VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ NÁVRH NEURONOVÝCH SÍTÍ VYUŽÍVAJÍCÍ GENERATIVNÍ KÓDOVÁNÍ

EVOLUTIONARY DESIGN OF NEURAL NETWORKS WITH GENERATIVE ENCODING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. TEREZA HYTYCHOVÁ**
AUTHOR

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

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2020/2021

Zadání diplomové práce

Studentka: **Hytychová Tereza, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Inteligentní systémy
Název: **Evoluční návrh neuronových sítí využívající generativní kódování**
Evolutionary Design of Neural Networks with Generative Encoding
Kategorie: Umělá inteligence
Zadání:

1. Zpracujte studii mapující využití evolučních algoritmů pro návrh umělých neuronových sítí. Zaměřte se na metody využívající generativní zakódování problému.
2. Navrhněte metodu využití evolučního algoritmu pro postupný vývin umělé neuronové sítě.
3. Navrženou metodu implementujte ve zvoleném programovacím jazyce.
4. Navrženou metodu ověřte na zvolených klasifikačních úlohách.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 2.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Sekanina Lukáš, prof. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 16. prosince 2020

Abstrakt

Cílem této práce je navrhnout a implementovat metodu pro návrh neuronové sítě, která bude využívat generativní kódování. Navržená metoda, která vychází z metody J. F. Millera, je založena na vytvoření modelu mozku, který je postupně vyvíjen, a ze kterého lze extrahovat klasickou neuronovou síť. Vývin mozku je řízen programy vytvořenými pomocí kartézského genetického programování. Implementace byla provedena v jazyce Python s použitím knihovny Numpy. Při experimentování se ukázalo, že metoda je schopná vytvářet neuronové sítě, které na menších datových sadách dosahují přesnosti přesahující 90 %. Metoda je zároveň schopna vytvářet neuronové sítě řešící více problémů naráz, za cenu mírného snížení dosažené přesnosti.

Abstract

The aim of this work is to design and implement a method for the evolutionary design of neural networks with generative encoding. The proposed method is based on J. F. Miller's approach and uses a brain model that is gradually developed and which allows extraction of traditional neural networks. The development of the brain is controlled by programs created using cartesian genetic programming. The project was implemented in Python with the use of Numpy library. Experiments have shown that the proposed method is able to construct neural networks that achieve over 90 % accuracy on smaller datasets. The method is also able to develop neural networks capable of solving multiple problems at once while slightly reducing accuracy.

Klíčová slova

neuronová síť, evoluční algoritmy, genetické programování, kartézské genetické programování, evoluční návrh neuronové sítě

Keywords

neural network, evolutionary algorithms, genetic programming, cartesian genetic programming, evolutionary development of neural network

Citace

HYTYCHOVÁ, Tereza. *Evoluční návrh neuronových sítí využívající generativní kódování.* Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Lukáš Sekanina, Ph.D.

Evoluční návrh neuronových sítí využívající generativní kódování

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením pana prof. Ing. Lukáše Sekaniny, Ph.D. Uvedla jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpala.

....
Tereza Hytychová
17. května 2021

Poděkování

Děkuji panu prof. Ing. Lukášovi Sekaninovi, Ph.D. za odborné rady poskytnuté při zpracování této práce.

Obsah

1 Úvod **3**

2 Umělé neuronové sítě **4**
2.1 Model umělého neuronu 4
2.2 Dopředné neuronové sítě 6
2.3 Učení dopředných neuronových sítí 7
2.3.1 Algoritmus zpětného šíření chyby (backpropagation) 7

3 Evoluční algoritmy **10**
3.1 Princip evolučních algoritmů 10
3.1.1 Reprezentace řešení 11
3.1.2 Selekční mechanismy 12
3.1.3 Vytváření jedinců pro novou generaci 12
3.2 Genetické algoritmy 13
3.3 Evoluční strategie 14
3.4 Evoluční programování 15
3.5 Genetické programování 15
3.6 Kartézské genetické programování 17
3.6.1 Reprezentace jedinců v CGP 17
3.6.2 Operátor mutace v CGP 19
3.6.3 Řídicí evoluční algoritmus v CGP 19

4 Metody pro návrh neuronových sítí pomocí evolučních algoritmů **21**
4.1 Přímé a nepřímé kódování 21
4.2 Přístupy založené na buněčné chemii 22
4.3 Přístupy využívající gramatiky 23
4.4 Metody NEAT a Hyper-NEAT 23
4.5 Metoda J. F. Millera pro vývin neuronové sítě řešící více problémů 24
4.5.1 Upravený model neuronu a dendritu 24
4.5.2 Princip růstu sítě 25
4.5.3 Parametry modelu 27
4.5.4 Vývoj soma a dendrit programu 27
4.5.5 Testování a výsledky 28

5 Návrh metody pro vývin neuronové sítě **30**
5.1 Návrh metody 30
5.2 Návrh testování 31

6 Implementace metody pro vývin neuronové sítě **33**
6.1 Popis implementace 33
6.1.1 Implementace mozku 34
6.1.2 Implementace soma a dendrit programu 36
6.1.3 Implementace neuronové sítě 36
6.2 Parametry a spuštění programu 37

7 Experimenty a výsledky **39**
7.1 Experimenty pro jednotlivé datasety 40
7.2 Experimenty pro návrh sítě řešící více problémů naráz 43
7.3 Výsledné soma a dendrit programy 46

8 Závěr **50**

Literatura **51**

Kapitola 1

Úvod

Ačkoliv vývoj umělých neuronových sítí začal už ve 40. letech 20. století, v posledních letech zájem o toto téma neustále rostl. To je způsobeno především skutečností, že neuronové sítě pomáhají s řadou problémů těžce řešitelných pomocí běžných algoritmů. Neuronové sítě se dnes využívají v celé řadě oborů pro řešení široké škály úloh z oblasti predikce, rozpoznávání či klasifikace. Úspěšné jsou například oblasti medicíny pro diagnostiku nádorových onemocnění z rentgenových snímků, dále v rozpoznávání značek automobilů ve snímcích z dopravních kamer, ve finanční sféře zase dokážou předpovědět vývoj cen různých komodit či identifikovat podezřelé transakce, rozsáhlé využití našly i v analýze textů či marketingu [9].

Výhodou neuronových sítí je především jejich schopnost se učit, která vychází z jejich inspirace fungováním lidského mozku. Aby ale učení bylo úspěšné, je potřeba mít i vhodnou vnitřní strukturu sítě. Zde nastupuje otázka, jak takovou strukturu vlastně určit. Jaký je vhodný počet neuronů v síti? A jak neurony ideálně propojit mezi sebou? Ačkoliv pro zodpovězení těchto otázek existují různá doporučení a běžně používané topologie sítí pro řešení určitých problémů, ne vždy jednoznačně vedou k úspěchu. Existují ovšem metody, založené na evolučních algoritmech, pro postupný vývin neuronových sítí, které se tento problém snaží řešit.

Cílem této práce bylo nastudovat problematiku neuronových sítí a evolučních algoritmů a dále seznámit se s metodami používanými pro evoluční návrh neuronových sítí. Na základě získaných znalostí a nastudovaných metod pak implementovat metodu pro evoluční návrh neuronových sítí, která bude využívat generativní kódování.

První dvě následující kapitoly této práce obsahují seznámení s důležitými pojmy a koncepty v oblasti neuronových sítí (kapitola 2) a evolučních algoritmů (kapitola 3). Další kapitola 4 se zabývá popisem aktuálního stavu výzkumu v oblasti evolučního návrhu neuronových sítí a prezentuje metody vývinu neuronových sítí využívající především nepřímého kódování. Následující kapitola 5 obsahuje návrh metody pro vývin neuronových sítí založený na metodě z předchozí kapitoly. Práce dále pokračuje kapitolou 6, která detailněji popisuje implementaci navržené metody, a především kapitolou 7 prezentující provedené experimenty a dosažené výsledky. Práci ukončuje závěr v kapitole 8.

Kapitola 2

Umělé neuronové sítě

Umělá neuronová síť je výpočetní model, který obecně slouží k aproximaci funkcí. Jak už bylo řečeno v úvodu, inspirace pro tvorbu umělých neuronových sítí byla nalezena v biologii, konkrétně ve fungování lidského mozku. Velkou výhodou neuronových sítí, která je odlišuje od striktně algoritmických metod aproximace, je jejich schopnost se učit na základě vstupů. Právě tato schopnost umožňuje pomocí neuronových sítí řešit i problémy, které by byly pro algoritmizaci moc složité a nebo jim nejsme schopni porozumět natolik, abychom je mohli algoritmizovat. Neuronové sítě navíc vynikají schopností odhalit v datech skryté vzory, což jim napomáhá k vyřešení zadaných problémů.

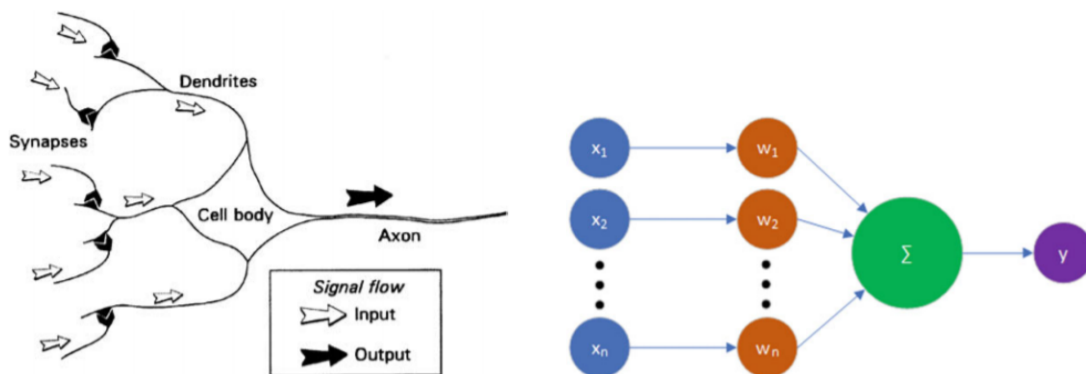
Architektura neuronových sítí je většinou založena na uspořádání do vrstev. Způsob propojení těchto vrstev je jednou z mnoha možností, jak neuronové sítě klasifikovat z hlediska jejich topologie. Různé možnosti propojení umožňují řešit různé problémy. Tato práce je zaměřena především na dopředné neuronové sítě, kde jsou jednotlivé vrstvy propojeny pouze jedním směrem (tzn. neexistují zde zpětné vazby), a které navíc obsahují menší počet vrstev. Základními prvky každé vrstvy jsou ale samotné neurony, které definují chování celé neuronové sítě. Je tedy nutné, aby byl nejprve představen model neuronu.

Tato kapitola proto dále pokračuje popisem modelu umělého neuronu a vysvětlením základních pojmů z oblasti problematiky neuronových sítí, které budou v rámci práce hojně používány a jejich pochopení je stěžejní pro pochopení celé práce. Dále jsou zde podrobněji rozebrány acyklické dopředné neuronové sítě a vysvětlen základní princip učení těchto sítí.

Pokud není uvedeno jinak, všechny informace v této kapitole týkající se neuronových sítí byly převzaty z [1], [8], [9], [11], informace v podkapitole 2.3 pak jsou převážně z [26], odkud jsou zároveň převzaty i v ní prezentované vzorce.

2.1 Model umělého neuronu

Neuron je základním stavebním kamenem každé neuronové sítě. Matematický model umělého neuronu byl inspirován skutečnými biologickými neurony, jaké se nacházejí v lidském mozku. Biologický neuron se skládá z těla, do kterého vede větší množství výběžků – dendritů, které slouží k přijmu informace od ostatních neuronů, a jednoho axonu, který slouží k šíření informací do ostatních neuronů. Na následujícím obrázku 2.1 lze vidět velmi zjednodušený model biologického neuronu ve srovnání s modelem umělého neuronu.



Obrázek 2.1: Zjednodušené schéma biologického neuronu a schéma umělého neuronu. Převzato z: [8] a [9].

Umělý neuron má n reálných vstupů označených x_1 až x_n, které jsou reprezentací dendritů reálných neuronů, a jeden výstup y, který reprezentuje axon biologického neuronu. „Dendrity" a „axony" umělého neuronu slouží stejně jako u biologického neuronu k propojování neuronů mezi sebou, což umožňuje tvorbu různě velkých neuronových sítí. Vstupy umělého neuronu mají navíc určeny váhy označené w_1 až w_n. Váhy jsou důležité jednak pro výpočet tzv. bázové funkce a aktivační funkce, které jsou reprezentovány v samotném těle neuronu a budou dále představeny později, ale především také pro učení sítě, protože to probíhá právě upravováním těchto vah.

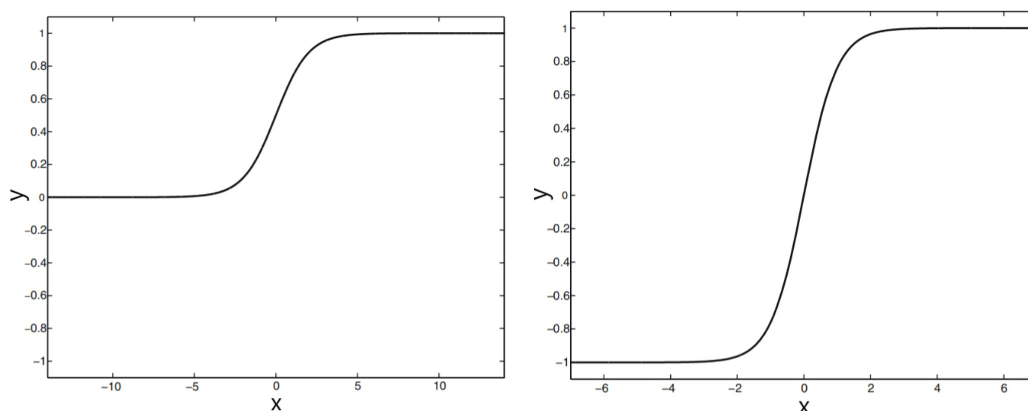
K modelu neuronu je v některých situacích dobré přidávat i tzv. bias. Jedná se o přivedení dalšího vstupu x_0 se stálou hodnotou 1 ale proměnnou váhou w_0. S tímto vstupem a váhou se dále zachází stejně jako se všemi ostatními vstupy a vahami a jeho váha je taktéž upravována při učení. Bias by měl pomoci zachytit případnou invariantní část predikce výstupu především v situacích, kdy je distribuce tříd na vstupu nevyvážená.

Dále bude představen princip funkčnosti umělého neuronu, který je následující. Při přivedení vstupních hodnot na vstupy neuronu se musí nejprve provést spočtení již zmíněné bázové funkce. Nejčastěji využívanou funkcí je lineární bázová funkce ve tvaru:

$$u = \sum_{k=1}^{n} x_k * w_k. \tag{2.1}$$

Jedná se o prostý součet součinů vstupů a jim odpovídajících vah. Výpočtem této bázové funkce získáme tzv. vnitřní potenciál neuronu u. Ten je ale ještě nutné transformovat na výstupní hodnotu y. Tu získáme spočtením tzv. aktivační funkce.

Aktivačních funkcí existuje velké množství. Jedná o funkce nelineární. Jako příklad aktivační funkce lze uvést například sigmoidu, ve tvaru $y = \frac{1}{1+e^{(-u)}}$, nebo hyperbolický tangens, který má tvar $y = tanh(u)$ (viz obrázek 2.2). Jedná se o spojité aktivační funkce, které jsou v neuronových sítích hojně využívané. Rozdíl mezi těmito dvěma funkcemi spočívá především v tom, že sigmoida na svém výstupu generuje výsledky v rozsahu $y \in [0, 1]$ (tzv. binární výstup), zatímco hyperbolický tangens poskytuje výsledky z intervalu $y \in [-1, 1]$ (tzv. bipolární výstup).



Obrázek 2.2: Příklad aktivační funkce neuronu (sigmoida a hyperbolický tangens). Převzato z: [1].

Poslední důležitou součástí popisu umělého neuronu je popis jeho učení. Jak už bylo zmíněno, učení jednoho neuron se provádí postupnou úpravou jeho vah. Na vstup neuronu se nejprve přiloží vstupní vektor $\vec{x}$ z trénovací množiny, pro který známe očekávaný výstup t. Na základě vstupů se vypočte výstup y. Pro úpravu vah se pak používá následující vzorec, kde $\vec{w}_{current}$ značí aktuální vektor vah v sítí před krokem učení a $\vec{w}_{new}$ nově vypočítaný vektor vah:

$$\vec{w}_{new} = \vec{w}_{current} + \mu(t - y)\vec{x}. \tag{2.2}$$

Symbol μ se nazývá koeficient učení, je volen z rozsahu $\mu \in (0, 1]$ a ovlivňuje rychlost samotného učení. Po shlédnutí vzorce je zřejmé, že je-li výstup y totožný o očekávaným výstupem t, váhy zůstávají stejné, protože na pravé straně vzorce by odečtením y od očekávaného výstupu vyšla 0 a zůstalo by jen $\vec{w}_{new} = \vec{w}_{current}$. V opačném případě se váhy upraví tak, aby se při příštím přiložení vstupu řešení přiblížilo očekávané hodnotě a došlo tak k minimalizaci chyby sítě.

Ve své podstatě i tento jeden umělý neuron tvoří neuronovou síť s n vstupy, která obsahuje jeden neuron s jedním výstupem y. Taková síť obsahující pouze jeden neuron dokáže rozdělit vstupní data do dvou tříd. Je ale schopna vyřešit pouze lineárně separovatelné úlohy. Proto přichází na řadu skládání neuronů do vrstev a vytváření složitějších neuronových sítí.

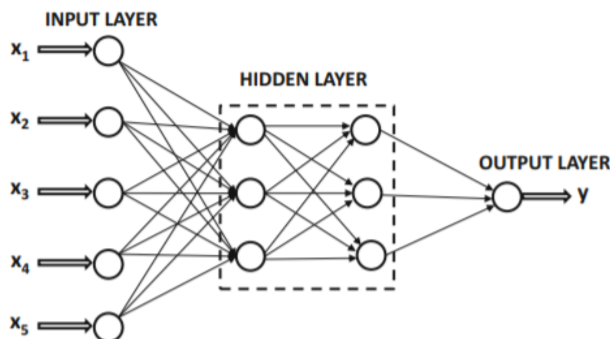
2.2 Dopředné neuronové sítě

Dopředná neuronová síť je síť obsahující větší množství neuronů, které jsou mezi sebou propojeny tak, že výstup jednoho neuronu je vstupem jednoho nebo i více dalších neuronů. Neurony jsou zde uspořádány do vrstev, konkrétně do jedné vstupní vrstvy, jedné nebo více vrstev skrytých a jedné výstupní vrstvy. Při konstrukci dopředné neuronové sítě je potřeba rozhodnout o počtu skrytých vrstev. Teoreticky by měla stačit jediná vrstva, sítě s více vrstvami jsou ale lepší ve zobecňování a mohou tedy mít po naučení lepší odezvy na neznáme vstupy.

Dále je nutné rozhodnout o počtu neuronů v jednotlivých vrstvách. První vrstva zpravila obsahuje počet neuronů odpovídající počtu vstupů. Skryté a výstupní vrstvy pak mohou obsahovat libovolné množství neuronů v závislosti na řešeném problému. Při příliš nízkém počtu neuronů ve skrytých vrstvách se síť nebude schopna na daný problém naučit, příliš vysoký počet neuronů pak může mít za následek nedostatečnou zobecňovací schopnost

sítě, tedy přeučení. Takto přeučená síť není schopna správně reagovat na nové, dříve neviděné vstupy. Počtu neuronů i počtu vrstev je proto vhodné při implementaci sítě věnovat pozornost a experimentovat s nimi.

Ve vrstvených dopředných sítích dále platí, že neurony jedné vrstvy mohou být propojeny jen s neurony ve vrstvě následující a to pouze v jednom směru (zleva doprava směrem k vrstvě výstupní). Propojení neuronů v rámci jedné vrstvy je tedy zakázáno, stejně jako jakékoliv cykly, které by mohly vzniknout propojením neuronů v opačném směru. Příklad dopředné neuronové sítě je na obrázku 2.3. Dopředné sítě obsahující velké množství skrytých vrstev pak nazýváme hluboké neuronové sítě.



Obrázek 2.3: Příklad dopředné neuronové sítě s pěti vstupními neurony ve vstupní vrstvě, dvěma skrytými vrstvami po třech neuronech a jedním neuronem ve vrstvě výstupní. Převzato z: [1].

Dopředné sítě využívají stejnou lineární bázovou funkci (2.1), jaká byla prezentována pro jeden umělý neuron a taktéž mohou využívat i stejné aktivační funkce, tedy sigmoidální funkci a hyperbolický tangens, ale i mnoho dalších.

Při výpočtu výstupů (tzv. odezvy sítě) se pak postupuje po jednotlivých vrstvách. Nejprve spočtou svoje výstupy pomocí bázové a aktivační funkce neurony ve první vrstvě a tyto výstupy předají další vrstvě, která také spočte svoje výstupy a tak se postupuje až do vrstvy výstupní, kde je pak možné přečíst konečný vektor výstupů. Učení dopředné sítě je ale trochu složitější a nejčastěji se provádí za pomoci metody gradientního sestupu a algoritmu zpětného šíření chyby, který bude popsán dále.

2.3 Učení dopředných neuronových sítí

Učení pouze jednoho neuronu je možné podle vzorce 2.2, protože chyba sítě je přímou funkcí vah. U sítí obsahujících více vrstev tento postup ale není možný, protože chyba sítě je vyjádřena chybovou funkcí, jež je složitou kompozitní funkcí vah z předchozích vrstev. Proto se pro učení vícevrstvých acyklických dopředných sítí používá metoda zpětného šíření chyby, známá taktéž jako backpropagation.

2.3.1 Algoritmus zpětného šíření chyby (backpropagation)

Jediným předpokladem pro použití algoritmu zpětného šíření chyby je, že aktivační funkce všech neuronů v síti musí být diferencovatelné. Cílem algoritmu backpropagation je minimalizovat chybovou funkci (někdy nazývaná také ztrátová funkce). Chybová funkce vyjadřuje

odchylku odezvy sítě od očekávaných výstupních hodnot. Výběr chybové funkce závisí na typu problému. V původním algoritmu backpropagation je chybová funkce vyjádřena jako polovina součtu kvadartických chyb:

$$E_p = \frac{1}{2}\sum_{j=1}^{m}(d_{pj} - o_{pj})^2. \tag{2.3}$$

Symbol m v této rovnici udává počet výstupních neuronů a p značí jeden prvek z trénovací množiny. Chyba je potom minimalizována pomocí metody gradientního sestupu. Minimalizace se provádí posunem váhového vektoru $\vec{w}$ ve směru záporného gradientu chybové funkce E_p, jak je naznačeno zde:

$$\Delta\vec{w} = -\mu\nabla E_p \tag{2.4}$$

$$\nabla E_p = \frac{\partial E_p}{\partial \vec{w}}.$$

Ve vzorci 2.4 ∇E_p značí gradient chyby E_p a μ je již zmíněný koeficient učení. Volba správného koeficientu μ je zde velmi důležitá, protože při nesprávném nastavení této hodnoty může dojít k problémům při učení. Koeficient učení totiž pomáhá upravovat velikost kroku, s jakým se váhový vektor posunuje ve směru záporného gradientu. Při zvolení příliš nízkého koeficientu učení se může stát, že neuronová síť bude konvergovat příliš pomalu a optimální řešení nebude nalezeno, naopak při příliš vysoké hodnotě koeficientu může být nejoptimálnějšímu řešení přeskočeno, gradientní sestup tak nikdy nedosáhne globálního minima a dojde k divergenci.

Vlastní gradientní sestup může být realizován pomocí několika přístupů. Nejpoužívanějším přístupem je tzv. stochastický gradientní sestup (SGD), kdy se z trénovací množiny náhodně vybírá pouze jeden prvek a změna vah se provádí ihned po jeho zpracování. Existují ovšem i metody dávkové, kdy se úpravy vah provádějí až po zpracování části nebo dokonce všech prvků z trénovací množiny. Stochastický gradientní sestup je nejvíce prospěšné volit pokud se očekává, že se na cestě ke správnému řešení nachází více lokálních minim, v opačném případě jsou vhodnější přístupy dávkové.

Pro samotnou změnu váhy i-tého vstupu j-tého neuronu ve vrstvě l pak platí vztah:

$$\Delta^l w_{ji} = \mu^l\delta_j{}^l x_i, \tag{2.5}$$

kde μ je koeficientu učení, ${}^l x_i$ je

$${}^l x_i = \frac{\partial^l u_j}{\partial^l w_{ji}} \tag{2.6}$$

a ${}^l\delta_j$ je derivace chyby podle vnitřního potenciálu neuronu definovaná jako

$${}^l\delta_j = -\frac{\partial E}{\partial^l y_j} * \frac{\partial^l y_j}{\partial^l u_j}. \tag{2.7}$$

Při odvození výpočtu derivace chyby podle vnitřního potenciálu ${}^l\delta_j$ se rozlišuje, zda se jedná o výpočet pro poslední vrstvu nebo některou s předcházejících vrstev. Pro výpočet derivace v poslední vrstvě L byl odvozen vzorec

$${}^L\delta_j = -\frac{\partial E}{\partial^L y_j} * \frac{\partial^L y_j}{\partial^L u_j} \tag{2.8}$$

a pro výpočet derivace předcházejících vrstev má ${}^{l-1}\delta_j$ tvar

$$ {}^{l-1}\delta_j = \sum_{k=1}^{n_l} (\partial^l w_{kj} {}^l\delta_k) \frac{\partial^{l-1} y_j}{\partial^{l-1} u_j}. \tag{2.9} $$

Průběh algoritmu backpropagation je následující. Prvním krokem algoritmu je výběr náhodného prvku p z trénovací množiny. Vstupní vektor $\vec{x}$, který reprezentuje tento prvek, se přiloží na vstup sítě. Počínaje první vrstvou postupně všechny vrstvy sítě vypočtou výstupy svých neuronů a je zjištěna odezva sítě. Pro každý výstupní neuron se pak spočítá jeho příspěvek k chybě pomocí chybové funkce 2.3 a také derivace chyby podle vnitřního potenciálu podle rovnice 2.8.

Poté se přistupuje k vlastní fázi zpětné propagace. Zde se postupně od konce neuronové sítě počínaje předposlední vrstvou spočítá derivace chyby každého jednotlivého neuronu podle jeho vnitřního potenciálu dle rovnice 2.9. Po provedení tohoto výpočtu musí dojít k dalšímu průchodu sítí, tentokrát opět v opačném směru od první vrstvy směrem k poslední. Při tomto posledním dopředném průchodu se na základě vypočtených derivací spočítá přírůstek vah pro každý neuron dle rovnice 2.5 a tento přírůstek je přičten původní hodnotě vah. Tímto způsobem je v síti proveden jeden krok učení. Algoritmus backpropagation pokračuje opět od prvního kroku tak dlouho, dokud nejsou splněny předem určené ukončovací podmínky, nebo dokud celková chyba sítě není menší než předem zvolená minimální chyba a nebo dokud není vyčerpán předem určený počet epoch učení.

Kapitola 3

Evoluční algoritmy

Tak jako neuronové sítě i evoluční algoritmy se snaží o napodobení biologických systémů, konkrétně o napodobení biologické evoluce podle Darwinovy teorie vývoje druhů a jejích různých rozšíření. Evoluční algoritmy založené na této teorii využívají myšlenku, že jedinci populace, kteří jsou nejzdatnější, přežívají a předávají svoji genetickou informaci jedincům následující generace, čímž dojde k postupnému zlepšování vlastností celé populace.

Podobně jako u neuronové sítě i evoluční algoritmy jsou vhodné pro řešení problémů, o nichž nemáme podrobné znalosti, nebo takových problémů, které jinými metodami řešitelné nejsou. Jejich hlavní využití leží v oblasti optimalizačních problémů, zejména u komplexních optimalizačních problémů s větším množstvím lokálních optim. Za optimalizační problém označujeme problém, kdy je nutné najít optimální řešení v množině možných kandidátních řešení. Pro některé optimalizační problémy sice existují přesné algoritmy, kterými je možné problém vyřešit, ale často jsou časově a výpočetně náročné. Zde přichází na řadu evoluční algoritmy, které mohou být schopné poměrně rychle poskytnout vyhovující řešení. Snížení výpočetního času má za cenu to, že nalezené řešení nemusí být optimální.

Tato kapitola obsahuje přehled problematiky evolučních algoritmů, se zaměřením na genetické programování. Zvláštní pozornost je zde věnována především variantě genetického programování zvané kartézské genetické programování (CGP), jelikož bude v práci využíváno. Pokud není uvedeno jinak, všechny informace v této kapitole byly čerpány z [5], [17], [22] a [24], informace v kapitole 3.6 o kartézském genetickém programování byly pak navíc čerpány z [13].

3.1 Princip evolučních algoritmů

Evoluční algoritmy patří mezi stochastické prohledávací algoritmy. Do kategorie evolučních algoritmů zařazujeme genetické algoritmy, evoluční programování, evoluční strategie a v neposlední řadě také genetické programování. Všechny tyto metody mají společnou terminologii a všechny pracují na principu náhodných změn navrhovaného řešení.

Hledání optimálního řešení je v evolučních algoritmech založeno na tvorbě populací. Jedinci populace představují jednotlivá řešení daného problému. Každého jedince populace, tedy každé řešení problému, je nutné ohodnotit pomocí vhodné hodnotící funkce nazývané fitness funkce. Zpravidla čím vyšší je hodnota funkce fitness, tím vyšší je kvalita řešení.

Dalším krokem je výběr vhodných jedinců z populace, kteří se využijí pro vytvoření množiny potomků. Tito vybraní jedinci se označují jako rodiče. Vybírají se na základě fitness a pro jejich výběr existuje řada selekčních mechanismů. Selekční mechanismy se

snaží především o to, aby byli pro výběr rodičů upřednostněni jedinci s vyšší fitness, ale správný selekční mechanismus zároveň zajistí, že nově vzniklá populace potomků bude dostatečně různorodá. Různé selekční mechanismy se taktéž využívají k výběru jedinců do nové populace. Nová populace pak může být složena pouze z potomků a nebo může obsahovat i jedince z předchozí populace (včetně rodičů).

Hodnocení jedinců a tvorba nové populace se cyklicky opakuje až do určitého počtu iterací nebo dokud nejsou splněny jiné, předem stanovené ukončovací podmínky. Jedna iterace algoritmu se v souladu s biologickou terminologií označuje také jako generace. Jako řešení problému je pak vrácen jedinec populace s nejvyšší dosaženou hodnotou fitness. Obecná konstrukce evolučního algoritmu je pseudokódem popsána v algoritmu 1 převzatého z [5].

Algoritmus 1 Základní konstrukce evolučních algoritmů.

```
i = 0
Inicializuj první populaci P_0 o velikosti M
Ohodnoť každého jedince v populaci P_i pomocí fitness funkce
repeat
   Vyber rodiče z P_i
   Vytvoř K potomků a ohodnoť je
   Vyber jedince do nové populace P_{i+1}
   i = i + 1
until Nejsou splněny ukončovací podmínky
return  Jedinec s nejlepší hodnotou fitness
```

Velikost populace M bývá konstantní po celou dobu provádění programu, stejně tak i počet potomků K, kteří se vytvoří v rámci jedné iterace. Populace bývá běžně inicializována náhodnými jedinci. Vytvořením prvotní populace z náhodných jedinců je zajištěno to, že prohledávání začne na různých místech stavového prostoru, a zároveň je vyřešen jeden z problémů standardních gradientních metod pro hledání řešení, u kterých existuje větší riziko, že budou konvergovat k bodu, který se nachází nejblíže počátečnímu řešení.

Pro funkčnost evolučních algoritmů je velmi důležitý výběr funkce fitness. Funkce fitness udává, co je to zlepšení v kontextu daného problému, a slouží k vytváření selekčního tlaku na populaci. Pokud je vybrána správně, evoluce vede k postupném zkvalitňování populace.

3.1.1 Reprezentace řešení

Kromě fitness funkce ovlivňuje kvalitu evolučního algoritmu i způsob zakódování problému. Pro zakódovaného jedince se v kontextu evolučních algoritmů používá název chromozom případně genotyp. Každý takový genotyp je řetězcem, který je složen z jednotlivých genů. Jeden genotyp kóduje jeden nebo více fenotypů, tedy kandidátů na řešení úlohy.

Chromozom (genotyp) může mít proměnnou délku a je reprezentován různě podle toho, co je zakódováno v jeho genech. Může se jednat například o binární reprezentaci chromozomu, kde každý gen nabývá hodnoty 1 nebo 0. Dále je možné použít i celočíselnou nebo reálnou reprezentaci chromozomu, kde hodnoty genů jsou celá popř. reálná čísla, nebo jinou reprezentaci, například ve formě grafů.

Aby mohl být genotyp ohodnocen, musí být nejprve transformován na odpovídající fenotyp pomocí vhodného dekódovacího procesu. Dekódovací proces by měl být snadno

proveditelný. Ohodnocení fenotypu získaného po dekódování pak udává fitness příslušného genotypu.

3.1.2 Selekční mechanismy

Pro výběr rodičů jsou využívány selekční mechanismy. Nejjednodušší formou selekce je tzv. deterministická selekce, kdy je pro reprodukci vybrán určitý počet jedinců s nejvyšší hodnotou fitness.

Další, velmi využívanou metodou selekce je proporcionální selekce, kde platí, že čím vyšší je fitness jedince, tím vyšší je pravděpodobnost jeho výběru. Proporcionální selekce je často implementována pomocí algoritmu tzv. rulety, kde je jednotková úsečka rozdělena na úseky podle velikosti normalizovaných hodnot fitness jedinců v populaci a poté je náhodně vybráno číslo z intervalu (0, 1), které identifikuje na úsečce odpovídajícího jedince. Problémem ruletového výběru je ovšem fakt, že populace může obsahovat jednoho jedince s velmi vysokou fitness oproti ostatním a ruletový algoritmus pak téměř vždy vybere stejného jedince, což zpravidla vede k degeneraci populace. Tento problém lze řešit selekcí podle pořadí, kdy jsou jedinci v populaci seřazeni podle fitness a pravděpodobnost výběru jedince je pak přímo úměrná jeho pořadí.

Další častou metodou výběru rodičů je výběr turnajový, kdy je z populace náhodně vybrán určitý počet jedinců a z nich se vybere dle fitness nejlepší z nich. Turnaj se opakuje tolikrát, dokud není vytvořen potřebný počet potomků.

Co se týká samotného výběru jedinců do nové populace, i zde existuje několik možností. Stejně jako u výběru rodičů lze využít některý ze selekčních mechanismů. Je ale nutné učinit rozhodnutí, zda budou do nové populace zařazeni pouze jedinci pouze z řad potomků, nebo i členů předchozí populace (tzv. překrývání populací). Zde se někdy zavádí i koncept tzv. elitismu, kdy se daný počet nejlepších jedinců z předchozí populace vždy dostane do populace nové.

3.1.3 Vytváření jedinců pro novou generaci

Samotná tvorba nových jedinců je založena na aplikaci genetických operátorů, konkrétně křížení a mutace. V tomto kroku dochází k předávání informace mezi rodičem a potomkem a k tvorbě nových kandidátních řešení problému.

Mutace

Operátor mutace se aplikuje pouze na jednoho jedince a vytváří z něj jedince mutovaného. Při mutaci dochází k procházení jednotlivých genů jedince a s určitou pravděpodobností se mění jejich hodnota. Typickým příkladem takové změny je v binární reprezentaci jedince změna genu s hodnotou 0 na 1 a opačně. Ukázka mutace pro binární reprezentaci řešení je na obrázku 3.1.

Obrázek 3.1: Ukázka mutace při binární reprezentaci jedince. Převzato z: [5].

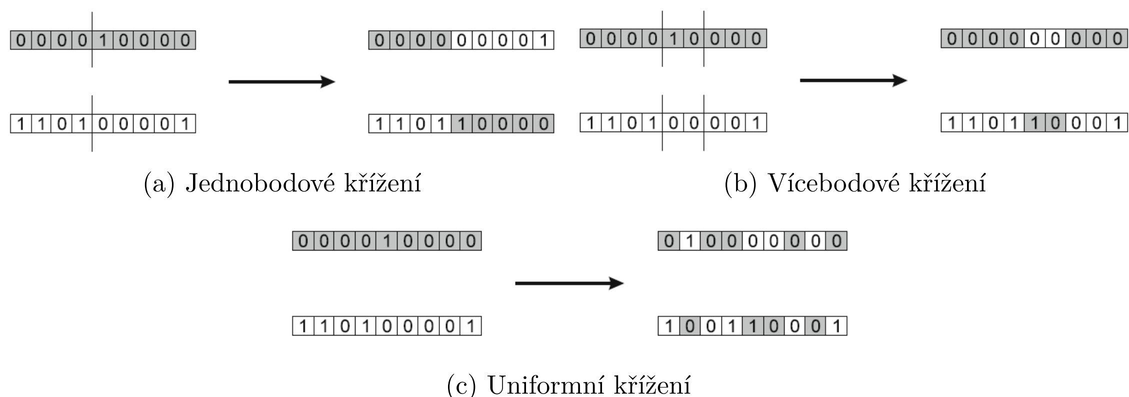
Pravděpodobnost mutace se volí velmi malá, což umožňuje velmi jemné změny řešení po malých krocích. Mutace tak generuje potomky velmi podobné svým rodičům a proto je díky ní možné najít takové jedince, kteří lépe vyhovují okolním podmínkám. Z hlediska

prohledávání stavového prostoru jde tak o operátor sloužící spíše k lokálnímu prohledávání okolí rodiče.

Křížení

Operátor křížení je naopak oproti mutaci nutné aplikovat na dva či více jedinců. Vychází z konceptu pohlavního rozmnožování v přírodě a také proto se často provádí právě nad dvěma rodiči. Dochází při něm ke vzájemné výměně části genů mezi rodičovskými jedinci.

Běžně používaným mechanismem křížení je křížení jednobodové či vícebodové. V případě jednobodového křížení mezi dvěma jedinci se nejprve náhodně stanoví bod křížení. Poté se z původních rodičů vytvoří potomek tak, že dojde k prohození jejich genů nacházejících se za tímto bodem křížení. Vícebodové křížení se od jednobodového liší pouze v tom, že stanoveno několik bodů křížení a prohození genů mezi rodiči je provedeno za každým tímto bodem. Další variantou křížení je křížení uniformní, kdy je pro každý gen rozhodnuto, jestli bude mezi rodiči prohozen nebo ne. Tyto mechanismy křížení jsou zobrazeny na obrázku 3.2. Při křížení dochází ke vzniku potomků, kteří se mnohem více liší od svých rodičů než v případě mutace, a jedná se zde oproti mutaci spíše o globální prohledávání.



Obrázek 3.2: Ukázka křížení. Převzato z: [5].

3.2 Genetické algoritmy

Jednou z podmnožin evolučních algoritmů jsou genetické algoritmy. Jedná se o nejznámější typ evolučních algoritmů. Jejich struktura v základní podobě přesně odpovídá struktuře evolučního algoritmu, jaká byla popsána v pseudokódu 1.

Základní podobu jednoduchého genetického algoritmu (Simple genetic algorithm, SGA) popsal ve svojí práci Goldberg. Tato základní varianta využívá binární zakódování problému, jednobodové křížení v rámci páru jedinců, mutaci ve formě inverze bitu, výběr rodičů je realizován proporcionální selekcí implementovanou pomocí algoritmu rulety a algoritmus tak pracuje následovně. Z přecházející populace o pevně dané velikosti se vybere počet rodičů stejný jako je velikost generace. Náhodným zamícháním těchto rodičů se vytvoří páry. Počet těchto párů je také stejný jako velikost populace. Mezi rodiči se provede operace křížení, čímž se vytvoří potomci. Tito potomci jsou s určitou pravděpodobností podrobeni mutaci. Tato nově vytvořená populace dále postupuje do další iterace, dokud není dosaženo řešení. V základní verzi algoritmu je předcházející populace nahrazena novou kompletně,

principy elitismu a překrývání populací byly zavedeny až v pozdějších modifikacích tohoto algoritmu pro zrychlení konvergence.

Jednoduché genetické algoritmy jsou dnes stále používané pro řešení jednodušší problémů, kde postačí binární reprezentace. Jejich zkoumání pomohlo k vzniku dalších modifikací genetického algoritmu například přidáním překrývání populací nebo vytvořením dalších metod křížení. Dnes genetické algoritmy běžně využívají vícebodové i uniformní křížení a selekce rodičů se provádí některou z vyváženější metod, například pomocí turnaje. Přispěly také k zjištění, že je vhodné zavést jiné formy reprezentace než jsou binární řetězce, a tak je pro genetické algoritmy dnes běžná i celočíselná reprezentace či reprezentace řešení reálnými čísly, umožňující řešit větší škálu problémů než tomu bylo v době vzniku těchto algoritmů.

3.3 Evoluční strategie

Do evolučních algoritmů dále řadíme evoluční strategie. Evoluční strategie byly vytvořeny v Německu a jejich autory jsou Rechenberg a Schwefel. Používají se pro optimalizaci vektorů reálných čísel. Hlavní operací v evolučních strategiích je mutace, která zde probíhá přičtením náhodného čísla ke každému prvku rodičovského vektoru.

Základní varianta evoluční strategie se označuje jako ES(1+1), kde je z jednoho rodiče mutací vytvořen jeden potomek, který je přijat pouze tehdy, pokud má vyšší fitness než rodič. Další alternativa základní evoluční strategie má označení ES(1,1), zde je rodič nahrazen potomkem vždy. Potomek vzniká mutací, při které dojde k vygenerování náhodných čísel z Gaussova rozložení, která jsou přičtena k jednotlivým prvkům rodičovského vektoru. Využívá se Gaussovo rozložení s nulovou střední hodnotou a rozptylem σ, kde σ určuje velikost kroku mutace. Jedním z průlomů v oblasti evolučních strategií bylo nalezením mechanismu pro úpravu velikosti kroku mutace a vznik tzv. pravidla jedné pětiny. Parametr σ je v něm modifikován tak, aby přibližně jedna pětina potomků měla vyšší fitness než jejich rodiče. V případě, že je výrazně více potomků lepších nebo naopak horších, dojde ke zvýšení nebo naopak snížení vlivu mutace.

Obě základní varianty evoluční strategie využívaly pouze populaci o velikosti jedna. V dnešní době se ovšem používají populace obsahující více jedinců a pracuje se se dvěma variantami reprodukce označovanými ES($\mu+\lambda$) a ES(μ,λ). Parametr μ značí počet jedinců v populaci a parametr λ počet potomků vygenerovaných v jednom cyklu, který se volí větší než počet rodičů. Rozdíl mezi těmito variantami spočívá v tom, že v ES($\mu+\lambda$) jsou nejlepší jedinci do nové populace vybíráni jak z množiny potomků tak rodičů, zatímco v ES(μ,λ) jsou nejlepší jedinci vybíráni pouze z množiny potomků. Za více používanou se považuje ES(μ,λ), protože oproti ES($\mu+\lambda$) díky odstranění rodičů snadněji opouští lokální optima a neuchovává starší řešení, která nejsou schopná sledovat směr optimálního řešení.

Evoluční strategie obsahující více jedinců v populaci vedly k vytvoření sofistikovanějších mechanismů pro adaptaci kroku a k vytvoření konceptu tzv. samoadptace parametrů. Ta spočívá v tom, že některé parametry jsou zakódovány v chromozomech a vyvíjí se společně s vývojem řešení. Samoadaptace je dnes běžnou součástí moderních evolučních strategií. Oproti genetickým algoritmům aplikují evoluční strategie větší množství selekčního tlaku a jsou mnohem agresivnější než jednoduché genetické algoritmy.

3.4 Evoluční programování

Do evolučních algoritmů řadíme i evoluční programovávaní, jehož autorem je Lawrence Fogel. Původním účelem evolučního programování bylo vyvinout umělou inteligenci a simulovat evoluci jako proces učení. Jedinci zde byly reprezentováni konečnými automaty.

Dnes, ačkoli evoluční programování vzniklo nezávisle, v podstatě splynulo s evolučními strategiemi díky společným rysům. Stejně jako evoluční strategie i evoluční programování začalo využívat reprezentaci pomocí reálných čísel, případně úplně jinou aplikačně specifikou reprezentaci vyhovující danému problému. Mutace je prováděna také jako u evolučních strategií za pomoci Gaussovského rozdělení. I v evolučním programování se aplikuje koncept samoadaptace. Rozdíl je možné nalézt až v užívaném selekčním mechanismu. V evolučním programování každý rodič vygeneruje právě jednoho potomka. Populace potomků a rodičů je potom sloučena do jedné populace a v ní poté jedinci soupeří turnajovou selekcí o přežití. Křížení se v evolučním programování nepoužívá, protože každý jedinec zde představuje odlišný druh, které se mezi sebou nemísí.

3.5 Genetické programování

Poslední variantou evolučního algoritmu je genetické programování. Oproti ostatním variantám je genetické programování novější přístup, byl totiž přestaven až na konci 80. let minulého století a jeho rozvoj se nejvíce zasloužil John Koza.

Ačkoliv vlastní algoritmus genetického programování pracuje stále na stejném principu představeném v algoritmu 1, použití genetického programování se od ostatních evolučních algoritmů liší. Nevyužívá se totiž k řešení optimalizačních problémů nýbrž k automatickému vyvíjení celých programů z elementárních funkcí. Cílem genetického programování je tedy z několika základních funkcí, operátorů, konstant a proměnných vyvinout program, který bude co nejlépe popisovat chování zadaného problému.

Genetické programování je jedním z nástrojů pro symbolickou regresi. Symbolická regrese je proces, který se zabývá hledáním funkce, která nejlépe aproximuje zadaná data. Na rozdíl od klasické regrese, která se snaží optimalizovat parametry předem dané funkce, odvozuje symbolická regrese z dat nejen parametry, ale i funkci samotnou. Pro nalezení této funkce symbolická regrese obecně nevyžaduje žádné apriorní informace o vztahu mezi vstupy a výstupy. Předpokladem pro symbolickou regresi je, že výstupní hodnota závisí na vstupních hodnotách. Genetické programování umožňuje symbolickou regresi řešit využitím principu evolučních algoritmů. [23]

Protože řešením genetického programování je samotný program, tedy spustitelná struktura, musí být použit vhodný způsob její reprezentace. Právě reprezentace řešení je v genetickém programování specifická. Program nelze jednoduše reprezentovat chromozomem ve tvaru řetězce znaků o pevné délce, jak tomu bylo například u genetických algoritmů, ale výsledné struktury musejí mít proměnnou délku. V genetickém programování se proto nejčastěji používá reprezentace pomocí stromů.

Stromové struktury ztvárňují programy do podoby syntaktických stromů skládajících se z terminálů a elementárních funkcí. Terminály představují vstupy do programu a konstanty, množina funkcí potom obsahuje předem daný počet elementárních funkcí. Tyto elementární funkce by měly být jednoduché a především rychle vyčíslitelné. Příkladem takových funkcí mohou být funkce jako součet, rozdíl, násobení, sinus, tangens a podobně.

Funkce jsou ve stromu reprezentovány uzly a terminální symboly tvoří listy stromu. Výsledná funkce je pak tvořena uzly stromu podle jejich přirozeného pořadí ve stromu, tedy jako při průchodu inorder.

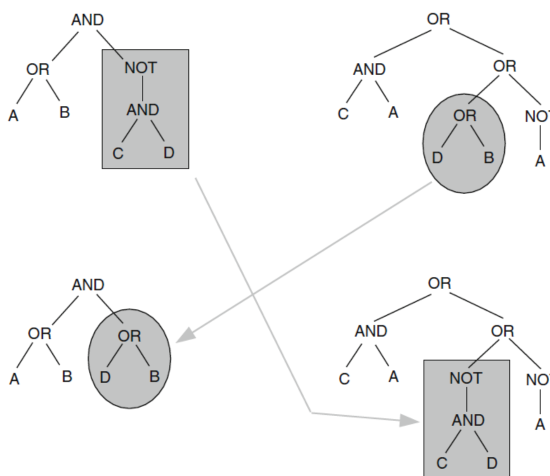
Stromová struktura umožňuje modelovat programy poměrně efektivně, avšak její zavedení si vynucuje použití odlišných způsobů inicializace populace než je pouhé náhodné naplnění řetězce hodnotami. Pro tvorbu populací se proto nejčastěji používají metody Grow, Full nebo Ramped Half-and-Half.

Při využití metody Grow jsou uzly v hloubce menší než je maximální hloubka stromu vybírány náhodně jak z množiny funkcí tak z terminálů. Metoda Grow začíná umístěním náhodné funkce jako kořene stromu, dále jsou vytvořeny uzly tvořící potomky kořene a jsou naplněny buď funkcí nebo terminálem. V případě že byla do uzlu vybrána funkce, provede se rekurzivní volání metody Grow na daný uzel, které se opakuje dokud není dosazen terminální uzel nebo dokud nebylo dosaženo maximální hloubky stromu, kde už mohou být dosazeny pouze terminály.

Metoda Full nepovoluje umístění terminálů do uzlů, které mají menší hloubku než maximální hloubka stromu. Výsledkem této metody je strom, jehož všechny větve dosahují maximální hloubky, na rozdíl od metody Grow, kde se délka jednotlivých větví liší.

Protože metody Grow a Full omezují různorodost vzniklých stromů, byla zavedena metoda Ramped Half-and-Half, která je kombinací metod předešlých. Při vytváření populace je vytvořena nejprve část populace o velikost 1/d (kde d je maximální hloubka stromu), která se inicializuje stromy o maximální hloubce 1, další 1/d část populace stromy o maximální hloubce 2 a tak dále až do maximální hloubky d. V každém těchto zlomků populace je polovina jedinců vytvořena pomocí metody Grow a druhá pomocí metod Full. Tento přístup umožňuje vznik počáteční populace s větší mírou diverzity mezi jedinci.

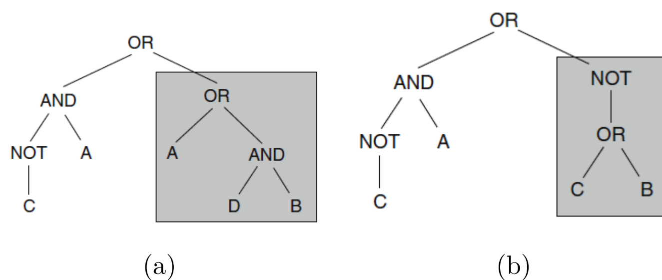
V genetickém programování se také využívají operátory křížení a mutace. Pro křížení se vybírají dva rodiče, v nichž je dále vybrán jeden uzel jako bod křížení. Část stromu počínaje bodem křížení se oddělí jako podstrom a potomci jsou vytvořeni výměnou těchto podstromů mezi rodiči. Příklad křížení je na obrázku 3.3.



Obrázek 3.3: Ukázka křížení při použití stromové reprezentace. Převzato z: [22].

Mutace probíhá náhodným vybráním jednoho uzlu. Počínaje tímto vybraným uzlem se část stromu odstraní a je nahrazena jiným, náhodně vygenerovaných podstromem (viz

obrázek 3.4). Tvorba náhodného nahrazujícího podstromu je omezena maximální hloubkou stromu, ovšem jeho velikost může být i větší než původní velikost rodiče před mutací.



Obrázek 3.4: Ukázka jedince před mutací (a) a po mutaci (b). Převzato z: [22].

Zjišťování hodnoty fitness probíhá při genetickém programování spouštěním vyvíjených programů a zhodnocením výsledků získaných jejich prostřednictvím.

Při řešení úlohy pomocí genetického programování je tedy potřeba kromě běžných parametrů genetického programování jako je velikost populace, počet generací atd., definovat také množinu terminálů a funkcí.

V kontextu genetického programování je nutné zmínit, že při aplikaci evolučních algoritmů na program reprezentovaný stromem může dojít k jevu, který nazýváme bloat. Tento jev vzniká tím, že v průběhu generací může narůstat velikost chromozomu bez zvyšování jeho fitness. V takovém chromozomu je reprezentováno velké množství částí programu, které nepřispívají zlepšení programu a nepřibližují ho k řešení a při následné optimalizaci kódu by je bylo možné odstranit. Bloat může způsobovat vážný problém, protože velké množství takových redundantních operací může mít za následek zbytečný nárůst výpočetního času vyvíjeného programu a v nejhorším případě může dojít k nedostatku paměti při jeho vyvíjení. Navíc program ovlivněný bloatem je často těžce čitelný. Proto jsou pro genetickém programování vyvíjeny různé techniky, které se snaží bloatu zabránit a zvýšit tak kvalitu genetickým programováním vyvíjených programů.

Kromě stromové reprezentace, která byla výchozím bodem pro genetické programovaní, jsou dalšími častými reprezentacemi reprezentace lineární či grafová. Grafová reprezentace je využívaná například v kartézském genetickém programování, což je specifická varianta genetického programování, která bude více popsána v následující kapitole 3.6. Kartézské genetické programování je zároveň jediná forma genetického programování, která netrpí fenoménem bloatu, ačkoli dosud nebylo přesně objasněno, proč tomu tak je.

3.6 Kartézské genetické programování

Kartézské genetické programování (zkráceně CGP) je forma genetického programování využívající pro reprezentaci jedinců orientované grafy. Tato metoda byla poprvé představena v roce 1999 J. F. Millerem a zpočátku našla využití především v oblasti návrhu kombinačních obvodů. Dnes se ovšem kromě návrhu obvodů využívá ve větším množství oborů, například i pro tvorbu neuronových sítí nebo symbolickou regresi.

3.6.1 Reprezentace jedinců v CGP

Pro reprezentaci jedinců jsou v CGP využívány orientované acyklické grafy. Tyto grafy jsou modelovány pomocí dvou-dimenzionální mřížky, kde jednotlivé body v mřížce představují

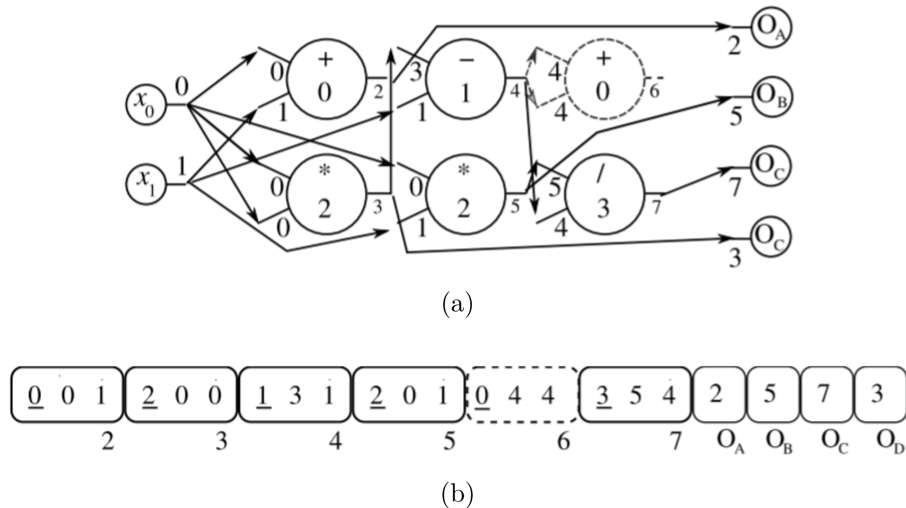
jednotlivé výpočetní uzly grafu. Velikost grafu je v CGP omezena počtem sloupců n_c a počtem řádků n_r. Součin těchto parametrů $L_n = n_c * n_r$ zároveň určuje maximální počet výpočetních uzlů v mřížce.

Každý uzel mřížky reprezentuje jednu funkci a je zakódován v chromozomu pomocí několika (obvykle celočíselných) hodnot. První z těchto hodnot je adresa daným uzlem reprezentované funkce ve vyhledávací tabulce funkcí. Další zakódované hodnoty jsou hodnoty určující místa, ze kterých funkce v uzlu získává své vstupy. Počet takto zakódovaných hodnot závisí na aritě (počtu vstupů) použitých funkcí. Vstupní hodnoty mohou uzly získávat pouze z uzlů z předcházejících sloupců mřížky nebo ze samotných vstupů programu, nikoli ze stejného sloupce z jakého samy pocházejí nebo sloupců následujících.

Vstupy programu a uzly mřížky jsou pro identifikaci adresovány pomocí číselných hodnot. Vstupy dostávají čísla od 0 do $n_i - 1$, kde n_i je počet vstupů. Následující uzly jsou pak adresovány sloupec po sloupci od n_i do $n_i + L_n - 1$.

Výstupy programu mohou být v grafu reprezentovány pomocí výstupních uzlů označených O_i. Výstupní uzly už neobsahují funkce, ale pouze hodnotu uzlu, ze kterého má být přečten výsledek. V chromozomu jsou pak výstupy, jejichž počet je značen n_o, zakódovány přidáním n_o hodnot obsahujících adresy uzlů s výsledkem na konec chromozomu. Ukázka grafu jedince a jemu odpovídajícímu chromozomu je na obrázku 3.5.

Díky tomuto kódování je dosaženo toho, že délka chromozomu je vždy konstantní, ale velikost zakódovaného programu je variabilní, protože některé uzly mřížky v závislosti na propojení nemusí být pro výsledný program vůbec využity. Uzly které využité jsou, pak označujeme jako aktivní. Pouze aktivní uzly mají podíl na fenotypu daného chromozomu. Aktivní uzly se mohou změnit na neaktivní a naopak pomocí mutace, což je také jediný standardní operátor, který se v CGP využívá, protože křížení se neukázalo být v kontextu CGP prospěšné.



Obrázek 3.5: Ukázka jedince reprezentovaného grafem (a) a chromozomu, do kterého je zakódován (b). Čárkovaně jsou označeny uzly a části chromozomu nepotřebné pro výpočet. Převzato z: [13].

Kromě nastavení počtu sloupců a řádků se v CGP vyskytuje další důležitý parametr l nazývaný L-back, určující míru propojitelnosti uzlů. Tento parametr udává z kolika před-

cházejících sloupců může uzel vzít hodnotu na svůj vstup, tedy například v případě $l = 1$ může být uzel propojen pouze s uzly v přecházející vrstvě.

3.6.2 Operátor mutace v CGP

Evoluce řešení je v CGP řízena pomocí operátoru mutace. Při mutaci je v chromozomu vybrán gen, jehož hodnota je potom změněna na náhodně vygenerovanou hodnotu novou. Musí být ovšem zajištěno, že nová hodnota genu je legální pro danou instanci CGP. Je-li pro mutaci vybrána hodnota určující adresu funkce, musí být nová hodnota validní adresa jiné funkce z vyhledávací tabulky. Pokud je vybrána pro mutaci hodnota značící propojení s jiným uzlem, pak musí být nová hodnota po mutaci validní adresa jiného předcházejícího uzlu, který je s respektem k parametru L-back vzdálen nejdále o l sloupců. V případě mutace genů představujících výstupní uzly jsou pak validními hodnotami adresy výstupu kteréhokoli uzlu nebo hodnoty vstupů vyvíjeného programu. Počet genů, které budou podrobeny mutaci v rámci jednoho cyklu algoritmu, je nastavitelným parametrem.

Mutace má velký vliv na fenotyp. I malá změna v genotypu může vyvolat velkou změnu fenotypu. V CGP ale běžně dochází k tomu, že mutace je aplikována na neaktivní uzly a ke změně fenotypu nějakou dobu nemusí dojít. Takovou mutaci, která nemění fitness jedince, nazýváme neutrální mutace. Mutaci, při které dojde ke zvýšení fitness, nazýváme mutace adaptivní. Série neutrálních mutací následovaná mutací adaptivní může může mít za následek aktivaci doposud neaktivních uzlů, jejichž mutace nějakou dobu probíhala na pozadí, a tím způsobit velkou změnu fenotypu. Díky této skutečnosti mutace způsobí velký posun v prohledávání stavového prostoru a v CGP tak nahrazuje operátor křížení, používaný v běžných evolučních algoritmech pro výraznější změny kandidátních jedinců.

3.6.3 Řídicí evoluční algoritmus v CGP

Evoluční algoritmus prohledávání používaný v CGP je podobný variantě evolučních strategií ES$(1 + \lambda)$. Parametr λ je většinou volen na hodnotu 4. Populace se skládá z $1 + \lambda$ jedinců. Tvorba nové populace probíhá tak, že se vybere jeden jedinec z původní populace s nejlepší hodnotou fitness. Ten se vloží do nové populace a zároveň je použit jako rodič pro dalších λ jedinců, kteří z něj vzniknou aplikací operátoru mutace a jsou také vloženi do nové populace společně s rodičem.

V CGP se může stát, že více jedinců v populaci má stejnou hodnotu fitness, která je v dané populaci nejlepší. Pak je nutné vyřešit otázku, kterého z nejlepších jedinců vybrat jako rodiče pro další členy nové populace. V CGP je tento problém řešen tak, že se jako rodič použije jedinec, který nebyl rodičem v předchozí generaci.

Celý prohledávací algoritmus pracuje následujícím způsobem. Nejprve je vygenerováno $1 + \lambda$ náhodných jedinců, kteří tvoří počáteční populaci. Jedinci populace jsou poté ohodnoceni pomocí vhodné fitness funkce a na základě výsledku je vybrán nejlepší jedinec, který bude dál použit jako rodič. Z rodiče je pak vygenerováno λ potomků pomocí mutace. Rodič a jeho potomci spolu vytvoří novou populaci a algoritmus znova pokračuje opět jejím ohodnocením a výběrem rodiče, dokud není nalezeno řešení nebo do splnění maximálního počtu generací či jiné ukončovací podmínky. Podoba algoritmu je popsána v následujícím pseudokódu 2.

Algoritmus 2 Prohledávací algoritmus používaný pro CGP.

```
Inicializuj první populaci o velikosti 1+λ
Ohodnoť všechny jedince populace pomocí funkce fitness
while Řešení nebylo nalezeno nebo nejsou splněny ukončovací podmínky do
  Vyber nejlepšího jedince v populaci jako rodiče
  for (i = 0, i < λ, i++) do
    Aplikuj operátor mutace na rodiče a vytvoř potomka C_i
  end for
  Vytvoř novou populaci obsahující rodiče a všechny jeho potomky C_i
  Ohodnoť všechny jedince populace pomocí funkce fitness
end while
```

Kapitola 4

Metody pro návrh neuronových sítí pomocí evolučních algoritmů

Máme-li problém a rozhodneme se ho řešit pomocí neuronové sítě, je nutné navrhnout architekturu sítě, tedy její topologii, propojení mezi neurony a tvar přenosové funkce jednotlivých neuronů. Architektura sítě má velký vliv na úspěšnost řešení problému, které je síť schopná poskytnout. Běžně se o architektuře sítě rozhoduje experimentálně metodou pokusů a omylů nebo na základě předchozích zkušeností a neexistují žádné zaručené postupy, jak síť navrhnout vždy správně. Je zřejmé, že tento přístup není moc vhodný a hledání správné architektury experimentálně může být zbytečně zdlouhavé. Proto se brzy po objevení neuronových sítí objevily snahy o nalezení metod, které by se daly použít k návrhu architektury sítě. Tyto metody bývají často založeny na kombinaci evolučních algoritmů a neuronových sítí, tzv. neuroevoluci.

Tato kapitola proto obsahuje popis metod vytvořených pro návrh neuronových sítí. Aby bylo možné neuronovou síť postupně vyvíjet pomocí technik evolučních algoritmů, musí být možné síť vhodně reprezentovat. Pro reprezentaci neuronové sítě se používají dva typy kódování: přímé a nepřímé. V následujíc podkapitole jsou proto nejprve vysvětleny rozdíly mezi přímým a nepřímým kódování neuronové sítě. Kapitola poté pokračuje přehledem dvou přístupů k evolučnímu návrhu neuronových sítí v podobě low-level přístupu simulujícího buněčnou chemii a high-level přístupu využívajícího gramatiky [21]. Následuje bližší pohled na metodu NEAT a její variantu Hyper-NEAT a nakonec popis metody využívající CGP navržený J. F. Millerem [14]. Informace v podkapitole 4.1 byly převzaty z [24] a [25], další zdroje informací jsou uváděny přímo v příslušných podkapitolách.

4.1 Přímé a nepřímé kódování

Při kódování architektury neuronové sítě pro evoluci do chromozomu je nutné se rozhodnout, kolik informací bude v chromozomu zakódováno. Pokud je v chromozomu zakódována celá architektura sítě včetně vah, pak mluvíme o tzv. *přímém kódování*. Při přímém kódování je už je buď předem rozhodnuto o topologii sítě a budou se vyvíjet pomocí evoluce pouze váhy (učení sítě pomocí evoluce), nebo mohou být při evoluci prováděny změny i v zakódované topologii. Při zakódování topologie je ale nutné mít na paměti, že s rostoucí velikostí neuronové sítě poroste úměrně i délka chromozomu, který ji kóduje, a pro velké sítě může být výpočetní náročnost až příliš vysoká. Přímé kódování proto nebývá považováno za dobře

škálovatelné, ovšem v poslední době se ukázalo jako úspěšné i při návrhu velmi složitých konvolučních sítí viz [12].

Kromě přímého kódování existuje i *kódování nepřímé*, kde jsou v chromozomu zakódovány jen některé důležité rysy sítě, a architektura sítě je pak specifikována jiným způsobem, například pomocí speciálních pravidel. V případě nepřímého kódování se začíná s malou neuronovou sítí (může být složena i z jednoho neuronu) a z tohoto počátečního bodu je síť dále vyvíjena přidáváním či odebíráním neuronů či celých vrstev sítě a spojení mezi nimi na základě pravidel.

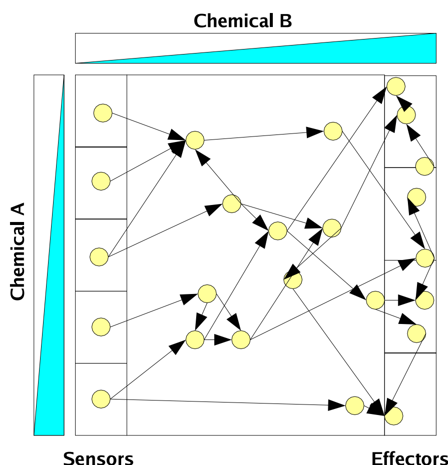
4.2 Přístupy založené na buněčné chemii

Metody postavené na buněčné chemii vycházejí z práce Alana Turinga, který popsal matematický model difúze a reakce, které zapříčiňují vznik v přírodě se vyskytujících vzorů. Motivací k použití tohoto přístupy je popsání procesů biologického vývoje na co nejnižší úrovni abstrakce a tím získání přirozených a komplexních řešení. [21]

Cangelosi a kol. [4] navrhl přístup snažící se zjednodušeně popsat přirozený vývoj pomocí růstu a větvení axonů společně s dělením a posunem buněk.

Rust a kol. přišel na přístup kódující v genomu vývojové parametry, které ovlivňují momenty větvení dendritů a interakci s atraktanty umístěnými v prostoru a vedou ke vzniku neuronů požadovaných vlastností [16].

Návrh představený v Balaam [2] popisuje dvou-dimenzionální prostředí s chemickým gradientem v každé dimenzi a rozděluje úlohy neuronů podle jejich umístění v prostoru na vstupní, skryté či výstupní, jak je znázorněno na obrázku 4.1. Genom neuronů se skládá fakticky ze sedmi vektorů funkcí, které berou za své vstupy úrovně chemikálií a potenciál neuronu daný rodičovským neuronem. Funkce slouží k výpočtu parametrů biasu, časové konstanty, energie, inkrementu růstu, směru růstu, vzdálenosti růstu a váhy nového spojení. Tyto parametry jsou dále v každém časovém kroku využity k vyhodnocení tvorby nových neuronů a jejich umístění a spojení.



Obrázek 4.1: Znázornění dvoudimenzionálního prostředí s chemickými gradienty a rozdělením úlohy neuronů podle jejich umístění. Převzato z: [2].

4.3 Přístupy využívající gramatiky

Gramatické metody využívají množiny přepisovacích pravidel a staví na práci Aristida Lindenmayera a jeho L-systému, který vznikl pro popis růstů rostlin. Kromě kontextových či bezkontextových variant gramatik ale existují i varianty využívající stromy instrukcí nebo orientované grafy. [21]

Gruau [7] navrhl přístup využívající grafové gramatiky k vytváření modulárních neuronových sítí, které se skládají z menších neuronových sítí pro řešení jednotlivých částí problému. K reprezentaci těchto řešení se rozhodl použít nepřímé buněčné kódování, u kterého prokázal, že i přes vyšší výpočetní náročnost než u přímého kódování dosahuje lepších výsledků bez nutnosti manuálního nastavení parametrů sítě [6].

Výhody nepřímého kódování dále potvrdil i Kitano [10] ve své práci inspirované L-systémy, ve které využil přepisovací pravidla ke vzniku matice sousednosti tvořící neuronovou síť a později Boers a Kuiper [3], kteří se taktéž rozhodli využít L-systému k vytvoření modulární neuronové sítě.

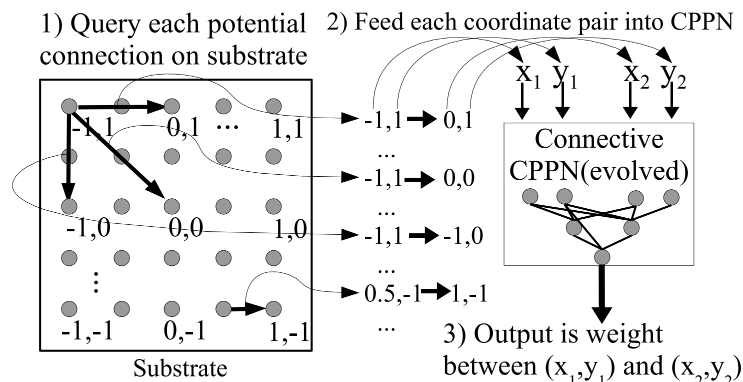
4.4 Metody NEAT a Hyper-NEAT

Metoda NEAT (NeuroEvolution of Augmenting Topologies, česky Neuroevoluce rozšiřujících topologií) navrhnutá Stanleym [20] staví na postupném rozšiřování neuronové sítě společně s využitím operace křížení a zvýšení šance zachování nových řešení.

Úvodní kandidátní řešení obsahují pouze jednoduché sítě složené ze vstupní a výstupní vrstvy neuronů. Tyto sítě jsou pak podle potřeby rozšiřovány kromě úpravy vah také přidáváním spojení mezi nepropojené neurony či vložením nových neuronů do existujících spojení. Značení historie mutací umožňuje křížení kandidátních řešení díky znalosti společných částí genů i bez nutnosti složité analýzy jejich topologie. Míra rozdílu mezi kandidáty je navíc využita ke shlukování podobných kandidátů do skupin, které sdílí průměrnou fitness skupiny a umožňují přežití nových mutací po dobu potřebnou k jejich lepšímu upravení.

Metoda NEAT byla později rozšířena do podoby Hyper-NEAT (Hypercube-based NEAT, česky NEAT založená na hyperkostce) v Stanley [19]. Tato metoda využívá dvou vlastností velké části přírodních struktur včetně lidského mozku: symetrie a lokálnosti. Symetrie umožňuje znovu využití jednotlivých částí genomu. Lokálnost souvisí s významností blízko sebe umístěných částí a zvýšenou mírou jejich vzájemné spolupráce. Pro možnost dosažení těchto vlastností pak používá oproti původní variantě nepřímé kódování, které se navíc více hodí pro generování rozsáhlejších sítí.

Tento algoritmus mapuje vstupní data do 2D mřížky a k tvorbě sítě založené na vzdálenost bodů využívá Connective CPPN (Connective Compositional Pattern Producing Network, česky spojovací síť produkující kompoziční vzory) navrhnuté v Stanley [18]. Tento algoritmus bere jako vstup kartézské souřadnice vstupního a výstupního uzlu a produkuje váhu spojení mezi těmito body, jak lze vidět na obrázku 4.2. Spojení s příliš malou váhou mohou být následně kompletně eliminovány. Samotné Connective CPPN oproti běžným neuronovým sítím v tomto případě obsahují funkce produkující symetrické či periodické výstupy a její topologie a funkce jsou vyvíjeny procesem NEAT.



Obrázek 4.2: Znázornění výpočtu vah mezi vstupními a výstupními body mřížky pomocí Connective CPPN. Převzato z: [19].

4.5 Metoda J. F. Millera pro vývin neuronové sítě řešící více problémů

J. F. Miller [14] představil metodu pro vývoj programů, které dokážou vyvinout neuronovou síť schopnou vyřešit několik problémů zároveň bez nutnosti trénovat síť zvlášť pro každý z problémů. Vývin takové sítě je zde řízen právě pomocí programů, které jsou vytvářeny pomocí CGP. Konkrétně se jedná o dva programy, tzv. *soma program*, který dovoluje neuronům se hýbat, měnit se, odumírat či replikovat, a *dendrit program*, který umožňuje růst a změny dendritů a taktéž jejich odumírání a replikaci. Na základě provádění těchto programů pak vznikne struktura neuronů a dendritů, ze které lze extrahovat klasickou neuronovou síť pomocí procesu nazvaného *snapping*. Stěžejním bodem jeho metody jsou jeho upravené návrhy modelu neuronu a dendritu, které umožňují růst struktury neuronové sítě.

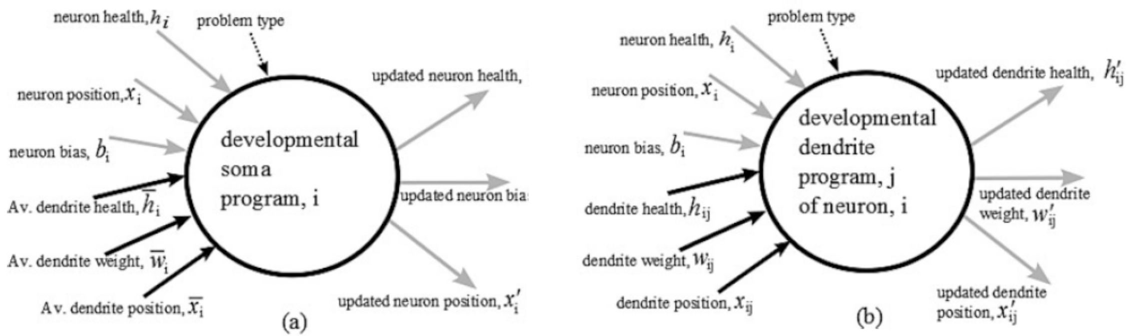
4.5.1 Upravený model neuronu a dendritu

Aby bylo možné síť vyvíjet pomocí neuronů a dendritů, byly v článku představeny speciální modely neuronu i dendritu. Model sítě sestavený z těchto upravených neuronů a dendritů Miller označuje jako mozek.

Jak je vidět na obrázku 4.3, neuron má oproti klasickému modelu neuronu více vstupů a to svoje zdraví, bias, svoji pozici, dále průměrné zdraví, průměrnou délku a průměrnou pozici všech dendritů, které jsou k němu připojeny, a navíc typ problému. Typ problému značí, zda se jedná o neuron výstupní vrstvy nebo ne a pokud ano, ke kterému z více řešených problémů tento neuron patří, aby se daný neuron mohl chovat správně pro každý z daných problémů. Výstupem takového neuronu po provedení soma programu je potom nová hodnota zdraví, biasu a pozice neuronu.

Model dendritu se také liší od klasického modelu. Jeho vstupy jsou jeho zdraví, váha a také pozice, zdraví, bias a typ problému jeho rodičovského neuronu. Výstupem po provedení dendrit programu je nová hodnota váhy, zdraví a pozice dendritu. Jelikož je úprava váhy výstupem dendrit programu, podílí se dendrit program i na učení sítě.

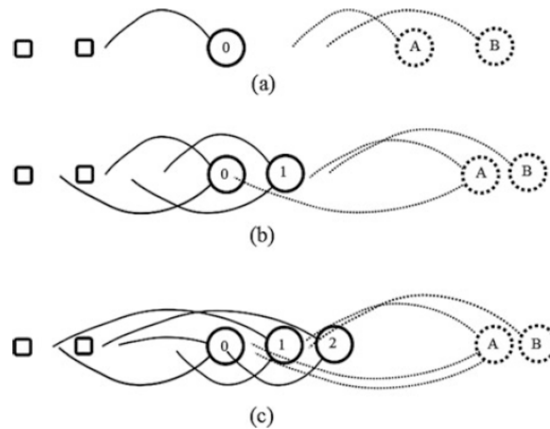
Hodnota zdraví slouží jak u dendritu tak neuronu pro určení, kdy má neuron či dendrit odumřít a kdy má být naopak vytvořen nový. Pozice je důležitá pro vyhledání nejbližšího neuronu pro dendrit při snappingu. Hodnota všech parametrů je v rozsahu [-1, 1].



Obrázek 4.3: Model neuronu (a) a dendritu (b) pro vývin neuronové sítě/mozku podle Millera. Převzato z: [14].

4.5.2 Princip růstu sítě

Ukázku růstu sítě (mozku) s využitím upraveného modelu neuronů a dendritů lze vidět na obrázku 4.4. Na této ukázce bude vysvětlen princip růstu mozku. Vstupy jsou na obrázku označeny čtverečky, kružnice ohraničené plnou čarou reprezentují nevýstupní neurony a jejich dendrity jsou také označeny plnou čarou. Tečkované kružnice reprezentují výstupní neurony a k nim patřící dendrity jsou také značeny tečkovaně. Výstupní neurony se mohou spojovat svými dendrity pouze s nevýstupními neurony nebo vstupy a nemohou se replikovat či umírat. Celý vývin mozku je řízen spouštěním soma a dendrit programu. Extrakce finální neuronové sítě z tohoto modelu mozku probíhá tak, že se dendrity podle svojí pozice připojí k nejbližšímu neuronu nalevo, tomuto procesu se říká snapping.



Obrázek 4.4: Ukázka vývinu sítě pomocí soma a dendrit programu. Převzato z: [14].

Na začátku (viz obrázek 4.4 (a)) je mozek inicializován jedním nevýstupním neuronem s jedním dendritem a obsahuje také dva výstupní neurony (A, B). Výsledná neuronová síť by v tomto počátečním stavu po aplikaci snappingu obsahovala 3 neurony. První nevýstupní neuron (0) by byl připojen na druhý vstup, a dva výstupní neurony (A, B) by byly připojeny k tomuto nevýstupnímu neuronu.

Po provedení prvního kroku vývoje (viz obrázek 4.4 (b)), kdy byl proveden soma a dendrit program v jednotlivých neuronech, došlo k tomu, že nevýstupnímu neuronu (0) přibyl jeden dendrit a tento neuron se také replikoval a vytvořil další nevýstupní neuron (1). Tento

nový neuron je kopií prvního neuronu (0) a má i stejné dendrity, ale jeho pozice se zvýšila o uživatelem definovanou vzdálenost. Provedením soma a dendtrit programu ve výstupních neuronech zde navíc došlo ke změně pozice výstupních neuronů A a B (oba se posunuly doprava), k prodloužení jejich dedritů a neuronu A přibyl navíc jeden dendrit, jehož pozice je nastavena na polovinu pozice rodičovského neuronu. Nový dendrit neuronu A by tedy ve výsledné neuronové síti byl připojen k neuronu 0. Druhý dendrit neuronu A a dendrit neuronu B by byl snappingem připojen k neuronu 1. Oba dendrity neuronu 1 byl byly připojeny ke druhému vstupu společně s jedním dendritem neuronu 0 a druhý dendrit neuronu 0 by byl připojen k prvnímu vstupu.

Po provedení dalšího kroku vývoje (viz 4.4 (c)) došlo k replikaci neuronu 1, který vytvořil nevýstupní neuron 2. Délka některých dendritů se po provedení programů zvětšila a neuronu A přibyl další dendrit. Výsledná síť po tomto kroku by tak obsahovala 5 neuronů, kde jeden z dendritů neuronu 0 a jeden z dendritů neuronu 1 by byl připojen k prvnímu vstupu. K druhému vstupu by byl připojen jeden dendrit neuronů 0, 1 a 2. K neuronu 0 by byly připojeny dva dendrity neuronu A a jeden dendrit patřící neuronu 2 a k neuronu 1 by byly připojeny dva dendrity neuronu B. K neuronu 2 není v tomto modelu připojen zprava žádný dendrit, je tedy redundatní, výsledné neuronové sítě by byl odstraněn a ta by tak nakonec obsahovala pouze 4 neurony.

Přehled celého řízení vývinu sítě pro řešení více problémů a získání jejího celkového ohodnocení fitness je v algoritmu 3 převzatého z [14]. Zde si lze všimnout, že vývoj celého modelu rozdělen do dvou fází, fáze „pre“ a „while“. Fáze „pre“ je fáze počátečního růstu, při které se pouze vyvíjí mozek, ale nedochází zde k extrakci výsledné sítě a nejsou jí předloženy ani žádné trénovací vstupy. Při fázi „while“ je prováděn další vývin mozku, ale poté také extrakce výsledné sítě a předložení trénovacích vstupů a vyhodnocování přesnosti sítě. Počet vývojových kroků obou těchto fází je dán nastavitelnými parametry (NDS_{pre}, NDS_{while}) a stejně tak i počet epoch, udávající kolikrát se provádí celá „while“ fáze (N_{ep}).

Algoritmus 3 Algoritmus pro vývin sítě a získání celkové fitness

Inicializuj mozek
Načti parametry pro fázi „pre“
Aktualizuj mozek NDS_{pre}-krát spuštěním soma a dendrit programu
Načti parametry pro fázi „while“
repeat
  Aktualizuj mozek NDS_{while}-krát spuštěním soma a dendrit programu
  Extrahuj z mozku výslednou neuronovou síť pro každý řešený problém
  Předlož každé síti trénovací vstupy a vypočítej přesnost pro každý problém
  Vypočti fitness jako průměr přesnosti všech sítí
  Pokud se snížila fitness, ukonči smyčku učení a vrať předchozí fitness
until Nebyl vyčerpán počet epoch učení N_{ep}
Vrať fitness

Na začátku tohoto algoritmu dochází k inicializaci mozku umístěním daného počtu počátečních nevýstupních neuronů a výstupních neuronů. Počet výstupních neuronů je dán součtem všech výstupů pro všechny řešené problémy. Každý výstup pro každý jednotlivý problém je tedy reprezentován jedním unikátním neuronem. Model umožňuje nastavení rozdílných parametrů pro fázi „pre“ a „while“, aby růst mozku mohl probíhat v každé z fází jiným způsobem. Dalším krokem je tedy načtení příslušných „pre“ parametrů. Pak následuje aktualizace mozku. Ta probíhá pomocí spuštění soma a dendrit programu. Jejich

spuštěním ve všech neuronech dojde v mozku ke změnám a vzniku nového modelu mozku. Tato aktualizace mozku je provedena NDS_{pre}-krát.

Poté se algoritmus přesune do fáze „while“. I pro tuto fázi je nejprve nutné načíst příslušné parametry. Pak je spuštěna smyčka, ve které pokračuje tak dlouho, dokud se nesníží celková fitness kandidátního řešení. Na začátku této smyčky se nejprve opět NDS_{while}-krát aktualizuje mozek, tentokrát s využitím příslušných „while“ parametrů. Po této aktualizaci je provedena extrakce výsledné sítě pro každý jednotlivý problém za použití snappingu. Takto extrahovaným sítím jsou předloženy jejich odpovídající trénovací vstupy a spočtena jejich přesnost. Celková fitness celé sítě se potom vypočte jako průměr všech získaných přesností pro všechny problémy. Pokud dojde v rámci smyčky, ke snížení fitness, smyčka je ukončena a jako celková fitness je vrácena předchozí hodnota, jinak se v ní pokračuje až do vyčerpání počtu epoch N_{ep}. Výsledkem tohoto algoritmu je tedy hodnota fitness, která udává kvalitu výsledné sítě.

4.5.3 Parametry modelu

Vývoj neuronové sítě tímto způsobem závisí na velkém množství nastavitelných parametrů. Jejich kompletní přehled je možné nalézt v [14]. Jedním z nastavitelných parametru je maximální počet neuronů, který může takto vyvíjená síť obsahovat. Stejně tak je nutné nastavit i počet dendritů, které může jeden neuronu mít. Tyto parametry zaručují, že síť nemůže přerůst a také že se nemůže sama eliminovat. Dalším vstupním parametrem modelu je počáteční počet nevýstupních neuronů. Počet výstupních neuronů naopak závisí na řešeném problému. Předem určena je také maximální a minimální hodnota pozice neuronů a přírůstek pozice, který udává, jak moc je nutné neuron posunout v případě kolize s jiným neuronem na stejné pozici. Důležitými nastavitelnými parametry jsou také hranice, které určují při jakých hodnotách zdraví budou neurony či dendrity odumírat a při jakých budou vznikat nové. Dále je možné nastavit parametry upravující velikost změny výstupů získaných ze soma a dendrit programu a upravit tak váhu takto získaných změn.

4.5.4 Vývoj soma a dendrit programu

Stěžejní částí Millerovy metody jsou právě soma program a dendrit program, protože řídí celý růst sítě. Soma program má celkem sedm vstupů a tři výstupy. Vstupy soma programu jsou zdraví neuronu, jeho bias a pozice a dále průměrné zdraví, váha a pozice dendritů a typ problému. Výstupy soma programu jsou nová hodnota zdraví, biasu a pozice. Úkolem soma programu je tedy vypočítat novou hodnotu zdraví, novou pozici a bias. Na základě nové hodnoty zdraví pak daný neuronu buď odumře nebo se naopak replikuje. Nová pozice neuronu může ovlivnit, které dendrity jsou k němu připojeny. Obě tyto hodnoty tedy mají vliv na topologii výsledné neuronové sítě a hodnota biasu neuronu ovlivňuje odezvu výsledné sítě.

Dendrit program má také sedm vstupů a tři výstupy. Vstupy dendrit programu jsou zdraví, pozice a váha dendritu, typ problému a zdraví, bias a pozice jeho rodičovského neuronu. Výstupy programu jsou nová hodnota zdraví, pozice a váhy. Podobně jako u soma programu, zdraví dendritu udává, zda má být daný dendrit odstraněn nebo má vzniknout nový. Pozice dendritu ovlivňuje, k jakému neuronu má dendrit nejblíže a tím tedy to, k jakému neuronu bude připojen při snappingu. Váha dendritu ovlivňuje odezvu výsledné sítě.

Tyto programy jsou vyvíjeny pomocí CGP, jehož cílem je vytvořit oba tak, aby jejich spouštěním bylo možno vyvinout síť s co nejvyšší přesností pro všechny dané problémy.

Oba programy jsou v CGP reprezentovány pomocí lineární konfigurace, kdy počet řádků mřížky CGP je 1 a parametr L-back je nastaven na celkový počet sloupců.

Miller ve svém článku navrhuje celkem 23 primitivních funkcí pro využití v CGP, které mají až tři vstupy. Využívá ale pouze podmnožinu z nich, konkrétně funkce step, add, sub, mult, xor a istep (viz tabulka 4.1), z nichž všechny mají maximálně dva vstupy z_0 a z_1. Všechny funkce pracují nad reálnými čísly reprezentovanými datovým typem float.

Délka chromozomu pro reprezentaci soma a dendrit programů byla zvolena na 800 uzlů. Pro řízení evoluce programů byla zvolena Goldmannova mutace, která postupně mutuje program v náhodných bodech, dokud není změněn nějaký aktivní gen. Vývoj obou programů obsahoval 20 000 generací a byla použita strategie ES(1+5). Právě soma a dendrit program jsou výsledkem celé metody a výsledná síť je vyvíjena podle nich. Ohodnoceny jsou v průběhu provádění hlavního algoritmu CGP pomocí algoritmu 3. Článek bohužel neuvádí přesný způsob vývinu programů. Není tedy jasné, jestli jsou oba programy vyvíjeny současně stejným způsobem, nebo každý zvlášť, ani žádné další detaily jejich vývoje. Konkrétně zmiňuje jen typ mutace, parametry CGP a způsob výpočtu fitness.

Funkce	**Definice**
step	if $z_0 < 0$ then 0 else 1
add	$(z_0 + z_1)/2$
sub	$(z_0 - z_1)/2$
mult	$z_0 z_1$
xor	if ($z_0 > 0$ and $z_1 > 0$) then -1 else if ($z_0 < 0$ and $z_1 < 0$) then -1 else 1
istep	if $z_0 < 1$ then 0 else -1

Tabulka 4.1: Tabulka primitivních funkcí a jejich definic. Převzato a upraveno z: [14].

4.5.5 Testování a výsledky

Millerovi se tedy podařilo vytvořit metodu, která dokáže vyvinout pomocí CGP dvojici programů. Pomocí těchto programů lze na základě Millerova principu pro růst sítě vyvinout neuronovou síť řešící několik problémů současně.

Pro testování sítí, které Miller vyvinul pomocí výsledných programů, byly využity tři běžné datové sady pro klasifikaci. Jednalo se o datové sady s poměrně podobným počtem vstupů a malým počtem tříd. Konkrétně o dataset pro rozpoznání rakoviny, dataset pro rozpoznávání diabetu a také dataset pro klasifikaci skla. Popis těchto datasetů je v tabulce 4.2.

Název	**Počet vstupů**	**Počet tříd**	**Velikost trénovací sady**	**Velikost validační sady**	**Velikost testovací sady**
Breast Cancer Wisconsin	9	2	350	175	174
Glass Identification	9	6	107	54	53
Pima Indians diabetes	8	2	384	192	192

Tabulka 4.2: Popis použitých datasetů.

Miller provedl s těmito datasety řadu experimentů, kde zkoumal, zda je pro vyšší přesnost výsledných sítí nutné povolit pohyb při vývinu sítě všem neuronům, pouze nevýstupním neuronům nebo pouze výstupním neuronům, nebo zda ho úplně zakázat. Nejlepších

výsledků dosáhl v případě, kdy byl povolen pohyb pouze výstupních neuronů, kde se výsledná průměrná přesnost vytvořené neuronové sítě pro všechny tři problémy na testovacím datasetu pohybovala okolo hodnoty 0,72. V ostatních případech bylo dosaženo průměrné hodnoty přesnosti pouze kolem 0,69. Neuronová síť vytvořená na základě jeho soma a dendrit programů byla tedy schopna vcelku dobře řešit kombinaci těchto tří problémů.

Další provedeným testem byl zjišťován význam parametru N_{ep}, tedy na jakou hodnotu je optimální nastavit počet epoch fáze „while“. Provedené testy ukázaly, že více opakování fáze „while“ nevede ke zlepšení výsledků a nejlepších výsledků bylo dosaženo pro $N_{ep} = 1$.

Kapitola 5

Návrh metody pro vývin neuronové sítě

Tato kapitola prezentuje návrh metody pro vývin neuronové sítě. Zde prezentovaná metoda vychází z metody navržené J. F. Millerem [14] představené v předchozí kapitole a snaží se ji napodobit, ovšem bude se lišit v částech, které v článku nebyly prezentovány. Využívají se zde také znalosti získané v předchozích kapitolách 2, 3 a 4.

5.1 Návrh metody

Návrh metody pro vývin neuronové sítě vychází přímo z metody J. F. Millera. Nejprve bude ale provedena implementace pouze pro řešení jednoho problému. Až po ověření funkčnosti bude implementace rozšířena o možnost vývinu sítě řešící více problémů zároveň. Využiji modely upraveného neuronu a dendritu, které byly prezentovány v podkapitole 4.5.1. Metoda bude také založena na vývoji soma a dendrit programů, které budou řídit vývin samotné sítě i její učení. Síť tedy bude vyvíjena pomocí těchto programů stejným principem, prezentovaným v podkapitole 4.5.2. Protože J. F. Miller ve své práci prokázal, že je vhodné povolit pohyb pouze výstupních neuronů, budu v návrhu uvažovat pouze tuto variantu.

Co se týká parametrů pro CGP, taktéž využiji navrženou strategii ES(1+5), stejnou délku chromozomu a lineární konfiguraci CGP, kde všechny uzly mohou být připojeny s libovolným předchozím sloupcem. Bude také využita stejná množina primitivních funkcí (viz tabulka 4.1).

Millerův článek explicitně neuvádí hlavní algoritmus, ve kterém probíhá vývoj soma a dendrit programů pomocí CGP, ale obsahuje algoritmus pro získání fitness těchto programů, který byl prezentován v algoritmu 3 v předchozí kapitole. Dendrit a soma programy budou tedy vyvíjeny souběžně a budou reprezentovat pravidla pro růst mozku, ze kterého bude možné po aplikaci těchto programů extrahovat výslednou neurnovou síť. Přesnost sítě, která bude při vývinu soma a dendrit programů průběžně extrahována, bude tedy sloužit jako fitness pro ohodnocení soma a dendrit programů.

Na začátku metody bude pomocí primitivních funkcí náhodně vytvořeno šest chromozomů reprezentujících soma program a šest chromozomů reprezentujících dendrit program. Z těchto programů bude vytvořeno šest dvojic, z nichž každá bude reprezentovat jednu neuronovou síť. Tyto dvojice tvoří počáteční populaci.

Pro ohodnocení soma a dendrit programu pak bude spuštěn algoritmus 3. Ten bude spuštěn pro každou z dvojic soma program-dendrit program zvlášť. Na začátku tohoto al-

goritmu dojde k inicializaci mozku vložením počátečního počtu nevýstupních a výstupních neuronů. Poté dojde nejprve ke spuštění fáze „pre“, ve které se dojde k růstu mozku pro danou dvojici programů. Jelikož soma a dendrit programy budou inicializovány náhodně, každá dvojice programů v populaci by měla síť z počátečních neuronů vyvinout trochu jiným způsobem. Následně se přistoupí k fázi „while“, kdy dojde k dalšímu spuštění soma a dendrit programu a dalšímu růstu mozku s mírně odlišnými parametry. Poté bude z mozku extrahována výsledná neuronová síť. Síť bude vyhodnocena na trénovacích vstupech a vypočte se její přesnost. V případě řešení více problému současně, se bude neuronová síť extrahovat pro každý problém zvlášť, přesnost jednotlivých sítí bude sečtena a vydělena počtem problémů pro získání celkové fitness.

Po získání fitness pro všechny dvojice bude možné jednotlivé kandidátní dvojice programů seřadit podle úspěšnosti a vybrat nejlepší dvojici, která se použije jako rodič pro následující generaci. V této vybrané dvojici se zvlášť na soma a dendrit pogram aplikuje Goldmannova mutace, pomocí které se vytvoří dalších pět dvojic programů pro novou generaci. Těchto pět potomků společně s rodičem vytvoří novou populaci a algoritmus bude pokračovat další generací.

Proces vývinu programů se bude opakovat po předem určený počet generací a bude se sledovat hodnota fitness. Výsledkem celé metody budou dva programy, soma a dendrit program, které mohou být využity k růstu sítě z předem daného počátečního stavu. Takto vyvinutá síť pak by měla být schopna řešit zároveň několik problémů, na kterých byla natrénována při vývinu jejích soma a dendrit programů.

Celý proces je tedy kombinací tradiční řídicí smyčky CGP (Algoritmus 2) a Millerova algoritmu pro získání fitness vyvíjených programů (Algoritmus 3). Hlavní řídicí smyčka tedy bude vypadat takto:

Algoritmus 4 Algoritmus pro vývoj soma a dendrit programu.

Náhodně vytvoř $(1 + \lambda)$ chromozomů reprezentujících soma program
Náhodně vytvoř $(1 + \lambda)$ chromozomů reprezentujících dendrit program
Z těchto programů vytvoř $(1 + \lambda)$ dvojic soma program-dendrit program
Inicializuj první populaci těmito dvojicemi
while Nevypršel počet generací **do**
  Zavolej algoritmus 3 a získej fitness pro všechny dvojice
  Vyber nejlepší dvojici jako rodiče
  for (i = 0, i $<$ λ, i++) **do**
    Aplikuj operátor Goldmannovy mutace pro každý program v dvojici rodiče a vytvoř potomka C_i
  end for
  Vytvoř novou populaci obsahující rodiče a všechny jeho potomky C_i
end while
Vrať dvojici programů s nejvyšší fitness

5.2 Návrh testování

Pro testování metody budou použity některé datasety použité J. F. Millerem v článku [14]. Využití stejných datasetů umožní porovnat mnou dosažené výsledky s výsledky dosaženými v originálním článku a ověřit funkčnost metody.

Datasety použité v [14] pocházejí ze sady referenčních problémů pro neuronové sítě Proben1 [15]. Pro exprimentování byly vybrány dva z použitých datasetů. Prvním je dataset obsahující informace o diagnóze rakoviny prsu, Breast Cancer Wisconsin. Jeho úkolem je klasifikovat nádor do dvou tříd jako benigní či maligní, na základě devíti vstupů ve formě reálných čísel. Dataset obsahuje celkem 699 vzorků. Dalším použitým datasetem bude dataset Pima Indians diabetes. Tento dataset má 8 vstupů také ve formě reálných čísel, značících výsledky testů a osobní data o pacientovi, na základě kterých je možné rozhodnout, zda pacient má diabetes nebo ne. Tento dataset obsahuje celkem 768 vzorků. Data v datových sadách Proben1 jsou rozdělena na trénovací, validační a testovací data v poměru 50 %, 25 %, 25%. Dle [14] budou využita pouze trénovací a testovací data a validační budou zanedbána.

Dále budou pro testování použity i jiné známé datové sady, získané z UCI Machine Learning Repository[1]. Konkrétně se bude jednat o dataset Iris[2], pro klasifikaci kosatců a dataset Bank Note Authentication[3], pro klasifikaci bankovek. Tyto datasety budou převedeny do stejného formátu jako ve sbírce Proben1, aby bylo možné je načíst stejným způsobem. U těchto nových datových sad jsem se rozhodla pro rozdělení dat v poměru 75%, 25%. Celkem 75% vzorků tak bude použito jako trénovací data a 25% pro testování.

Po převedení do stejného formátu má datová sada Iris celkem 4 vstupy ve formě reálných čísel a obsahuje celkem 150 vzorků, které je třeba klasifikovat do 3 tříd. Dataset Bank Note Authentication obsahuje taktéž 4 vstupy ve formě reálných čísel, 2 boolovské výstupy a celkem 1372 vzorků.

Metoda bude nejprve otestována na jednotlivých problémech zvlášť, čímž bude možné ověřit, že je metoda vhodná i pro vývin soma a dendrit programů umožnujících navrhnout neuronovou síť řešící jeden problém. Dále bude otestován vývin programů pro více problémů zároveň. Výsledky budou porovnávány s výsledky dosaženými J. F. Millerem v [14].

[1]https://archive.ics.uci.edu/ml/datasets.php
[2]https://archive.ics.uci.edu/ml/datasets/iris
[3]https://archive.ics.uci.edu/ml/datasets/banknote+authentication

Kapitola 6

Implementace metody pro vývin neuronové sítě

Implementace metody navržené v předchozí kapitole byla provedena v jazyce Python ve verzi 3.9 a využívá knihovnu Numpy. Knihovna Numpy byla zvolena proto, že je optimalizovaná pro práci s daty a poskytuje podporu pro práci s poli, která jsou rychlejší než běžné listy jazyka Python. Navíc obsahuje i velké množství užitečných matematických funkcí, které je možné provést nad celým polem.

Program se skládá z celkem šesti souborů, z nichž každý implementuje jednu část navržené metody. Soubor cgp.py obsahuje implementaci kartézského genetického programování, soubor brain.py obsahuje kód sloužící k vývinu mozku na základě soma a dendrit programů získaných pomocí kartézského genetického programovaní. Soubor neuralNetwork.py obsahuje funkce umožnující extrakci neuronové sítě z vyvinutého mozku a její výpočet pro získání přesnosti na testovacích a trénovacích datech. Hlavní smyčka programu (viz algoritmus 4) a výpis a ukládání výsledků jsou realizovány v souboru main.py. Dalším souborem je soubor dataLoader.py, který slouží k načtení datových sad pro trénování a testování neuronové sítě. A nakonec také soubor params.py, který obsahuje velké množství nastavitelných parametrů pro inicializaci a vývin mozku a také parametry pro CGP, jako je počet generací nebo velikost populace.

Detailnější popis implementace je uveden v podkapitole 6.1, další podkapitola 6.2 ukazuje jak nastavovat parametry programu a jak program spustit a vysvětluje obsah souborů vytvořených při běhu programu.

6.1 Popis implementace

Prvním krokem programu a tedy i prvním krokem v implementaci bylo načtení datových sad. Pro jejich načtení byla v souboru dataLoader.py vytvořena třída DataLoader, která obsahuje dvě metody. Metodu `load`, která načte data ze souboru s koncovkou .dt a rozdělí je na testovací a trénovací, a metodu `print_data_info`, která slouží k vypsání informací o načtené datové sadě na výstup.

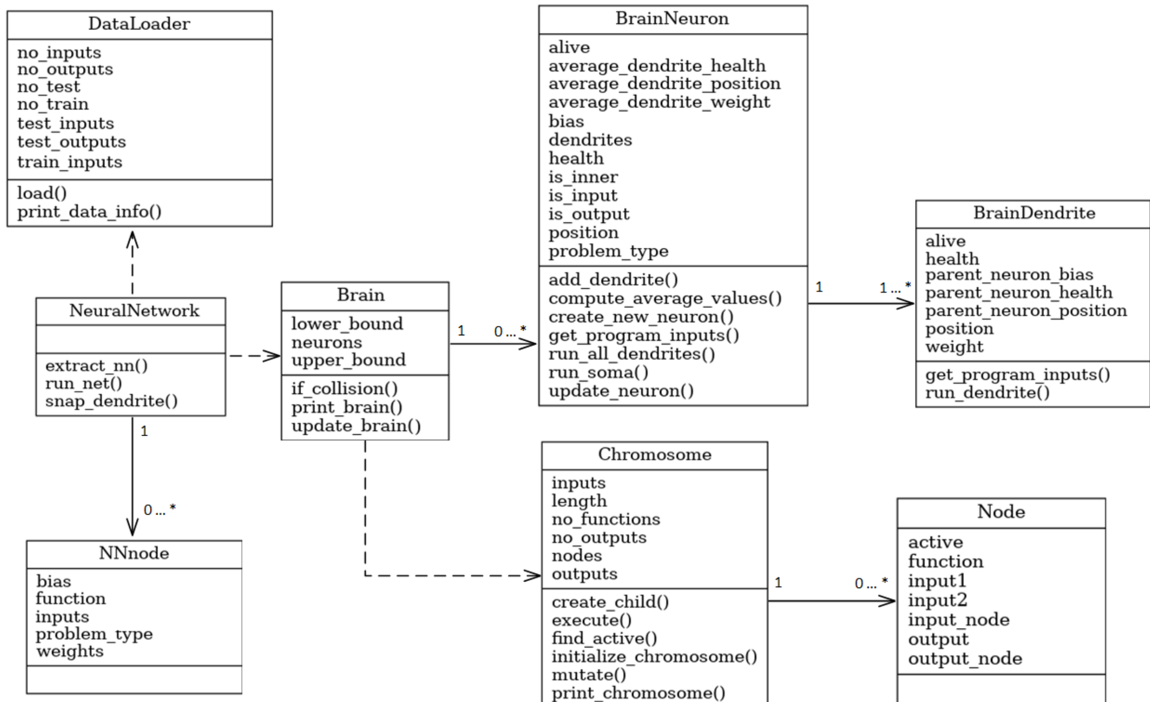
Dalším krokem bylo provedení inicializace mozku. Logika mozku je implementována v souboru brain.py a obsahuje několik tříd. Třídu Brain, která implementuje mozek samotný, a třídy BrainNeuron a BrainDendrit, které mozek využívá.

Po inicializaci mozku následuje vytvoření populace soma a dendrit programů, které bude možné v mozku spustit. Soma a dendrit program jsou reprezentovány pomocí chromozomů

v CGP, jejichž evoluce je řízena mutací. CGP je implementováno v souboru cgp.py a obsahuje třídu Chromosome, která slouží k reprezentaci programů, a třídu Node, reprezentující jeden uzel v chromozomu CGP. Soubor také obsahuje vyhledávací tabulku primitivních funkcí pro použití v uzlech CGP.

Po inicializaci populace se spustí hlavní smyčka programu, tak jak byla popsána v algoritmu 4. V rámci smyčky je nutné zjistit v každé iteraci fitness všech jedinců v populaci, tedy extrahovat neuronovou síť z mozku a spustit ji pro trénovací data. K tomuto účelu byly v souboru neuralNetwork.py implementovány třídy NeuralNetwork a NNnode. Po zjištění fitness a nalezení nejlepšího jedince proběhne vytvoření nových jedinců pro další populaci pomocí mutace soma a dendrit programu nejlepšího jedince. Nejlepší jedinec populace vždy bývá umístěn do další populace nezměněn. Po uběhnutí stanoveného počtu generací a tedy kompletního ukončení vývinu soma a dendrit programu je jako výsledek vrácen soma a dendrit program a neuronová síť naučená pro každý řešený problém. Výsledná neuronová síť je poté spuštěna na testovacích datech a je zjištěna její přesnost.

Všechny vytvořené třídy potřebné pro vývin mozku, soma a dendrit programů a vytvoření neuronové sítě jsou detailněji popsány dále. Vzájemné závislosti tříd je možné si prohlédnout v diagramu tříd na obrázku 6.1.



Obrázek 6.1: Diagram tříd implementovaného programu.

6.1.1 Implementace mozku

Hlavní třídou implementující mozek je třída Brain. Ta má uložen seznam neuronů, které se vyskytují v mozku, a rozsah pozic, na které je možné umisťovat neurony. Třída dále obsahuje metodu `init`, pro inicializaci mozku, metodu `update_brain` pro provedení vývinu mozku, metodu `if_collision`, která má za úkol zajistit, že neurony přidávané do mozku nebudou vloženy na stejnou pozici, a metodu `print_brain` pro výpis informací o neuronech nacházejících se v mozku.

Pro inicializaci mozku je metodě `init` nutné zadat počáteční váhy dendritů a biasy neuronů. Jelikož způsob počáteční inicializace vah a biasů nebyl v [14] popsán, jsou váhy pro dendrity a biasy neuronů v mozku vygenerovány náhodně z intervalu [-1, 1]. Náhodná inicializace byla zvolena proto, že je běžnou praxí při učení neuronových sítí. Náhodné váhy a biasy se generují nové pro každou generaci, protože při implementaci bylo pozorováno, že toto rozhodnutí přispívá k větší diverzitě populace. Vyšší diverzita populace by měla přispět k rychlejší konvergenci k lepšímu řešení.

Nejdůležitější metodou mozku je metoda `update_brain`, kde nejprve dochází k načtení správných parametrů v závislosti na fázi vývinu (fáze „pre" nebo „while") a především zde proběhne spuštění soma a dendrit programu, na základě nichž jsou v mozku provedeny změny. Nejprve jsou na základě programů provedeny změny pro všechny nevýstupní neurony, které zůstávají na stejné pozici. Na základě nové hodnoty zdraví buď v mozku zůstanou, nebo odumírají. Je povolena jejich replikace při překročení nastavené hranice zdraví. Pokud je neuron zreplikován, je k jeho pozici přičtena hodnota parametru mn_{inc}, provedena kontrola pomocí metody `if_collision` a teprve poté je přidán do mozku.

Dalším krokem je aktualizace hodnot pro výstupní neurony, kterým je sice povolen pohyb, ale nemohou se replikovat ani odumírat, aby zůstal zachován správný počet výstupů pro řešené problémy. Změny hodnot ve výstupních neuronech probíhají na základě soma a dendrit programu stejně jako u neuronů nevýstupních.

Metoda `update_brain` také kontroluje, zda počet neuronů v mozku nepřekročil maximální hranici a v případě, že by k takové situaci došlo, zachová dostatek místa v mozku pro výstupní neurony tím, že omezí počet vložených nevýstupních neuronů.

Každý neuron v mozku je instancí třídy BrainNeuron. Třída BrainNeuron, která slouží k reprezentaci neuronů, je rozsáhlejší a obsahuje celkem 8 metod. Pro každý neuron je uchovávána informace o tom, jestli se jedná o vstupní, nevýstupní nebo výstupní neuron, jestli je živý, a o hodnotách jeho zdraví, biasu a pozici. Neuron také obsahuje seznam svých dendritů a informace o jejich průměrných hodnotách, které slouží jako vstupy pro soma program.

Stěžejními metodami třídy BrainNeuron jsou metody `update_neuron` a `run_all_dendrites`, které slouží k aktualizaci informací v neuronu v průběhu vývinu mozku. V metodě `update_neuron` dochází ke změnám hodnot týkajících se samotného neuronu na základě výstupů získaných ze soma programu. Jsou zde vypočteny nové hodnoty pozice, biasu a zdraví. Metoda `run_all_dendrites` se poté stará o úpravu hodnot ve všech dendritech neuronu dle dendrit programu a řídí tak jejich pohyb, replikaci a odumírání. Také kontroluje, zda nebyla překročena maximální či minimální hranice dendritů v mozku. V případě překročení maximální hranice počtu dendritů, přeruší jejich přidávání. Pokud by naopak došlo k odumření všech dendritů, zachová jeden původní dendrit rodičovského neuronu.

Dále třída BrainNeuron obsahuje metodu pro vytvoření nového neuronu se stejnými parametry, která se využívá při replikaci, a metodu pro přidání dendritu, která vytvoří zcela nový dendrit s vahou 1 a pozicí na polovině pozice rodičovského neuronu. Další metody jsou spíše pomocného charakteru a slouží například k získání všech vstupů pro soma program, spuštění soma programu či výpočet průměrných hodnot dendritů neuronu.

Poslední třídou důležitou pro vývin mozku je také třída BrainDendrite, reprezentující jednotlivé dendrity neuronu. Zde je pro každý dendrit uložena informace o jeho zdraví, váze a pozici a dále hodnoty biasu, pozice a zdraví rodičovského neuronu. Tato třída obsahuje 3 metody, z nichž nejdůležitější je metoda `run_dendrite`, ve které dochází k aktualizaci hodnot dendritu na základě výstupů dendrit programu. Dále obsahuje metodu pro získání

vstupních hodnot pro dendrit program a samozřejmě metodu pro inicializační metodu pro vytvoření dendritu.

6.1.2 Implementace soma a dendrit programu

Programy jsou reprezentovány pomocí třídy Chromosome, která implementuje metody potřebné pro mutaci a spuštění soma a dendrit programů. V každém chromozomu je uložen seznam vstupů, seznam funkčních uzlů chromozomu a seznam výstupních uzlů. Všechny uzly jsou reprezentovány instancemi třídy Node, která slouží k uchování informací o uzlu, konkrétně o tom, zda se jedná o vstupní, výstupní nebo funkční uzel, o čísle funkce přiřazené uzlu dle označení funkcí ve vyhledávací tabulce a o hodnotě na výstupu uzlu.

Pro vytvoření nového chromozomu je nutné použít metodu `initialize_chromosome`, která se postará o naplnění chromozomu náhodnými funkčními a výstupními uzly. Implementována je pouze lineární konfigurace mřížky CGP s maximální hodnotou L-back, tudíž každý uzel může být připojen na kterýkoliv předcházející. Dále třída Chromosome obsahuje metodu `find_active` sloužící pro nalezení aktivních uzlů. Za aktivní uzel se považuje uzel, z něhož vede cesta od nějakého výstupního uzlu až do libovolného vstupu. Každý výstupní uzel je vždy aktivní. Metoda `find_active` je důležitá pro další implementovanou metodu s názvem `execute`, která slouží pro výpočet výstupů chromozomu. Implementována je také metoda `mutate`, která provádí Goldmanovu mutaci nad chromozomem CGP, tedy provádí změny v uzlech do té doby, dokud není změněn nějaký aktivní uzel. Tato metoda se používá k vytvoření potomků nejlepšího jedince v populaci.

6.1.3 Implementace neuronové sítě

Poslední důležitou částí programu je extrakce samotné neuronové sítě z mozku. K tomuto účelu byla vytvořena třída NeuralNetwork obsahující 3 metody, metodu `extract_nn` pro extrakci neuronové sítě, metodu `snap_dendrite`, která má za úkol najít neuron nejbližší k danému dendritu a metodu `run_net`, pro výpočet výstupů neuronové sítě pro vzorky ze zadané datové sady a spočtení celkové přesnosti sítě. Neuronová síť je vytvářena postupným přidáváním neuronů reprezentovaných třídou NNnode. Tyto neurony sítě mají uloženy pozice neuronů se kterými jsou propojeny a k nim i odpovídající hodnoty dendritů, hodnotu biasu a hodnotu určující ke kterému problému daný neuron přísluší, pokud se jedná o neuron výstupní, aby bylo možno identifikovat výstupy při řešení více problémů zároveň.

Metoda `extract_nn` nejprve seřadí neurony obsažené v mozku podle pozice. Poté jsou postupně zleva procházeny jednotlivé neurony. Pro každý neuron totiž musí být zjištěno, ke kterým předcházejícím neuronům je svými dendrity připojen. Jelikož dendrity v mozku mohou růst oběma směry, ale je potřeba extrahovat dopřednou síť, pro každý dendrit neuronu je nutné jeho pozici přepočítat tak, aby se nacházel nalevo od rodičovského neuronu. Po tomto převodu je zavolána metoda `snap_dendrite`, která nalezne k dané pozici dendritu nejbližší neuron zleva. Index pozice tohoto nejbližšího neuronu v seřazeném poli je poté přidán do pole pozic neuronů uložených u daného neuronu. Po provedení této akce pro všechny neurony a jejich dendrity vznikne síť reprezentovaná polem instancí třídy NNnode.

Stěžejní metodou, pro celý program, je metoda `run_net`, protože v ní dochází k výpočtu přesnosti neuronové sítě a tudíž se používá k výpočtu fitness. Zde jsou na vstupy neuronové sítě postupně přikládány vstupní vzorky a je vypočítána odezva sítě. Po vypočtení odezvy pro každý vzorek datové sady se provede kontrola, zda výsledek poskytnutý sítí odpovídá očekávanému výstupu a pokud ano, je zvýšen čítač počítající počet správných klasifikací.

Celková přesnost sítě je potom vypočtena vydělením počtu správně klasifikovaných vzorků počtem všech vzorků datasetu.

6.2 Parametry a spuštění programu

Nastavení parametrů programu se provádí v souboru param.py. Soubor obsahuje celkem 32 nastavitelných parametrů. Jejich přehled a komentář vysvětlující jejich význam je uveden v tabulce 6.1. Před spuštěním je nutné mít datové sady, pro které se bude program spouštět, umístěny ve složce data. Všechny také musí být ve formátu odpovídajícímu datasetům ze sady Proben1 a mít koncovku .dt. Další podmínkou pro spuštění je mít nainstalovanou knihovnu Numpy.

Parametr	**Komentář**
NN_{max}	Maximální počet neuronů v mozku
N_{init}	Počet vnitřních neuronů mozku při inicializaci
DN_{max}	Maximální počet dendritů pro jeden neuron
ND_{init}	Počet dendritů neuronu při inicializaci
NDS_{pre}	Počet kroků vývinu mozku ve fázi „pre"
NDS_{whi}	Počet kroků vývinu mozku ve fázi „while"
MN_{inc}	Hodnota zvýšení pozice neuronu při kolizi
I_u	Maximální pozice, do které jsou v mozku umístěny vstupní neurony
O_l	Minimální pozice, od které jsou v mozku umístěny výstupní neurony
Vývojové parametry	
$NH_{death}\text{-}pre$ ($NH_{death}\text{-}while$)	Hranice zdraví pro odumření neuronu ve fázi „pre" („while")
$NH_{birth}\text{-}pre$ ($NH_{birth}\text{-}while$)	Hranice zdraví pro narození neuronu ve fázi „pre" („while")
$DH_{death}\text{-}pre$ ($DH_{death}\text{-}while$)	Hranice zdraví pro odumření dendritu ve fázi „pre" („while")
$DH_{birth}\text{-}pre$ ($DH_{birth}\text{-}while$)	Hranice zdraví pro narození dendritu ve fázi „pre" („while")
$\delta_{sh}\text{-}pre$ ($\delta_{sh}\text{-}while$)	Hodnota změny zdraví neuronu ve fázi „pre" („while")
$\delta_{sp}\text{-}pre$ ($\delta_{sp}\text{-}while$)	Hodnota změny pozice neuronu ve fázi „pre" („while")
$\delta_{sb}\text{-}pre$ ($\delta_{sb}\text{-}while$)	Hodnota změny biasu neuronu ve fázi „pre" („while")
$\delta_{dh}\text{-}pre$ ($\delta_{dh}\text{-}while$)	Hodnota změny zdraví dendritu ve fázi „pre" („while")
$\delta_{dp}\text{-}pre$ ($\delta_{dp}\text{-}while$)	Hodnota změny pozice dendritu ve fázi „pre" („while")
$\delta_{dw}\text{-}pre$ ($\delta_{dw}\text{-}while$)	Hodnota změny váhy dendritu ve fázi „pre" („while")
Parametry CGP	
$Chromosome_{len}$	Délka chromozomu
$NumCGP_{gens}$	Počet generací
$Num_{population}$	Velikost populace

Tabulka 6.1: Tabulka nastavitelných parametrů programu.

Spuštění programu je možné pomocí příkazu `python main.py` (popř. `python3 main.py`). Po spuštění program na svůj standardní výstup nejprve vypíše informace o načtených datasetech (počty vstupů a výstupů a počet testovacích a trénovacích vzorků), poté se spustí samotný vývin sítě. V této fázi program průběžně vypisuje informaci o čísle generace a aktuální nejvyšší hodnotě fitness. Po skončení programu je vypsána na výstup také informace o hodnotě přesnosti klasifikace pro jednotlivé problémy na testovacích datech.

Taktéž se vytvoří soubor s výsledky results.txt, který bude obsahovat informace o vývoji fitness v průběhu provádění programu. Ukázka tohoto souboru je na obrázku 6.2. Číslo v prvním sloupci značí číslo generace, ve které došlo ke změně fitness, číslo ve druhém sloupci uvádí novou hodnota fitness. Na konci takového jednoho běhu následují dva řádky, kdy na prvním z nich je zapsána přesnost pro jednotlivé problémy a celková fitness celého řešení na trénovacích datech a na druhém řádku poté tytéž informace pro testovací data.

```
----------------------------------------------------------------------
0 0.5116439048360778
7 0.5471045189882807
12 0.5657961077733275
26 0.6485689252336448
P1: 0.9 P2: 0.671875 P3: 0.37383177570093457 TOTAL TRAIN: 0.6485689252336448
P1: 0.9080459770114943 P2: 0.6302083333333334 P3: 0.5283018867924528 TOTAL TEST: 0.6888520657124269
----------------------------------------------------------------------
```

Obrázek 6.2: Ukázka z výsledného results.txt souboru vytvořeného programem.

Kapitola 7

Experimenty a výsledky

Tato kapitola obsahuje přehled provedených experimentů a jejich výsledků. Experimenty byly prováděny kvůli jejich urychlení na superpočítači Barbora, který je součástí národního superpočítačového centra IT4Innovations provozovaného Technickou univerzitou Ostrava. Výpočty na Barboře byly prováděny na CPU. Program je ale možné spustit i na běžném počítači.

Pro experimenty budou využity datasety navržené v kapitole 5.2: Breast Cancer Wisconsin (dále jen Cancer), Pima Indians diabetes (dále jen Diabetes), Iris a Banknote Authentication (dále jen Banknotes). Výchozími parametry pro všechny experimenty byly parametry prezentované v [14] a dále byly upravovány pro právě řešený problém. Výchozí nastavení parametrů pro vývin mozku je vidět v tabulce 7.1 a parametry použité v CGP v tabulce 7.2.

Parametr	**Hodnota**	**Hodnota**
NN_{max}	20	
N_{init}	5	
DN_{max}	40	
ND_{init}	5	
NDS_{pre}	8	
NDS_{whi}	3	
MN_{inc}	0,03	
I_u	–0,6	
O_l	0,8	
Vývojové parametry	Fáze „pre“	Fáze „while“
NH_{death}	–0,6	–0.58
NH_{birth}	0,308	0,8
DH_{death}	–0,404772	–0,38
DH_{birth}	–0,2012	0,85
δ_{sh}	0,1	0,01
δ_{sp}	0,1	0,01
δ_{sb}	0,07	0,0402
δ_{dh}	0,1	0,01
δ_{dp}	0,2032	0,01
δ_{dw}	0,1	0,02029

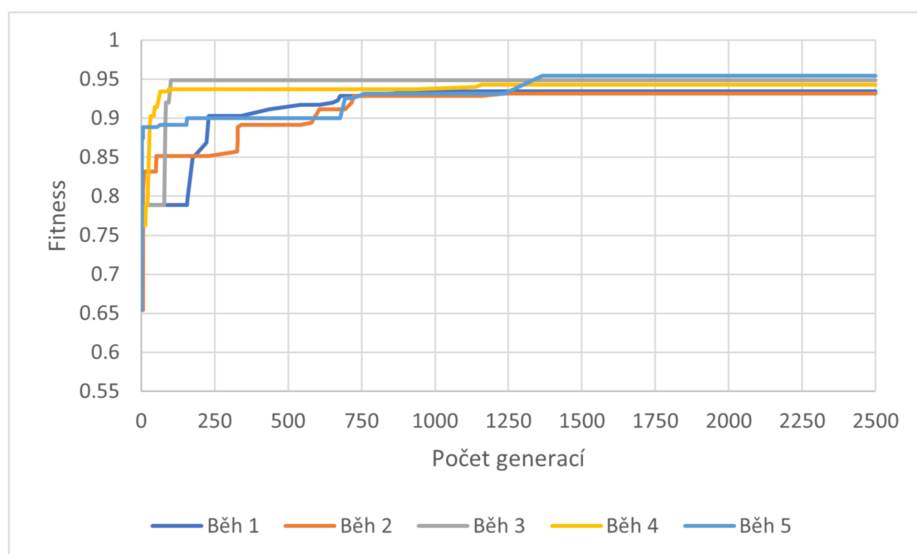
Tabulka 7.1: Tabulka výchozích parametrů pro vývin mozku použitých pro experimenty.

Parametry CGP	Hodnota
Délka chromozomu	800
Počet generací	1000
Použitá strategie	1+5

Tabulka 7.2: Tabulka výchozích parametrů pro CGP použitých pro experimenty.

Parametr, který musel být nastaven odlišně oproti parametrům v článku, byl počet provedených generací, protože běh 20000 generací trvá programu i několik hodin a nebyl vhodný pro rozsáhlejší testování. Bylo tedy nejprve nutné zjistit, po jakém počtu generací se vývoj ustálí a kdy nastává okamžik, od kterého se přestává zvyšovat fitness řešení.

Pro zjištění této hranice jsem provedla 5 nezávislých běhů programu s datovou sadou Cancer a zaznamenávala jsem průběh nejvyšší fitness pro každou generaci do grafu. Pro každý běh byl počet generací omezen na 2500. Výsledný graf je na obrázku 7.1. Tam si lze všimnout, že fitness řešení nejprve prudce stoupá během prvních několika generací, přibližně do 750. probíhá vývoj fitness nejrychleji a od generace 1000 se fitness přestává měnit. Počet generací pro vývin sítě jsem na základě tohoto zjištění omezila na 1000 a experimenty byly dále prováděny s tímto počtem.



Obrázek 7.1: Graf vývoje fitness v průběhu generací pro pět nezávislých běhů programu.

7.1 Experimenty pro jednotlivé datasety

Po zjištění potřebného počtu generací pak byly prováděny experimenty na jednotlivých datových sadách zvlášť. Pro vyhodnocení byl program spouštěn třicetkrát pro každou datovou sadu a výsledky z všech těchto běhů byly zpracovány do tabulek. Jeden běh programu pro řešení jednoho problému trvá průměrně 20 minut.

Cílem těchto experimentů bylo zjistit, jak si implementovaná metoda povede při vývinu neuronové sítě pro jeden problém a zda vůbec vyvinutá síť bude schopna prokázat nějakou klasifikační schopnost. Ukázalo se, že implementovaná metoda opravdu dokáže nalézt neuronovou síť schopnou klasifikace.

Pro všechny datové sady jsem se snažila upravit parametry z tabulky 7.1 na takové, které povedou na nejvyšší hodnotu přesnosti. Protože parametrů je velké množství, bylo jejich nastavení měněno podle znalosti řešeného problému a implementované metody. U datových sad, kde se výchozí parametry podařilo upravit tak, že se zvýšila přesnost klasifikace na dané datové sadě, je uvedena tabulka změn výchozích parametrů.

První experiment byl proveden nad datovou sadou Cancer. Pro tuto datovou sadu se nepodařilo najít lepší parametry než výchozí (viz tabulka 7.1). Dosažené výsledky pro trénovací i testovací data jsou v uvedeny v tabulce 7.3. Průměrná přesnost řešení zde dosahovala hodnoty téměř 0,94 na trénovacích a 0,95 na testovacích datech. Navíc celková přesnost sítě nikdy neklesla pod hodnotu 0,9 a u testovacích dat se držela dokonce nad hodnotou 0,92. Výsledek odpovídá výsledkům dosaženým v [14], kde průměrná přesnost na trénovacích datech byla taktéž 0,94 a na testovacích 0,95. Hodnota přesnosti na testovacích datech, byla navíc vyšší než na datech trénovacích, což taktéž odpovídá zjištěním v [14].

Dataset Cancer		
Data	trénovací	testovací
Průměr	0,9367	0,9540
Medián	0,9371	0,9540
Maximum	0,9600	0,9826
Minimum	0,9057	0,9253

Tabulka 7.3: Tabulka přesnosti pro datovou sadu Cancer.

Dalším testovanou datovou sadou byla sada Diabetes. Zde byla provedena úprava parametrů (viz tabulka 7.4), protože tato úprava přinesla zvýšení přesnosti. I přes úpravy parametrů zde bylo dosaženo trochu horších výsledků něž v prvním experimentu. Většinou se podařilo správně klasifikovat průměrně pouze kolem 72 % vzorků trénovacích dat a 70 % vzorků testovacích dat (viz tabulka 7.5). Snížení přesnosti zde bylo ovšem očekávané, protože přesnost dosažená v [14] se také pohybovala kolem hodnoty 0,7 a výsledky tedy odpovídají očekáváním.

Tato datová sada se pro metodu ukázala jako trochu náročnější, jelikož nebyla schopná najít řešení které by dokázalo správně klasifikovat více než 76 % testovacích vzorků a na rozdíl od datové sady Cancer zde navíc přesnost na trénovacích datech ve většině případů nepřevýšila přesnost při ověření na datech testovacích. Nicméně nižší hodnoty přesnosti neuronových sítí pro tento problém nemusí být způsobeny selháním metody, ale spíše kvalitou samotné datové sady, pro kterou je bez dodatečných úprav dat složitější najít klasifikátor s přesností vyšší než 80 %. Ve srovnání provedeném v [14] s jinými metodami klasifikace dosáhly odlišné klasifikační metody pro tento dataset nejvyšší přesnosti 0,79.

Parametr	Původní hodnota	Nová hodnota
NN_{max}	20	15
O_l	0,8	–0,5

Tabulka 7.4: Parametry změněné oproti výchozím parametrům.

Dataset Diabetes		
Data	trénovací	testovací
Průměr	0,7206	0,7050
Medián	0,7279	0,7031
Maximum	0,7474	0,7604
Minimum	0,6849	0,6406

Tabulka 7.5: Tabulka přesnosti pro datovou sadu Diabetes.

Dále byly provedeny experimenty nad dalšími datovými sadami. První byla známá datová sada Iris. Pro tento dataset byly upraveny parametry v tabulce 7.6. Tabulka 7.7 ukazuje hodnoty přesnosti dosažené s těmito parametry. Je patrné, že na této datové sadě bylo dosaženo velmi dobrých výsledků. Na trénovacích datech bylo dosaženo průměrné přesnosti 0,94 a na testovacích dokonce 0,98. Vyšší hodnota přesnosti na testovacích datech je způsobena skutečností, že se vyvinuté síti při většině běhů podařilo dosáhnout 100 % úspěšnosti při klasifikaci testovacích vzorků, což je důkazem dobré klasifikační schopnosti sítě.

Parametr	Původní hodnota	Nová hodnota
NN_{max}	20	25

Tabulka 7.6: Parametry změněné oproti výchozím parametrům.

Dataset Iris		
Data	trénovací	testovací
Průměr	0,9387	0,9833
Medián	0,9464	1,0000
Maximum	0,9554	1,0000
Minimum	0,8839	0,9387

Tabulka 7.7: Tabulka přesnosti pro datovou sadu Iris.

Další použitou datovou sadou byla sada Banknotes. Pro tuto datovou sadu byla také provedena úprava parametrů (viz tabulka 7.8). Na této datové sadě bylo rovněž dosaženo dobrých výsledků, které jsou uvedeny v tabulce 7.9. Průměrná přesnost na trénovacích i na testovacích datech dosahovala hodnoty 0,89.

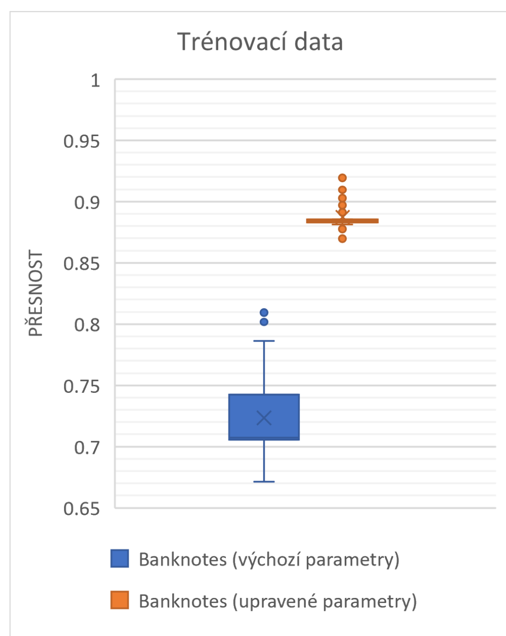
Parametr	Původní hodnota	Nová hodnota
NN_{max}	20	11
O_l	0,8	–0,25

Tabulka 7.8: Parametry změněné oproti výchozím parametrům.

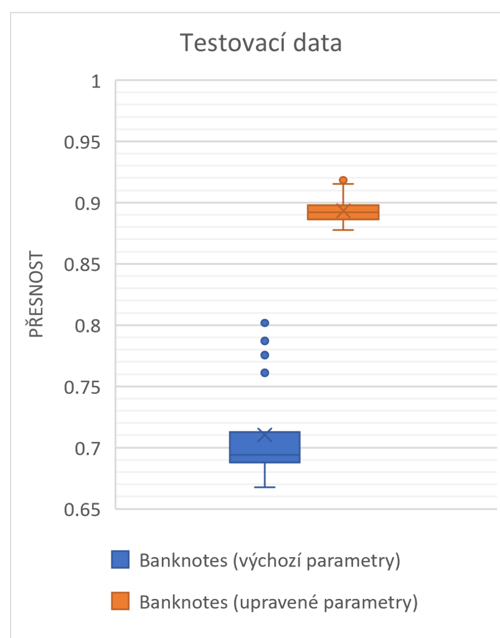
Dataset Banknotes		
Data	trénovací	testovací
Průměr	0,8874	0,8934
Medián	0,8844	0,8921
Maximum	0,9193	0,9184
Minimum	0,8698	0,8776

Tabulka 7.9: Tabulka přesnosti pro datovou sadu Banknotes.

Při experimentech byla také potvrzena hypotéza, že metoda je citlivá na správné nastavení parametrů. Nejvíce se osvědčilo provádět změny maximálního počtu neuronů a dále změny hodnoty pozice, od které se mají do mozku umisťovat výstupní hodnoty. Protože jsou datové sady poměrně malé, stačí v některých případech ke zvýšení přesnosti omezit pouze maximální počet neuronů. Velký vliv na výslednou přesnost má ovšem změna hranice výchozí pozice výstupních neuronů. Změna pozice totiž do velké míry ovlivňuje, k jakým neuronům v mozku se připojí výstupní neurony a mění tak vývin topologie neuronové sítě. Grafy na obrázcích 7.2 a 7.3 ukazují, jak se změnily hodnoty přesnosti při změně parametrů oproti výchozímu nastavení pro dataset Banknotes. Je vidět, že mírná změna parametrů způsobila velký posun v hodnotách přesnosti.



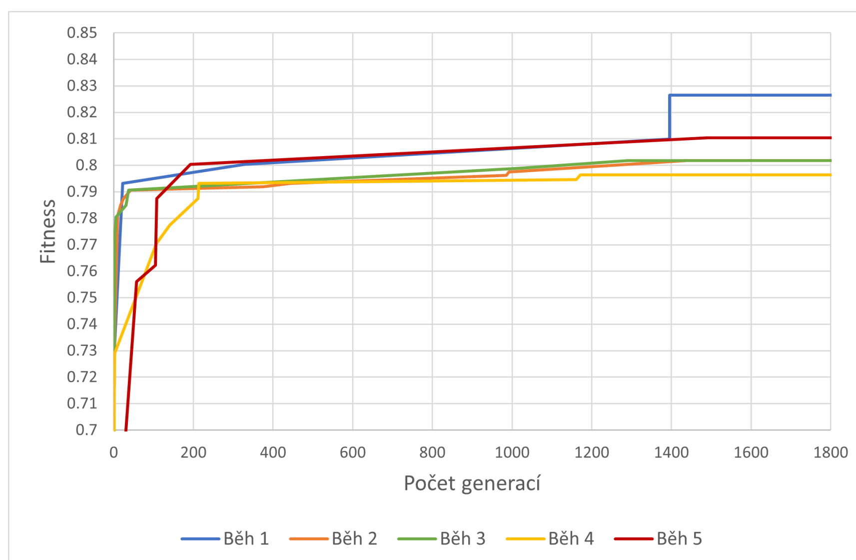
Obrázek 7.2: Srovnání přesnosti pro různá nastavení pro trénovací data.



Obrázek 7.3: Srovnání přesnosti pro různá nastavení pro testovací data.

7.2 Experimenty pro návrh sítě řešící více problémů naráz

Dále byly prováděny experimenty pro návrh neuronové sítě, která zvládne vyřešit více problémů naráz. Zde byly provedeny 2 experimenty pro dvojice problémů. Použité výchozí parametry byly opět parametry prezentované v tabulce 7.1. Byla ale provedena změna pro počet generací, který byl zvýšen na 1800 generací. Toto zvýšení bylo zavedeno proto, aby měla síť více času se dostatečně přizpůsobit oběma řešeným problémům. Jeden běh programu pro návrh sítě řešící více problémů trval průměrně 40 minut.



Obrázek 7.4: Průběh hodnoty fitness v průběhu generací při řešení více problémů.

Graf průběhu změn hodnot fitness v průběhu generací pro 5 nezávislých běhů lze vidět na obrázku 7.4, kde je patrné, že kolem generace 1000 i po ní ještě často dochází ke změnám fitness a že navýšení počtu generací v tomto případě je tedy pro program prospěšné.

Prvním experimentem pro řešení vícero problémů jednou neuronovou sítí, byl experiment nad kombinací datových sad Cancer a Diabetes. Nejlepších výsledků bylo dosaženo po úpravě výchozích parametrů na parametry uvedené v tabulce 7.10. Změna parametru O_l byla provedena proto, že při experimentování bylo pozorováno, že při této nové hodnotě parametru se zvedne přesnost sítě pro klasifikaci datasetu Diabetes. Pro původní hodnotu parametru 0,8 totiž docházelo k tomu, že výsledná síť byla správně naučena na klasifikaci pro datovou sadu Cancer, ale přesnost na datasetu Diabetes byla až o 15 % nižší než očekávané hodnoty.

Tabulka 7.11 obsahuje celkovou hodnotu přesnosti pro obě datové sady. Celková hodnota přesnosti se v rámci programu počítá jako součet přesností pro všechny řešené problémy vydělený počtem problémů. Celková přesnost dosažená zde pro kombinaci datových sad Cancer a Diabetes se průměrně pohybovala kolem hodnoty 0,8 pro trénovací i testovací data.

Tabulky 7.12 a 7.13 ukazují hodnoty přesnosti dosažené společně vyvíjenou neuronovou sítí na jednotlivých problémech. Z těchto tabulek vyplývá, že při řešení více problémů zároveň došlo k poklesu přesnosti pro oba problémy, oproti přesnosti dosažené při vývinu neuronové sítě pro každý problém zvlášť. U datové sady Cancer byl pokles přesnosti velmi nízký. Průměrná přesnost klesla pouze o hodnotu 0,0041 na trénovacích datech a 0,0003 na testovacích datech, což se dá považovat za zanedbatelné. U datové sady Diabetes byl pokles průměrné přesnosti vyšší, a to 0,0476 u trénovacích dat a 0,0647 u dat testovacích. Vyšší pokles přesnosti pro datovou sadu Diabetes je pravděpodobně způsoben tím, že nalezení řešení pro dataset Cancer, jakožto jednodušší z obou datasetů, bylo pro pro program snazší a zvyšování přesnosti na něm tedy probíhalo rychleji. Toto rychlejší zvyšování přesnosti pro jeden problém mohlo způsobit, že místo toho, aby se správná topologie sítě tvořila pro oba problémy zaráz, jak by bylo nejvíce žádoucí, program upřednostňoval řešení problému Cancer, protože přiblížení k jeho lepšímu řešení mělo za následek vyšší zvýšení jeho fitness než při snaze zlepšit přesnost na druhém datasetu.

Parametr	Původní hodnota	Nová hodnota
NN_{max}	20	25
O_l	0,8	–0,5
Počet generací	1000	1800

Tabulka 7.10: Parametry změněné oproti výchozím parametrům.

Datasety Cancer a Diabetes		
Data	trénovací	testovací
Průměr	0,8028	0,7970
Medián	0,8018	0,7947
Maximum	0,8266	0,8463
Minimum	0,7932	0,7788

Tabulka 7.11: Tabulka celkové přesnosti při návrhu sítě pro datasety Cancer a Diabetes.

Dataset Cancer		
Data	trénovací	testovací
Průměr	0,9326	0,9537
Medián	0,9343	0,9540
Maximum	0,9514	0,9770
Minimum	0,9057	0,9080

Tabulka 7.12: Tabulka přesnosti pro síť řešící více problémů pro dataset Cancer.

Dataset Diabetes		
Data	trénovací	testovací
Průměr	0,6730	0,6403
Medián	0,6693	0,6354
Maximum	0,7474	0,7500
Minimum	0,6589	0,6094

Tabulka 7.13: Tabulka přesnosti pro síť řešící více problémů pro dataset Diabetes.

Výsledky dosažené na této kombinaci datových sad je opět možné porovnat s výsledky dosaženými v [14]. Při srovnaní tabulek 7.12 a 7.14 lze vidět, že průměrné hodnoty pro datovou sadu Cancer jsou v podstatě totožné a liší se až v řádu tisícin, kromě hodnoty minima, které je u zde dosažených výsledků mírně vyšší. Při srovnání výsledků pro datovou sadu Diabetes (tabulky 7.13 a 7.15) se navržené řešení liší více pro trénovací sadu vzorků. Průměrná hodnota přesnosti je u ní přibližně o 0,04 nižší. Na testovací sadě dat se dosažené výsledky liší jen mírně.

Dataset Cancer		
Data	trénovací	testovací
Průměr	0,9397	0,9534
Medián	0,9471	0,9598
Maximum	0,9657	0,9942
Minimum	0,8771	0,8391

Tabulka 7.14: Tabulka přesnosti dosažené v [14] pro dataset Cancer.

Dataset Diabetes		
Data	trénovací	testovací
Průměr	0,7094	0,6622
Medián	0,7031	0,6510
Maximum	0,7526	0,7500
Minimum	0,6693	0,6094

Tabulka 7.15: Tabulka přesnosti dosažené v [14] pro dataset Diabetes.

V dalším experimentu byla vyvíjena neuronová síť pro kombinaci datových sad Iris a Banknotes. Výchozí parametry byly opět upraveny dle tabulky 7.16. Protože u předchozího experimentu se osvědčilo posunutí hodnoty O_l na hodnotu nalezenou jako nejlepší pro složitější dataset Diabetes, byla podobná korekce vyzkoušena i zde a také se osvědčila. Hodnoty celkové přesnosti jsou uvedeny v tabulce 7.17. Celková přesnost zde dosahovala průměrné hodnoty 0,81 pro trénovací a 0,8 pro testovací data.

Výsledky přesnosti pro jednotlivé problémy jsou uvedeny v tabulkách 7.18 a 7.19. Opět zde můžeme pozorovat snížení hodnot přesnosti pro síť navrhovanou pro oba problémy zároveň oproti samostatné variantě. Pro datovou sadu Banknotes došlo ke snížení průměrné přesnosti přibližně o 0,1, což je větší pokles, než by bylo žádoucí. Maximální dosažená hodnota přesnosti je zde podobně vysoká jako průměrná hodnota dosažené u sítě řešící jeden problém. Pro dataset Iris, který byl samostatně řešen velice úspěšně, taktéž došlo k poklesu přesnosti, přibližně o 0,06 na trénovacích datech a 0,1 na testovacích datech. Pro datovou sadu Iris se ale i při řešení společně s datovou sadou Banknotes podařilo najít síť, která zvládne klasifikovat 100 % testovacích dat datasetu Iris.

Zajímavé je také to, že minimální přesnost jednotlivých problémů je v tomto experimentu více vzdálena od hodnot průměrné přesnosti oproti ostatním experimentům. Tato skutečnost by mohla poukazovat na to, že nějaký z parametrů nebyl nastaven zcela správně, nebo že by bylo potřeba provedení většího množství generací, aby mohl vývin sítě u řešení s pomalejším vývojem pokročit trochu dále.

Parametr	Původní hodnota	Nová hodnota
NN_{max}	20	18
O_l	0,8	–0,25
Počet generací	1000	1800

Tabulka 7.16: Parametry změněné oproti výchozím parametrům.

Datasety Iris a Banknotes		
Data	trénovací	testovací
Průměr	0,8098	0,7934
Medián	0,7985	0,7664
Maximum	0,8964	0,9315
Minimum	0,7696	0,7105

Tabulka 7.17: Tabulka celkové přesnosti při návrhu sítě pro datasety Iris a Banknotes.

Dataset Banknotes		
Data	trénovací	testovací
Průměr	0,7915	0,7727
Medián	0,7696	0,7318
Maximum	0,8756	0,8863
Minimum	0,6832	0,7026

Tabulka 7.18: Tabulka přesnosti pro síť řešící více problémů pro dataset Banknotes.

Dataset Iris		
Data	trénovací	testovací
Průměr	0,8795	0,8930
Medián	0,8929	0,8947
Maximum	0,9554	1,0000
Minimum	0,6875	0,6053

Tabulka 7.19: Tabulka přesnosti pro síť řešící více problémů pro dataset Iris.

Metoda by teoreticky měla být schopna vyřešit i kombinaci více problémů současně. Pro úspěšné řešení libovolného množství problémů by ovšem musela být nalezena přesná příčina poklesu přesnosti při návrhu sítě pro kombinaci více datových sad. Ve zde provedených experimentech se ukázalo, že přidáním jedné datové sady může být způsoben pokles přesnosti na jednotlivých datasetech až o 10 %. V případě, že by se podobný pokles opakoval i po přidání další datové sady, klasifikační schopnost vytvořených sítí by rychle klesala a pro více problémů by se hodnota přesnosti pro jednotlivé datové sady mohla dostat na velmi špatnou úroveň.

Pokles hodnot přesnosti při kombinování řešení více datových sad může být způsoben například i tím, že metoda může preferovat současné řešení jen velmi podobných datových sad. Je ovšem těžké odhadnout, v čem podobnost spočívá, protože i přesto, že zde prezentovaná kombinace datových sad Banknotes a Iris obsahovala datové sady se stejným počtem vstupů a počtem výstupů lišícím se pouze o 1, pokles přesnosti byl vyšší než by bylo vhodné. Obě datové sady ale obsahovaly velmi odlišný počet vzorků, což také může hrát roli ve výsledných hodnotách přesnosti.

7.3 Výsledné soma a dendrit programy

Tato kapitola obsahuje ukázku výsledných soma a dendrit programů vytvořených při evoluci pomocí CGP. Statistiky počtu uzlů v soma a dendrit programech pro návrh sítě řešící jeden problém jsou uvedeny v tabulce 7.20. Lze si všimnout, že velikost dendrit programů byla průměrně o 12 uzlů vyšší než u soma programů. Pro soma program tedy bylo použito průměrně přibližně 9 % uzlů a pro dendrit program přibližně 10 % uzlů z celkových 800 uzlů dostupných v chromozomu CGP.

	Počet uzlů v CGP			
Program	Průměrný	Medián	Minimální	Maximální
Soma program	72	70	33	134
Dendrit program	81	81,5	35	124

Tabulka 7.20: Tabulka ukazující statistiku aktivních uzlů v soma a dendrit programech pro návrh sítě pro jeden problém.

Tabulka 7.21 ukazuje statistiku pro počet uzlů v soma a dendrit programech při vývinu sítě pro řešení dvou problémů současně. Při srování obou uvedených tabulek je zajímavé, že počet použitých uzlů byl průměrně stejný pro soma a dendrit programy pro návrh sítě pro jeden problém jako u programů pro návrh sítě řešící dva problémy současně.

	Počet uzlů v CGP			
Program	Průměrný	Medián	Minimální	Maximální
Soma program	74	70	45	110
Dendrit program	82	80	42	117

Tabulka 7.21: Tabulka ukazující statistiku aktivních uzlů v soma a dendrit programech pro ávrh sítě pro dva problémy.

Následující tabulky 7.22 a 7.23 obsahují ukázku výsledného soma a dendrit programu pro návrh sítě řešící jeden problém. Konkrétně tyto programy byly vyvinuty pomocí CGP pro návrh neuronové sítě provádějící klasifikaci datové sady Cancer. Výsledná neuronová síť vytvořená při jejich použití měla trénovací a testovací klasifikační přesnost 95 %. V obou programech byly použity všechny ze šesti dostupných funkcí.

Tabulka 7.22 obsahuje reprezentaci soma programu tak, jak byla implementována. Program tedy obsahuje nejprve 7 vstupních uzlů, reprezentujících postupně zdraví, bias a pozici neuronu a průměrné zdraví, váhu a pozici jeho dendritů. Následuje seznam vnitřních uzlů, kde každý uzel má přiřazeny dva vstupy. Tyto vstupy odkazují pomocí id na některého z aktuálnímu uzlu předcházejících uzlů. Každý uzel má navíc přiřazenu funkci, kterou pro své vstupy vykonává. Program má také tři výstupy, určující postupně novou hodnotu zdraví, pozice a biasu. Tyto tři výstupy jsou reprezentovány pomocí upravených uzlů, které nemají funkce a mají pouze jeden vstup který odkazuje na id uzlu, který má být v chromozomu uzlem výstupním.

V tabulce 7.23 je uveden dendrit program. Jeho reprezentace je stejná jako u soma programu, má tedy 7 vstupních uzlů, vnitřní uzly se dvěma vstupy odkazující na uzly předcházející a přiřazenou funkcí, a tři upravené výstupní uzly určující novou hodnotu váhy, pozice a zdraví dendritu.

Vstupy			
Id uzlu	Výstup uzlu		
0	Zdraví neuronu		
1	Bias neuronu		
2	Pozice neuronu		
3	Průměrné zdraví dendritů		
4	Průměrná váha dendritů		
5	Průměrná pozice dendritů		
6	Typ problému		
Vnitřní uzly			
Id uzlu	Id vstupu 1	Id vstupu 2	Funkce
7	2	4	add
8	1	4	i_step
10	5	1	step
11	0	0	i_step
12	8	3	xor
13	5	12	add
14	13	2	xor
15	0	3	sub
16	7	3	xor
17	7	11	mult
18	12	10	xor
19	16	7	sub
20	5	1	xor
21	6	6	i_step
22	16	0	sub
23	12	18	sub
26	0	6	add
27	20	0	mult
28	26	2	step
29	3	17	sub
31	20	7	mult
33	28	17	step
34	29	12	i_step
35	27	29	sub
36	17	35	add
40	36	22	step
41	40	35	mult
42	15	21	i_step

(pokračování)			
Id uzlu	Id vstupu 1	Id vstupu 2	Funkce
43	1	16	step
48	11	22	xor
51	1	16	i_step
52	17	8	mult
54	18	6	xor
55	6	40	sub
56	42	13	step
57	36	13	i_step
60	23	3	sub
61	54	31	sub
62	35	41	step
66	34	48	mult
73	52	66	i_step
79	56	20	i_step
85	43	57	add
86	26	55	step
91	21	51	mult
105	62	86	xor
115	79	91	add
133	19	2	add
136	36	55	step
137	85	33	xor
157	137	61	mult
187	136	133	mult
215	14	48	i_step
241	105	62	xor
252	215	73	sub
255	60	252	step
279	157	115	i_step
305	29	241	i_step
404	279	31	step
447	305	255	mult
Výstupy			
Id uzlu	Id vstupu	Výstup	
-	447	Zdraví	
-	404	Pozice	
-	187	Bias	

Tabulka 7.22: Ukázka výsledného soma programu.

Vstupy			
Id uzlu	Výstup uzlu		
0	Zdraví dendritu		
1	Váha dendritu		
2	Pozice dendritu		
3	Pozice rodičovského neuronu		
4	Zdraví rodičovského neuronu		
5	Bias rodičovského neuronu		
6	Typ problému		
Vnitřní uzly			
Id uzlu	Id vstupu 1	Id vstupu 2	Funkce
7	3	4	step
8	6	5	mult
9	5	2	step
10	8	6	sub
11	8	1	xor
12	5	10	xor
13	4	1	sub
14	3	2	mult
15	3	6	mult
16	9	5	mult
17	14	13	add
18	4	15	sub
19	9	7	add
20	5	12	xor
22	8	17	xor
23	20	1	mult
25	9	0	xor
27	25	15	xor
28	0	10	i_step
30	27	4	add
31	16	6	step
32	17	14	sub
34	1	11	xor
35	19	22	i_step
37	22	20	xor
39	16	30	mult
42	0	37	sub
45	20	3	i_step
46	34	28	i_step
47	22	8	step
51	16	25	sub

(pokračování)			
Id uzlu	Id vstupu 1	Id vstupu 2	Funkce
53	8	12	xor
55	12	11	sub
56	47	7	step
57	45	32	add
58	39	51	add
59	17	5	mult
61	51	13	xor
62	57	46	step
68	23	61	step
72	34	37	xor
73	18	2	xor
76	47	73	sub
79	53	56	step
80	56	55	add
85	35	72	i_step
86	58	59	xor
113	6	23	add
114	13	79	step
118	76	58	xor
127	86	68	xor
130	31	118	mult
138	130	85	xor
139	42	15	sub
140	80	55	xor
145	28	127	i_step
154	10	51	add
188	145	62	xor
194	114	30	xor
200	194	8	sub
224	113	140	sub
252	188	4	mult
253	252	154	sub
326	253	139	i_step
465	326	138	xor
Výstupy			
Id uzlu	Id vstupu	Výstup	
-	200	Váha	
-	465	Pozice	
-	224	Zdraví	

Tabulka 7.23: Ukázka výsledného dendrit programu.

Kapitola 8

Závěr

Cílem této práce bylo navrhnout a implementovat metodu pro evoluční návrh neuronové sítě pomocí generativního kódování.

V rámci práce byly nejprve nastudovány potřebné znalosti týkající se neuronových sítí a jejich vývinu pomocí evolučních algoritmů. Práce také poskytuje přehled některých metod využívaných pro vývin umělých neuronových sítí a detailněji popisuje metodu J. F. Millera, pro návrh neuronových sítí schopných řešit více problémů.

Na základě těchto znalostí se podařilo implementovat metodu vycházejí právě z práce J. F. Millera, která zvládne s využitím generativního kódování vytvořit neuronovou síť. Metoda spočívá v postupném vývoji mozku složeného z dendritů a neuronů, ze kterého je možné extrahovat po ukončení jeho vývoje tradiční neuronovou síť. Vývin mozku je řízen programy vytvořenými pomocí kartézského genetického programování. Metoda byla implementována v jazyce Python s využitím knihovny Numpy.

Neuronová síť navržená pomocí této metody dokáže řešit jednodušší klasifikační problémy s poměrně vysokou přesností. Experimenty prokázaly, že výsledná přesnost sítě je závislá na konkrétní datové sadě, nicméně pro jednodušší datasety se podařilo dosáhnout úspěšnosti přesahující 90 % jak pro trénovací tak i testovací data. Metoda je také citlivá na správné nastavení parametrů, kde i menší úpravy mohou způsobit velké změny v přesnosti výsledné neuronové sítě. Experimenty byly provedeny pro některé datové sady použité v [14], ale také pro jiné datové sady.

V práci se také podařilo implementovat i variantu metody umožnující návrh neuronové sítě řešící několik klasifikačních problémů naráz. Zde sice přesnost jednotlivých sítí při testování mírně klesla, ale i přesto výsledné sítě vykazovaly poměrně dobrou klasifikační schopnost. Výhodou současného vývinu sítě pro více problémů je skutečnost, že není nutné trénovat síť pro každý problém zvlášť a což by mělo přispět ke snížení výpočetní náročnosti neuronových sítí.

Další možné rozšíření práce by mohlo spočívat například v nalezení metody, která bude schopná nastavit všech 35 parametrů pro vývin mozku tak, aby bylo pro každý daný problém dosaženo co nejlepších výsledků. Nalezení správného nastavení všech parametrů se totiž ukázalo jako složitý problém, který je náročné řešit systematicky, a proto v rámci této práce probíhalo nastavení spíše intuitivně na základě dobré znalosti metody a řešených problémů. Dalším možným vylepšením by mohlo být zrychlení celé metody, aby mohla být lépe spouštěna s vyšším počtem generací.

Literatura

[1] Aggarwal, C. C. *Neural Networks and Deep Learning: A Textbook.* 1. vyd. Springer, 2018. ISBN 978-3-319-94462-3.

[2] Balaam, A. Developmental neural networks for agents. In:. Springer Verlag, 2003, sv. 2801, s. 154–163. ISBN 3540200576.

[3] Boers, E. J. a Kuiper, H. *Biological metaphors and the design of modular artificial neural networks.* Nizozemsko, 1992. Dimplomová práce. Leiden University.

[4] Cangelosi, A., Parisi, D. a Nolfi, S. Cell division and migration in a 'genotype' for neural networks. *Network (Bristol).* Taylor & Francis. 1994, sv. 5, č. 4, s. 497–515. ISSN 0954-898X. Dostupné z: http://www.tandfonline.com/doi/abs/10.1088/0954-898X_5_4_005.

[5] Eiben, A. a Smith, J. *Introduction to Evolutionary Computing (Natural Computing Series).* 2. vyd. Springer, 2015. ISBN 978-3-662-44873-1.

[6] Gruau, F., Whitley, D. a Pyeatt, L. A Comparison between Cellular Encoding and Direct Encoding for Genetic Neural Networks. In: *Proceedings of the 1st Annual Conference on Genetic Programming.* Cambridge, MA, USA: MIT Press, 1996, s. 81–89. ISBN 0262611279.

[7] Gruau, F. Automatic Definition of Modular Neural Networks. *Adaptive Behavior - ADAPT BEHAV.* Září 1994, sv. 3, s. 151–183. DOI: 10.1177/105971239400300202.

[8] Gurney, K. *An Introduction to Neural Networks.* 1. vyd. CRC Press, 1997. ISBN 1-85728-673-1.

[9] Joshi, A. V. *Machine Learning and Artificial Intelligence.* 1. vyd. Springer, 2019. ISBN 978-3-030-26621-9.

[10] Kitano, H. Designing Neural Networks Using Genetic Algorithms with Graph Generation System. *Complex Syst.* 1990, sv. 4.

[11] Kriesel, D. *A Brief Introduction to Neural Networks.* 2007. Dostupné z: http://www.dkriesel.com/en/science/neural_networks.

[12] Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K. et al. NSGA-Net: Neural Architecture Search Using Multi-Objective Genetic Algorithm. In:. New York, NY, USA: Association for Computing Machinery, 2019, s. 419–427. GECCO '19. DOI: 10.1145/3321707.3321729. ISBN 9781450361118.

[13] Miller, J. F. *Cartesian Genetic Programming*. 1. vyd. Springer, 2011. ISBN 978-3-642-17309-7.

[14] Miller, J., Wilson, D. a Cussat Blanc, S. Evolving Developmental Programs That Build Neural Networks for Solving Multiple Problems. In: *Genetic Programming Theory and Practice XVI*. Springer, 2019, s. 137—178. GEVO : Genetic and Evolutionary Computation book series. ISBN 9783030047344.

[15] Prechelt, L. PROBEN1 — A set of benchmarks and benchmarking rules for neural network training algorithms. Leden 1994.

[16] Rust, A., Adams, R. a Bolouri, H. Evolutionary Neural Topiary: Growing and Sculpting Artificial Neurons to Order. *Artificial Life VII: Procs. of the 7th Interntional Conferen*. MIT Press. 2000.

[17] Sekanina, L. *Evoluční hardware: od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. 1. vyd. Academia, 2009. ISBN 978-80-200-1729-1.

[18] Stanley, K. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*. New York: Kluwer Academic Publishers-Plenum Publishers. 2007, sv. 8, č. 2, s. 131–162. ISSN 1389-2576.

[19] Stanley, K. O., D'Ambrosio, D. B. a Gauci, J. A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artificial Life*. MIT Press. 2009, sv. 15, č. 2, s. 185–212. ISSN 1064-5462.

[20] Stanley, K. O. a Miikkulainen, R. Efficient Evolution Of Neural Network Topologies. In: Langdon, W. B., Cantu Paz, E., Mathias, K. E., Roy, R., Davis, D. et al., ed. *Proceedings of the Genetic and Evolutionary Computation Conference*. Piscataway, NJ: San Francisco, CA: Morgan Kaufmann, 2002, s. 1757–1762. Dostupné z: http://nn.cs.utexas.edu/?stanley:cec02.

[21] Stanley, K. O. a Miikkulainen, R. A Taxonomy for Artificial Embryogeny. *Artificial Life*. MIT Press. 2003, sv. 9, č. 2, s. 93–130. ISSN 1064-5462.

[22] Vanneschi, L. a Poli, R. *Genetic Programming — Introduction, Applications, Theory and Open Issues*. Springer, 2012.

[23] Vladimir, P. *Symbolic regression as a surrogate model in evolutionary algorithms*. Praha, CZ, 2016. Diplomová práce. České vysoké učení technické v Praze. Fakulta elektrotechnická. Katedra počítačů. Vedoucí práce Petr, P. Dostupné z: https://dspace.cvut.cz/handle/10467/65280.

[24] Volná, E. *Evoluční algoritmy a neuronové sítě*. Ostravská univerzita, 2012. Dostupné z: https://www.researchgate.net/publication/47063096_Neuronove_site_a_geneticke_algoritmy.

[25] Vosol, D. *Využití evolučních algoritmů při učení neuronových sítí*. Brno, CZ, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis-file/19256/19256.pdf.

[26] ZBOŘIL, F. V. *2. Acyklické a dopředné neuronové sítě. Algoritmus backpropagation.* [Přednáška pro předmět Soft Computing]. FIT VUT v Brně. Dostupné z: https://www.fit.vutbr.cz/study/courses/SFC/private/20sfc_2.pdf.