v různých úlohách byly potřebné diametrálně odlišné hodnoty a jejich nalezení může být náročné. U všech řešených úloh tvořil algoritmus NEAT menší sítě, než byly potřebné k optimálnímu učení algoritmy BP a BPTT. V klasifikační úloze byl algoritmus BP v průměru schopný dosáhnout nižších hodnot chyby sítě (MSE - Mean Squared Error) než algoritmus NEAT, obzvláště na trénovacích datech. Algoritmus NEAT zaznamenal vyšší úspěšnost klasifikace, ovšem oba algoritmy dosáhly úspěšnosti nad 90%.

Faktorem, který hraje v algoritmu NEAT značnou roli, je optimální nastavení parametrů. Užitečnou úpravou by mohl být metaoptimalizátor, který se pokusí odhadnout vhodné nastavení parametrů. Parametry by také mohlo být možné měnit za běhu s cílem vyváznout z lokálních extrémů a stagnující populace.

Další možnou úpravou algoritmu je začlenění LSTM (Long Short-Term Memory) buněk, které umožňují udržovat v sítí pamět po delší dobu. Vzhledem k faktu, že LSTM buňky obsahují více vstupů, by bylo potřeba zvážit jejich prvotní napojení na zbytek sítě.

Literatura

[1] Angeline, P. J.; Saunders, G. M.; Pollack, J. B.: An evolutionary algorithm that constructs recurrent neural networks. *Neural Networks, IEEE Transactions on*, ročník 5, č. 1, 1994: s. 54–65.

[2] Bertsimas, D.; Tsitsiklis, J.: Simulated annealing. In *Statistical Science*, ročník 8, 1993, s. 10–15.

[3] Eiben, A.; Smith, J.: *Introduction to Evolutionary Computing, 2nd edition*. Springer, 2015, ISBN 3662448734.

[4] Gers, F.: *Long short-term memory in recurrent neural networks*. Dizertační práce, Universität Hannover, 2001, dostupné: 14.5. 2016.
URL http://felixgers.de/papers/phd.pdf

[5] Gomez, F. J.; Miikkulainen, R.: Solving non-Markovian Control Tasks with Neuroevolution. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, s. 1356–1361.
URL http://dl.acm.org/citation.cfm?id=1624312.1624411

[6] Gruau, F.: Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm. 1994.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.5939&rep=rep1&type=pdf

[7] Gruau, F.; Whitley, D.; Pyeatt, L.: A Comparison between Cellular Encoding and Direct Encoding for Genetic Neural Networks. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, MIT Press, 1996, s. 81–89.

[8] Guo, J.: BackPropagation Through Time. 2013, dostupné: 14.5. 2016.
URL http://ir.hit.edu.cn/~jguo/docs/notes/bptt.pdf

[9] Gurney, K.: *An Introduction to Neural Networks*. Bristol, PA, USA: Taylor & Francis, Inc., 1997, ISBN 1857286731.

[10] Hornby, G. S.; Globus, A.; Linden, D. S.; aj.: Automated antenna design with evolutionary algorithms. 2006.

[11] Kassahun, Y.; Sommer, G.: Efficient reinforcement learning through evolutionary acquisition of neural topologies. In *In 13th European Symposium on Artificial Neural Networks (ESANN)*, 2005.

[12] Koehn, P.: Combining genetic algorithms and neural networks: The encoding problem. 1994.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.7330&rep=rep1&type=pdf

[13] Kárný, M.; Warwick, K.; Kůrková, V.: The Psychological Limits of Neural Computation. In *Dealing with Complexity*, Perspectives in Neural Computing, Springer London, 1998, ISBN 978-3-540-76160-0, s. 252–263, doi:10.1007/978-1-4471-1523-6_17.
URL http://dx.doi.org/10.1007/978-1-4471-1523-6_17

[14] Lichman, M.: UCI Machine Learning Repository. 2013, dostupné: 14.5. 2016.
URL http://archive.ics.uci.edu/ml

[15] Mitchell, M.: *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998, ISBN 0262631857.

[16] Moscato, P.; Cotta, C.; Mendes, A.: *New Optimization Techniques in Engineering*, kapitola Memetic Algorithms. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, ISBN 978-3-540-39930-8, s. 53–85, doi:10.1007/978-3-540-39930-8_3.
URL http://dx.doi.org/10.1007/978-3-540-39930-8_3

[17] Pedersen, M. E. H.; Chipperfield, A. J.: Tuning differential evolution for artificial neural networks. *HL0803. Hvass Laboratories*, 2008.

[18] Rojas, R.: *Neural Networks: A Systematic Introduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1996, ISBN 3-540-60505-3.

[19] Schaul, T.; Bayer, J.; Wierstra, D.; aj.: PyBrain – The Python Machine Learning Library. 2016, dostupné: 14.5. 2016.
URL http://www.pybrain.org/pages/download

[20] Simon, D.: *Evolutionary Optimization Algorithms*. Wiley, 2013, ISBN 0470937416.

[21] Slowik, A.; Bialko, M.: Training of artificial neural networks using differential evolution algorithm. In *Human System Interactions, 2008 Conference on*, IEEE, 2008, s. 60–65.

[22] Sopena, J. M.; Romero, E.; Alquezar, R.: Neural networks with periodic and monotonic activation functions: a comparative study in classification problems. In *Artificial Neural Networks, 1999. ICANN 99. Ninth International Conference on (Conf. Publ. No. 470)*, ročník 1, IET, 1999, s. 323–328.

[23] Stanley, K. O.: neural networks research group.
http://nn.cs.utexas.edu/?neat-c, dostupné: 9.12. 2015.

[24] Stanley, K. O.: *Efficient Evolution of Neural Networks through Complexification*. Dizertační práce, The University of Texas at Austin, 2004.

[25] Stanley, K. O.; Miikkulainen, R.: Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, ročník 10, 2002: s. 99–127.

[26] Stephenson, C.: Hebbian neural networks and the emergence of minds. 2010, dostupné: 14.5. 2016.
URL http://guava.physics.uiuc.edu/~nigel/courses/569/Essays_Fall2010/Files/Stephenson.pdf

[27] Storn, R.; Price, K.: Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, ročník 11, č. 4: s. 341–359, ISSN 1573-2916, doi:10.1023/A:1008202821328.
URL http://dx.doi.org/10.1023/A:1008202821328

[28] Trask, A.: Anyone Can Learn To Code an LSTM-RNN. 2015, dostupné: 14.5. 2016.
URL https://iamtrask.github.io/2015/11/15/anyone-can-code-lstm/

[29] Wikimedia: Wikimedia Commons. Dostupné: 31.12. 2015.
URL https://commons.wikimedia.org

Přílohy

Seznam příloh

Příloha A

Obsah CD

- ./src – Složka se zdrojovými kódy a programovou částí DP.
- ./src/readme.txt – Soubor s dalšími informacemi o obsahu složek a návody ke spuštění.
- ./src/NEAT – Složka s projektem NEAT.
- ./src/BP_BPTT – Složka se skripty k experimentům s algoritmy BP a BPTT.
- ./src/NEAT/store/NEAT.jar – Umístění jar archivu s implementací algoritmu NEAT.
- ./doc – Složka obsahující zdrojové texty DP pro LaTeX.
- ./doc/dp.pdf – Diplomová práce ve formátu pdf.

Příloha B

Tabulka s výsledky učení logické funkce XOR

Rychlost učení	Setrvačnost	S. neuronů	Úspěšnost	Iterací
0,01	0,0	2	0/10	-
0,01	0,0	5	2/10	654,5
0,01	0,0	10	2/10	2916,0
0,01	0,1	2	0/10	-
0,01	0,1	5	1/10	3675,0
0,01	0,1	10	6/10	3027,0
0,01	0,25	2	0/10	-
0,01	0,25	5	2/10	1601,5
0,01	0,25	10	3/10	3209,33
0,01	0,5	2	1/10	2492,0
0,01	0,5	5	1/10	3921,0
0,01	0,5	10	9/10	1978,77
0,01	0,75	2	0/10	-
0,01	0,75	5	8/10	1948,75
0,01	0,75	10	10/10	1577,4
0,01	0,9	2	0/10	-
0,01	0,9	5	10/10	1082,8
0,01	0,9	10	10/10	548,3
0,05	0,0	2	1/10	2875,0
0,05	0,0	5	8/10	2003,0
0,05	0,0	10	10/10	1155,1
0,05	0,1	2	1/10	3552,0
0,05	0,1	5	8/10	2111,5
0,05	0,1	10	10/10	672,6
0,05	0,25	2	2/10	2563,0
0,05	0,25	5	10/10	1610,1
0,05	0,25	10	10/10	608,4
0,05	0,5	2	3/10	2040,0
0,05	0,5	5	10/10	1094,8
0,05	0,5	10	10/10	473,1

0,05	0,75	2	7/10	1378,57
0,05	0,75	5	10/10	621,6
0,05	0,75	10	10/10	310,1
0,05	0,9	2	6/10	1017,83
0,05	0,9	5	10/10	217,9
0,05	0,9	10	10/10	163,5
0,1	0,0	2	3/10	2925,33
0,1	0,0	5	10/10	1494,3
0,1	0,0	10	10/10	611,9
0,1	0,1	2	7/10	2308,71
0,1	0,1	5	10/10	1114,7
0,1	0,1	10	10/10	428,2
0,1	0,25	2	4/10	767,5
0,1	0,25	5	10/10	963,1
0,1	0,25	10	10/10	521,2
0,1	0,5	2	6/10	1741,83
0,1	0,5	5	10/10	670,8
0,1	0,5	10	10/10	222,5
0,1	0,75	2	6/10	826,16
0,1	0,75	5	10/10	252,4
0,1	0,75	10	10/10	126,8
0,1	0,9	2	7/10	290,28
0,1	0,9	5	10/10	126,0
0,1	0,9	10	10/10	72,6
0,2	0,0	2	5/10	1359,2
0,2	0,0	5	10/10	702,1
0,2	0,0	10	10/10	233,9
0,2	0,1	2	8/10	1154,75
0,2	0,1	5	10/10	619,8
0,2	0,1	10	10/10	273,8
0,2	0,25	2	7/10	1533,14
0,2	0,25	5	10/10	535,6
0,2	0,25	10	10/10	294,1
0,2	0,5	2	6/10	1172,5
0,2	0,5	5	10/10	253,9
0,2	0,5	10	10/10	121,7
0,2	0,75	2	6/10	1040,5
0,2	0,75	5	10/10	193,3
0,2	0,75	10	10/10	60,7
0,2	0,9	2	8/10	144,12
0,2	0,9	5	10/10	75,6
0,2	0,9	10	10/10	42,3
0,5	0,0	2	8/10	728,75
0,5	0,0	5	10/10	235,1
0,5	0,0	10	10/10	116,1

0,5	0,1	2	4/10	736,0
0,5	0,1	5	10/10	199,1
0,5	0,1	10	10/10	116,7
0,5	0,25	2	5/10	550,4
0,5	0,25	5	10/10	167,7
0,5	0,25	10	10/10	77,3
0,5	0,5	2	7/10	343,28
0,5	0,5	5	10/10	142,0
0,5	0,5	10	10/10	66,8
0,5	0,75	2	7/10	156,0
0,5	0,75	5	10/10	67,5
0,5	0,75	10	10/10	37,3
0,5	0,9	2	8/10	484,5
0,5	0,9	5	10/10	47,5
0,5	0,9	10	10/10	31,4
0,75	0,0	2	6/10	386,5
0,75	0,0	5	10/10	155,3
0,75	0,0	10	10/10	78,6
0,75	0,1	2	7/10	459,57
0,75	0,1	5	10/10	179,5
0,75	0,1	10	10/10	86,5
0,75	0,25	2	9/10	363,88
0,75	0,25	5	10/10	157,9
0,75	0,25	10	10/10	55,5
0,75	0,5	2	9/10	210,33
0,75	0,5	5	10/10	85,7
0,75	0,5	10	10/10	63,7
0,75	0,75	2	8/10	154,0
0,75	0,75	5	10/10	58,0
0,75	0,75	10	10/10	36,9
0,75	0,9	2	7/10	96,14
0,75	0,9	5	10/10	40,4
0,75	0,9	10	10/10	27,3
0,9	0,0	2	10/10	446,8
0,9	0,0	5	10/10	145,8
0,9	0,0	10	10/10	64,7
0,9	0,1	2	7/10	423,85
0,9	0,1	5	10/10	156,8
0,9	0,1	10	10/10	63,3
0,9	0,25	2	6/10	287,33
0,9	0,25	5	10/10	149,0
0,9	0,25	10	10/10	45,0
0,9	0,5	2	8/10	169,37
0,9	0,5	5	10/10	69,3
0,9	0,5	10	10/10	42,7

0,9	0,75	2	7/10	160,71
0,9	0,75	5	10/10	52,2
0,9	0,75	10	10/10	34,8
0,9	0,9	2	5/10	250,0
0,9	0,9	5	10/10	30,5
0,9	0,9	10	10/10	27,9

Tabulka B.1: Kompletní výsledky experimentů s logickou funkcí XOR pro různé parametry algoritmu BP.

Příloha C

Tabulka s výsledky učení binární rekurentní sčítačky

Rychlost učení	Setrvačnost	S. neuronů	Úpěšnost	Iterací
0,1	0,0	2	0/10	-
0,1	0,0	5	8/10	884,75
0,1	0,0	10	9/10	715,44
0,1	0,0	25	0/10	-
0,25	0,0	5	8/10	897,25
0,25	0,0	10	9/10	466,55
0,25	0,0	25	1/10	1031,0
0,5	0,0	5	10/10	591,0
0,5	0,0	10	10/10	733,0
0,5	0,0	25	1/10	821,0
0,9	0,0	5	10/10	518,0
0,9	0,0	10	10/10	448,0
0,9	0,0	25	0/10	-
0,1	0,1	5	6/10	1099,33
0,1	0,1	10	9/10	586,55
0,1	0,1	25	0/10	-
0,1	0,5	5	9/10	733,22
0,1	0,5	10	9/10	545,44
0,1	0,5	25	1/10	771,0
0,1	0,9	5	10/10	388,0
0,1	0,9	10	9/10	636,55
0,1	0,9	25	4/10	838,5
0,5	0,0	5	9/10	465,44
0,5	0,0	10	8/10	447,25
0,5	0,0	25	2/10	581,0
0,5	0,1	5	10/10	626,0
0,5	0,1	5	10/10	626,0
0,5	0,1	10	9/10	488,77
0,5	0,1	25	3/10	504,33
0,5	0,5	5	10/10	539,0

0,5	0,5	10	9/10	449,88
0,5	0,5	25	3/10	327,66
0,5	0,9	5	5/10	853,0
0,5	0,9	10	5/10	845,0
0,5	0,9	25	0/10	-

Tabulka C.1: Kompletní výsledky experimentů s binární rekurentní sčítačkou pro různé parametry algoritmu BPTT.