VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

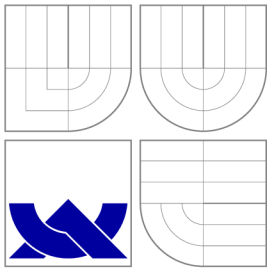
AKCELEROVANÉ NEURONOVÉ SÍTĚ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

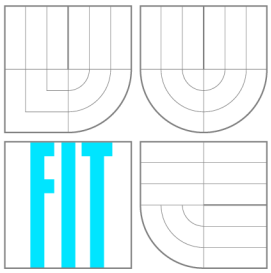
AUTOR PRÁCE
AUTHOR

MICHAL FLAX

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELEROVANÉ NEURONOVÉ SÍTĚ

ACCELERATED NEURAL NETWORKS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL FLAX

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTIN KRČMA

BRNO 2015

Abstrakt

Tato práce se zabývá simulací neuronových sítí a algoritmem Backpropagation. Simulace je akcelerována pomocí standardu OpenMP. Aplikace také umožňuje modifikovat strukturu neuronových sítí a simulovat tak nestandardní chování sítě.

Abstract

This thesis deals with neural network simulation and the Backpropagation algorithm. The simulation is accelerated using the OpenMP standard. The application is also able to modify the structure of neural networks and thus simulate their non-standard behavior.

Klíčová slova

Neuronové sítě, Backpropagation, akcelerace, OpenMP, modifikace struktury.

Keywords

Neural networks, Backpropagation, acceleration, OpenMP, modification of structure.

Citace

Michal Flax: Akcelerované neuronové sítě, bakalářská práce, Brno, FIT VUT v Brně, 2015

Akcelerované neuronové sítě

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Martina Krčmy.

.....
Michal Flax
14. května 2015

Poděkování

Rád bych poděkoval Ing. Martinu Krčmovi za odborné vedení, přístup, rady a materiály vedoucí k vypracování této práce.

© Michal Flax, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Bakalářská práce je zaměřena na vytvoření aplikace, která bude sloužit jako simulátor umělých neuronových sítí. Neuronové sítě jsou datové struktury, které jsou složeny z umělých neuronů. Ty představují zjednodušený model biologického neuronu a výsledná funkce neuronové sítě je určena způsobem propojení jednotlivých neuronů mezi sebou. Základní funkce neuronových sítí spočívají ve schopnosti učit se a generalizovat informace. Naučené znalosti jsou reprezentovány pomocí hodnot spojů mezi jednotlivými neurony.

Simulátor bude také umožňovat úpravu struktury neuronové sítě oproti běžným typům sítí. Mezi tyto úpravy patří přidání nestandardního spoje mezi neurony, odebrání spoje nebo neuronu a nastavení stálé výstupní hodnoty spoje nebo neuronu.

Zmíněné možnosti změny struktury jsou v simulátoru obsaženy z toho důvodu, že simulátor by měl umožnit nestandardní chování neuronové sítě a simulovat tak různé poruchy a nezvyklé chování. To je například zaseknutí spoje nebo neuronu, vznik nezvyklého spoje mezi dvěma neurony a znemožnění funkčnosti spoje nebo neuronu.

Aplikace je dále navržena s použitím standartu OpenMP, který umožňuje paralelizovat vybrané části zdrojového kódu. V simulátoru je používán, protože nevýhoda neuronových sítí je časová náročnost potřebná pro běh simulace. Cílem návrhu je pokusit se zmenšit čas potřebný pro výpočet a učení se neuronové sítě.

Návrh simulátoru je také vytvořen s ohledem na jednoduchou rozšiřitelnost. Tím je umožněno přidávat nové typy sítí a algoritmů.

V následujících kapitolách této práce jsou popsány základní principy neuronových sítí, jejich vlastnosti a nejvíce používané typy sítí. Dále následuje kapitola o standartu OpenMP a část obsahující popis návrhu a implementace vlastní aplikace. V předposlední kapitole jsou potom popsány experimenty a jejich výsledky.

Kapitola 2

Neuronové sítě

Kapitola se zabývá popisem neuronových sítí. Na začátku kapitoly je stručně popsána historie tohoto vědního oboru. Následuje podkapitola zabývající se obecnými vlastnostmi neuronových sítí. Následně jsou podrobněji popsány nejznámější a nejpoužívanější typy neuronových sítí.

Za počátek vzniku oboru neuronových sítí je považován rok 1943, kdy Warren McCulloch a Walter Pitts publikovali matematický model neuronu [16]. V roce 1949 Donald Hebb zveřejnil učící pravidlo pro nastavení váhy mezi dvěma neurony [6]. V roce 1947 Frank Rosenblatt navrhl model perceptronu, který je zobecněním základního modelu neuronu pro reálný obor čísel a navrhl pro něj učící algoritmus [21]. V roce 1959 Bernard Widrow vytvořil model neuronu ADALINE a popsal k němu učící pravidlo [25].

V roce 1969 byla vydána práce Perceptrons, ve které se Marvin Minsky a Seymour Papert snažili diskreditovat obor neuronových sítí. Jejich argumentace byla založena na faktu, že jeden perceptron nedokáže vyřešit logickou funkci XOR. Tento problém lze sice vyřešit pomocí dvouvrstvé perceptronové sítě, ale protože pro vícevrstvou perceptronovou síť v té době nebyl znám učící algoritmus, pozastavila tato práce úspěšně zájem o neuronové sítě až do začátku 80. let [18].

V letech 1982 a 1984 publikoval John Hopfield na základě poznatků z oblasti magnetických materiálů Hopfieldovu síť a popsal použiti energetické funkce pro její učení se [8, 9]. V roce 1982 popsal Teuvo Kohonen myšlenku kompetičního učení a navrhl Kohonenovy samoorganizační mapy a učící vektorovou kvantizaci [11]. V roce 1986 zveřejnily David Rumelhart, Geoffrey Hinton a Ronald Williams práci, ve které publikovali učící algoritmus zpětného síření chyby (backpropagation) pro vícevrstvou perceptronovou síť [22]. V následujících letech se vícevrstvá perceptronová síť stala nejvíce známou a používanou.

2.1 Popis neuronových sítí

Pojem umělá neuronová síť označuje abstraktní matematický model, který je inspirován strukturou lidského mozku. Skládá se z umělých neuronů, které jsou vytvořeny na základě biologických neuronů a jednotlivých vazeb mezi nimi. Architektura sítě je určena počtem neuronů a způsobem jejich propojení [14]. Právě schopnost adaptovat váhy těchto vazeb umožňuje učení neuronové sítě. Způsoby učení lze rozdělit na učení s učitelem a bez učitele, které budou podrobněji popsány v kapitole 2.1.3.

2.1.1 Formální neuron

Matematický model formálního neuronu je založen na biologickém neuronu. Každý neuron má x_1 až x_n vstupů, které jsou ohodnoceny w_1 až w_n váhami a jeden výstup y. Jednotlivé váhy určují míru vlivu příslušného vstupu na stav neuronu. Vážený součet vstupů určuje vnitřní potenciál neuronu ξ (2.1). Přímý výstup neuronu y je popsán vztahem (2.2), kde f je aktivační (přenosová) funkce a její druh je zvolený v závislosti na aktuálním výpočtu. Některé běžně používané funkce jsou uvedeny v kapitole 2.1.2 [14]. Ilustrace formálního neuronu je uvedena na ilustraci 2.1 [15].

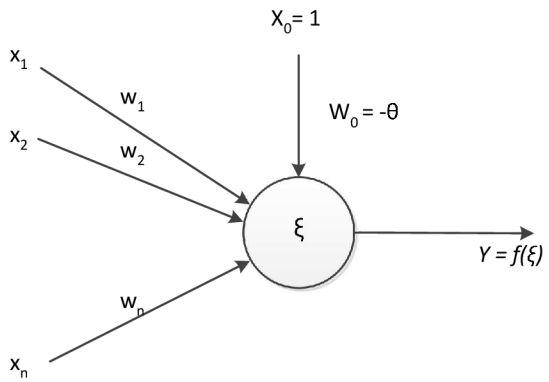
Matematickou úpravou je možné docílit toho, že aktivační funkce f bude mít nulový práh. Místo toho budeme práh chápat jako váhu (bias) formálního vstupu neuronu x_0 s konstantní hodnotou $x_0 = 1$, jehož váha je určena zápornou hodnotou prahu $w_0 = -\Theta$ [14].

Hodnota biasu umožňuje posouvat graf aktivační funkce po ose x (záporná hodnota vpravo a kladná vlevo) a plní tak stejnou funkci jako práh aktivační funkce. Výhodou oproti standardní implementaci prahu je efektivnější výpočet, protože v rámci struktury sítě se jedná pouze o další váhu v síti. Je možné říci, že neurony s nenulovou bias hodnotou, by se měly naučit i takové vstupy, které se neurony s nulovým prahem nejsou schopné naučit [1, 3].

V historii neuronových sítí se vyskytoval ještě další pohled na funkci prahu u jednotlivých neuronů. Jednalo se o hodnotu, které musel dosáhnout vážený součet jednotlivých vstupů příslušného neuronu, aby byla nastavena a aktivována výstupní hodnota tohoto neuronu [24].

$$\xi = \sum_{i=0}^{n} x_i w_i \tag{2.1}$$

$$y = f(\xi) \tag{2.2}$$



Obrázek 2.1: Formální neuron

2.1.2 Aktivační Funkce

V neuronových sítích může být použit různý typ aktivační funkce, tato kapitola obsahuje základní popis nejvíce používaných aktivačních funkcí.

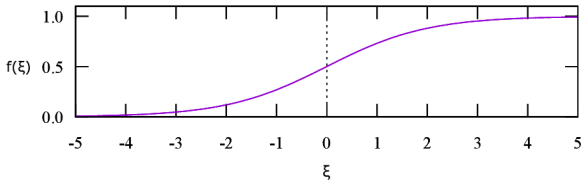
Jedná se o standardní sigmoidu (2.3) [14], hyperbolický tangens (2.4) [2], nespojitou skokovou funkci (2.5) [26] a saturovanou lineární funkci (2.6) [14].

$$f(\xi) = \frac{1}{1 + e^{-\xi}} \tag{2.3}$$

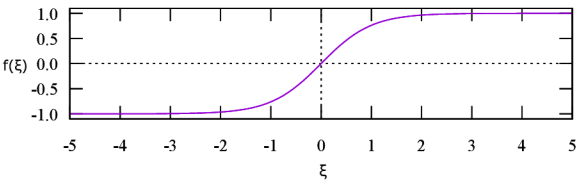
$$f(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}} \tag{2.4}$$

$$f(\xi) = \begin{cases} 1 & \xi > 0 \\ \xi & \xi = 0 \\ 0 & \xi < 0 \end{cases} \tag{2.5}$$

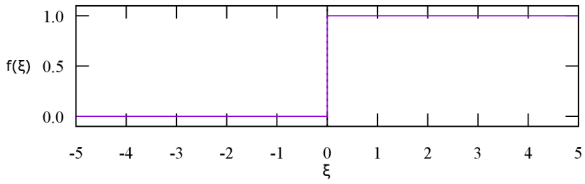
$$f(\xi) = \begin{cases} 1 & \xi > 1 \\ \xi & 0 \leq \xi \leq 1 \\ 0 & \xi < 0 \end{cases} \tag{2.6}$$



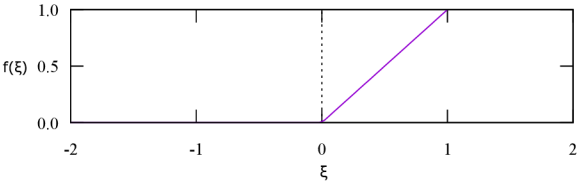
(a) Sigmoida



(b) Hyperbolický tangens



(c) Nespojitá skoková funkce



(d) Saturovaná lineární funkce

2.1.3 Dělení neuronových sítí

Ačkoli se neuronové sítě skládají ze stejných umělých neuronů, můžou být jejich vlastnosti a charakteristika odlišná v závislosti na struktuře sítě. V této kapitole jsou popsány některé způsoby klasifikace neuronových sítí a to podle principu učení, uspořádání vrstev a směru výstupu.

Princip učení lze rozdělit na dva druhy, učení s učitelem a bez učitele. Pro učení s učitelem jsou pro testovací data známé i požadované správné výstupy ze sítě. Učení potom probíhá za pomoci srovnávání aktuálních výstupů se vzorovými daty. Sítě používající učení bez učitele naopak nepotřebují správné řešení a dokáží se učit samostatně [19].

Podle struktury uspořádání neuronů je možné dělení na vícevrstvé a nevrstvené neuronové sítě. Ve vícevrstvých sítí jsou neurony organizované do vrstev. Jednotlivé neurony jsou propojené s neurony ve vedlejších vrstvách a nejsou spojeny s jinými neurony v dané vrstvě. Příkladem je například Vícevrstvá perceptronová síť [19].

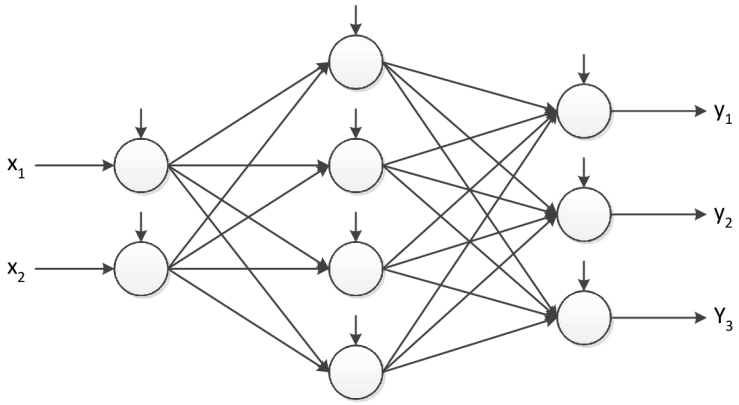
Struktura sítě může být také uspořádaná bez zřetelného uspořádání do jednotlivých vrstev. Zde může být každý neuron spojen s libovolným neuronem a neuron může být zároveň vstupní í výstupní. Typická příklad je například Hopfieldova síť [19].

V závislosti na směru výstupních vazeb, je možné neuronové sítě rozdělit na rekurentní a nerekurentní sítě. U rekurentního způsobu propojení mohou být výstupy jednotlivých

neuronů přivedeny i na vstup stejného neuronu nebo vrstvy. Jedná se zpětnou vazbu a typický příklad je Hopfieldova síť [19].

2.2 Vícevrstvá perceptronová síť

Vícevrstvá perceptronová síť je v současnosti nejvíce známý a rozšířený typ neuronové sítě. Jde o vícevrstvou síť složenou z perceptronů, která funguje na principu učení s učitelem. Síť se skládá z jedné vstupní vrstvy, $0 - n$ skrytých vrstev a jedné výstupní vrstvy. Sousední vrstvy jsou mezi sebou plně propojeny, to znamená, že každý neuron v dané vrstvě je propojen s každým neuronem v sousedních vrstvách. Z důvodu, že vstupní vrstva pouze přeposílá vstupní hodnoty, jsou možné dva způsoby určování počtu vrstev v síti. První způsob neuvažuje vstupní vrstvu jako regulérní vrstvu neuronů, druhý ano [19]. Tato práce používá druhý způsob zápisu.



Obrázek 2.3: Příklad vícevrstvé perceptronové sítě se strukturou 2-4-3.

2.2.1 Výpočet ve vícevrstvé síti

Na začátku výpočtu je na vstupy sítě nastaven vstupní vektor, který je v další fázi výpočtu postupně transformován přes jednotlivé vrstvy sítě až k výstupům sítě. Postupně se vyhodnocují neurony v jednotlivých vrstvách, kdy platí, že pro následující vrstvu je nutné znát aktuální výstupy neuronů v předcházející vrstvě. Ty jsou vypočteny podle vztahů pro vnitřní potenciál (2.7) a výstup neuronu (2.8), kde index i označuje neurony v předchozí vrstvě, index j neurony v aktuální vrstvě, f aktivační funkci neuronu a w_{ij} váhu spoje mezi i-tým neuronem v předcházející vrstvě a j-tým neuronem v aktuální vrstvě. Výstupní hodnoty sítě se přímo rovnají výstupním hodnotám neuronů v poslední vrstvě [15].

$$\xi_j = \sum_i y_i w_{ij} \tag{2.7}$$

$$y_j = f(\xi_j) \tag{2.8}$$

Platí, že druh aktivační funkce lze zvolit libovolně, ale pro nediferencovatelné funkce nebude správně fungovat učící pravidlo zpětného síření (backpropagation).

2.2.2 Učení ve vícevrstvé síti

Cílem učení ve vícevrstvé perceptronové síti je takové nastavení hodnot jednotlivých vah, aby byl rozdíl mezi skutečnými a požadovánými výstupními hodnotami minimální. To znamená minimalizovat hodnotu celkové chyby sítě, která je definována jako součet chyb jednotlivých testovacích vzorů (2.10). Chyba odpovídající testovacímu vzoru je určena vztahem (2.9), kde index k označuje testovací vzory, index j jednotlivé neurony ve výstupní vrstvě a t_j značí j-tou hodnotu výstupního vektoru náležejícímu trénovacímu vzoru k [15, 19].

$$E_k = \frac{1}{2} \sum_j (t_j - y_j)^2 \tag{2.9}$$

$$E = \sum_k E_k \tag{2.10}$$

Na začátku procesu učení jsou jednotlivé váhy inicializovány na náhodné malé hodnoty, například v intervalu $(-0.5, 0.5)$ [19] nebo $\langle -\frac{2}{s}, \frac{2}{s} \rangle$ [15], kde s je počet vstupních vazeb neuronu, do kterého vazba vstupuje. Následně jsou váhy postupně aktualizovány podle vzorce (2.11), kde $\Delta w_{ij}^{(n)}$ je přírůstek, o který se změní daná váha. Tato hodnota je určena vztahem (2.12), který můžeme dále nepovinně upravit pomocí vzorce (2.13) [19].

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} + \Delta w_{ij}^{(n)} \tag{2.11}$$

$$\Delta w_{ij}^{(n)} = -\eta \left(\frac{\partial E}{\partial w_{ij}} \right) \tag{2.12}$$

$$\Delta w_{ij}^{(n)} = -\eta \left(\frac{\partial E}{\partial w_{ij}} \right) + \alpha \Delta w_{ij}^{(n-1)} \tag{2.13}$$

Parametr η se označuje jako parametr učení (learning rate) a umožňuje ovlivňovat rychlost a konvergenci k požadovanému výsledku. Jedná se o pozitivní konstantu, která ovlivňuje rychlost snižování chyby sítě [15].

Parametr α se nazývá parametr hybnosti (momentum rate) a představuje setrvačnost pohybu po chybové funkci. V rámci vztahu (2.13) slouží ke zmenšení pravděpodobnosti uváznutí chybové funkce v lokálním minimu, což je běžný problém gradientních metod. Pokud funkce míří k lokálnímu minimu, je možné ho překročit o jeden krok, jehož délka je určena velikostí předchozího kroku a parametrem hybnosti [15].

Přesné hodnoty parametru η a α, které by byly obecně vhodné nelze stanovit a volí se v závislosti na zvoleném výpočtu. Velikost parametrů je možné v průběhu učení měnit a ovlivňovat tak rychlost učení.

2.2.3 Backpropagation

Metoda zpětného šíření (backpropagation) nebo jiným názvem zobecněné delta pravidlo, je učící algoritmus, který umožňuje vypočítat hodnotu parciální derivace $\frac{\partial E}{\partial w_{ij}}$ ze vzorce (2.12). Bylo publikováno roku 1986 D. Rumelhaltem, G. Hintonem a R. Williamsem [22].

Protože pro parciální derivaci chyby podle vah sítě lze použít pravidlo pro derivaci složené funkce, je možné odvodit vztah (2.14) [15]. První parciální derivaci $\frac{\partial \xi_j}{\partial w_{ij}}$ lze určit pomocí substituce $\xi_j = \sum_k w_{kj} y_k$ a odvodit tak vztah (2.15) [19].

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j}\frac{\partial \xi_j}{\partial w_{ij}} \tag{2.14}$$

$$\frac{\partial \xi_j}{\partial w_{ij}} = y_i \tag{2.15}$$

Tvar parciální derivace $\frac{\partial y_j}{\partial \xi_j}$ je závislý na zvoleném typu aktivační funkce a jedná se o její derivaci. Pokud je aktivační funkce sigmoida (2.3), platí vztah (2.16), kde λ určuje strmost aktivační funkce [15].

$$\frac{\partial y_j}{\partial \xi_j} = \lambda y_j(1 - y_j) \tag{2.16}$$

Zbývající parciální derivace $\frac{\partial E}{\partial y_j}$ má rozdílný tvar pro výstupní a skryté vrstvy. Její hodnota je získána metodou zpětného šíření, která spočívá v postupném procházením vrstev sítě směrem od výstupní vrstvy. Opačný směr je dán tím, že pro výpočet parciální derivace v aktuální vrstvě je potřebné znát její hodnotu v předchozí vrstvě ve směru od výstupní vrstvy. Hodnota pro výstupní vrstvu je přímo určena rozdílem mezi skutečným a požadovaným výstupem (2.17). Pro skryté vrstvy potom platí vztah (2.18) [15].

$$\frac{\partial E}{\partial y_j} = y_j - t_j \tag{2.17}$$

$$\frac{\partial E}{\partial y_j} = \sum_r \frac{\partial E}{\partial y_j}\lambda y_r(1 - y_r)w_{rj} \tag{2.18}$$

Po dosazení odvozených vztahů do vzorce (2.14) dostaneme pro výstupní vrstvu vztah (2.19) a pro skryté vrstvy (2.20).

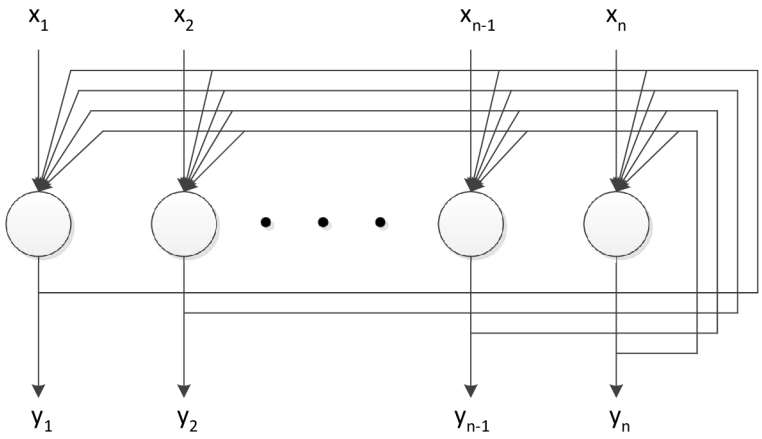
$$\Delta w_{ij}^{(n)} = -\eta y_i \lambda y_j(1 - y_j)(y_j - t_j) \tag{2.19}$$

$$\Delta w_{ij}^{(n)} = -\eta y_i \lambda y_j(1 - y_j)\sum_r \frac{\partial E}{\partial y_j}\lambda y_r(1 - y_r)w_{rj} \tag{2.20}$$

2.3 Hopfieldova síť

Další známý a často používaný typ neuronové sítě je Hopfieldova síť. Používá se primárně jako autoasociativní paměť a pro řešení optimalizačních problémů. Jde o jednovrstvou a rekurentní síť. V síti jsou všechny neurony v roli vstupních i výstupních neuronů, proto je počet neuronů roven počtu vstupů a výstupů. Jednotlivé neurony v Hopfieldově síti jsou mezi sebou plně propojeny, tj. výstup jednotlivých neuronů je přiveden na vstupy všech ostatních neuronů. Protože platí, že váhy vazeb mezi dvěma neurony jsou stejné v obou směrech $w_{ij} = w_{ji}$, je někdy Hopfieldova síť nazývána symetrickou. Protože v síti neexistuje zpětná vazba v rámci jednoho neuronu, jinak řečeno výstup neuronu není přiveden na vstup stejného neuronu, platí vztah $w_{ii} = 0$ [15].

Existují dvě hlavní omezení, pokud je Hopfieldova síť použita jako autoasociativní paměť. První je relativně malá kapacita paměti spolu s velkou paměťovou náročností počtu vah v síti. Ten je kvadraticky závislý na počtu vstupů (O^2) [15]. Například pro 50 vstupů bude síť obsahovat $50^2 = 2500$ vah.



Obrázek 2.4: Příklad struktury Hopfieldovy sítě.

Kapacita paměti je závislá na počtu neuronů n a počtu trénovacích vzorů s. Pro minimalizování pravděpodobnosti chyby při rozpoznávání vzoru, je nutné dodržet vztah $s \leq 0.138n$ [15].

Druhá nevýhoda této sítě je ta, že při malé Hammingově vzdálenosti mezi trénovacími vzory, která vyjadřuje rozdíl mezi jednotlivými vzory, může dojít k nepřesné identifikaci předloženého vzoru.

Výhodou Hopfieldovy sítě je to, že pro každý naučený vzor, je schopná správně klasifikovat i jeho inverzní vzor [15].

2.3.1 Učení v Hopfieldově síti

Učení v Hopfieldově spočívá ve výpočtu hodnot jednotlivých vah na základě předložených vzorů a po tomto nastavení se váhy již dále nemění.

Vztah pro výpočet hodnot vah je odvozen z energetické funkce (2.21), kde indexy i a j označují jednotlivé neurony a index k vybraný trénovací vzor. Pro naučení vybraného vzoru je potřeba minimalizovat hodnotu energetické funkce v určitém bodě, kdy se zároveň nesmí poškodit již naučené informace. Z tohoto důvodu je možné funkci (2.21) rozložit na (2.22), kde první část určuje příspěvek aktuálního vzoru k energetické funkci a druhá část jsou příspěvky od už naučených vzorů [15].

$$E = -\frac{1}{2}\sum_{i}\sum_{j\neq i} w_{ij}x_i x_j \tag{2.21}$$

$$E = -\frac{1}{2}\sum_{i}\sum_{j\neq i} w_{ij}^{k}x_i x_j - \frac{1}{2}\sum_{i}\sum_{j\neq i} w_{ij}^{*}x_i x_j \tag{2.22}$$

Pro úspěšné naučení vzoru je potřeba minimalizovat jeho příspěvek k energetické funkci. Je tedy nutné minimalizovat výraz (2.23). Protože vstupní hodnoty mohou nabývat i záporných hodnot, ale hodnota x_{ki}^2 je vždy kladná, je možné vztah upravit na (2.24) [15].

$$\sum_{i}\sum_{j\neq i} w_{ij}^{k}x_{ki}x_{kj} \tag{2.23}$$

$$\sum_i \sum_{j \neq i} w_{ij}^k x_{ki} x_{kj} = \sum_i \sum_{j \neq i} x_{ki}^2 x_{kj}^2 \tag{2.24}$$

Z předchozího vztahu je možné přímo odvodit vzorec pro výpočet vah aktuálního trénovacího vzoru (2.25). Výsledná hodnota vah pro všechny trénovací vzory je následně určena jako součet vah jednotlivých vzorů (2.26) [15].

$$w_{ij}^k = x_{ki} x_{kj} \tag{2.25}$$

$$w_{ij} = \sum_k x_{ki} x_{kj} \tag{2.26}$$

2.3.2 Výpočet v Hopfieldově sítí

Rozpoznávání předloženého vzoru je v Hopfieldově sítě založeno na opakované aktualizaci výstupů jednotlivých neuronů. Jakmile se výstupy všech neuronů po dvě následující iterace nezmění, je možné výstupy neuronů považovat za konečný výstup sítě [15].

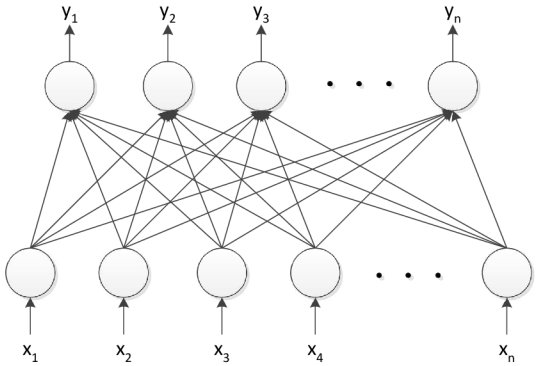
Na začátku výpočtu v čase $t(0)$ jsou výstupy neuronů nastaveny na hodnoty vstupů sítě. Následně je podle zvolených kriterií vybrán neuron a jeho stav je aktualizován na základě vztahů (2.27) a (2.28) [14]. Tento postup je opakován, dokud výstupy všech neuronů nejsou ve stabilním stavu.

$$\xi_j^{(t-1)} = \sum_{i=1}^{n} w_{ji} y_i^{(t-1)} \tag{2.27}$$

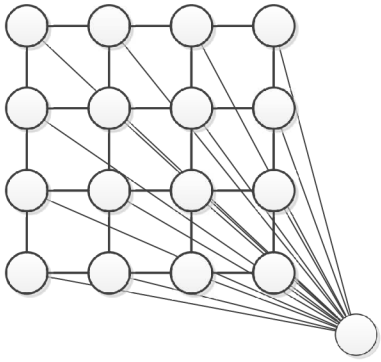
$$y_j^{(t)} = \begin{cases} 1 & \xi_j^{(t-1)} > 0 \\ y_j^{(t-1)} & \xi_j^{(t-1)} = 0 \\ -1 & \xi_j^{(t-1)} < 0 \end{cases} \tag{2.28}$$

2.4 Kohonenovy samoorganizační mapy

Kohonenova síť se řadí mezi sítě, které je založena na principu učení bez učitele. Jde o dvouvrstvou síť, kde první vrstva slouží pouze k přenosu vstupních hodnot na vstupy všech neuronů v následující vrstvě. Druhá vrstva je složena z tzv. Kohonenových neuronů a nazývá se Kohonenova vrstva (někdy také kompetiční). Jednotlivé neurony v první vrstvě jsou spojeny s každým neuronem v druhé vrstvě. Neurony kompetiční vrstvy mohou být mezi sebou dále propojeny. Struktura vzniklá pomocí těchto postranních vazeb se nazývá topologická mřížka. Jeden z možných tvarů topologické mřížky je uveden na následujícím obrázku.



(a) Příklad struktury Kohonenovy sítě.



(b) Příklad čtvercové topologické mřížky.

2.4.1 Výpočet a učení v Kohonenově síti

Výpočet v Kohonenově síti je založen porovnávání neuronů v kompetiční vrstvě. Nejprve je pro každý neuron vypočítána vzdálenost mezi vstupním vzorem a váhami neuronu podle vztahu (2.29), kde index i označuje neurony vstupní vrstvy a index j jednotlivé neurony v kompetiční vrstvě. Následně je vybrán neuron s nejmenší vzdáleností a tento neuron je aktivován. Zbylé neurony zůstávají neaktivní [15].

$$d_j = \sum_{i=1}^{n} (x_i - w_{ij})^2 \tag{2.29}$$

Učení v Kohonenově síti je také založeno na vztahu (2.29). Pro každý předložený trénovací vzor je nejdříve podle vztahu (2.29) nalezen neuron s nejmenší vzdáleností a následně jsou podle vzorce (2.30) upraveny váhy příslušející k tomuto neuronu a neuronů v jeho okolí. Tento postup je opakován do vyčerpání předepsaného počtu iterací, protože Kohonenova síť nemá definovanou chybovou funkci a z tohoto důvodu nemůžeme určit hodnotu chyby, při které by bylo učení ukončeno [15].

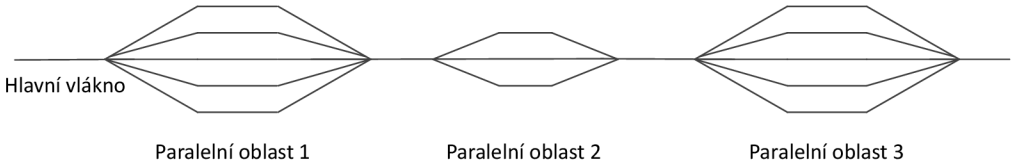
Tvar a velikost okolí neuronu jsou určeny topologickou mřížkou a nezáleží na skutečné poloze jednotlivých neuronů. Okolí se většinou během výpočtu zmenšuje a na konci zahrnuje pouze aktivní neuron.

$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)h(v,t)(x_i(t) - w_{ij}(t)) \tag{2.30}$$

Kapitola 3

OpenMP

OpenMP je standard pro podporu paralelního programování se sdílenou pamětí pro programy vytvářené v jazycích C/C++ a Fortran. Je vyvíjen mezinárodním konsorciem firem od roku 1997. Jeho implementace je postavena na principu vícevláknové paralelizace, kde hlavní vlákno vytvoří skupinu vedlejších vláken s kterými se následně dělí o práci. Po ukončení paralelní oblasti skupina vláken zaniká a proces se synchronizuje v původním vlákně (fork-join model) [10][20].



Obrázek 3.1: Příklad fork-join modelu.

Použití OpenMP je založeno na direktivách, kterými programátor označuje části programu, které mají být vykonávány paralelně. Protože OpenMP nezajišťuje synchronizaci vláken a paměti, musí být ošetřena programátorem jinak hrozí riziko chyb v programech. Direktivy je také možné přidávat postupně a paralelizovat tak už existující aplikaci [10]. V C/C++ jsou direktivy označeny začátečním řetězcem `#pragma omp`.

Sdílení paměti je založeno na systému, kdy má každé vlákno přístup do sdílené paměti a dále vlastní místo v paměti přístupné pouze pro konkrétní vlákno. V základním nastavení jsou všechny proměnné sdílené (`shared`) mezi vlákny. Pokud je proměnná označena jako soukromá (`private`), jsou vytvořeny její kopie v soukromé části paměti jednotlivých vláken, které ovšem nejsou inicializovány a neobsahují tak hodnotu originální proměnné (výjimku představuje klauzule `firstprivate`) [20].

3.1 Základní direktivy

Podkapitola popisuje základní direktivy a klauzule standardu OpenMP. Nejdříve jsou uvedeny direktivy a za nimi následuje seznam některých klauzulí, které lze použít v uvedených direktivách. Popis jednotlivých položek byl vytvořen na základě zdrojů [10][20] a [5].

`#pragma omp parallel [clause[ clause ...]`

Direktiva označuje oblast, která bude paralelně prováděna více vlákny. Pomocí speci-

ální klauzule `number_threads(integer)` lze zvolit počet vytvořených vláken. Jednotlivé direktivy `parallel` je možné zanořovat, pokud je nastavena systémová proměnná `OMP_NESTED` na hodnotu `true`.

`#pragma omp for [clause[ clause ...]`
Jednotlivé iterace cyklu označeného touto direktivou budou rozděleny mezi více vláken a vykonávány paralelně. Počet iterací musí být znám před začátkem zpracovávání cyklu. Pomocí klauzule `schedule(kind,[chunk_size])` lze nastavit způsob přidělování iterací jednotlivým vláknům.

`#pragma omp section [clause[ clause ...]`
Direktiva umožňuje zpracovávat rozdílné části kódu paralelně. Každou část provede pouze jedno vlákno.

`#pragma omp single [clause[ clause ...]`
Sekce označená touto direktivou bude zpracována pouze vláknem, které jako první dorazí k této direktivě.

`#pragma omp master [clause[ clause ...]`
Direktiva označuje blok, který provede pouze hlavní vlákno.

`#pragma omp critical [clause[ clause ...]`
Direktiva označuje kritický blok kódu, který bude prováděn v aktuálním čase pouze jedním vláknem.

`#pragma omp barrier [clause[ clause ...]`
Direktiva slouží jako bariéra. Jednotlivá vlákna mohou pokračovat pouze za předpokladu, že všechny vlákna dorazila k bariéře.

`private(list)`
Určuje seznam proměnných (`list`), které budou označeny jako soukromé. Každé vlákno bude mít vlastní kopii jednotlivých proměnných.

`firstprivate(list)`
Stejný význam jako klauzule `private`, ale jednotlivé kopie proměnných budou inicializovány na hodnotu originální proměnné.

`shared(list)`
Seznam `list` určuje, které proměnné budou sdíleny mezi vlákny. Jedná se o standardní nastavení všech proměnných.

`reduction(operator:list)`
Na konci označené paralelní sekce je provedena redukce kopií soukromých proměnných z jednotlivých vláken pomocí zvoleného operátoru a hodnota je uložena do původní proměnné.

Kapitola 4

Návrh simulátoru

Návrh struktury simulátoru a tříd z kterých se skládá vychází ze struktury umělých neuronových sítí. Tento základ je dále rozšířen s ohledem na umožnění budoucího rozšíření aplikace o další typy neuronových sítí, aktivačních funkcí a učících nebo výpočetních algoritmů. Také je vytvořen s ohledem na možnost přímé modifikace jednotlivých prvků neuronové sítě.

4.1 Základní princip návrhu

Jádro simulátoru je obsaženo ve třídě `NeuralNetwork`, která představuje rozšířený model neuronové sítě. Objekt této třídy se stará o většinu práce spojené s přípravou na běh simulace. Zajišťuje vytváření jednotlivých objektů, ze kterých se skládá neuronová síť, zpracovávání konfiguračních a datových souborů, inicializaci sítě a další potřebné činnosti pro samotný běh simulace.

Základní model neuronové sítě je založen na objektech tříd, které reprezentují jednotlivé části neuronové sítě. Jedná se o třídy modelující umělý neuron, vstupní a výstupní body neuronové sítě a vazby mezi jednotlivými objekty v neuronové síti (neurony, vstupy, výstupy). Příklad struktury je uveden na ilustraci 4.1.

Průběh simulace je následně řízen pomocí zvolené metody třídy `AlgorithmContainer`. Tato třída obsahuje a zapouzdřuje do jednoho celku metody, které implementují jednotlivé učící a výpočetní algoritmy. Tento způsob implementace umožňuje oddělit algoritmy od jádra simulátoru. Cílem této myšlenky je oddělit implementaci jednotlivých algoritmů od implementace samotné neuronové sítě a vytvořit tak logicky strukturovaný návrh, který bude umožňovat rozšiřování aplikace o další učící algoritmy bez větších zásahů do ostatních zdrojových tříd.

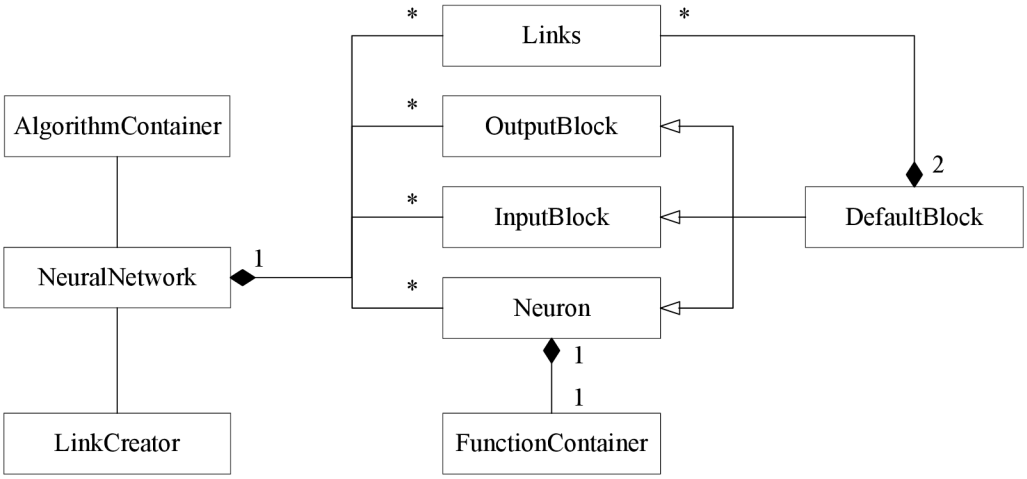
Na stejném principu je postaven návrh třídy `FunctionContainer`, která obsahuje jednotlivé aktivační funkce a třídy `LinkCreator` jejíž metody vytváří vazby mezi neurony, vstupní a výstupní body neuronové sítě a tím vytváří požadovanou strukturu určenou na základě zvoleného typu neuronové sítě.

Obrázek 4.1: Příklad stavební struktury neuronové sítě.

4.2 Návrh funkce tříd a jejich struktury

Základem návrhu aplikace je třída `NeuralNetwork`, která vykonává činnosti nutné pro běh simulace. Jedná se o konfiguraci a inicializaci sítě na základě informací v konfiguračním souboru, vytváření požadovaného počtu neuronů a inicializaci jejich aktivačních funkcí, načtení vstupních dat, průběžný výpis výstupních hodnot sítě aj. Dále také ukládá konfiguraci sítě po skončení simulace. Také jsou zde uloženy specifické konstanty potřebné pro běh jednotlivých algoritmů, které je možné nastavit v rámci konfigurace.

Objekt této třídy dále obsahuje ukazatele na vytvořené objekty jednotlivých tříd, které implementují části neuronových sítí.



Obrázek 4.2: Zjednodušený třídní UML diagram navržených tříd.

4.2.1 Třídy implementující části neuronové sítě

Jednotlivé stavební objekty, které tvoří implementovaný model neuronové sítě, jsou objekty tříd reprezentující umělý neuron, vstupní bod sítě, výstupní bod sítě a vazbu mezi dvojicí objektů předchozích tříd. Třídy `Neuron`, `InputBlock` a `OutputBlock` jsou implementovány jako odvozené třídy od třídy `DefaultBlock`. Tímto je umožněno navázat k objektu třídy `Links`, který reprezentuje vazbu mezi dvěma bloky, všechny typy objektů v neuronové sítí. Následuje stručný popis funkce jednotlivých tříd, které reprezentují objekty v tomto modelu.

Třída `Neuron` reprezentuje model umělého neuronu. Jeho hlavní funkce spočívá ve výpočtu nové výstupní hodnoty neuronu. Toho je docíleno pomocí sečtením vstupních hodnot od jednotlivých vstupních vazeb náležejících k objektu a pomocí aktivační funkce, která je implementována jako metoda ve třídě `FunctionContainer`.

Třídy `InputBlock` a `OutputBlock` modelují jednotlivé vstupy a výstupy sítě a slouží pouze k nastavení vstupních hodnot sítě a poskytnutí výstupních hodnot.

Třída `Links` modeluje jednotlivé vazby mezi objekty v síti. Každý objekt odvozený od této třídy je přiřazen ke dvěma stavebním objektům, kde první slouží jako vstupní objekt a druhý jako výstupní. Hlavní funkce vazebního objektu je založena na vyžádání výstupní hodnoty od vstupního objektu a jeho distribuci na vstup výstupního objektu. Protože každý

spoj může mít určenou váhu, je možné výstupní hodnotu upravit pomocí této váhy a až poté distribuovat na výstup vazby.

4.2.2 Kontejnerové třídy

Návrh také obsahuje několik tříd, které seskupují implementaci podobných operací do stejného místa. Tyto třídy jsou navrženy z důvodu, aby byly jednotlivé možnosti rozšiřování aplikace logicky strukturované a oddělené od ostatních zdrojových kódů aplikace. Jedná se o třídu, která obsahuje aktivační funkce, třídu obsahující algoritmy pro výpočet a trénování neuronové sítě a třídu, která vytváří strukturu vazeb mezi neurony podle zadaného typu sítě.

Třída `FunctionContainer` představuje knihovnu aktivačních funkcí. Jednotlivé aktivační funkce jsou uloženy jako samostatné metody.

Třída `AlgorithmContainer` obsahuje implementaci algoritmů pro výpočet a trénování neuronové sítě. Jednotlivé metody z této třídy řídí samotný průběh simulace.

Třída `LinkCreator` slouží k implementaci jednotlivých metod, které vytváří požadovanou strukturu neuronové sítě. Funkce třídy je založena na návrhu, kdy objekt třídy `NeuralNetwork` vytvoří pouze požadovaný počet neuronů a následně je předán ke zpracování vybrané metodě v této třídě. Ta vytvoří jednotlivé vazby mezi neurony podle požadovaného typu. Dále také vytváří vstupní a výstupní body neuronové sítě a inicializuje hodnoty jednotlivých prahů a vah.

4.3 Rozšířitelnost aplikace a modifikace struktury sítí

Princip návrhu pro umožnění rozšíření aplikace je založen na umístění jednotlivých způsobů rozšíření do samostatných tříd (4.2.2). Tím je docíleno struktury aplikace, kde pro rozšíření aplikace není nutné významněji zasahovat do tříd, které tvoří základ simulátoru.

Aplikace také umožňuje modifikovat neuronové sítě v menším měřítku než je celková struktura. Návrh umožňuje následující úpravy. Jde o změnu počáteční hodnoty vybrané váhy, změnu hodnoty prahu vybraného neuronu, změnu aktivační funkce vybraného neuronu, přidání nové nestandardní vazby mezi neurony, odebrání vybrané vazby mezi neurony, odebrání určeného neuronu nebo nastavení trvalé výstupní hodnoty neuronu nebo vazby.

Návrh aplikování zmíněných modifikací je založen na návrhu, kdy je nejprve vytvořena standardní struktura neuronové sítě pomocí třídy `LinkCreator` a následně jsou provedeny modifikace zadané v konfiguračním souboru.

Kapitola 5

Implementace

Kapitola je zaměřena na podrobný popis implementace simulátoru. Obsahuje například popis fungování simulátoru, technické specifikace aplikace, popis formátu vstupních a výstupních dat, podrobný popis jednotlivých tříd nebo instrukce a vysvětlení pro rozšiřování možností aplikace.

5.1 Technické specifikace

Práce byla vytvořena v programovacím jazyce C++ za použití standardu C++11 (ISO/IEC 14882:2011). Dále byl použit standard OpenMP pro paralelní programování se sdílenou pamětí . Překlad programu byl proveden pomocí překladače g++(gcc). Při tvorbě byla použita verze g++ 4.7.1. Dále byla aplikace přeložena a otestována i pomocí verze 4.8.4.

5.2 Formát vstupních a výstupních dat

Tato podkapitola popisuje požadovaný formát vstupních dat v konfiguračních a datových souborech. Ve stejném formátu je konfigurace sítě uložena po ukončení simulace. Dále je popsán formát výstupních dat reprezentujících vypočtené hodnoty.

5.2.1 Konfigurační soubor

Konfigurační soubor obsahuje popis neuronové sítě, specifikaci parametrů učení a popis jednotlivých modifikací neuronové sítě oproti standardní struktuře. Číslování neuronů začíná na hodnotě 1.

Formát konfiguračního souboru podporuje řádkové komentáře, které jsou uvozené pomocí znaku `#`. Vše za tímto znakem je programem ignorováno. Následující části souboru jsou povinné a musí být uvedeny na začátku konfiguračního souboru:

type = `name` – Určuje jaký typ neuronové sítě bude vytvořen. `name` udává název typu.

structure = `NxNxN` – Určuje počet neuronů obsažený v jednotlivých vrstvách neuronové sítě. `N` je celé číslo, které určuje počet neuronů v dané vrstvě. Znak **`x`** slouží jako oddělovač počtu neuronů v jednotlivých vrstvách.

activation = `name` **`x`** `name ; default` **`x`** `<-min,max>, value, value` – Popisuje aktivační funkci pro jednotlivé vrstvy neuronů, `name` určuje název aktivační funkce, `value`

je hodnota jednotlivých parametrů ve formě reálného čísla. Aktivační funkce musí být zadána pro všechny vrstvy, nebo obsahovat pouze jediný údaj, který bude platit pro všechny vrstvy. Výjimku tvoří sítě typu `feedforward/shortcut/kohonen` u kterých se nespecifikuje aktivační funkce pro první (vstupní) vrstvu neuronů.

Jako první a povinná část se udávají jména aktivačních funkcí pro jednotlivé vrstvy oddělené pomocí znaku **x**. Následuje nepovinná část oddělená pomocí znaku '**;**', která specifikuje nepovinné parametry aktivační funkce. Parametry jednotlivých aktivačních funkcí jsou odděleny znakem **x**. Pokud má funkce více parametrů jsou oddělené čárkou. Jako první parametr se uvádí obor hodnot funkce ve tvaru <`min,max`>, potom následují zbylé parametry. Obor hodnot nemusí být uveden a lze zadat pouze hodnoty parametrů. Pokud je zadána hodnota určitého parametru, je nutné uvést i hodnoty všech předcházejících parametrů.

Pokud je uveden chybný počet parametrů, jsou při menším počtu dosazeny standardní hodnoty, pří větším počtu je zbytek ignorován. Dále pokud je zadána aktivační funkce pro každou vrstvu sítě a ne jednotně pro všechny vrstvy, musí být zadané parametry opět pro každou vrstvu. Pokud je místo hodnoty parametrů uvedeno `default` jsou použity standardní hodnoty parametrů.

Následují nepovinné části konfiguračního souboru. V následujícím popisu `X` a `Y` reprezentují celá kladná čísla a *value* reprezentuje reálné číslo.

w`X,Y` = `value` – Nastavuje váhu vazby mezi dvěma neurony. `X` a `Y` udávají čísla neuronů, mezi kterými se vazba nachází.

bias`X` = `value` – Nastaví práh vybraného neuronu na zvolenou hodnotu. *X* určuje číslo neuronu a *value* hodnotu, na kterou bude práh nastaven.

activation`X` = `name ; param` – Změní aktivační funkci u zvoleného neuronu. Formát je stejný jako u povinné části popisující aktivační funkci. `X` určuje číslo neuronu.

stuck n`X` = `value` – Výstup neuronu označeného číslem `X` bude mít stálou hodnotu `value`.

stuck w`X,Y` = `value` – Vazba mezi neurony číslo `X` a `Y` bude jako výstupní hodnotu trvale poskytovat hodnotu zadanou parametrem `value`.

remove n`X` – Odstraní neuron s číslem `X`. Implementace je provedena jako příkaz `stuck nX = 0.0`.

remove w`X,Y` – Odstraní vazbu mezi neurony určenými čísly `X` a `Y`.

add w`X,Y` = `value` – Vytvoří novou vazbu mezi neurony určenými čísly `X` a `Y`. Váha vazby bude inicializována na hodnotu `value`.

epoch_limit = `value` – Nastaví maximální počet epoch učení na hodnotu `value`.

epsilon = `value` – Nastaví cílenou hodnotu přesnosti učení na hodnotu `value`.

training = `name ; param` – Popisuje parametry učícího algoritmu, `name` určuje název algoritmu, `param` označuje parametry učení. Název a parametry musí být odděleny znakem `;`. Jednotlivé parametry jsou odděleny znakem čárky.

Parametry **epoch_limit** a **epsilon** lze zadat i jako argumenty při spuštění programu. Tyto hodnoty budou mít větší prioritu než hodnoty zadané v konfiguračním souboru.

5.2.2 Vstupní datový soubor

První řádek ve vstupních souborech je ignorován z důvodu kompatibility s běžně dostupnými testovacími a trénovacími daty, které na prvním řádku obsahují informace o počtu trénovacích vzorů a počtu vstupu a výstupů.

Formát vstupních dat požaduje zapsání vstupních hodnot ve tvaru vektorů, kde jsou jednotlivé hodnoty v rámci vektorů oddělené mezerami. Dále je požadováno, aby každý vektor byl na samostatném řádku. V normálním módu následují jednotlivé vstupní vektory za sebou. Pro trénovací a testovací se střídají vstupní vektory s požadovaným výstupními vektory.

Následuje příklad normálních a trénovacích dat pro síť se čtyřmi vstupy a dvěma výstupy.

```
0.1   0.2   0.3   0.4
0.5   -0.6  0.7   0.8
-0.1  1.0   0.1   0.2

0.1   0.2   0.3   0.4
1.0   -0.5
0.5   0.6   0.7   0.8
-1.0  0.0
```

5.2.3 Formát výstupních dat

Formát výstupních dat je závislý na zvoleném módu výpočtu. V normální volbě jsou vypisovány pouze výstupní vektory ve stejném formátu jako vstupní vektory. Při volbě testovacího módu jsou jednotlivé výstupy formátovány v následujícím tvaru: `[vstupní vektor] =>[výstupní vektor] ; target = [požadovaný vektor] ; diff = [vektor diferencí] ; error = MSE chyby` . Hodnota `error` je vypočtena na základě vztahu (5.1), kde index i označuje jednotlivé výstupy sítě, y_i i-tý výstup sítě, t_i i-tou hodnotu požadovaného vektoru a n celkový počet výstupů. Následuje ukázka výstupních testovacích dat pro síť se třemi vstupy a jedním výstupem.

```
[0.1, 0.2, 0.3]  =>  [0.5];  target = [1.0];  diff = [0.5];  error = 0.25
```

$$error = \frac{1}{n}\sum_{i}(t_i - y_i)^2 \tag{5.1}$$

5.3 Argumenty aplikace

Aplikace požaduje při spuštění dva povinné argumenty, které určují konfigurační soubor s popisem neuronové sítě a soubor se vstupními daty. Dále je možné zadat následující nepovinné argumenty. Pokud pomocí argumentů nebude zvolen žádný běhový režim, bude aplikace spuštěna ve standardním výpočetním režimu. Argumenty `--train` a `--test` nelze kombinovat.

--test Aplikace bude spuštěna v testovacím režimu.

--train Aplikace bude spuštěna v trénovacím režimu.

--save=`name` Po ukončení simulace bude konfigurace sítě uložena do souboru zadaného řetězcem `name` ve formátu popsaném v kapitole 5.2.1. Pokud tento argument není zadaný, konfigurace sítě bude vypsána na standardní výstup.

--epsilon=`value` Nastaví cílenou hodnotu přesnosti na hodnotu `value`. Pokud byla hodnota přesnosti zadaná v konfiguračním souboru, je tato hodnota přepsána hodnotou `value`.

--epoch_limit=`value` Nastaví maximální počet epoch na hodnotu `value`. Pokud byl maximální počet epoch zadaný v konfiguračním souboru, je tato hodnota přepsána hodnotou `value` určenou v argumentu.

Aplikace dále umožňuje parametrem `--help` vypsat stručnou nápovědu. Tento parametr může být zadaný pouze samostatně.

5.4 Tok programu a obecný popis implementace

Kapitola popisuje tok programu a vnitřní implementaci simulátoru. Neobsahuje podrobný popis jednotlivých tříd a metod, který je uveden v následujících kapitolách.

Po spuštění aplikace je nejdříve vytvořen objekt třídy `NeuralNetwork`, který představuje model neuronové sítě a jádro aplikace. Následně je řízení programu předáno metodě `init` tohoto objektu, která nejprve otevře a zkontroluje konfigurační a vstupní soubory a následně postupně volá metody pro vytvoření a konfiguraci modelu neuronové sítě, vyhodnocení argumentů programu a načtení vstupních dat.

Metoda `createNetwork` zajišťuje vytvoření a konfiguraci neuronové sítě. Nejdříve z konfiguračního souboru načte povinné parametry (typ sítě, počet a velikost jednotlivých vrstev a jejich aktivační funkce). Pokud jsou tyto údaje správně načteny, je zavolána metoda `createNeurons`, která vytvoří požadovaný počet objektů reprezentující neurony a pomocí metody `createFunctionContainer` vytvoří a inicializuje objekty představující aktivační funkce jednotlivých neuronů. Jakmile jsou tyto objekty úspěšně vytvořeny, je řízení předáno objektu třídy `LinkCreator` (5.6), který na základě zvoleného typu sítě zavolá vybranou metodu této třídy a v ní vytvoří a inicializuje jednotlivé objekty reprezentující vazby, vstupy a výstupy v síti.

Po vytvoření a konfiguraci vybraného typu sítě, je řízení programu opět navráceno metodě `createNetwork`, která načte zbývající data z konfiguračním souboru a na jejich základě provede úpravy sítě oproti standardní struktuře.

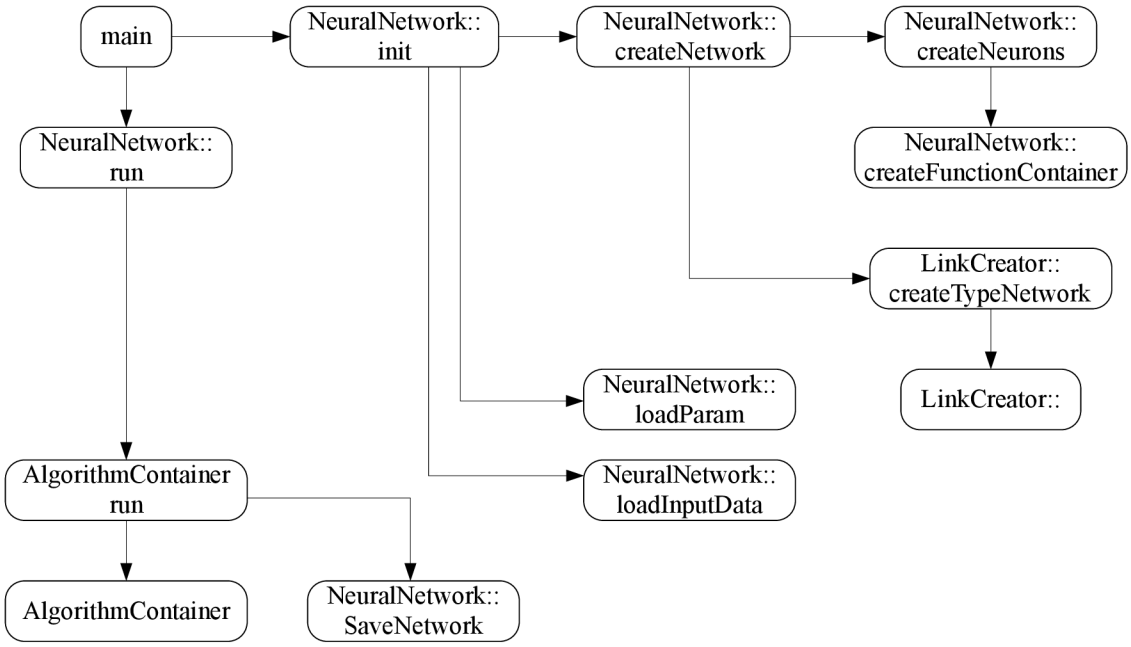
Po úspěšném vytvoření a konfiguraci sítě je zavolána metoda `loadParam`, která vyhodnotí a zkontroluje argumenty zadané při spuštění aplikace.

Poté je zavolána metoda `loadInputData`, která načte data ze vstupního souboru, zkontroluje jejich správný formát a uloží je ve formě jednotlivých vektorů. Data jsou čtena po jednotlivých řádcích a každý řádek je ukládán jako samostatný vektor pomocí knihovní třídy `std::vector`. Pokud je aplikace spuštěna ve standardním režimu, jsou všechny načtené vektory ukládány jako vstupní data, pokud v testovacím nebo trénovacím režimu, jsou jednotlivé sudé vektor uloženy jako požadovaná výstupní data. Po ukončení této metody vrátí metoda `init` řízení programu opět funkci `main` a aplikace je připravena pro spuštění simulace.

Funkce `main` v dalším kroku spustí simulaci pomocí metody `run` náležející k vytvořenému objektu třídy `NeuralNetwork`. Tato metoda vytvoří objekt třídy `AlgorithmContainer`

(5.8) a spustí metodu `run` tohoto objektu, která v závislosti na zvoleném typu sítě a režimu aplikace určí, jaká metoda v této třídě bude spuštěna a provede simulaci.

Po korektním ukončení simulace je řízení programu předáno metodě `saveNetwork` u již vytvořeného objektu třídy `NeuralNetwork`, která uloží aktuální konfiguraci neuronové sítě. Po uložení konfigurace je aplikace korektně ukončena.



Obrázek 5.1: Diagram datových toků v aplikaci.

5.5 Popis jednotlivých tříd

V kapitole jsou popsány jednotlivé třídy a jejich funkce v aplikaci. Výjimku představují třídy `FunctionContainer`, `LinkCreator` a `AlgorithmContainer`, které jsou podrobněji popsány v samostatných kapitolách.

NeuralNetwork

Implementace této třídy je založena na návrhu, který chápe neuronovou síť jako samostatný celek a objekt. Metody této třídy jsou slouží pro vytváření a konfiguraci neuronové sítě, načítání vstupních dat, konfiguraci jednotlivých vstupů a výstupů, výpis aktuálních výstupních hodnot a další činnosti potřebné pro bezproblémový chod simulace. Technicky řečeno objekt této třídy řídí většinu činností v aplikaci před zahájením běhu samotné simulace a po jejím ukončení. Samotná simulace je pak řízena vybranou metodou z třídy `AlgorithmContainer`.

Objekt této třídy také obsahuje ukazatele na vytvořené objekty ostatních tříd, ze kterých je složen samotný model neuronové sítě, načtená vstupní data a konfigurační data popisující neuronovou síť a její modifikace. Také jsou zde uloženy jednotlivé konstanty a proměnné specifické pro jednotlivé učící a výpočetní algoritmy ve třídě `AlgorithmContainer`, jejichž

hodnotu lze zadávat jednotlivými parametry položky `--training` v konfiguračním souboru. Třída je implementována v souborech `NeuralNetwork.h` a `NeuralNetwork.cpp`.

DefaultBlock

Třída `DefaultBlock` plní v aplikaci funkci rodičovské třídy pro všechny třídy, které představují jednotlivé stavební bloky, ze kterých je složena neuronová síť (`InputBlock`, `OutputBlock`, `Neuron`). Tento způsob implementace je zvolen z důvodu, aby k objektu třídy `Links`, který představuje vazbu v síti, bylo možné připojit libovolný prvek neuronové sítě, který zároveň bude plnit požadovanou základní funkcionalitu (poskytování výstupní hodnoty). Třída je popsána v souboru `Blocks.h`.

InputBlock

Objekty této třídy reprezentují v aplikaci vstupní body neuronové sítě. V návrhu jsou implementovány z důvodu, aby vstup do sítě u vstupních neuronů, mohl být implementován formou objektu třídy `Links` a zároveň vstupní hodnoty nebyly nastavovány přímo do tohoto objektu třídy `Links`. Samotná funkce objektů této třídy spočívá pouze v poskytování vstupní hodnoty, která je nastavena zvenčí objektu a samotný objekt ji nemění. Třída je odvozena od třídy `DefaultBlock` a je implementována v souboru `IOBlocks.h`.

OutputBlock

Objekty této třídy představují výstupní body z neuronové sítě. V návrhu jsou přítomny z důvodu strukturovanosti implementace, aby jednotlivé výstupní hodnoty nebyly vypisovány přímo z objektu třídy `Neuron`. Místo toho je vytvořen objekt této třídy, který je připojen k objektu reprezentující výstupní neuron pomocí objektu třídy `Links` a následně pouze poskytuje výstupní hodnotu daného neuronu. Třída je odvozena od třídy `DefaultBlock` a je implementována v souboru `IOBlocks.h`.

Neuron

Třída `Neuron` implementuje model umělého neuronu (2.1.1). Funkce jednotlivých objektů této třídy spočívá ve vyhodnocení vstupních vazeb, následné aktualizaci hodnoty vnitřního potenciálu (2.1) a výpočtu nové výstupní hodnoty objektu pomocí vybrané aktivační funkce (2.2), která je implementována formou ukazatele na vybranou metodu třídy `FunctionContainer`. K objektu této třídy je připojen objekt třídy `FunctionContainer`, který konfiguruje aktivační funkci. Vstupní a výstupní body těchto objektů jsou implementovány pomocí objektů třídy `Links`, které představují jednotlivé vstupní a výstupní vazby neuronu. Objektu je také možné nastavit trvalou výstupní hodnotu a simulovat tak poruchu vybraného neuronu. Třída je odvozena od třídy `DefaultBlock` a je implementována v souborech `Neuron.h` a `Neuron.cpp`.

Links

Objekty této třídy představují vazby mezi jednotlivými objekty v neuronové síti (nejčastěji neurony). Ke každému objektu je proto přiřazena dvojice objektů třídy `DefaultContainer`, nebo tříd od ní odvozených, kde první objekt představuje vstupní objekt vazby a druhý objekt výstupní objekt. Funkce jednotlivých objektů této třídy je založena na poskytnutí aktuální výstupní hodnoty vstupního objektu vazby v okamžiku, kdy výstupní objekt vazby

požádá o aktuální výstupní hodnotu. Výstupní objekt dále může požádat o výstupní hodnotu upravenou pomocí váhy vazby a objekt vazby poskytne výstupní hodnotu vynásobenou hodnotou své váhy. Třída dále umožňuje simulovat poruchu vazby v neuronové síti, kdy je objekt uzamknut a dále poskytuje pouze trvale nastavenou výstupní hodnotu, která je nezávislá na výstupní hodnotě vstupního objektu a váze vazby. Třída je implementována v souborech `Links.h` a `Links.cpp`.

5.6 Třída LinkCreator a implementované typy sítí

Třída `LinkCreator` je navržena na základě návrhu na umožnění zapouzdření jednotlivých metod pro vytváření struktury jednotlivých typů neuronových sítí do samostatné oddělené třídy. Tato třída je implementována v souborech `LinkCreator.cpp` a `LinkCreator.h`.

Na začátku vytváření a konfigurace modelu neuronové sítě, jsou v aplikaci vytvořeny pouze objekty reprezentující umělé neurony. Následně je na základě zadaného typu sítě vybrána odpovídající metoda této třídy, která vytvoří jednotlivé vazby mezi neurony (objekty třídy `Links`) a inicializuje jejich váhy. Také jsou vytvořeny požadované vstupní a výstupní objekty a u vybraných typů sítí inicializovány hodnoty prahů u neuronů. Po skončení metody aplikace obsahuje kompletní sestavený model požadovaného typu neuronové sítě ve standardním formátu.

Následující části kapitoly popisují požadavky na metody této třídy a popis standardně implementovaných typů sítí.

5.6.1 Požadavky na jednotlivé metody

Pro správnou funkci aplikace musí implementované metody splňovat následující požadavky:

1. Deklarace metody musí být ve tvaru `int jméno_metody(NeuralNetwork &)`[1]. Argument slouží k předání objektu, který obsahuje vytvářenou neuronovou síť.
2. Vytvoření požadovaného počtu objektů tříd `InputBlock` a `OutputBlock`, které představují vstupní body a výstupní body neuronové sítě.
3. Vytvořit a inicializovat jednotlivé objekty třídy `Links`, které reprezentují vazby mezi jednotlivými neurony v neuronové síti.

V průběhu druhého kroku jsou vyvářeny objekty sloužící jako vstupy a výstupy neuronové sítě. Jejich počet musí odpovídat počtu vstupních a výstupních neuronů, protože na základě počtu vytvořených objektů je při následném načítání vstupních dat kontrolována správná délka vstupních a výstupních datových vektorů, která musí odpovídat počtu vytvořených vstupních a výstupních objektů.

Pro vytvoření vstupního objektu jsou nutné následující kroky:

1. Vytvoření objektu třídy `InputBlock` pomocí operátoru `new`.
2. Vytvoření objektu třídy `Links` příkazem `new Links(*in,*out,1.0)` který reprezentuje vazbu mezi vstupním bodem a vstupním neuronem. Argument `in` představuje ukazatel na vytvořený vstupní objekt a `out` ukazatel na objekt třídy neuron[2], který představuje vstupní neuron.

[1] Bude-li v kapitole odkazováno na třídu `NeuralNetwork`, je tím myšlen objekt zadaný povinným argumentem metody.

[2] Jedná se o už vytvořené objekty, dostupné metodou `getNeuronList` třídy `NeuralNetwork`. Totéž platí pro všechny objekty třídy `Neuron` zmíněné v této kapitole.

3. Uložení ukazatele na vytvořený vstupní objekt pomocí metody `setNewInput` třídy `NeuralNetwork`.
4. Přidání nové vstupní vazby k objektu třídy `Neuron` použitém v druhém bodu pomocí metody `setInputLinks`.

Pro výstupní bod pak platí následující:

1. Vytvoření objektu třídy `OutputBlock` pomocí operátoru `new`.
2. Vytvoření objektu třídy `Links` příkazem `new Links(*in,*out,1.0)`, který bude sloužit jako výstupní bod sítě. Argument `in` představuje ukazatel na objekt třídy Neuron, který reprezentuje výstupní neuron a `out` ukazatel na vytvořený výstupní objekt třídy `OutputBlock`.
3. Uložení ukazatele na vytvořený výstupní objekt pomocí metody `setNewOutput` třídy `NeuralNetwork`.
4. Přidání nové vstupní vazby k výstupnímu objektu třídy `Neuron`, který byl použitý v druhém bodu pomocí metody `setOutputLinks`.

Třetí požadavek zahrnuje vytvoření jednotlivých objektů třídy `Links` představujících vazby mezi neurony. Výsledná struktura neuronové sítě je určena způsobem propojení neuronů a směrem jednotlivých vazeb. Pro každý vytvořený vazební objekt je nutné implementovat následující kroky:

1. Příkazem `new Links(*in, *out, weight)` vytvořit nový objekt třídy `Links`, který reprezentuje vazbu mezi dvěma neurony, kde argumenty `in` a `out` představují ukazatele na rozdílné objekty třídy `Neuron` a `weight` je hodnota, na kterou bude inicializována váha vytvořené vazby. Vazba je orientována směrem od objektu `in` k objektu `out`.
2. Vytvořenému objektu metodou `setID(id1,id2)` přiřadit identifikační čísla vstupního (`id1`) a výstupního (`id2`) neuronu použitých v bodě 1.
3. Uložit ukazatel na vytvořený objekt do objektu neuronové sítě metodou `setNewLinks`.
4. Přiřadit vytvořený objekt jako novou výstupní vazbu k objektu vstupního neuronu z prvního bodu metodou `setOutputLinks` a jako novou vstupní vazbu k objektu výstupního neuronu z prvního bodu metodou `setInputLinks`.

5.6.2 Typy sítí v aplikaci

Následuje popis jednotlivých typů sítí, které jsou v aplikaci standardně k dispozici. U každého typu sítě je uveden identifikační řetězec, který označuje zvolený typ v konfiguračním souboru a v rámci vnitřní identifikace v aplikaci, název tohoto typu sítě a popis struktury vytvořené jednotlivými objekty třídy `Links`.

Vícevrstvá perceptronová síť

Identifikační řetězec: `feedforward`. Vytvořená struktura sítě je založena na plném propojení neuronů v sousedních vrstvách. Postupně jsou procházeny všechny neurony v jednotlivých vrstvách s výjimkou výstupní vrstvy a jsou vytvořeny vazby směřující z jednotlivých neuronů zpracovávané vrstvy do všech neuronů následující vrstvy. Počet vytvořených objektů reprezentujících výstupní vazby neuronu je tedy stejný jako počet neuronů v následující vrstvě a počet vstupních objektů je určený počtem neuronů v předcházející vrstvě. Výjimku představují vstupní neurony, které obsahují pouze jediný vstupní objekt třídy `Links`

směřující z objektu třídy `Inputblock` a výstupní neurony, které vlastní pouze jediný výstupní objekt třídy `Links`, který představuje vstupní vazbu do objektu třídy `OutputBlock`. Z toho plyne, že počet vytvořených vstupních objektů je určen počtem neuronů v první zadané vrstvě a počet vytvořených výstupních objektů počtem neuronů v poslední vrstvě.

Celkový vytvořený počet vazeb mezi neurony (objektů třídy `Links`), je možné popsat vztahem (5.2), kde i představuje pořadí vybrané vrstvy, $size$ počet vrstev sítě a x velikost vrstvy i.

$$links = \sum_{i=1}^{size-1} x_i x_{i+1} \tag{5.2}$$

Váhy jednotlivých vytvořených vazeb jsou inicializovány na náhodnou hodnotu z intervalu $\langle -\frac{2}{s}, \frac{2}{s} \rangle$, kde s určuje počet vstupů neuronu, do kterého vazba směřuje [15].

V rámci vytváření sítě, jsou také inicializovány hodnoty prahů u jednotlivých neuronů ve všech vrstvách s výjimkou vstupní vrstvy na hodnotu z uvedeného intervalu.

Standardní hodnoty parametrů trénovacího algoritmu Backpropagation jsou nastaveny na 0.5 v případě parametru učení a 0.9 v případě parametru hybnosti. Při nastavování hodnot parametrů pomocí konfigurační položky `training`, představuje první uvedená hodnota parametr učení a druhá uvedená hodnota parametr hybnosti.

Vícevrstvá perceptronová síť - shortcut varianta

Identifikační řetězec: `shortcut`. Struktura tohoto typu sítě je založena na struktuře vícevrstvé perceptronové sítě. Rozdíl spočívá v tom, že objekty jednotlivých neuronů neobsahují vstupní vazby pouze od objektů neuronů v poslední předchozí vrstvě, ale od objektů neuronů ve všech předchozích vrstvách. Počet vytvořených objektů třídy `Links` mezi jednotlivými neurony je možné popsat vztahem (5.3), kde i označuje pořadí aktuální vrstvy, $size$ počet vrstev, j jednotlivé předchozí vrstvy a x velikost aktuální vrstvy. Pro ostatní vlastnosti tohoto typu sítě platí stejné vztahy jako v případě předchozího typu `feedforward`. Pouze hodnoty jednotlivých vah a prahů, jsou inicializovány na hodnotu z intervalu $\langle -0.5, 0.5 \rangle$.

$$links = \sum_{i=2}^{size} \sum_{j=1}^{i-1} x_j x_i \tag{5.3}$$

Kohonenova síť

Identifikační řetězec: `kohonen`. Kohonenova síť je složena ze dvou vrstev neuronů, kde každý neuron v první (vstupní) vrstvě je propojen se všemi neurony v druhé (kohonenově) vrstvě. Počet vytvořených objektů třídy `Links` mezi dvěma objekty třídy `Neuron` tedy odpovídá součinu velikostí první a druhé vrstvy. Hodnoty jednotlivých vah jsou inicializovány na hodnotu z intervalu $\langle -0.5, 0.5 \rangle$. Protože každý vstupní neuron představuje jeden vstup sítě, počet vytvořených objektů třídy `Output` odpovídá velikosti první vrstvy neuronů a protože každý neuron v Kohonenově vrstvě je zároveň výstupní, je počet vytvořených objektů třídy `InputLinks` určen velikostí druhé vrstvy neuronů. Typ sítě je vytvářen v metodě `kohonen`, která také inicializuje hodnotu parametru učení na 0.8 a rozsah okolí neuronů na hodnotu 0. Při nastavování hodnot parametrů pomocí konfigurační položky `training`, představuje první uvedená hodnota typ topologické mřížky, druhá hodnota velikost okolí neuronů a třetí hodnota parametr učení.

5.7 Třída FunctionContainer a implementované funkce

Třída `FunctionContainer` slouží k zapouzdření jednotlivých aktivačních funkcí do samostatné třídy. Výběr a konfigurace jednotlivých aktivačních funkcí je poté prováděn přes jednotlivé objekty této třídy a tím je umožněno logicky oddělit část zdrojového kódu, která implementuje aktivační funkce, od hlavní části aplikace. Touto strukturou implementace je rovněž umožněno přidávat do aplikace nové aktivační funkce, aniž by to vyžadovalo úpravu ostatních části aplikace.

Deklarace jednotlivých metod této třídy musí odpovídat následujícímu formátu: `double název_metody(double)`. Ke každé nové aktivační funkci je také vyžadována druhá metoda, která bude implementovat derivaci této aktivační funkce a její deklarace bude ve stejném formátu.

5.7.1 Dostupné aktivační funkce

V této části jsou popsány implementované aktivační funkce, které jsou k dispozici. Také jsou uvedeny k nim náležející derivace. Vždy je uvedeno klíčové slovo, které identifikuje konkrétní aktivační funkci v rámci konfiguračního souboru a v rámci vnitřní struktury aplikace a popis této funkce.

Hodnoty parametrů a obor hodnot funkce (`min,max`) je možné zadat v konfiguračním konfiguračním souboru, v opačném případě jsou použity standardní hodnoty. Pokud má funkce více parametrů, je požadováno je v konfiguračním souboru uvádět v pořadí v jakém jsou uvedeny u popisu jednotlivých aktivačních funkcí. Pro každý zadaný parametr je nutné uvést všechny předchozí parametry, které jsou v pořadí před tímto parametrem.

- `sigmoid` - Standardní sigmoida (2.3) je implementována ve tvaru (5.4). Implementace pomocí funkce hyperbolický tangens z knihovny `cmath` je zvolena z důvodu menší časové náročnosti v porovnání se vztahem obsahujícím mocninu Eulerova čísla (funkce `exp` knihovny `cmath`). Pro zjištění časové náročnosti byl použit algoritmus [23]. Derivace funkce je implementována podle vztahu (5.5) převzatého z [26], který byl upraven pro obecné hodnoty. Standardní hodnoty parametrů: $= 0.0, max = 1.0, \lambda = 1.0$. Parametry aktivační funkce: λ - parametr strmosti.

$$y(\xi) = (max - 0.5(max - min)) + 0.5(max - min)tanh(0.5\xi\lambda) \tag{5.4}$$

$$y' = \lambda(y - min)\left(1 - \frac{y - min}{max - min}\right) \tag{5.5}$$

- `ramp` - Saturovaná lineární funkce (2.6) je implementována v obecném tvaru (5.6) [26]. Derivace této funkce je skoková funkce (5.7). Standardní hodnoty parametrů: $min = 0.0, max = 1.0, c = 0.0, d = 1.0$. Parametry aktivační funkce: c - minimální hodnota na ose x, d - maximální hodnota na ose x.

$$y(\xi) = \begin{cases} min & \xi < c \\ max & \xi > d \\ min + \frac{(max-min)(\xi-c)}{d-c} & c \leq \xi \leq d \end{cases} \tag{5.6}$$

$$y' = \begin{cases} 1 & y \geq 0 \\ 0 & y < 0 \end{cases} \tag{5.7}$$

- `step` - Nespojitá skoková funkce (2.5). V aplikaci je implementována ve tvaru (5.8), kdy standardní hodnoty parametrů jsou: $min = 0.0, max = 1.0$. Derivace této funkce je Diracova δ-funkce (5.9).

$$y(\xi) = \begin{cases} max & \xi \geq 0 \\ \xi & \xi = 0 \\ min & \xi < 0 \end{cases} \tag{5.8}$$

$$\delta(x) = \begin{cases} +\infty & x \neq 0 \\ 0 & x = 0 \end{cases} \tag{5.9}$$

- `tanh` - Funkce hyperbolický tangens (2.4) je implementována v obecném tvaru pomocí funkce `tanh` z knihovny `cmath` (5.10) [26]. Derivace funkce je implementována na základě vztahu (5.11) [26], který je upraven do obecného tvaru (5.12). Standardní hodnoty parametrů: $min = -1.0, max = 1.0, \lambda = 1.0$. Parametry aktivační funkce: λ - parametr strmosti.

$$y(\xi) = 0.5(min + max + (max - min)tanh(\lambda\xi)) \tag{5.10}$$

$$y' = (1 - y)^2 \tag{5.11}$$

$$y' = 0.5(max - min)\lambda \left(1 - \left(\frac{2y - (min + max)}{max - min}\right)^2\right) \tag{5.12}$$

- `simple` - Pomocná aktivační funkce, která pouze vrací vstupní hodnotu na základě vztahu: $y = x$.

- `rbf` - Radiální spojitá bázová funkce ve tvaru (5.13) [26]. Derivace této aktivační funkce má tvar (5.14). Standardní hodnoty parametrů: $\sigma = 1.0$. Parametry aktivační funkce: σ - parametr určuje šířku grafu funkce na ose x.

$$y(\xi) = e^{-(\xi/\sigma)^2} \tag{5.13}$$

$$y' = \frac{2}{\sigma^2} e^{-(\frac{y^2}{\sigma^2})} y \tag{5.14}$$

- `sigmoid_2` - Další varianta funkce sigmoid (2.3) je implementovaná na základě vztahů pro aproximaci této funkce (5.15), (5.16) a (5.17), které byly převzaty z [12]. Interval L je nastaven na hodnotu 4, která představuje odhadovaný rozsah střední části standardní funkce sigmoid a parametry β a θ ovlivňují sklon a funkce v její středové části (interval $\langle -L, L \rangle$) (2.3). Protože tato aktivační funkce není vhodná pro učení ve vícevrstvé perceptronové síti, je místo její derivace použita pomocná metoda `simple`.

$$H_s(x) = \begin{cases} x(\beta + \theta x) & x \in \langle -L, 0 \rangle \\ x(\beta - \theta x) & x \in (0, L \rangle \end{cases} \tag{5.15}$$

$$G_s(x) = \begin{cases} -1 & x \in (-\infty, -L\rangle \\ H_s(x) & x \in (-L, L) \\ 1 & x \in \langle L, \infty) \end{cases} \tag{5.16}$$

$$y(\xi) = \frac{1}{2} G_s(\xi) + \frac{1}{2} \tag{5.17}$$

$$\beta = \frac{2}{L} \tag{5.18}$$

$$\theta = \frac{1}{L^2} \tag{5.19}$$

5.8 Třída AlgorithmContainer a implementované algoritmy

`AlgorithmContainer` je třída, která slouží k zapouzdření jednotlivých výpočetních a učících algoritmů pro neuronové sítě do samostatné třídy a tím zvyšuje přehlednost struktury aplikace. Objekt této třídy a její jednotlivé metody řídí průběh simulace. Jednotlivé algoritmy jsou implementovány formou samostatných metod, jejichž definice musí splňovat následující formát: `int jméno_metody(NeuralNetwork &)`. Pokud je do aplikace přidáván nový algoritmus, musí nová metoda splňovat tento formát a v metodě `run` je potřeba určit, pro který typ sítí bude tento algoritmus používán.

Tato kapitola obsahuje podrobný popis implementovaných algoritmů a jejich pseudokódy.

5.8.1 Vícevrstvá perceptronová síť - vypočet

Algoritmus 1: feedforward_calcul

```
1 for početVstupníchVektorů
2     Nastavení hodnot vybraného vstupního vektoru na vstupy sítě.
3     for početNeuronů
4         ξ^new = Σ w_i x_i
5         y^new = f(ξ^new)
6     end
7     Výpis aktuálních výstupních hodnot.
8 end
```

Algoritmus je implementován v metodě `feedforward_calcul`. Je použitý pro výpočet výstupních hodnot vícevrstvé perceptronové sítě a také v případě její `shortcut` varianty. Jeho implementaci popisuje uvedený pseudokód.

Nejprve jsou jednotlivé hodnoty z příslušného vstupního datového vektoru nastaveny na příslušné vstupy sítě. Následně je pro každý neuron v síti postupně vypočtena nová hodnota vnitřního potenciálu ξ a nová výstupní hodnota y, která je následně nastavena na výstup tohoto neuronu. Po ukončení vyhodnocení všech neuronů je na standardní výstup vypsán výstupní datový vektor v požadovaném formátu. Jakmile jsou takto zpracovány všechny vstupní datové vektory, je metoda ukončena a aplikace se korektně ukončí.

Pro správnou funkci tohoto algoritmu je nutné, aby ukazatele na jednotlivé objekty reprezentující neurony, byly uloženy v rámci vektoru v pořadí, které je určeno pořadím

vrstvy ve které se neuron nachází. Tato podmínka je splněna v konfigurační části aplikace, kde se jednotlivé objekty třídy `Neuron` vytváří v pořadí podle jednotlivých vrstev, kdy jsou nejdříve vytvořeny objekty reprezentující neurony ve vstupní vrstvě, následně první skryté vrstvě atd.

5.8.2 Vícevrstvá perceptronová síť - Backpropagation

Algoritmus 2: backpropagation

```
1  for maximálníPočetEpoch
2      errGlobal = 0
3      for početTrénovacíchVzorů
4          err = 0, innerEpoch = 0
5          Nastavení trénovacího vzoru k na vstupy sítě.
6          do
7              for početNeuronů
8                  ξ^new = Σ w_i x_i
9                  y^new = f(ξ^new)
10             end
11             Výpočet aktuální hodnoty chyby err.
12             #pragma omp parallel forr
13             for Počet výstupních neuronů
14                 Výpočet parciálních derivací ve výstupní vrstvě.
15             end
16             #pragma omp parallel for
17             for početSkrytýchVrstev          ▷ Procházení směrem od výstupní vrstvy.
18                 Výpočet parciálních derivací neuronů v dané skryté vrstvě.
19             end
20             #pragma omp parallel for
21             for početVšechVazebMeziNeurony
22                 Výpočet nové hodnoty váhy jednotlivých vazeb mezi neurony.
                   Nastavení nové hodnoty váhy pro objekt reprezentující odpovídající
                   vazbu vazbu.
23             end
24             innerEpoch = innerEpoch + 1
25         while err > ε && innerEpoch < 100;
26         errGlobal = errGlobal + err
27     end
28     if (errGlobal * (1/PN)) < ε
29         Opětovné vyhodnocení chyby všech trénovacích vzorů a globální chyby.
30         if (errGlobalCheck * 1/PN) < ε
31             return
32     if aktuálníEpocha == interval
33         Zálohování aktuální konfigurace.
34 end
```

Uvedený pseudokód popisuje implementovaný algoritmus Backpropagation, který slouží pro učení vícevrstvé perceptronové sítě a její shortcut varianty. Při tvorbě a návrhu této implementace bylo vycházeno z verzí a popisu tohoto algoritmu v [19] a [26]. Algoritmus

je implementován v metodě `backpropagation`. Základní myšlenka zvolené implementace spočívá ve snaze naučit síť postupně vybraný trénovací vzor a teprve jakmile je naučen, přejít na učení dalšího trénovacího vzoru.

Maximální počet celkových iterací algoritmu je určený maximálním počtem provedených epoch, který lze zadat v konfiguračním souboru nebo pomocí argumentů aplikace. Bez nastavení je hodnota nastavena na 1000. Pokud celková chyba sítě klesne pod zadanou hodnotu ε, je dosaženo požadované přesnosti a algoritmus je ukončen.

Na začátku každé epochy je vynulována globální chyba sítě a pro každý trénovací vzor k je spuštěn vnitřní cyklus, který je ukončen, pokud je tréninkový vzor naučen na požadovanou přesnost ($E_k <= \varepsilon$), nebo pokud je překročen povolený počet vnitřních epoch (standardně 100 iterací).

Před spuštěním vnitřního cyklu je na vstupy sítě nastaven zvolený trénovací vzor. Následují jednotlivé iterace, ve kterých jsou nejdříve vypočteny aktuální výstupní hodnoty sítě. Způsob tohoto výpočtu je stejný jako v algoritmu (1).

Následně jsou vypočteny a uloženy parciální derivace pro jednotlivé neurony. Kdy jsou nejdříve vypočteny parciální derivace pro neurony ve výstupní vrstvě a následně pro neurony v jednotlivých skrytých vrstvách, které se vyhodnocují postupně směrem od výstupní vrstvy.

V dalším kroku jsou upraveny váhy jednotlivých vazeb. Protože ukazatele na jednotlivé objekty reprezentující vazby jsou uloženy ve společném vektoru a aktualizace vah nezávisí na hodnotě ostatních vah, je možné implementovat tuto část implementovat for cyklu a paralelizovat pomocí OpenMP.

Po skončení učícího cyklu pro vybraný trénovací vzor, je aktuální chyba tohoto vzoru přičtena ke globální chybě sítě na základě vztahu (2.10).

Po zpracování všech testovacích vzorů je globální chyba sítě E převedena na hodnotu MSE sítě na základě vztahu (5.20), kde P určuje počet trénovacích vzorů, N počet výstupních bodů sítě a ostatní části odpovídají vztahům (2.9) a 2.10) [13]. Tato hodnota je následně porovnána s hodnotou požadované přesnosti a pokud platí $MSE < \varepsilon$ je znovu provedeno vyhodnocení všech trénovacích vzorů a globální chyby. Pokud i v tom případě platí vztah $MSE_{check} < \varepsilon$ je dosaženo požadované přesnosti a algoritmus je ukončen. V opačném případě je provedena další epocha učení.

Tento postup vyhodnocení chyby je prováděn z toho důvodu, že při učení testovacích vzorů mohou být poškozeny předchozí naučené vzory a postupná globální chyba nemusí být přesná.

Algoritmus také umožňuje v metodě nastavit hodnotu proměnné, která určuje po jakém počtu provedených epoch bude zálohována konfigurace sítě (standardně 1024) do souboru `defaultBackup.txt`. Formát tohoto souboru je stejný jako formát popsaný v části 5.2.1 a jméno souboru je možné změnit v konstruktoru třídy `NeuralNetwork`.

$$MSE = \frac{1}{PN}E = \frac{1}{2PN}\sum_{k}\sum_{j}(t_{kj} - y_j)^2 \tag{5.20}$$

5.8.3 Kohonenova síť - výpočet

Algoritmus pro vyhodnocení jednotlivých vstupních vzorů v Kohonenově síti byl implementován na základě [19] a je obsažen v metodě `kohonen`. Implementaci popisuje uvedený pseudokód.

Pro každý vstupní vzor jsou provedeny následující činnosti. Nejdříve je na vstupy neuronů ve vstupní vrstvě nastaven příslušný vstupní vektor k a jsou aktualizovány výstupní hodnoty těchto neuronů, které označíme indexem i. Neurony ve vstupní vrstvě nemají standardní aktivační funkci, místo které je použita funkce `simple` a pouze přeposílají vstupní hodnotu na výstup.

V následující části je pro každý neuron v Kohonenově (kompetiční) vrstvě vyhodnocena vzdálenost od předloženého vzoru na základě vztahu (5.21), kde n je počet neuronů ve vstupní vrstvě [19]. Následně je vybrán neuron s nejmenší hodnotou vzdálenosti. U tohoto neuronu je aktualizována výstupní hodnota a jeho identifikační číslo je vypsáno na standardní výstup.

$$D(w_j, x) = \sum_{i=1}^{n} (w_{ij} - x_i)^2 \tag{5.21}$$

Algoritmus 3: kohonen

```
1  for početVstupníchVzorů
2      Nastavení trénovacího vzoru k na vstupy sítě.
3      #pragma omp parallel for
4      for početVstupníchNeuronů
5          ξ^new = Σ w_i x_i
6          y^new = f(ξ^new)
7      end
8      #pragma omp parallel for
9      for početNeuronůKompetičníVrstvy - j
10         distance_j = 0
11         for početVstupníchNeuronů - i
12             distance_j += (w_ij − x_i)^2
13         end
14     end
15     Určení neuronu v kompetiční vrstvě s nejmenší hodnotou vzdálenosti distance.
16     Aktualizace vnitřního potenciálu a výstupní hodnoty vítězného neuronu.
17     Výpis identifikačního čísla vítězného neuronu na standardní výstup
18 end
```

5.8.4 Kohonenova síť - učení

Algoritmus pro trénování Kohonenovy sítě vychází z algoritmu pro výpočet v této síti a byl implementován na základě informací v [19]. Je implementován v metodě `kohonenTrain`.

Princip algoritmu spočívá v nalezení neuronu v Kohonenově vrstvě, který má nejmenší vzdálenost vah od trénovacího vzoru a následné upravě vah náležejících k tomuto neuronu. Pokud jsou topologickou mřížkou určeny sousední neurony, jsou upraveny váhy i u těchto neuronů.

Nejdříve je pomocí stejného algoritmu jako při výpočtu (5.21) nalezen neuron s nejmenší vzdáleností. U tohoto neuronu jsou následně upraveny jednotlivé váhy vstupních vazeb podle vztahu (5.22), kde α označuje parametr učení (learning rate) a jeho hodnotu je doporučeno volit v intervalu $0 \leq \alpha \leq 1$ [19].

$$w_j^{t+1} = w_j^{(t)} + \alpha(x^{(t)} - w_j^{(t)}) \tag{5.22}$$

V následující části je na základě zvolené topologie a zvolené velikosti okolí vyhodnoceno, které neurony patří do okolí vybraného neuronu a jejich identifikační čísla jsou uloženy do datového vektoru. Tento vektor v následujícím for cyklu identifikuje neurony, u kterých jsou následně také změněny váhy vstupních vazeb na základě vztahu (5.22).

Zjištění identifikačních čísel a až následně úprava jednotlivých vah, je implementována z důvodu paralelizace zpracovávání určených neuronů.

Standardní hodnoty parametrů algoritmu jsou zvoleny na hodnoty $\alpha = 0$ a rozsah 0, který označuje okolí obsahující pouze vybraný neuron a žádné sousední neurony.

Algoritmus 4: kohonenTrain

```
1  for maximálníPočetEpoch
2      Algoritmus (3).
3      for početVstupníchVazebVítěznéhoNeuronu
4          w_j^(t+1) = w_j^(t) + α(x^(t) − w_j^(t))
5      end
6      Určení identifikačních čísel neuronů v určeném okolí vítězného neuronu.
7      #pragma omp parallel for
8      for početVybranýchNeuronů
9          for početVstupníchVazebVybranéhoNeuronu
10             w_j^(t+1) = w_j^(t) + α(x^(t) − w_j^(t))
11         end
12     end
13 end
```

5.9 Rozšiřování aplikace

Tato část popisuje potřebné úpravy pro rozšíření aplikace. Jedná se o možnost přidání nového typu neuronové sítě, nové aktivační funkce a dalších algoritmů pro výpočet a učení v neuronových sítích.

Nová aktivační funkce

Pro přidání nové aktivační funkce do aplikace jsou nutné provést následující úpravy ve třídě `FunctionContainer`, která je uložena v souborech `FunctionContainer.cpp/.h`. Řetězec identifikující aktivační funkci v aplikaci, je stejný jako název funkce zadávaný v konfiguračním souboru. Pokud aktivační funkce nebude používána v algoritmu Backpropagation, je možné vynechat druhý bod a ve čtvrtém bodě použít například předdefinovanou funkci `simple`.

1. Vytvořit novou metodu, která představuje aktivační funkci a musí mít návratovou hodnotu typu `double` a pouze jediný povinný argument typu `double`.
2. Vytvořit další metodu, která bude plnit roli derivace aktivační funkce a musí mít návratovou hodnotu typu `double` a jediný povinný argument také typu `double`.
3. V metodě `getFunction` přidat podmínku, která bude vracet ukazatel na nově vytvořenou metodu implementující aktivační funkcí, pokud bude zadáno klíčové slovo identifikující tento typ aktivační funkce.

4. V metodě `getDerivFunction` přidat podmínku vracející ukazatel na vytvořenou metodu obsahující derivaci vytvořené aktivační funkce, pokud bude zadáno klíčové slovo identifikující tento typ aktivační funkce.
5. V metodě `setType` přidat podmínku, která při zadání vytvořené aktivační funkce inicializuje obor hodnot funkce uložený v proměnných `rangeMin` a `rangeMax`. Dále uloží do vektoru `params` metodou `push_back` standardní hodnoty všech parametrů aktivační funkce v pořadí, ve kterém budou zadávány v konfiguračním souboru. Dále je možné uložit do vektoru `values` předem vypočítané konstanty, aby je nebylo nutné vyhodnocovat v každém průběhu opětovně.

Nový typ neuronové sítě

Pokud je do aplikace přidáván nový typ neuronové sítě, je nutné provést následující kroky ve třídě `LinkCreator`, která je implementována v souborech `LinkCreator.c/h`.

1. Vytvořit novou metodu třídy `LinkCreator`, která bude mít povinný návratový typ `int` a jediný povinný argument, kterým bude odkaz na objekt třídy `NeuralNetwork`. Metoda musí vytvořit vazební objekty třídy `Links` mezi jednotlivými neurony a vstupní a výstupní objekty. Muže také inicializovat hodnoty jednotlivých vah a prahů neuronů.
2. V metodě `createTypeNetwork` třídy `LinkCreator` přidat novou větev podmínky, která bude volat nově vytvořenou metodu a předá jí argument této metody, kterým je objekt obsahující neuronovou síť.

Dále jsou nutné následující úpravy ve třídě `NeuralNetwork`.

1. V metodě `getTrainingProperties` určit, jaký tvar bude mít řetězec, ve kterém budou v konfiguračním souboru uloženy parametry trénovacího algoritmu pro tento typ sítě.
2. V metodě `setTrainingParameters` přidat zpracování parametrů trénovacího algoritmu pro tento typ sítě, které jsou načteny z konfiguračního souboru z položky `training`. Formát uložených parametrů je identický jako v předcházejícím bodu.
3. Pokud je to potřebné, vytvořit ve třídě `NeuralNetwork` proměnné reprezentující parametry učících a trénovacích algoritmů, které bude tento typ sítě používat.
4. Pokud první vrstva sítě obsahuje neurony, které pouze přeposílají vstupní hodnoty a neplní výpočetní funkci, je možné upravit funkci `firstLayer` ve třídě `NeuralNetwork`, aby pro tento typ vracela hodnotu `true`. Tím bude při konfiguraci sítě zajištěno, že první vrstvě neuronů nebude možné nastavovat aktivační funkci v konfiguračním souboru v rámci globální aktivační funkce, ale neuronům v této vrstvě bude nastavena speciální aktivační funkce `simple`, která pouze vrátí neupravenou vstupní hodnotu.

Poslední úprava je nutná ve třídě `AlgorithmContainer`, kde je nutné upravit metodu `run` a určit jaký typ tréninkového a učícího algoritmu bude pro tento typ sítě používán.

Přidání nového učícího algoritmu

Pro přidání nového učícího nebo výpočetního algoritmu, jsou nutné následující dva kroky:

1. Ve třídě `AlgorithmContainer` vytvořit novou metodu, která bude mít požadovaný tvar deklarace `int jméno_metody(NeuralNetwork &)`.
2. Upravit metodu `run` ve třídě `AlgorithmContainer`, která určuje jaký trénovací nebo výpočetní algoritmus bude spuštěn pro konkrétní typ neuronové sítě.

5.10 Paralelizované oblasti pomocí OpenMP

V této části jsou popsány jednotlivé části algoritmů, které jsou paralelizovány pomocí OpenMP a výhody a nevýhody této implementace.

Hlavní výhoda i nevýhoda při použití OpenMP spočívá v časové náročnosti výpočtu, která je závislá na zvoleném počtu vrstev v síti a počtu neuronů v nich obsažených. Při menším počtu vrstev a neuronů (Př. XOR, 3-bitová parita) jsou implementované algoritmy značně časově náročnější než při výpočtu bez použití OpenMP. Teprve při velkém počtu neuronů a vazeb mezi nimi (tisíce a více) se projevuje časová úspora. Toto chování je dáno tím, že ve většině algoritmů není možné paralelizovat velké nedělitelné celky výpočtu, ale pouze menší a stále se opakující části. To způsobuje, že režie OpenMP je pro menší sítě časově náročnější než samotný výpočet a tato nevýhoda je smazána až při větších sítích, kdy lze paralelizovat velké objemy výpočtu.

Všechny paralelizované části jednotlivých algoritmů jsou spouštěny direktivou `#pragma omp parallel for`, protože se vždy jedná o kolekci více objektů stejného typu (vazby, neurony), které vykonávají stejné operace. Je tedy možné rozdělit jednotlivé iterace for cyklů mezi více vláken a zpracovávat více objektů souběžně. V direktivě není použita klauzule `schedule`, protože vzhledem k implementaci je nejvhodnější standardní režim `static`, který rozdělí iterace na stejně velké části a přidělí je jednotlivým vláknům.

Backpropagation

V implementaci algoritmu Backpropagation jsou obsaženy tři části, které lze smysluplně paralelizovat. Jedná se o výpočet jednotlivých parciálních derivací neuronů v síti, modifikaci vah jednotlivých vazeb mezi nimi a výpočet nových výstupních hodnot sítě, který ale v implementaci není z důvodu časové úspory implementovaný paralelně.

Toto řešení je zvoleno z důvodu, že pro menší sítě umožňuje časovou úsporu ušetřením jedné úrovně cyklů a režie OpenMP. Na paralelizovanou implementaci lze aplikaci přepnout odkomentováním její implementace, která je obsažena v metodě `Backpropagation`, zakomentováním původního for cyklu a rekompilací aplikace.

Při výpočtu parciálních derivací jednotlivých neuronů je paralelizován výpočet na základě rozdělení neuronů do jednotlivých vrstev. Vždy jsou současně vyhodnoceny neurony pouze v jediné vrstvě a tyto objekty jsou pro zpracování rozděleny mezi více vláken.

Největší časovou úsporu je možné získat u paralelizace změny vah v síti. Jednotlivé objekty reprezentující váhy jsou uloženy v rámci vektoru ukazatelů a protože nezáleží na pořadí změny jednotlivých vah, je možné úpravu jednotlivých objektů rozdělit mezi více vláken.

Kohonenova síť

V implementaci algoritmů pro výpočet a učení v kohonenově síti je vhodné paralelizovat dvě hlavní oblasti. Jedná se o výpočet vzdálenosti jednotlivých neuronů od vstupního vektoru a v učícím algoritmu dále o úpravu vah vítězného neuronu a sousedních neuronů.

Výpočet jednotlivých vzdáleností je část, kde paralelizace umožňuje v případě velkého počtu neuronů dosáhnout značné časové úspory. To je možné díky tomu, že pro každý neuron je nutné vykonat stejnou práci a protože nezáleží na jejich pořadí, je možné vytvořit vytvořit více vláken pouze jednou a rozdělit celou práci mezi ně. Tím je ušetřena nadbytečná režie u OpenMP.

V trénovacím algoritmu jsou dále paralelizovány části, které upravují váhy vítězného neuronu a váhy neuronů v určeném okolí. Časová úspora u první části je závislá na počtu vstupních neuronů, protože se úprava jednotlivých objektů vah rozdělí mezi více vláken. Opět platí, že pro větší počet vah se potřebný čas snižuje a naopak. U úpravy vah sousedních neuronů je časová úspora závislá na počtu neuronů ve vybraném okolí.

Kapitola 6

Experimenty

V této kapitole jsou popsáné sítě a problémy, které byly použité pro testování aplikace. Jsou uvedena data naměřená u vytvořené aplikace a knihovny FANN [4]. Experimenty byly měřeny a testovány na stroji obsahujícím čtyřjádrový procesor Intel Core i5-2500K 3.30GHz, 8GB RAM a operační systém Windows 7 Professional ve verzi Service Pack 1. Naměřená časová náročnost byla měřena přímo v rámci jednotlivých metod implementujících dané algoritmy a není tak ovlivněna vytvářením sítě, ukládáním dat a dalšími činnostmi aplikace. Naměřený čas u všech použitých experimentů je uvedený v sekundách. Byly testovány následující problémy:

XOR

Jedná se o standardní problém řešící logickou funkci exkluzivní disjunkce (XOR). Pro testování byla použita síť typu `feedforward` se strukturou 2-4-1, která byla převzata z [17]. Byla použita aktivační funkce `sigmoid`, hodnota $\varepsilon = 0.001$ a maximální počet trénovacích epoch byl nastaven na 4000. Následuje pravdivostní tabulka pro tuto logickou funkci:

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

Tabulka 6.1: Logická funkce XOR.

3-bitová parita

Síť řeší problém 3-bitové sudé parity. Byl použit typ sítě `feedforward` se strukturou 3-8-1. Počet neuronů ve skryté vrstvě byl určen na základě informací z [7]. Jako aktivační funkce byla použita funkce `sigmoid`, hodnota ε byla nastavena na 0.002 a maximální počet epoch byl nastaven na hodnotu 5000.

4-bitová parita

Příklad na řešení 4-bitové liché parity. Byla použita struktura sítě 4-8-1 typu `shortcut`. Počet neuronů ve skryté vrstvě byl určen na základě informací z [7]. Byla použita hodnota $\varepsilon = 0.002$, maximální počet epoch byl 5000 a jako aktivační funkce byla zvolena funkce `sigmoid`.

8-bitová parita

Příklad řeší 8-bitovou sudou paritu. Byl použit typ sítě `shortcut` se strukturou 8-8-8-1, aktivační funkce `sigmoid` a hodnota $\varepsilon = 0.01$. Maximální počet epoch byl nastaven na 5000.

Naměřená data

Následují tabulky s naměřenými daty. U jednotlivých položek bylo vždy provedeno 100 opakování a výsledné hodnoty byly zprůměrovány. Při určování průměrného počtu epoch byly do průměru uvažovány pouze opakování simulace, při kterých bylo dosaženo požadované přesnosti.

η, α	0.7, 0.9	0.7, 0.3	0.3, 0.9	0.3, 0.3	0.5, 0.6	0.5, 0.3	0.5, 0.9
XOR	4.26487	4.51739	4.56068	10.7537	4.06691	6.81128	2.03096
parity 3	0.70709	5.94534	2.04909	11.4751	12.1427	6.42216	1.262
parity 4	2.9755	7.96621	1.7891	45.6724	4.81899	24.788	2.16778
parity 8	180.965	17.5225	33.2689	29.1132	17.7995	23.0729	74.2058

Tabulka 6.2: Průměrný čas trvání simulace [s].

η, α	0.7, 0.9	0.7, 0.3	0.3, 0.9	0.3, 0.3	0.5, 0.6	0.5, 0.3	0.5, 0.9
XOR	77	95	134	165	91	111	36
parity 3	74	256	168	395	621	194	105
parity 4	182	171	81	1590	107	843	102
parity 8	521	103	375	114	110	102	597

Tabulka 6.3: Průměrný počet epoch učení.

η, α	0.7, 0.9	0.7, 0.3	0.3, 0.9	0.3, 0.3	0.5, 0.6	0.5, 0.3	0.5, 0.9
XOR	95%	97%	95%	93%	97%	94%	98%
parity 3	100%	98%	100%	94%	96%	97%	100%
parity 4	99%	98%	100%	73%	100%	88%	99%
parity 8	74%	100%	99%	100%	100%	100%	86%

Tabulka 6.4: Procenta dosažení požadované přesnosti.

η, α	0.7, 0.9	0.7, 0.3	0.3, 0.9	0.3, 0.3	0.5, 0.6	0.5, 0.3	0.5, 0.9
XOR	0.00072	0.00422	0.00151	0.01034	0.00353	0.00606	0.00088
parity 3	0.00316	0.01449	0.00947	0.04436	0.01421	0.02475	0.00425
parity 4	0.03505	0.05171	0.04098	0.07339	0.04185	0.05157	0.02071
parity 8	16.6574	2.12471	9.3168	1.028	1.83154	1.20576	15.3908

Tabulka 6.5: Průměrný čas trvání simulace při použití knihovny FANN [s].

η, α	0.7, 0.9	0.7, 0.3	0.3, 0.9	0.3, 0.3	0.5, 0.6	0.5, 0.3	0.5, 0.9
XOR	192	1284	443	3064	1034	1808	262
parity 3	245	1423	926	4339	1388	2434	406
parity 4	1098	2509	1070	3562	2047	2432	665
parity 8	1061	968	1547	668	685	739	2283

Tabulka 6.6: Průměrný počet epoch učení při použití knihovny FANN.

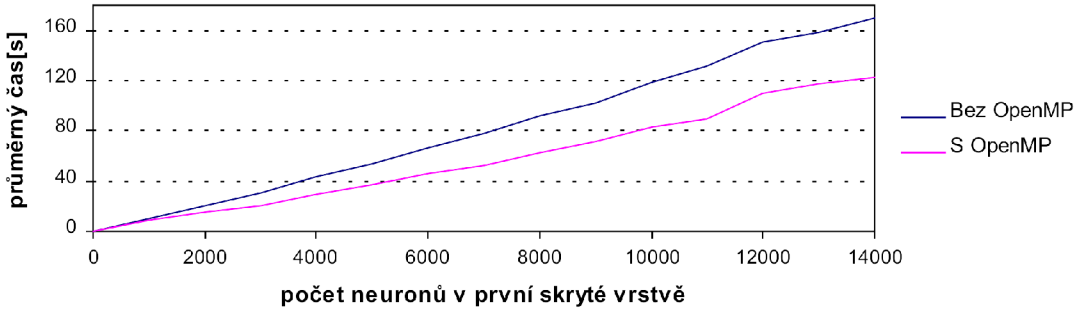
η, α	0.7, 0.9	0.7, 0.3	0.3, 0.9	0.3, 0.3	0.5, 0.6	0.5, 0.3	0.5, 0.9
XOR	100%	100%	100%	99%	100%	100%	100%
parity 3	99%	100%	100%	86%	100%	100%	100%
parity 4	85%	99%	75%	98%	100%	94%	92%
parity 8	1%	91%	35%	100%	92%	99%	4%

Tabulka 6.7: Procenta dosažení požadované přesnosti při použití knihovny FANN.

V další části experimentů byla měřena časová náročnost algoritmu BackPropagation. Ta byla měřena při použití OpenMP a bez použití OpenMP v závislosti na počtu neuronů ve vrstvách sítě. Pro měření byl použit problém 8-bitové parity, u kterého byl postupně zvyšován počet neuronů v první skryté vrstvě. Byly použité následující hodnoty: počet epoch učení 1, $\eta = 0.7$, $\alpha = 0.9$, $\varepsilon = 1e-009$. Tyto hodnoty byly zvoleny z důvodu, aby při měření časové náročnosti byla vždy provedena stejná míra práce a simulace nebyla předčasně ukončena na základě dosažení požadované přesnosti. Výsledné časy u jednotlivých hodnot představují průměr z deseti opakování simulace.

Počet neuronů	1000	2000	3000	4000	5000	6000	7000
Bez OpenMP	10.5639	21.0148	31.2603	43.7875	53.9664	65.8164	77.5025
S OpenMP	9.1023	15.5604	20.9747	29.9603	36.7336	45.5143	52.3539
Počet neuronů	8000	9000	10000	11000	12000	13000	14000
Bez OpenMP	91.6754	102.661	118.238	131.01	150.849	158.838	170.101
S OpenMP	62.6022	71.2321	82.8922	89.2895	109.887	117.633	122.032

Tabulka 6.8: Vliv OpenMP na časovou náročnost algoritmu Backpropagation [s].

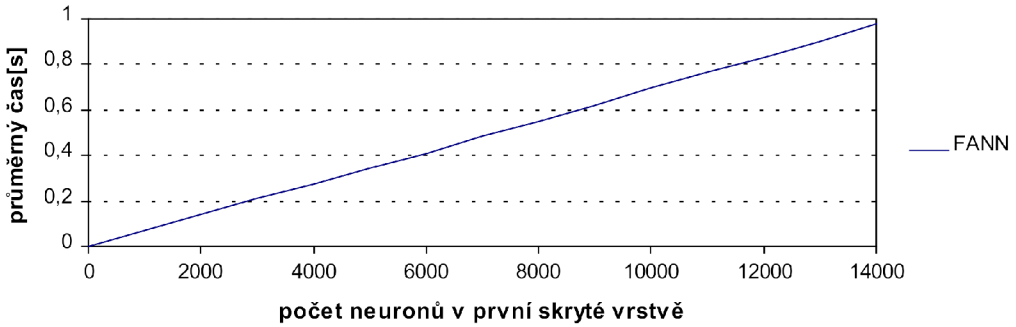


Obrázek 6.1: Vliv OpenMP na časovou náročnost algoritmu Backpropagation.

Stejný experiment byl simulován také za pomoci knihovny FANN.

Počet neuronů	1000	2000	3000	4000	5000	6000	7000
FANN	0.0692	0.1392	0.2709	0.2756	0.3468	0.4110	0.4846
Počet neuronů	8000	9000	10000	11000	12000	13000	14000
FANN	0.5527	0.6212	0.6961	0.7678	0.8344	0.9011	0.9758

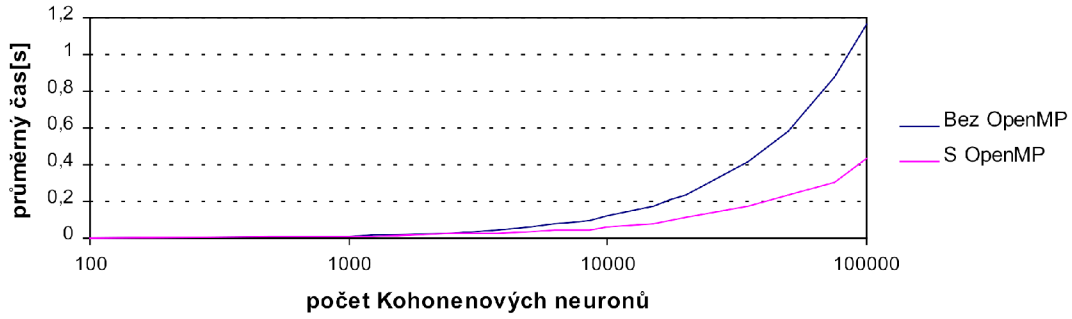
Tabulka 6.9: Časová náročnost simulace při použití knihovny FANN [s].



Obrázek 6.2: Časová náročnost simulace při použití knihovny FANN.

Kohonenova síť

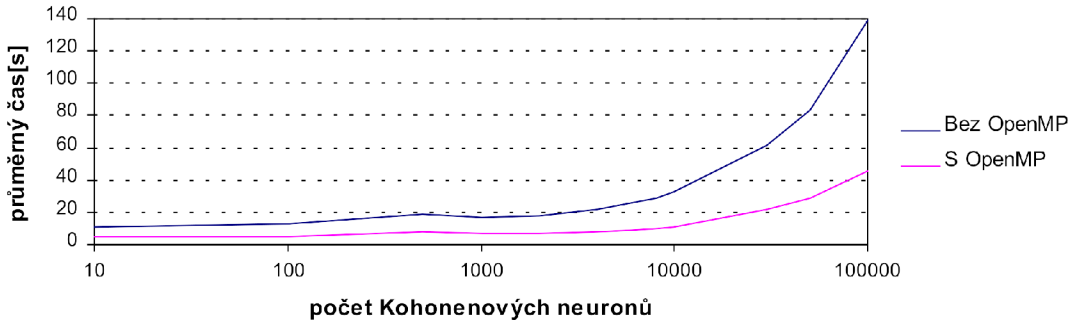
U Kohonenovy byl testován vliv akcelerace pomocí OpenMP na časovou náročnost simulace. Byla použita síť se třemi vstupními neurony a čtyřiceti vstupními vektory. Počet neuronů v Kohonenově vrstvě byl v průběhu testování měněn. Výsledné naměřené časové hodnoty jsou uvedeny v sekundách a představují průměr hodnot naměřených v rámci padesáti opakování simulace ve výpočetním režimu a pěti opakování simulace v trénovacím režimu. Při testování učení sítě byla použita topologická mřížka typu 1D, velikost okolí byla nastavena na hodnotu 5, parametr učení na hodnotu 0.8 a počet epoch na hodnotu 10.



Obrázek 6.3: Časová náročnost výpočtu v Kohonenově síti.

Počet neuronů	100	500	1000	1250	2500	3750
Bez OpenMP	0.00134	0.00574	0.01232	0.01442	0.02968	0.04384
S OpenMP	0.00388	0.00686	0.0112	0.01208	0.0205	0.0234
Počet neuronů	5000	6250	7500	8500	10000	15000
Bez OpenMP	0.05888	0.07412	0.08804	0.09788	0.12438	0.17324
S OpenMP	0.03388	0.04048	0.04632	0.0467	0.06296	0.07836
Počet neuronů	17250	20000	35000	50000	75000	100000
Bez OpenMP	0.2065	0.2357	0.4211	0.58018	0.87934	1.16106
S OpenMP	0.09664	0.11368	0.1744	0.23908	0.30452	0.4319

Tabulka 6.10: Časová náročnost výpočtu v Kohonenově síti [s].



Obrázek 6.4: Časová náročnost učení v Kohonenově síti.

Počet neuronů	10	100	500	1000	2000	4000
Bez OpenMP	11.085	13.1314	18.402	16.936	17.9362	21.5954
S OpenMP	4.6954	5.4336	7.945	6.7898	7.204	7.7692
Počet neuronů	6000	8000	10000	30000	50000	100000
Bez OpenMP	25.4516	29.184	32.2918	61.793	83.8674	139.471
S OpenMP	9.1918	9.907	11.3398	21.924	28.772	45.9326

Tabulka 6.11: Časová náročnost učení v Kohonenově síti [s].

Vyhodnocení experimentů

Z naměřených dat vyplývá, že simulace provedená v této aplikaci má větší časovou náročnost než simulace provedená pomocí volně dostupné knihovny FANN.

Z provedených experimentů vyplývá, že u jednoduchých problémů a sítí typu XOR nebo 3-bitová parita nemají různé hodnoty parametrů η a α větší vliv na přesnost simulace a ovlivňují spíše časovou náročnost simulace a počet potřebných epoch učení. U poněkud složitějších problémů typu 4-bitová a 8-bitová parita, už je možné sledovat vliv těchto parametrů na přesnost simulace. U 8-bitové parity bylo zjištěno, že při vysoké hodnotě parametru α se síť učí pomaleji nebo vůbec. U 4-bitové parity bylo změřeno, že při malých hodnotách parametrů η a α síť konverguje pomaleji a vzroste časová náročnost a počet potřebných epoch učení.

Dále byl měřen přínos použití OpenMP. Bylo změřeno, že při použití OpenMP v aplikaci klesá časová náročnost simulace u sítí, které obsahují cca. 1000 a více neuronů v jednotlivých vrstvách.

Backpropagation	29%
Kohonenova síť - výpočet	62%
Kohonenova síť - učení	29%

Tabulka 6.12: Průměrná časová úspora při použití OpenMP.

U algoritmu Backpropagation bylo zjištěno, že OpenMP zmenšuje čas potřebný k simulaci průměrně o 29% a průměrná procentuální velikost této časové úspory není příliš ovlivněna počtem neuronů v síti. U většiny ze zvoleného počtu neuronů se tato časová úspora pohybovala v rozmezí 25-35%.

U implementovaných algoritmů pro Kohonenovu síť bylo změřeno, že časová úspora je závislá na počtu neuronů v síti, kdy pro větší počet neuronů byla naměřena větší procentuální časová úspora. Výjimku představuje učení sítě s velmi malým počtem neuronů, maximálním počtem epoch učení a počtem vstupních vzorů, kdy použití OpenMP naopak simulaci zpomaluje. Při testování výpočtu se zvolenými počty neuronů průměrná časová úspora vzrůstala průměrně od 55% k 70% a průměrná časová úspora byla 62%. U učení v této síti vzrůstala časová úspora od 10% k 60-65% a průměrná hodnota časové úspory byla 29%. Při počtu neuronů v kohonenově vrstvě menším jak tisíc neuronů OpenMP simulaci zpomalovalo.

Kapitola 7

Závěr

V rámci této práce byla vytvořena aplikace simulující umělé neuronové sítě a učení těchto sítí algoritmem Backpropagation. Aplikace byla implementována tak, aby snadno umožňovala modifikaci struktury neuronových sítí oproti standardním typům sítí a umožňovala tak simulovat neobvyklé chování neuronových sítí a jejich poruchy. Při vytváření práce bylo také myšleno na možnost rozšiřování aplikace o nové typy aktivačních funkcí, sítí a algoritmů.

Dalším tématem práce byla akcelerace aplikace pomocí standardu OpenMP. Při použití OpenMP byla naměřena průměrná časová úspora 29% u algoritmu Backpropagation u sítě řešící 8-bitovou paritu, 29% u výpočtu v Kohonenově síti a 62% u učení v Kohonenově síti. Bylo také zjištěno, že průměrná časová úspora závisí na počtu neuronů a vazeb v síti a že u malých sítí s jednotkami až desítkami neuronů OpenMP simulaci spíše zpomaluje z důvodu vysoké režie při použití OpenMP.

U algoritmu Backpropagation bylo změřeno, že simulace dosáhla požadováné přesnosti učení v 95,6% testů u problému XOR, v 97,8% testů u 3-bitové parity, v 93,9% testů u 4-bitové parity a 94,1% testů u 8-bitové parity.

V porovnání s volně dostupnou knihovnou FANN byla zjištěno, že vytvořená aplikace vykazovala přesnější výsledky u problémů 4-bitové a 8-bitové parity a méně přesné výsledky u problémů XOR a 3-bitové parity. Bylo také změřeno, že aplikace má větší časovou náročnost než simulace provedená pomocí knihovny FANN.

V rámci budoucího vývoje aplikace by bylo možné zaměřit se na zlepšení paměťové náročnosti aplikace a ještě větší optimalizaci časové náročnosti. Také by bylo možné rozšířit aplikaci o další a rozsáhlejší možnosti modifikace sítí. Případně pro tyto modifikace implementovat samostatnou aplikaci s grafickým rozhraním pro úpravu konfiguračního souboru, které by nejspíš bylo pro rozsáhlejší modifikace sítě snadněji ovladatelné a přehlednější než ruční zadávání modifikací v konfiguračním souboru.

Literatura

[1] mgr inż. Adam Gołda; Ziaja, K.; Miernikowski, P.: Biased and non biased neurons. http://galaxy.agh.edu.pl/~vlsi/AI/bias/bias_eng.html [online].

[2] Bartsch, H.-J.: *Matematické vzorce*. SNTL - Nakladatelství technické literatury, 1983.

[3] Bias. http://www.heatonresearch.com/wiki/Bias [online], August 2011.

[4] Fast Artificial Neural Network Library. URL http://leenissen.dk/fann/wp/.

[5] FIT VUT v Brně: Praktické Paralelní Programování. http://www.fit.vutbr.cz/study/courses/PPP/public/, 2003.

[6] Hebb, D. O.: *The Organization of Behavior*. Wiley: New York, 1949.

[7] Hjelmas, E.; Munro, P. W.: A Comment on the Parity Problem. 1999.

[8] Hopfield, J. J.: Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the USA*, ročník 79, 1982: s. 2554–2558.

[9] Hopfield, J. J.: Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences of the USA*, ročník 81, 1984: s. 3088–3092.

[10] Katedra geofyziky MFF v Praze: Fortran 95 a paralelní programování, OpenMP. http://geo.mff.cuni.cz/~lh/NPRF039/index.htm [online], April 2013.

[11] Kohonen, T.: Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, ročník 43, 1982: s. 59–69.

[12] Krčma, M.: *Akcelerace neuronových sítí v FPGA*. Diplomová práce, FIT VUT v Brně, 2014.

[13] Leverington, D.: A Basic Introduction to Feedforward Backpropagation Neural Networks. http://www.webpages.ttu.edu/dleverin/neural_network/neural_networks.html [online], 2009.

[14] Šíma, J.; Neruda, R.: *Teoretické otázky neuronových sítí*. matfyzpress, 1996, ISBN 80-85863-18-9.

[15] Mařík, V.; Štěpánková, O.; Lažanský, J.; aj.: *Umělá inteligence (4)*. Academia, 2003, ISBN 80-200-1044-0.

[16] McCulloch, W. S.; Pitts, W.: A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, ročník 5, 1943: s. 115–133, ISSN 0092-8240.

[17] McCullock, J.: The XOR Problem. http://mnemstudio.org/neural-networks-backpropagation-xor.htm [online].

[18] Minsky, M. L.; Papert, S. A.: *Perceptrons*. The MIT Press, Cambridge MA, 1969, ISBN 0-262-63022-2.

[19] Munakata, T.: *Fundamentals of the New Artificial Intelligence*. Springer, 2008, ISBN 978-1-84628-838-8.

[20] OpenMP Architecture Review Board: OpenMP 4.0 Complete Specifications. http://openmp.org/wp/openmp-specifications/ [online], July 2013.

[21] Rosenblatt, F.: The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, ročník 65, 1958: s. 386–408.

[22] Rumelhalt, D. E.; Hinton, G. E.; Williams, R. J.: Learning internal representations by error propagation. In *Parallel distributed processing: explorations in the microstructure of cognition*, ročník 1, MIT Press, Cambridge, Massachusetts, USA, 1986, ISBN 0-262-68053-X, s. 318–362.

[23] Benchmark various sigmoid functions. https://gist.github.com/astanin/5270668.

[24] NN Session 1: What is a Neural Network? Weights & Thresholds. http://www.heatonresearch.com/wiki/NN_Session_1:_What_is_a_Neural_Network%3F_Weights_%26_Thresholds [online], March 2012.

[25] Widrow, B.: Generalization and information storage in networks of 'adaline neurons'. In *Self-Organizing Systems*, editace M. Yovitz; G. Jacobi; G. Goldstein, Spartan Book, Washington DC, 1962, s. 435–461.

[26] Zbořil, F. V.: Přednášky předmětu Soft Computing. http://www.fit.vutbr.cz/study/course-l.php?id=10197 [online], 2014.

Příloha A

Obsah CD

`\data` - Adresář obsahuje použité datové soubory.
- `*.txt` - Soubory s datovou sadou použitou pro trénování i testování sítě.
- `*.train` - Soubory s trénovací datovou sadou.
- `*.test` - Soubory s testovací datovou sadou.

`\examples` - Adresář obsahuje konfigurační soubory.
- `*.txt` - Konfigurační soubory.
- `xor_modify.txt` - Ukázka konfiguračního souboru s modifikacemi sítě.

`\source` - Adresář se zdrojovými soubory aplikace v jazyce C++.
- `\doxygen` - Adresář s dokumentací zdrojového kódu vytvořenou nástrojem Doxygen.
- `*.cpp` - Zdrojové soubory.
- `*.h` - Hlavičkové soubory.
- `Makefile` - Soubor pro přeloženi aplikace příkazem `make`.
- `BP.exe` - Už přeložený spustitelný soubor aplikace.

`\text` - Adresář se soubory pro vytvoření textové části práce pomocí systému LaTeX.
- `\fig` - Grafické soubory použité v této práci.
- `*.bst, *.tex, *.cls, *.bib` - Zdrojové soubory.
- `Makefile` - Soubor pro překlad příkazem `make`.

`projekt.pdf` - Soubor typu pdf s textem této práce.
`readme.txt` - Soubor obsahující nápovědu a základní informace o aplikaci.

Příloha B

Popis třídních proměnných a třídních metod

Kapitola stručně popisuje význam proměnných a metod vytvořených tříd, kromě tříd `FunctionContainer`, `AlgorithmContainer` a `LinkCreator`

U popisu jednotlivých třídních proměnných je implicitně uvažován modifikátor přístupu `private` a u třídních metod modifikátor přístupu `public`. Pokud je použitý jiný modifikátor přístupu než tyto implicitní, bude uveden u popisu příslušné proměnné nebo metody.

Třída NeuralNetwork

Třídní proměnné:

`std::string type`
: Identifikátor zvoleného typu sítě.

`std::vector<int> structure`
: Počet neuronů v jednotlivých vrstvách neuronové sítě.

`std::vector<std::string> activationNames`
: Identifikační názvy aktivačních funkcí. Jsou uloženy v pořadí zadaném v konfiguračním souboru.

`std::vector<std::string> activationParams`
: Parametry aktivačních funkcí v pořadí a tvaru zadaném v konfiguračním souboru.

`std::vector<int> activationChangesID`
: Identifikační čísla neuronů u kterých byla změněna aktivační funkce.

`std::vector<std::string> activationChangesNames`
: Identifikační názvy jednotlivých aktivačních funkcí u modifikovaných neuronů. Hodnota uložená na indexu `i` náleží k neuronu, který je určen identifikačním číslem uloženým na indexu `i` v proměnné `activationChangesID`.

`std::vector<std::string> activationChangesParam`
: Hodnoty parametrů aktivačních funkcí u modifikovaných neuronů. Hodnota uložená na indexu `i` náleží k neuronu, který je určen identifikačním číslem uloženým na indexu `i` v proměnné `activationChangesID`.

`std::vector<Links *> added_links`
: Obsahuje ukazatele na objekty reprezentující nestandardní vazby mezi neurony, které byly manuálně přidány v konfiguračním souboru.

`std::string outputFilename`
: Název výstupního souboru. Do tohoto souboru bude po ukončení simulace uložena

konfigurace neuronové sítě.

`std::string backupFilename`
: Název souboru do kterého může být v průběhu simulace zálohována aktuální konfigurace sítě.

`bool train`
: Příznak označuje trénovací režim aplikace.

`bool test`
: Příznak označuje testovací režim aplikace.

`bool standardOutput`
: Určuje jestli bude konfigurace sítě zapsána do souboru nebo na standardní výstup.

`double epsilon`
: Požadovaná přesnost pro ukončení učení neuronové sítě.

`int epochLimit`
: Maximální počet epoch učení.

`std::vector<double> inputValues`
: Vstupní datové vektory.

`std::vector<reqOutputValues>`
: Vektory s požadovánými výstupními hodnotami.

`std::vector<Neuron *> neurons`
: Vektor ukazatelů na objekty reprezentující neurony.

`std::vector<InputBlock *> inputs`
: Vektor ukazatelů na objekty reprezentující vstupní body sítě.

`std::vector<OutputBlock *> outputs`
: Vektor ukazatelů na objekty reprezentující výstupní body sítě. Pro výpis výstupních hodnot postačuje vypsat výstupní hodnoty těchto objektů.

`std::vector<Links *> links`
: Ukazatele na objekty reprezentující vazby mezi neurony v síti. Obsahuje pouze objekty, které mají na vstupu i výstupu objekt třídy `Neuron`.

`double bpr_alfa`
: Parametr hybnosti (momentum rate) v algoritmu Backpropagation.

`double bpr_learningRate`
: Parametr učení v (learning rate) v algoritmu Backpropagation.

`std::string koh_kohonenTopo`
: Typ topologické mřížky v Kohonenově síti.

`int koh_range`
: Rozsah okolí neuronů v Kohonenově síti.

`double koh_learningRate`
: Parametr učení pro učeni (learning rate) v Kohonenově síti.

Třídní metody:

`NeuralNetwork()`
: Konstruktor inicializuje následující proměnné: `type` = "", `train` = false, `test` = false, `standardOutput` = true, `outputFilename` = defaultName.txt, `backupFilename` = defaultBackup.txt, `epsilon` = 0.01, `epochLimit` = 10000.

`int init(int, char *[]);`
: Metoda postupně volá metody pro konfiguraci a vytváření sítě a připravuje tak aplikaci na simulaci. Také otevírá a kontroluje vstupní soubory.

`private: int createNetwork(std::ifstream &)`

Metoda slouží pro konfiguraci a vytváření neuronové sítě. Načítá a vyhodnocuje jednotlivé položky v konfiguračním souboru. Také volá metody pro vytváření neuronů a vazeb mezi nimi (třída `LinkCreator`.

`private: int createNeurons()`
: Vytváří objekty reprezentující jednotlivé neurony v aplikaci. Ke každému objektu volá funkci `createFunctionContainer`, která vytvoří objekt reprezentující aktivační funkci a ten je následně přiřazen k objektu konkrétního neuronu.

`private: int LoadParam(int, char *[])`
: Vyhodnotí argumenty zadané při spuštění aplikace. Pokud je zadán argument pro celkovou chybu sítě nebo maximální počet epoch učení, mají přednost před hodnotami, které jsou zadané v konfiguračním souboru.

`private: int loadInputData(std::ifstream &)`
: Metoda načte a zpracuje vstupní datový soubor, kde jeden řádek představuje jeden datový vektor. Dále zkontroluje správný formát, délku a počet vstupních vektorů.

`FunctionContainer * createFunctionContainer(std::string, std::string)`
: Metoda je vytvořena s parametrem přístupu `private`. Vytvoří objekt reprezentující aktivační funkci neuronu, který je nakonfigurován na základě parametrů metody.

`void setTrainingParameters(std::string, std::vector<std::string>)`
: Zpracovává a nastavuje parametry učících algoritmů, které jsou zadané položkou `training` v konfiguračním souboru. Parametry reprezentují typ sítě a hodnoty jednotlivých parametrů ve formátu řetězců.

`bool firstLayer(std::string)`
: Vráti hodnotu `true`, pokud neurony v první vrstvě neplní výpočetní funkci a pouze přeposílají vstupní hodnoty.

`void setNewLinks(Links *`
: Uloží ukazatel na objekt reprezentující vazbu mezi neurony.

`void setNewInput(InputBlock *)`
: Uloží ukazatel na vstupní objekt a vytvoří tak nový vstupní bod sítě. Pořadí vstupů je určeno pořadím při jejich uložení.

`void setNewOutput(OutputBlock *)`
: Uloží ukazatel na výstupní objekt a vytvoří tak nový výstupní bod sítě. Pořadí výstupů je určeno pořadím při jejich uložení.

`unsigned int getNumberOfInputs()`
: Vrátí počet vstupních datových vektorů.

`int getNumberOfNeurons()`
: Vrátí počet neuronů v neuronové síti.

`unsigned int getNumberOfOutputsNodes()`
: Vrátí počet výstupů z neuronové sítě.

`int getEpochLimit()`
: Vrátí hodnotu maximálního počtu epoch učení.

`double getEpsilon()`
: Vrátí hodnotu maximální povolené globální chyby sítě.

`std::vector<Links *> & getLinks()`
: Vráti vektor s ukazateli na objekty, které reprezentují vazby mezi neurony.

`std::string getType()`
: Vrátí identifikátor použitého typu neuronové sítě.

`std::vector<int> getStructure()`
: Vrátí vektor, který obsahuje počet neuronů v jednotlivých vrstvách sítě.

`std::vector<Neuron *> & getNeuronList()`
: Vrátí vektor, který obsahuje ukazatele na objekty reprezentující neurony v síti.

`std::vector<std::vector<double> > &getInputsValues()`
: Vrátí odkaz na vektor, který obsahuje uložené datové vektory.

`int run()`
: Metoda spouští simulaci a volá metodu `run` třídy `AlgorithmContainer`.

`int saveNetwork(bool = false)`
: Vypíše aktuální konfiguraci neuronové sítě. Standardně zapisuje na standardní výstup. Zadáním argumentu aplikace `--save`, je výpis přesměrován do zadaného souboru.

`private: std::string getTrainingProperties()`
: Vytváří řetězec pro uložení parametrů učícího algoritmu, který je následně zapsán v rámci položky `training` v konfiguračním souboru.

`double errOutput(int)`
: Vráti aktuální chybu neuronové sítě vypočtenou podle vztahu (B.1) [19]. Index i označuje pořadí výstupů sítě, y_j konkrétní výstup a t_j j-tou hodnotu požadováného výstupního vektoru pro odpovídající tréninkový vzor, který určuje parametr metody.

$$err = \frac{1}{2}\sum_j (t_j - y_j)^2 \tag{B.1}$$

`double diffOutput(int, int)`
: Vrací rozdíl mezi aktuální a požadovanou výstupní hodnotou. Konkretní výstup a trénovací vzor určují parametry metody.

`void printOutput(int = 0)`
: Vypíše aktuální výstupní hodnoty neuronové sítě na standardní výstup. Formát je určený režimem aplikace a je popsán v části (5.2.3).

`void setInputVector(int)`
: Nastaví na vstupy sítě hodnoty vstupního vektoru určeného parametrem metody.

`double getBprLearningRate(), double getBprAlfa()`
: Metody pro přístup k parametrům algoritmu Backpropagation,

`void setBprLearningRate(double), void setBprAlfa(double)`
: Metody pro nastavení hodnot jednotlivých parametrů algoritmu Backpropagation.

`std::string getKohonenTopo(), int getKohonenRange(),`
`double getKohonenLearnRate()`
: Metody pro přístup k parametrům učícího algoritmu pro Kohonenovu síť.

`void setKohonenTopo(std::string), void setKohonenRange(int),`
`void setKohonenLearnRate(double)`
: Metody pro nastavení jednotlivých parametrů učícího algoritmu pro Kohonenovu síť.

Třída DefaultBlock

Třídní proměnné:

`protected: double outputValue`
: Výstupní hodnota objektu.

Třídní metody:

`DefaultBlock()`
: Konstruktor inicializuje výstupní hodnotu na 0.0.

`virtual double getValue()`
: Vrátí výstupní hodnotu. V odvozených třídách může být metoda předefinována.

`void setValue(double)`
: Metoda umožňuje manuálně nastavit výstupní hodnotu objektu.

`virtual ~DefaultBlock()`
: Virtuální konstruktor.

Třída Neuron

Třídní proměnné:

`std::vector<Links *> inputLinks`
: Ukazatele na objekty třídy `Links`, které reprezentují vstupní vazby neuronu.

`std::vector<Links *> outputLinks`
: Ukazatele na objekty třídy `Links`, které reprezentují výstupní vazby neuronu.

`FunctionContainer *actFContainer`
: Ukazatel na objekt třídy `FunctionContainer` konfigurující aktivační funkci neuronu.

`double (FunctionContainer::*activationFunction)(double)`
: Ukazatel na metodu třídy `FunctionContainer` s vybranou aktivační funkcí.

`double (FunctionContainer::*derivateActFunction)(double)`
: Ukazatel na metodu třídy `FunctionContainer` s derivací aktivační funkce.

`double innerValue`
: Vnitřní potenciál neuronu.

`double lockValue`
: Výstupní hodnota uzamknutého neuronu příkazem `stuck nX` nebo `remove nX`.

`double bias`
: Hodnota prahu neuronu.

`int ID`
: Identifikační číslo neuronu.

`bool lockFlag`
: Příznak pro identifikaci uzamknutého neuronu.

Třídní metody:

`Neuron(int = 0)`
: Konstruktor inicializuje ID hodnotou určenou parametrem. Pokud není parametr zadán, je hodnota nastavena na 0. Dále inicializuje následující proměnné: `lockFlag` = false, `actFContainer` = NULL, `innerValue` = 0.0, `lockValue` = 0.0, `bias` = 0.0.

`int getID()`
: Metoda vrátí identifikační číslo neuronu.

`void newInnerValue()`
: Aktualizuje hodnotu vnitřního potenciálu neuronu na základě vztahu (2.1).

`void newValue()`
: Pomocí aktivační funkce vypočte a nastaví novou výstupní hodnotu neuronu.

`double getDerivative()`
: Vrátí hodnotu derivace aktivační funkce, vypočtenou na základě aktuální výstupní hodnoty. Pokud je neuron uzamknutý, místo derivace vrátí hodnotu `lockValue`.

`void setInputLinks(Links &)`
: Do vektoru `inputLinks` je uložen ukazatel na objekt reprezentující vstupní vazbu.

`void setOutputLinks(Links &)`
: Do vektoru `outputLinks` je uložen ukazatel na objekt reprezentující výstupní vazbu.

`void removeInputLinks(int)`
: Z vektoru `inputLinks` odebere ukazatel na indexu určeném parametrem metody.

`void removeOutputLinks(int)`
: Z vektoru `outputLinks` odebere ukazatel na indexu určeném parametrem metody.

`void lock(double)`
: Metoda uzamyká neuron a nastavuje hodnotu `lockValue` na hodnotu parametru.

`void unlock()`
: Metoda odemyká neuron.

`bool getLock()`
: Vráti hodnotu proměnné `lockFlag`.

`double getValue()`
: Vrátí aktuální výstupní hodnotu neuronu. Pokud je neuron uzamknutý, vrátí hodnotu proměnné `lockValue`.

`int getNumberOfOutputs()`
: Vrátí velikost vektoru `outputLinks`, která představuje počet vstupů do neuronu.

`Links & getOutputLink(int link)`
: Vrátí referenci na objekt reprezentující výstupní vazbu, který je uložen na indexu určeném parametrem metody v rámci vektoru `outputLinks`.

`double getBias()`
: Vrací hodnotu prahu neuronu.

`setBias(double)`
: Nastaví práh neuronu na hodnotu zadanou parametrem metody.

`int setFunctionContainer(FunctionContainer &)`
: Uloží ukazatel na objekt třídy `FunctionContainer` představující aktivační funkci.

`std::vector<Links *> getInputLinks()`
: Vrátí vektor s ukazateli na objekty reprezentující vstupní vazby neuronu.

`std::vector<Links *> getOutputLinks()`
: Vrátí vektor s ukazateli na objekty reprezentující výstupní vazby neuronu.

Třída InputBlock

Třídní metody:

`InputBlock()`
: Základní konstruktor této třídy.

Třída OutputBlock

Třídní proměnné:

`Links *link`
: Ukazatel na objekt třídy Links, který představuje vazbu z výstupního neuronu v síti.

Třídní metody:

`OutputBlock()`
: Základní konstruktor třídy.

`void setNewInputLink(Link & newLink)`
: Metoda přiřadí k objektu vstupní vazbu, která vychází z výstupního neuronu.

`double getValue()`
: Metoda vratí vstupní hodnotu objektu, tedy výstupní hodnotu z výstupního neuronu.

Třída Links

Třídní proměnné:

`DefaultBlock *input`
: Ukazatel na objekt, který představuje vstupní objekt vazby.

`DefaultBlock *output`
: Ukazatel na objekt, který představuje výstupní objekt vazby.

`double weight`
: Hodnota váhy vazby.

`double stuckValue`
: Výstupní hodnota vazby uzamknuté konfiguračním příkazem `stuck wX,Y = value`.

`bool lockFlag`
: Příznak signalizující uzamknutou vazbu příkazem `stuck wX,Y = value`.

`int IDInput, IDOutput`
: Identifikační čísla vstupního a výstupního objektu.

Třídní metody:

`Links(Default Block &inputBlock, DefaultBlock &outputBlock, double newWeight = 0.0)`
: Konstruktor nastaví ukazatele na vstupní a výstupní objekt a inicializuje váhu vazby. Vazba je implicitně nastavena odemknutá. Identifikační čísla vstupního a výstupního objektu jsou nastaveny na 0.

`void setWeight(double newWeight)`
: Nastavuje váhu vazby na hodnotu zadanou parametrem metody. Uzamknutá vazba váhu nemění.

`double getWeight()`
: Vrátí aktuální hodnotu váhy vazby.

`DefaultBlock & getInput()`
: Vrátí odkaz na vstupní objekt.

`DefaultBlock & getOutput()`
: Vrátí odkaz na výstupní objekt.

`double getValue()`
: Metoda vrací výstupní hodnotu vstupního objektu vazby. Pokud je vazba uzamknutá, je místo této hodnoty vrácena hodnota `stuckValue`

`double getFinalValue()`
: Vrátí výstupní hodnotu vstupního objektu vynásobenou hodnotou váhy vazby. Pokud je vazba uzamknutá, je místo této hodnoty vrácena hodnota `stuckValue`.

`lock(double)`
: Uzamyká vazbu a nastavuje proměnnou `stuckValue` na hodnotu parametru metody.

`unlock()`
: Odemyká vazbu a resetuje hodnotu `stuckValue` na hodnotu 0.0.

`getLock()`
: Vrátí hodnotu příznaku `lockFlag`.

`setID(int, int)`
: Metoda nastavuje identifikační čísla vstupního a výstupního objektu.

`int getInputID()`
: Vrátí identifikační číslo vstupního objektu.

`int getOutputID()`
: Vrátí identifikační číslo výstupního objektu.