Neurónové siete v C#

Neural networks in C#

Michal Pavlech

Diplomová práce
2009

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2008/2009

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Michal PAVLECH**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Neuronové sítě v C Sharp**

Zásady pro vypracování:

1. **Vypracujte literární rešerši na dané téma.**
2. **Zhodnoďte a porovnejte stávající řešení v zadané oblasti.**
3. **Vytvořte .Net assembly v jazyce C-sharp obsahující různé typy architektur neuronových sítí včetně důsledné dokumentace.**
4. **Na praktických příkladech klasifikace a predikce otestujte funkčnost a výkonnost vypracovaného řešení.**

Rozsah práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **DREYFUS, G. Neural Networks : Methodology and Applications. Berlin – Germany : Springer-Verlag Berlin Heidelberg, 2005. 497 s. ISBN-13 978-3-540-22980-3.**
2. **BOSE, N.K., LIANG, P. Neural network fundamentals with graphs, algorithms, and applications. [s.l.] : [s.n.], 1996. 478 s. ISBN 0-07-006618-3 .**
3. **ZELINKA, Ivan. Umělá inteligence : Neuronové sítě a genetické algoritmy. Brno : Brno – Vutium, 1998. 126 s. ISBN 80-214-1163-5.**

Vedoucí diplomové práce: **Ing. Ladislav Běhal**
Ústav aplikované informatiky

Datum zadání diplomové práce: **20. února 2009**

Termín odevzdání diplomové práce: **27. května 2009**

Ve Zlíně dne 13. února 2009

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Hlavným cieľom tejto práce je vytvoriť knižnicu na vytváranie a prácu s umelými neurónovými sieťami v jazyku C#.

V teoretickej časti je stručná história a popis fungovania neurónových sietí a učiacich algoritmov.

Praktická časť popisuje vytvorenú knižnicu. Popis je zameraný na verejne prístupné metódy jednotlivých tried. Ďalej sa v tejto časti nachádza popis vzorových aplikácií na prácu s knižnicou a stručný návod na prácu s ňou.

Klíčová slova: Neurónová sieť, neurón, backpropagation, perceptron, SOM, Hopfieldova sieť, strojové učenie

ABSTRACT

The main aim of this thesis is to create library that can be used to build and use artificial neural networks in C# programming language.

The theoretical part contains brief history and description of neural networks and learning algorithms.

The practical part describes the library itself. Description is focused on the publicly accessible methods of individual classes. In addition, this section describes sample applications to work with the library and a brief tutorial to work with it.

Keywords: Neural network, neuron, backpropagation, perceptron, SOM, Hopfield network, machine learning

Touto cestou by som chcel poďakovať vedúcemu mojej diplomovej práce Ing. Ladislavovi Běhalovi za odborné vedenie, rady a pripomienky.

Ďalej by som chcel poďakovať svojej rodine za podporu počas celej doby môjho štúdia.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.

Ve Zlíně

……...
Podpis diplomanta

OBSAH

ÚVOD

Pri vytváraní počítačov bola vždy snaha o napodobnenie a zefektívnenie činnosti ľudského mozgu. So zvyšovaním výkonu počítačov v priebehu dvadsiateho storočia sa stalo možným simulovanie dejov v mozgu pomocou softwaru. Priamym pozorovaním činnosti nervovej sústavy živých organizmov vznikli prvé modely nervových buniek, známych ako neuróny. Tieto modely sa stali základom pre tvorbu väčších funkčných celkov – umelých neurónových sietí. Odvážne vyhlásenia vedcov predpovedali širokú škálu využitia, od jednoduchých problémov až po komplexné úlohy z oboru umelej inteligencie.

Po počiatočnom nadšení a úspechoch záujem o neurónové siete upadol až do osemdesiatich rokov dvadsiateho storočia. Príčinou bola neexistencia účinného algoritmu učenia zložitejších sietí. Objavenie algoritmu backpropagation, spolu s novými architektúrami sietí pomohli oživiť toto odvetvie. V súčasnosti sú umelé neurónové siete predmetov výskumu na najprednejších svetových univerzitách a vo vývojových oddeleniach softwarových spoločností.

Predpokladom do budúcna je zvyšovanie nasadenia neurónových sietí v praxi, preto sú potrebné softwarové knižnice, ktoré umožňujú prácu s nimi. Pre jazyk C# už v súčasnosti existuje niekoľko knižníc na prácu s neurónovými sieťami. Tie sú ale zamerané značne jednoúčelovo, alebo implementujú len základné algoritmy učenia sietí. Pri tvorbe výsledného programu teda bude kladený dôraz na znovu použiteľnosť a aplikovanie pokročilejších metód učenia. Dôležitou súčasťou bude aj dokumentácia popisujúca činnosť jednotlivých tried a metód, spolu s návodom na používanie knižnice. Na demonštráciu použitia umelých neurónových sietí budú zhotovené ukážkové programy.

I. TEORETICKÁ ČÁST

1 POPIS NEURÓNOVÝCH SIETÍ

1.1 Stručná história

V roku 1943 napísali neurofyziológ Warren McCulloh a matematik Walter Pitts prácu o teoretickej činnosti neurónov. Pri snahe modelovať správanie neurónov v mozgu vymodelovali jednoduchú neurónovú sieť za pomoci elektrických obvodov. Tento model neurónu so sebou prinášal viacero problémov. Váhy na jednotlivých spojoch siete boli analyticky určované, teda sieť sa nedokázala učiť „za behu“. Ďalšie nevýhody modelu boli neschopnosť riešiť spojité a nebinárne problémy a skoková aktivačná funkcia.

V roku 1949 Donald Hebb publikoval *The Organization of Behavior*. V tejto svojej práci poukazoval na fakt, že neurálne spoje sú posilňované vždy, keď sú použité. Tento koncept je podstatný pri spôsobe, akým sa ľudia učia. To znamená, že ak sú dve nervové spojenia aktivované, spojenie medzi nimi sa zlepší.

Až zvyšovanie výkonnosti počítačov v päťdesiatych rokoch 20. storočia umožnil simulovanie hypotetických neurónových sietí. Prvýkrát sa o to pokúsil Nathanial Rochester z výskumných laboratórií IBM, bohužiaľ jeho snaha zlyhala.

Prelom nastal až v roku 1959 kedy Bernard Widrow a Marcian Hoff zo Stanfordskej univerzity vybudovali modeli nazvané ako ADALINE (ADAptive LINear Elements) a MADALINE (Multiple ADAptive LINear Elements, teda je zložený z niekoľkých sietí typu ADALINE). Obe siete boli založené na tzv. McCulloch-Pitts neurónoch. Úlohou ADALINE bolo rozoznávať binárne vzory, takže ak bol na jej vstup privádzaný prúd bitov z telefónnej linky, dokázala predpovedať nasledujúci bit. MADALINE bola prvá neurónová sieť aplikovaná na problém z reálneho sveta. Jej úlohou bolo za použitia adaptívneho filtra eliminovať ozveny pri telefonovaní. Aj keď je tento systém značne starý, je stále v komerčnej prevádzke (toto tvrdenie je ale spomenuté len v [1], žiadne ďalšie dôkazy podporujúce toto tvrdenie neboli nájdené).

Frank Rosenblatt využil závery Hebbovej práce, ktoré aplikoval na McCulloch-Pittsov model neurónu, čím dal vzniknúť perceptronu. V roku 1962 vydal knihu *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*, v ktorej popisuje správanie a základné vlastnosti jednovrstvej neurónovej siete typu percetron (perceptron pritom simuloval už v roku 1957).

V tom istom roku Widrow a Hoff vytvorili procedúru učenia, ktorá skúma hodnotu na vstupe predtým, ako ju váhy spojov upravia. Je založená na predpoklade, že aj keď jeden aktívny perceptron môže obsahovať veľkú chybu, je možné upraviť váhy a túto chybu rozložiť na celú sieť, alebo aspoň na priľahlé neuróny.

Kniha *Perceptrons: An Introduction to Computational Geometry* od Marvina Minského a Seymoura Paperta z roku 1969 poukazovala na slabosti siete perceptron. Autori v knihe kritizovali použiteľnosť tohto typu siete. Keďže sa jednalo o známych odborníkov z oboru, kniha mala za následok úpadok záujmu o neurónové siete na niekoľko najbližších rokov.

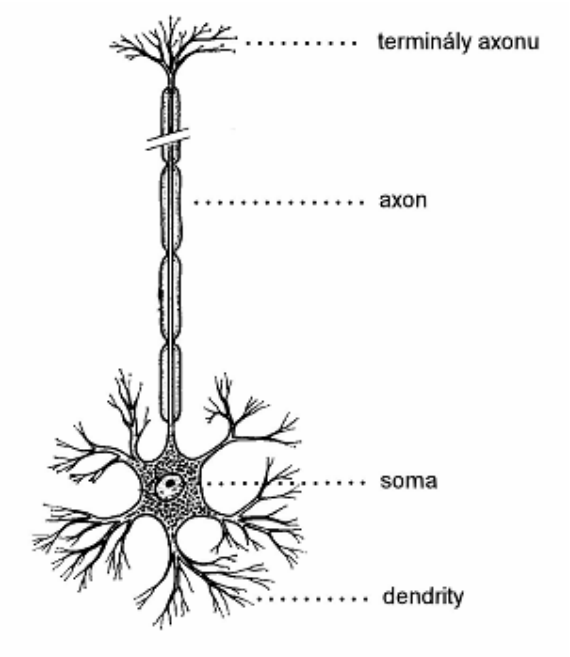
Záujem o neurónové siete bol obnovený až v roku 1982. John Hopfield prezentoval obojsmerný typ siete (dovtedy boli spoje medzi neurónmi len jednosmerné). V tom istom roku použili Reilly a Cooper hybridnú sieť, jednalo sa o viacvrstvová sieť, pričom každá vrstva mala inú stratégiu riešenia problémov.

S rozvojom viacvrstvových sietí bolo potrebné uplatniť Widrow-Hoffovo pravidlo o rozložení chyby na viac vrstiev. Simultánne sa to podarilo trom nezávislým tímom. Z výsledkov tohto výskumu vznikol algoritmus backpropagation. Názov je odvodený z činnosti algoritmu, kde chyby jednotlivých neurónov sú distribuované po sieti v smere od výstupu na vstup. Hybridné siete používali len 2 vrstvy, pri použití algoritmu backpropagation ich je spravidla viac. Dôsledkom je predĺženie doby učenia, ktorá môže prekročiť aj niekoľko tisíc iterácií.

1.2 Biologický základ

Nervová sústava živých organizmov a špecificky človeka, sprostredkuje vzťahy medzi vonkajším prostredím a organizmom, alebo jeho časťami. Takto zabezpečuje reakciu na vonkajšie podnety a vnútorné stavy organizmu. Tento proces prebieha šírením vzruchov z jednotlivých senzorov (receptory), ktoré umožňujú prijímať mechanické, tepelné, chemické a svetelné podnety. Tieto podnety ďalej pokračujú smerom k nervovým bunkám, ktoré signály spracúvajú a privádzajú k príslušným výkonným orgánom (efektory). Nervové vzruchy sa šíria po projekčných dráhach, kde dochádza k prvému predspracovaniu, kompresii a filtrácii informácie. Postupne sa dostávajú až do mozgovej kôry, ktorá je najvyšším riadiacim centrom nervového systému. Na povrchu mozgu je možné rozlíšiť šesť primárnych, vzájomne prepojených, projekčných oblastí približne

zodpovedajúcich zmyslom, v ktorých dochádza k paralelnému spracovaniu informácie. Komplexné spracovanie informácií, ktoré je základom pre vedomé riadenie činnosti efektorov, prebieha sekvenčne v tzv. asociačných oblastiach.

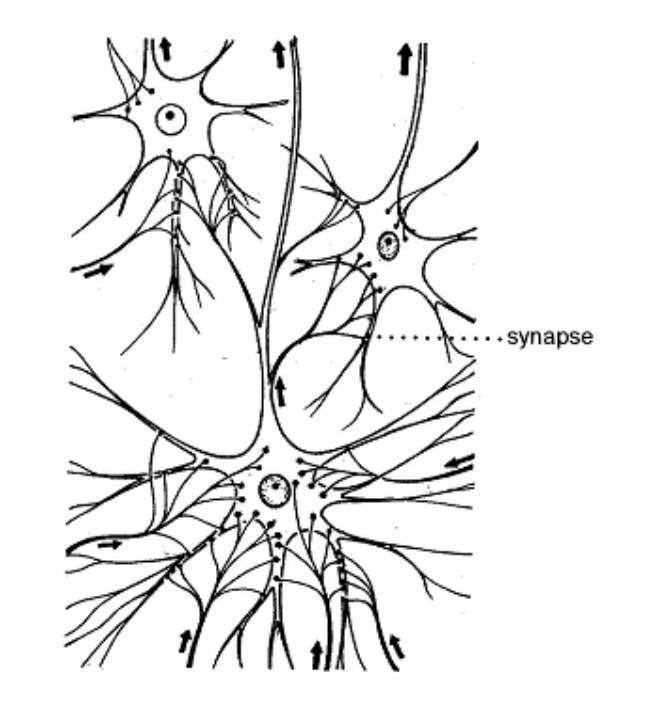


Obr. 1 Biologický neurón

Základným stavebným prvkom nervovej sústavy je nervová bunka (neurón). Mozgová kôra človeka obsahuje 13 až 15 miliárd neurónov, pričom každý môže byť spojený s ďalšími 5000 neurónmi. Neuróny sú špecializované bunky určené k prenosu, spracovaniu a uchovávaniu informácií nutných pre realizáciu životných funkcií organizmu.

Pre prenos signálov obsahuje neurón vstupné a výstupné prenosové kanály: dendrity a axón. Z axónu vybieha veľký počet vetiev, tzv. terminálov (viď. Obr. 1). Terminály sú zakončené blanou, ktorá sa dotýka výbežkov iných neurónov (tŕne dendritov). K prenosu informácie slúži unikátne medzineurónové rozhranie – synapsia (viď. Obr. 2). Miera synaptickej priepustnosti je nositeľom všetkých významných informácií behom života organizmu.

Z funkčného hľadiska je možné synapsie rozdeliť na excitačné, ktoré umožňujú rozšírenie vzruchu v nervovej sústave, a na inhibičné, ktoré spôsobujú útlm vzruchu. Predpokladá sa, že pamäťová stopa vzniká zakódovaním synaptických väzieb medzi receptorom a efektorom.



Obr. 2 Biologická neurónová sieť

Šírenie informácie je umožnené tým, že soma aj axón sú obalené membránou, ktorá má za určitých okolností schopnosť generovať elektrické impulzy. Tieto impulzy sú z axónu prenášané na dendrity iných neurónov synaptickými bránami, ktoré svojou priepustnosťou určujú intenzitu podráždenia ďalších neurónov. Takto podráždené neuróny po dosiahnutí určitej hraničnej hodnoty (prahu) generujú impulz a takto zabezpečujú šírenie príslušnej informácie. Po každom prechode signálu sa synaptická priepustnosť mení, čo je predpokladom pre pamäťovú schopnosť neurónov. Počas života organizmu podlieha zmene aj prepojenie neurónov. V priebehu učenia sa vytvárajú nové pamäťové stopy, naopak pri zabúdaní sa synaptické spoje prerušujú.

Nervová sústava človeka predstavuje veľmi zložitý systém, ktorý je stále predmetom skúmania. Uvedené princípy sú značne zjednodušené a ani zďaleka nepredstavujú ucelený popis problematiky, avšak sú dostatočné na vytvorenie modelu umelého neurónu a následne umelej neurónovej siete.

1.3 Model neurónu

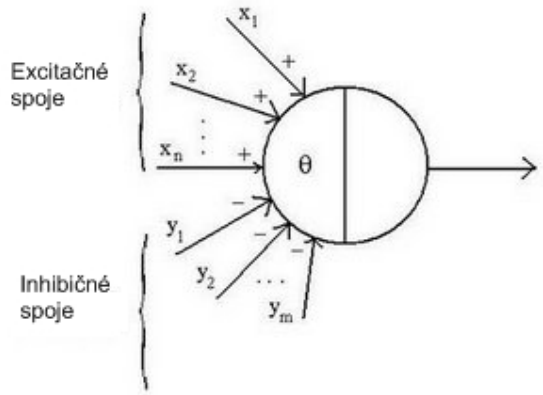
McCulloch a Pitts vytvorili model jednoduchých logických jednotiek, ktoré nazvali bunky, alebo neuróny. Úlohou týchto jednotiek bolo reprezentovať a napomáhať v analýze situácií,

ktoré nastávajú pri diskrétnych procesoch, či už v mozgu, v počítači, alebo kdekoľvek inde. Automaty vytvorené z týchto elementárnych jednotiek sa zvyčajne nazývajú neurónové siete.

McCulloch-Pitts model neurónu je jednoduchý dvojstavový automat. Z každého neurónu vychádza jedno výstupné vlákno, ktoré sa môže ďalej vetviť. Výstupné vlákna z rôznych neurónov sa nesmú spájať. Každé výstupné vlákno musí byť zakončené v neuróne (toto zakončenie sa môže nachádzať aj v pôvodnom neuróne, z ktorého vlákno vychádza), kde predstavuje vstupnú hodnotu. Povolené sú dva druhy ukončení: excitačné a inhibičné, pričom počet vstupov do neurónu je ľubovoľný.

Bunky sú prezentované ako konečné automaty a pracujú v diskrétnych časových intervaloch, ktoré sú pre všetky bunky synchrónne. V každom momente je bunka v jednom z dvoch stavov: aktívna (firing), alebo neaktívna (quiet). Pre tieto stavy sú pridelené príslušné výstupné signály, ktoré sú vysielané pomocou výstupného vlákna. Keďže existujú len dva výstupné stavy, je možné predstaviť si aktívny stav neurónu ako vyslanie impulzu, analogicky neaktívny stav ako nevytvorenie impulzu.

Zmena stavu bunky nastáva ako dôsledok pulzov privedených na jej vstupy. Prenosové vlastnosti bunky určuje veľkosť prahu neurónu (threshold). Bunka bude v čase *(k+1)* aktívna vtedy a len vtedy, ak v čase *k* počet aktívnych excitačných vstupov je rovný, alebo väčší ako prah neurónu. Pri prítomnosti aktívnych inhibičných vstupov zostáva bunka neaktívna.



Obr. 3 McCulloch-Pitts model neurónu

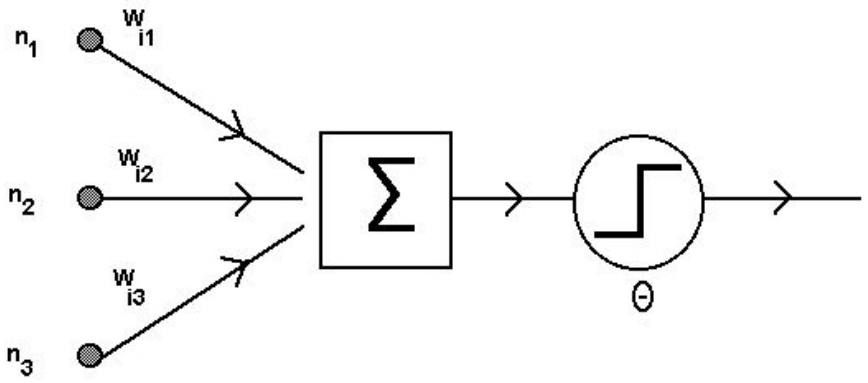
Sieť vytvorená z McCulloch-Pitts neurónov je založená na niekoľkých zjednodušujúcich predpokladoch:

1. Stav neurónu v čase *(k+1)* je závislý na jeho stave v čase *k* a na vstupoch do neurónu v danom čase.
2. Inhibičné vstupy sú považované za absolútne, teda jeden aktívny inhibičný vstup znemožňuje aktiváciu neurónu pri ľubovoľnom množstve aktívnych excitačných vstupov.
3. Medzi vstupom a výstupom z neurónu je pevne určený časový interval.

V neskorších modeloch bol predpoklad č. 2 zmenený, takže inhibícia už nebola absolútna, ale o aktivácii neurónov rozhodoval rozdiel počtu aktívnych excitačných a inhibičných vstupov. Z predpokladu č. 3 vyplýva, že umelá neurónová sieť pracuje synchrónne, na rozdiel od biologických sietí, v ktorých pracuje každý neurón v inom časovom rámci (je asynchrónna).

Každá neurónová sieť zložená z McCulloch-Pitts neurónov je konečný automat a každý konečný automat je ekvivalentný, a môže byť simulovaný neurónovou sieťou. V každom momente je celkový stav siete určený aktiváciou buniek, ktoré sú jej súčasťou.

Ďalším rozvojom práce McCullocha a Pittsa sú logické prahové jednotky (TLU-threshold logic units) s nastaviteľnými váhami. TLU je neurón s *n* vstupmi (x_1, x_2,...,x_n) a výstupom *y*.



Obr. 4 TLU

Neurón má *n+1* parametrov, menovite váhy (w_1 ,w_2,...,w_n) a prah neurónu θ. Výstup neurónu v diskrétnych časových momentoch *k = 1,2,...* je daný:

$$y(k+1) = \begin{cases} 1, & ak \sum_{i=1}^{n} w_i x_i > \theta \\ 0, & ak \sum_{i=1}^{n} w_i x_i \leq \theta \end{cases}$$

V prípade bipolárnej verzie je na výstupe neaktívneho neurónu hodnota -1.

Pozitívne hodnoty váh $w_i>0$ predstavujú excitačné synapsie, negatívne hodnoty váh $w_i<0$ predstavujú inhibičné synapsie.

Výstup z neurónu je často množinou *{α, 1}*, ktorá reprezentuje binárne stavy neurónu, kde *α* je nenulové číslo. Hodnota *α* môže byť -1 (v prípade bipolárnej verzie), alebo malé číslo väčšie ako nula. Nenulové hodnoty *α* spravidla povedú k rýchlejšej konvergencii takmer pri všetkých algoritmoch učenia. Pretože prírastky váh s nulovým vstupom sú nulové, použitím hodnôt rôznych od nuly je zabezpečená zmena váh vždy, keď je to potrebné.

Oba spomenuté modely neurónov sú diskrétne, výstupy biologických neurónov sú však spojité. Interakciu neurónov s okolím ovplyvňuje aj časový okamih, v ktorom nastala zmena potenciálu na synapsiách. Informácia nie je zakódovaná len pomocou veľkosti aktivácie spojov, ale aj rýchlosť aktivácie nesie určitú časť informácie. Reálne neuróny majú integračné časové oneskorenie kvôli ich kapacitancii. Priebeh hodnoty neurónu v čase je presnejšie popísaný spojitými funkciami než diskrétnymi hodnotami, použitými v TLU.

Celková hodnota vstupu x_i do *i*-teho neurónu je definovaná ako aproximácia potenciálu v some neurónu, na ktorý vplývajú excitačné a inhibičné synapsie spolu s prahom neurónu. Ak do neurónu neprichádzajú žiadne vstupné hodnoty mimo neurónovú sieť, a na jeho vstupy sú pridelené výstupné hodnoty y_j, kde *j=1,2,...,n* z *n* neurónov, potom je hodnota x_i:

$$x_i = \sum_{j=1}^{n} w_{ij} y_j - \theta_i$$

Výstup neurónu y_j predstavuje krátkodobý priemer miery aktivácie neurónu *j* a je definovaná ako:

$$y_j = f(\lambda x_j)$$

Kde λ je číslo väčšie ako nula. Unipolárny tvar aktivačnej funkcie *f()* je najčastejšie popísaný funkciou:

$$y_i = \frac{1}{1+e^{-\lambda x_i}} = \frac{1}{1+e^{-\lambda \sum_{j=1}^{n} w_{ij} y_j - \theta_i}}$$

Pre hodnoty λ blížiace sa k nekonečnu sa charakteristika aktivačnej funkcie blíži k charakteristike unipolárnej TLU (pre ďalšie používané aktivačné funkcie viď. 1.4).

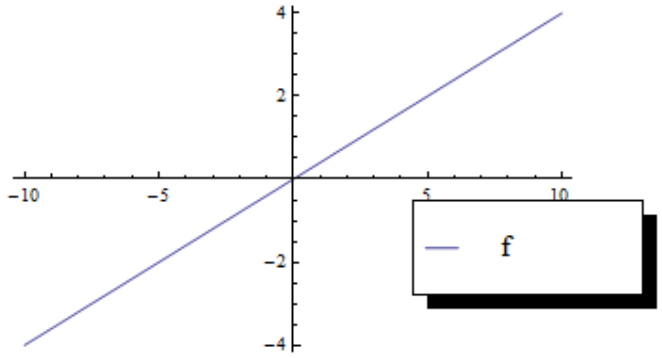
1.4 Aktivačné funkcie

Táto kapitola poskytuje popis aktivačných funkcií používaných v neurónoch s dopredným šírením. Výpočet derivácie je potrebný pre učenie gradientnými metódami (viď. 2.1)

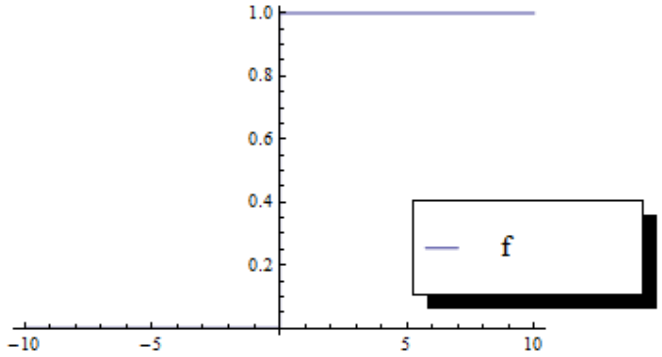
Tab. 1 Implementované aktivačné funkcie

Názov	**Aktivačná funkcia**	**Derivácia**
Lineárna funkcia	$y = s \cdot x$	$y' = s$
Skoková funkcia	$y = \begin{cases} 1, & pre\ x > \theta \\ 0, & pre\ x \le \theta \end{cases}$	-
Bipolárna skoková funkcia	$y = \begin{cases} 1, & pre\ x > \theta \\ -1, & pre\ x \le \theta \end{cases}$	-
Sigmoida	$y = \frac{1}{1+e^{-s*x}}$	$y' = s \cdot y \cdot (1-y)$
Bipolárna sigmoida	$y = -1 + \frac{2}{1+e^{-s*x}}$	$y' = 0.5 \cdot (1+y) \cdot (1-y)$
Hyperbolický tangens	$y = \tanh(s \cdot x)$	$y' = s \cdot (1-y^2)$
Gaussova funkcia	$y = e^{-(x^2 \cdot s^2)}$	$y' = -2 \cdot x \cdot y \cdot s^2$
Elliotova funkcia	$y = 0.5 + \frac{\frac{x \cdot s}{2}}{1+\lvert x \cdot s \rvert}$	$y' = s \cdot \frac{1}{2 \cdot (1+\lvert x \cdot s \rvert)^2}$
Bipolárna Elliotova funkcia	$y = \frac{x \cdot s}{1+\lvert x \cdot s \rvert}$	$y' = s \cdot \frac{1}{(1+\lvert x \cdot s \rvert)^2}$

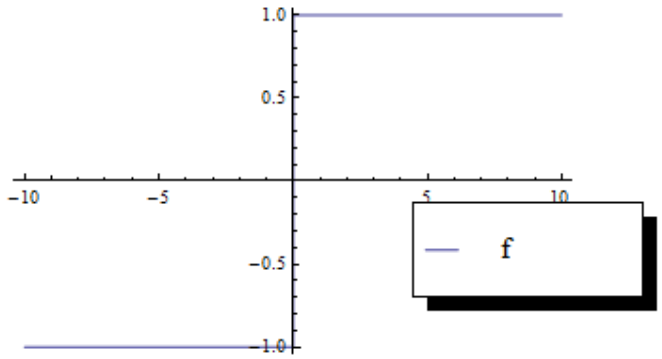
Premenná *y* predstavuje výstup z aktivačnej funkcie pre daný vstup, *x* je vstup do funkcie (suma ováhovaných vstupov do neurónu), *s* je strmosť aktivačnej funkcie. Vplyv strmosti na tvar funkcie je možné vidieť na Obr. 14 Sigmoida so strmosťou s=0,2, s=0,5, s=0,9.



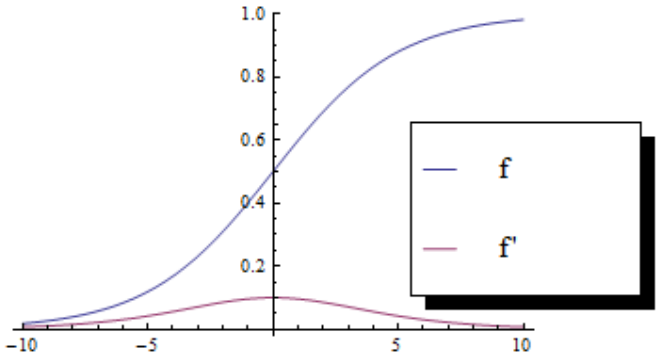
Obr. 5 Lineárna aktivačná funkcia



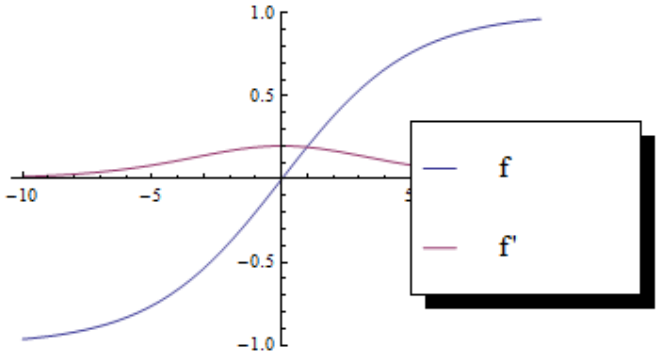
Obr. 6 Skoková funkcia



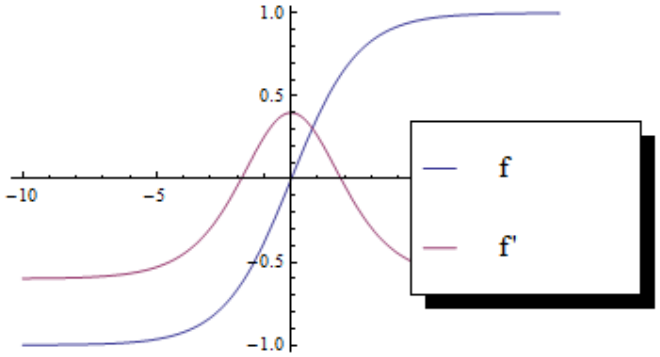
Obr. 7 Bipolárny perceptron



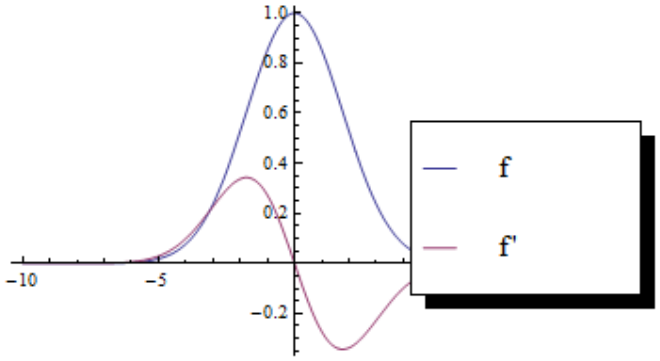
Obr. 8 Sigmoida



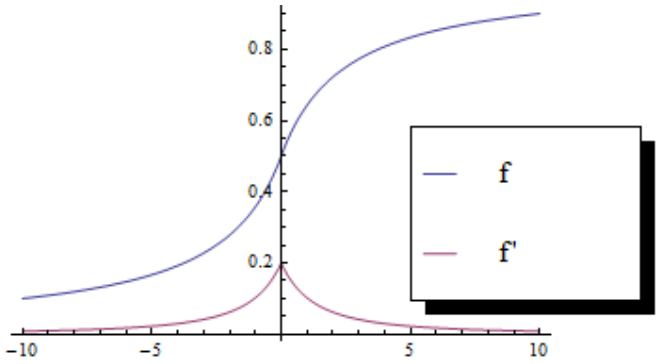
Obr. 9 Bipolárna sigmoida



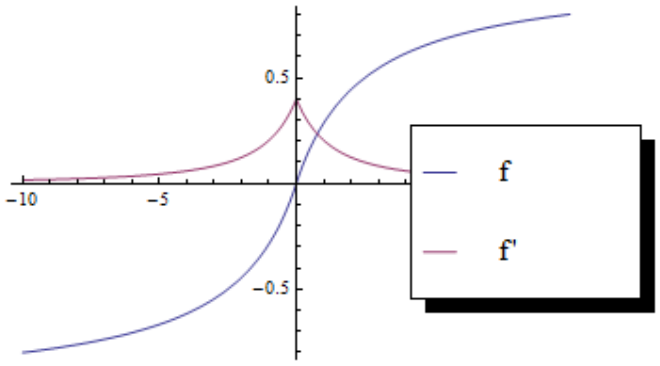
Obr. 10 Hyperbolický tangens



Obr. 11 Gaussova funkcia

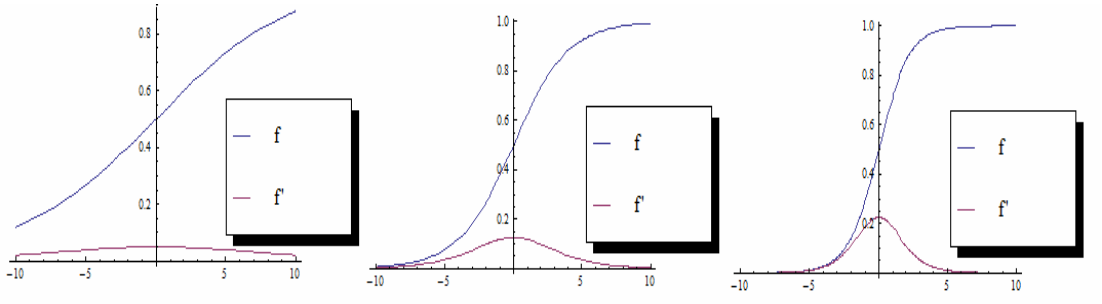


Obr. 12 Elliotova funkcia



Obr. 13 Bipolárna elliotova funkcia

Všetky grafy aktivačných funkcií boli vykreslené s rovnakou hodnotou strmosti *s=0,4* pre porovnanie ich priebehov.



Obr. 14 Sigmoida so strmosťou s=0,2, s=0,5, s=0,9

1.5 Vzdialenostné funkcie

Vzdialenostné funkcie sú potrebné pri učení a používaní SOM sietí. Sú určené na zisťovanie vzdialeností medzi vstupnými vektormi a neurónmi v sieti.

Tab. 2 Implementované vzdialenostné funkcie

Názov	**Vzdialenostná funkcia**
Euklidova vzdialenosť	$d = \sqrt{\sum_i (x_i - w_i)^2}$
Manhattan vzdialenosť	$d = \sum_i \lvert x_i - w_i \rvert$

Premenná *d* predstavuje výstup zo vzdialenostnej funkcie, teda vzdialenosť dvoch vektorov: vstupného vektoru *x* a vektoru váh *w*.

1.6 Siete s dopredným šírením

V súčasnosti najpoužívanejším typom neurónových sietí sú siete s dopredným šírením. Neuróny v sieti sú usporiadané vo vrstvách. Prvá vrstva sa nazýva vstupná a spravidla len mapuje svoje vstupy na výstupy. Jej využitie je možné pri transformácii vstupných hodnôt (normalizácia na určitú hodnotu, napr. interval (0;1)). Posledná vrstva sa nazýva výstupnou. Medzi vstupnou a výstupnou vrstvou sa môže nachádzať ľubovoľný počet vrstiev.

Počet neurónov vo vstupnej a výstupnej vrstve je daný riešeným problémom a spôsobom jeho číselnej reprezentácie.

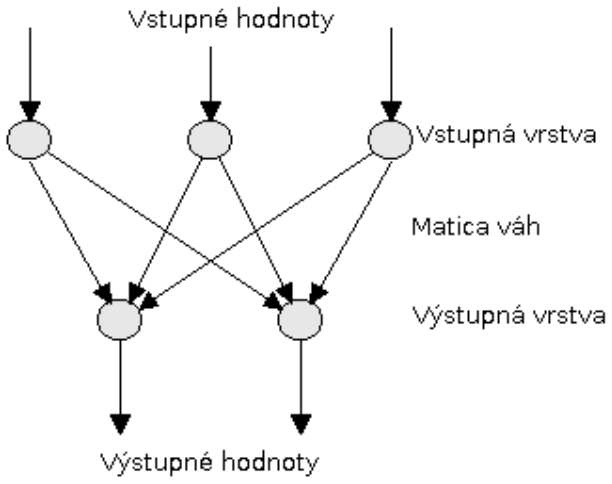
Vstupné vrstvy sa spravidla nerátajú pri určovaní počtu vrstiev v sieti. To znamená, že sieť so vstupnou, skrytou a výstupnou vrstvou je označovaná za dvojvrstvovú.

Neuróny vo všetkých vrstvách okrem vstupnej majú váhy, ktoré sú schopné učenia. Do každej skrytej a výstupnej vrstvy je zavedený jeden vstup, ktorý má za každých okolností nenulovú, konštantnú hodnotu. Tento vstup sa nazýva bias a pri učení zaručuje zmenu váh, aj keď sú všetky vstupy do neurónu nulové.

1.6.1 Perceptron

Jedným z prvých modelov neurónových sietí je sieť typu perceptron. Tento model bol navrhnutý Rosenblattom v roku 1957. Hlavným prínosom perceptronu bola jeho schopnosť učiť sa, táto prelomová vlastnosť spôsobila zvýšený záujem o výskum neurónových sietí.

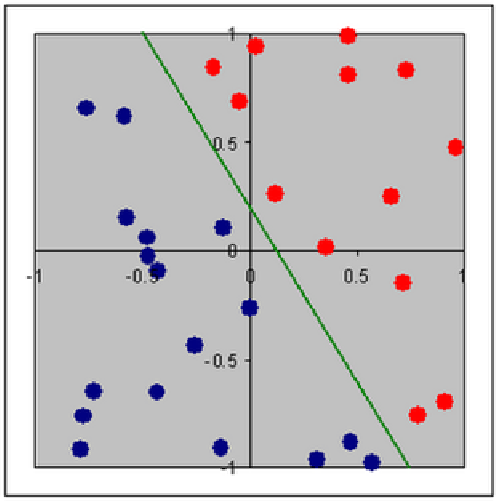
Perceptron je prepojená sieť neurónov, ktorá simuluje asociatívnu pamäť. Najjednoduchší perceptron je zložený zo vstupnej a výstupnej vrstvy, pričom obe vrstvy sú navzájom plne prepojené (každý neurón zo vstupnej vrstvy je spojený s každým neurónom z výstupnej vrstvy). Neuróny v jednej vrstve medzi sebou nemajú žiadne spojenia. Ku každému spoju medzi neurónmi je priradená váha, ktorej hodnota môže byť menená za účelom učenia siete.



Obr. 15 Perceptron

Pôvodná verzia perceptronu bola zložená z TLU, teda aktivačné funkcie neurónov mali skokový charakter.

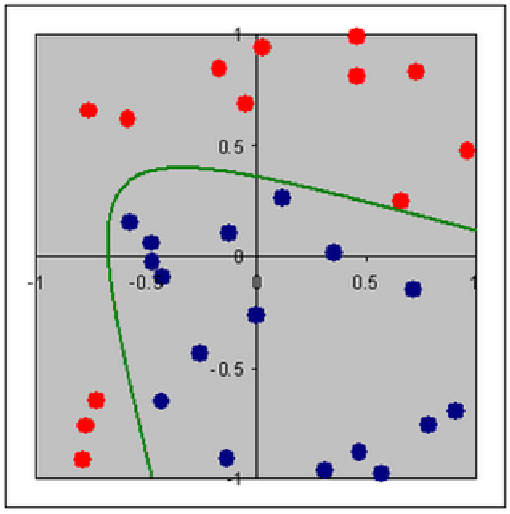
Najzásadnejším obmedzením perceptronu bola jeho jednovrstvová architektúra. Tá umožňovala riešiť len lineárne separabilné problémy, teda problémy, ktorých vzory je možné rozdeliť jednou priamkou (viď. Obr. 16).



Obr. 16 Lineárne separabilný problém

1.6.2 Viacvrstvá sieť s dopredným šírením

Ďalším rozvojom siete perceptron je viacvrstvová sieť s dopredným šírením (v niektorých zdrojoch nazývaná aj viacvrstvový perceptron). Rozdiel oproti perceptronu je v pridaní jednej, alebo viacerých skrytých vrstiev a v použití derivovateľných aktivačných funkcií neurónov.



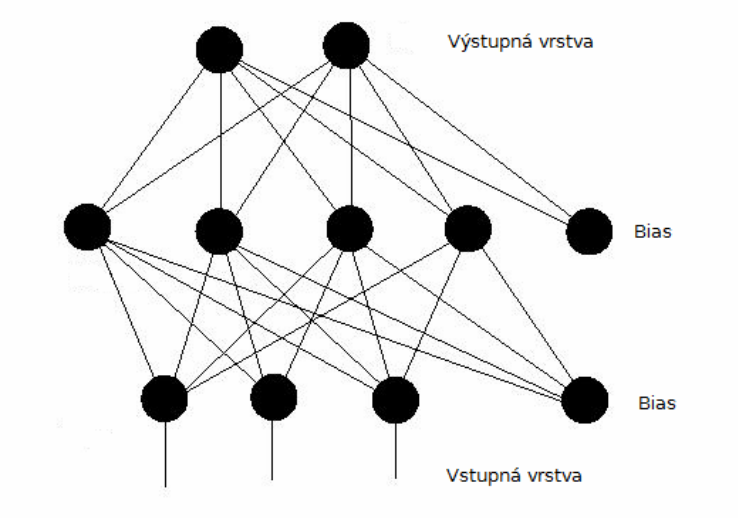
Obr. 17 Nelineárne separabilný problém

Siete so skrytými vrstvami sú schopné riešiť aj nelineárne separabilné problémy, a platí pre ne univerzálny aproximačný teorém. Tento teorém vyjadruje, že viacvrstvová sieť s dopredným šírením, ktorá obsahuje jednu skrytú vrstvu s konečným počtom neurónov a ľubovoľnými aktivačnými funkciami je schopná aproximovať ľubovoľnú spojitú funkciu na kompaktnej podmnožine R^N s ľubovoľnou presnosťou. Je predpoklad, že aktivačné funkcie vo výstupnej vrstve sú lineárne. Kurt Hornik v roku 1991 dokázal, že táto schopnosť siete nie je závislá na výbere aktivačnej funkcie, ale na viacvrstvovej štruktúre siete (pre odvodenie viď. [10]).

Kým pri perceptrone bol počet vrstiev pevne daný, a počet neurónov v nich bol daný riešeným problémom, pri viacvrstvových sieťach je možnosť ovplyvňovať ich vlastnosti počtom a veľkosťou skrytých vrstiev.

V súčasnosti neexistuje žiaden teoretický dôvod na používanie viac ako dvoch skrytých vrstiev. Avšak problémy, ktoré vyžadujú dve skryté vrstvy, sú vzácne a pre väčšinu problémov je dostatočná jedna skrytá vrstva.

Veľmi dôležitou časťou návrhu štruktúry neurónovej siete je určenie počtu neurónov v skrytých vrstvách. Použitie príliš malého počtu neurónov môže spôsobiť problém, nazývaný *underfitting* (voľne preložené ako podučenie). Ten nastáva, ak nie je v sieti dostatok neurónov, aby adekvátne pokryli signály v zložitej množine dát.



Obr. 18 Viacvrstvá sieť s dopredným šírením

Pri použití príliš veľkého množstva neurónov sa môžu vyskytnúť dva problémy. Prvým je *overfitting* (preučenie). Tento jav nastáva, ak má sieť takú kapacitu na spracovanie informácie, že obmedzený počet dát v trénovacej množine niex je schopná naučiť všetky neuróny. V praxi to znamená, že sieť je naučená na šumové hodnoty z trénovacej množiny a stratí svoje generalizačné vlastnosti. Druhý problém sa objavuje, ak je pri učení dostatok dát. Veľké množstvo neurónov spolu s veľkou trénovacou množinou môžu predĺžiť čas potrebný na učenie do takej miery, že učenie nebude prakticky použiteľné.

Na určenie počtu neurónov existuje viacero metód, napríklad: šľachtenie siete pomocou evolučných techník, kaskádová korelácia, orezávanie (pruning).

1.7 Hopfieldova sieť

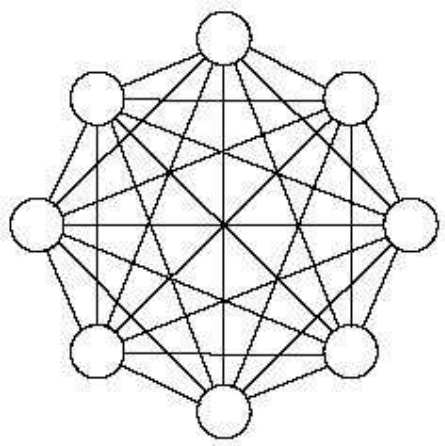
Jednou z najdôležitejších funkcií ľudského mozgu je schopnosť ukladať a znovu obnovovať spomienky. To umožňuje učiť sa z predchádzajúcich skúseností. Ľudské spomienky sú ukladané s asociatívnou vlastnosťou. Teda časti pamäte (napríklad výstupy z receptorov) nie sú v mozgu uložené izolovane alebo lokálne pre istú časť neurónov, pamäť je distribuovaná.

Asociatívne spoje medzi spomienkami umožňujú vybavovať si fakty, ktoré spolu súvisia – existujú medzi nimi asociácie. Dôležité je, že celkovú spomienku je možné obnoviť na základe spomenutia si len na časť. Prístup k informáciam je teda na základe ich obsahu, nie ich umiestnenia v neurónovej sieti. Takýto prístup k spomienkam dovoľuje vybaviť si

kompletnú informáciu na základe poškodených vstupov. Napríklad, ak človek uvidí zle zaostrenú fotografiu známej tváre, je schopný vybaviť si danú tvár s veľkou presnosťou.

Takýto prístup je značne odlišný od tradičného prístupu v počítačoch, kde sú špecifické informácie uložené na špecifických umiestneniach. Ak má počítač o umiestnení len čiastočné informácie obnovenie pôvodných dát je nemožné.

Za účelom simulovania daného správania ľudského mozgu vytvoril John Hopfield rekurentnú neurónovú sieť, ktorá nesie jeho meno.



Obr. 19 Hopfieldova sieť

Hopfieldova sieť je zložená z jednej vrstvy neurónov so skokovou aktivačnou funkciou, počet neurónov je rovnaký ako rozmer vstupného vektoru siete. Každý neurón je prepojený so všetkými ostatnými neurónmi, odtiaľ názov rekurentná sieť. Váhy spojení medzi neurónmi sú vnímané ako symetrická, štvorcová matica s nulovou diagonálou (neurón nemá spojenie sám so sebou). Symetrickosť matice váh je nutná podmienka pri vybavovaní si výstupných hodnôt [27].

Hopfieldova sieť nepoužíva chybovú funkciu, ale tzv. energetickú funkciu siete, ktorá je vlastná pre každý stav siete:

$$E = -\frac{1}{2}\sum_{i<j} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

Kde w_{ij} predstavuje váhu spojenia medzi neurónmi s indexmi i a j, pričom $i \neq j$, s_i je stav neurónu j a θ_i je prah neurónu i.

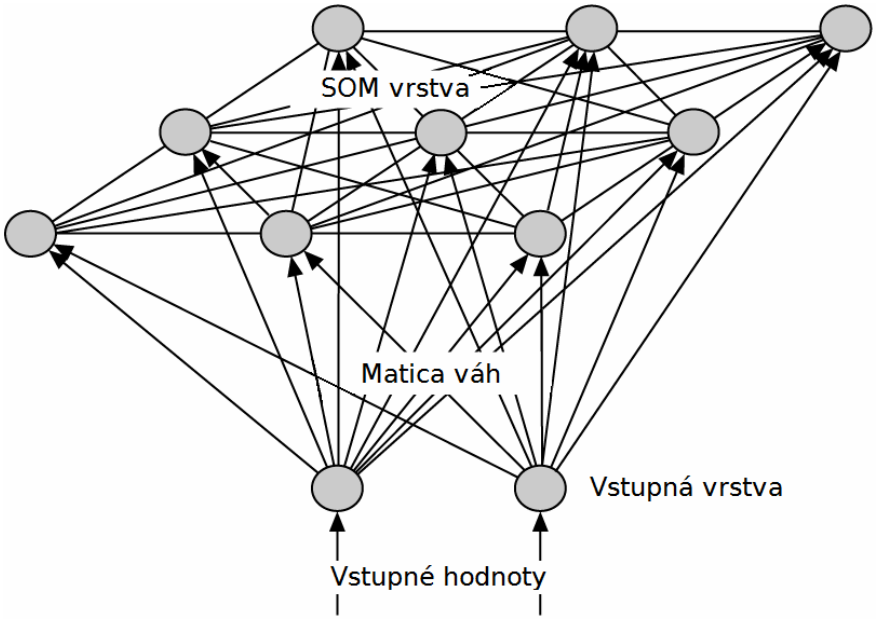
Ďalšou zvláštnosťou siete je, že učenie siete je jednorazové, ale vybavovanie prebieha iteratívne. Získanie výstupu zo siete na základe predloženého vstupu znamená konvergovať do lokálneho minima energetickej funkcie. Vybavovanie prebieha podľa nasledujúceho algoritmu:

```
1 z = 0
2 while z < prah výstupu begin
3     náhodný výber neurónu zo siete a predloženie zodpovedajúceho
      vstupu
4     ak nedošlo k zmene stavu neurónu zvýšenie hodnoty z o 1
5     inak z = 0
6 end;
```

1.8 SOM

SOM (Self Organizing Map) je neurónová sieť so schopnosťou samoorganizácie, niekedy nazývaná aj Kohonenova samoorganizačná mapa, podľa svojho tvorcu Teuvo Kohonena. Tento typ siete je určený na učenie pomocou algoritmov bez učiteľa, aby vytvoril nízko rozmernú (typicky len dvojrozmernú), diskrétnu reprezentáciu trénovacích vektorov. Takáto reprezentácia sa nazýva mapa. SOM sú odlišné od neurónových sietí s dopredným šírením v používaní tzv. vzdialenostnej funkcie na zachovanie topologických vlastností vstupov. Táto vlastnosť je užitočná pri vizualizácii mnoho rozmerných dát do dvojrozmerného priestoru.

SOM pozostáva z dvoch vrstiev neurónov: vstupnej vrstvy a pracovnej vrstvy, ktorá je zároveň výstupná. Každý neurón v pracovnej vrstve má váhy schopné učenia, ich počet zodpovedá veľkosti vstupného vektora siete: Na rozdiel od neurónov v sieťach s dopredným šírením neobsahujú neuróny bias. Neuróny sú usporiadané v obdĺžnikovej, alebo štvorcovej štruktúre. Aj keď sú neuróny v pracovnej vrstve zobrazované spojené medzi sebou (viď Obr. 20), tieto spojenia vyjadrujú len ich topologickú blízkosť, neslúžia na prenos informácie.



Obr. 20 Štruktúra Kohonenovej mapy

Výstupom zo siete nie je konkrétna hodnota, sieť len určí neurón, ktorého vzdialenosť od vstupného vektoru je najnižšia, tento proces sa nazýva aj mapovanie. Je tak možné určiť, v ktorej časti dvojrozmerného priestoru sa daný vektor nachádza. Z charakteru algoritmu učenia ([21]) vyplýva, že rovnaké, alebo podobné vektory budú umiestnené blízko seba. Vo výsledku je teda možné určovať príbuznosť vstupných dát podľa ich umiestnenia v sieti. Človek si totiž dobre vie predstaviť priestor najviac tretej dimenzie a tak musíme viacrozmerné vstupy upraviť. Hľadať totiž vzťahy medzi viacrozmernými vstupmi by pre človeka bola príliš zložitá úloha [24].

2 ALGORITMY UČENIA

Základným predpokladom pre použitie neurónových sietí je ich schopnosť učiť sa. Bez fungujúceho algoritmu učenia je neurónová sieť nevyužiteľná. V tejto kapitole sú popísané učiace algoritmy, ktoré sú aplikované v praktickej časti tejto práce.

2.1 Učenie s učiteľom

Učenie s učiteľom je skupina algoritmov, pri ktorom učenie prebieha na základe predložených dát a zodpovedajúcich požadovaných výstupov. Sieť sa snaží upraviť svoje váhy tak, aby jej výstup reagoval na vstupné dáta zodpovedajúcim požadovaným výstupom. Trénovacie dáta teda pozostávajú z dvojíc hodnôt.

2.1.1 Perceptron

Učiaci algoritmus určený pre učenie siete perceptron so skokovou aktivačnou funkciou. Jednotlivé kroky učenia sú rovnaké pre všetky neuróny vo výstupnej vrstve, nasledujúce kroky sú pre jeden neurón. Premenné prítomné pri učení: x - vektor vstupných hodnôt, $f(x)$ - aktuálny výstup z neurónu, y - požadovaný výstup, α - miera učenia, w_i - váha neurónu.

```
Inicializácia váh neurónu na náhodné hodnoty
for c<počet iterácií učenia begin
    for j<počet vstupných vzorov begin
        Výber dvojice vektorov z trénovacej množiny, a predloženie
        vstupov neurónu
        Výpočet výstupu y a chyby Δyj=yj-fj(x)
        for i<počet váh v neuróne begin
            Úprava všetkých váh neurónu wi= wi +α. Δyj
        end;
    end;
    Ak je chyba nulová, alebo dostatočne malá ukonči učenie
end;
```

Konvergenčný teorém pre učenie siete perceptron hovorí: Ak existuje súbor váh w^*, ktorý umožňuje previesť transformáciu $y=f(x)$, potom učenie konverguje k riešeniu (ktoré nemusí byť nutne totožné s w^*) v konečnom množstve krokov učenia bez ohľadu na počiatočné nastavenie váh (pre dôkaz viď. [11]).

Miera učenia je konštanta, kde: $0< \alpha \leq 1$. Miera učenia určuje, aká časť chyby neurónu sa bude podielať na zmene váh. Pri malých hodnotách bude potrebných viac krokov učenia na

konvergenciu, pri veľkých hodnotách bude učenie prebiehať rýchlejšie, prehľadávanie priestoru váh však nebude také dôkladné.

2.1.2 Delta pravidlo

Perceptron s lineárnymi aktivačnými funkciami je schopný prezentovať lineárne závislosti medzi vstupnými a výstupnými hodnotami siete. Takáto sieť je vhodná na aproximáciu lineárnych funkcií. Delta pravidlo využíva chybovú funkciu siete na úpravu váh:

$$E = \frac{1}{2}\sum_{n}(y_n - f_n(x))^2$$

Index n predstavuje pozíciu neurónu vo výstupnej vrstve. Táto chybová funkcia predstavuje mieru chyby siete pre jeden vzor učenia. Delta pravidlo aplikuje metódu záporného gradientu. Tá spočíva v úprave váh proporcionálne k derivácii chyby pre daný vzor vzhľadom na všetky váhy neurónu:

$$\Delta w_{nj} = -\alpha \frac{\partial E_n}{\partial w_{nj}}$$

Index j určuje pozíciu váhy v neuróne. Z tohto vzťahu je možné odvodiť výpočet prírastku pre jednotlivé váhy:

$$\Delta w_{nj} = \alpha(y_n - f_n(x))g'(h_n)x_j$$

Kde g predstavuje aktivačnú funkciu neurónu, h_n sumu ováhovaných vstupov do neurónu a $g'(h_n)$ deriváciu aktivačnej funkcie vzhľadom k jej vstupom.

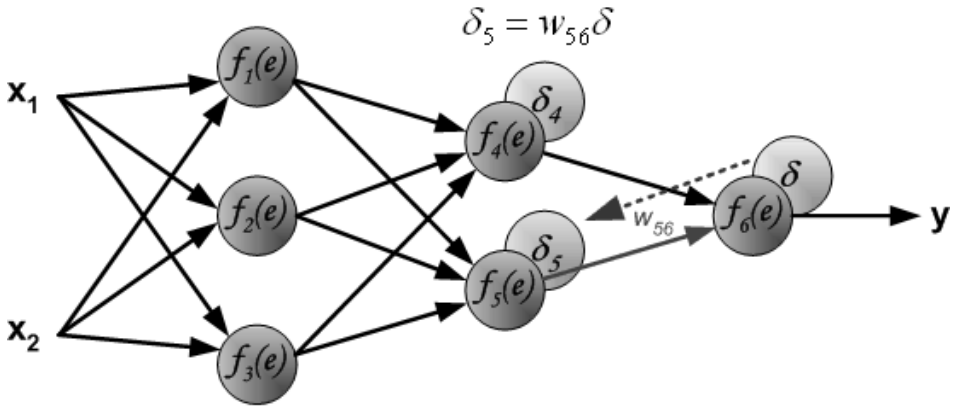
Konvergencia tohto učiaceho algoritmu, na rozdiel od učenia perceptron, nie je zaručená.

2.1.3 Backpropagation

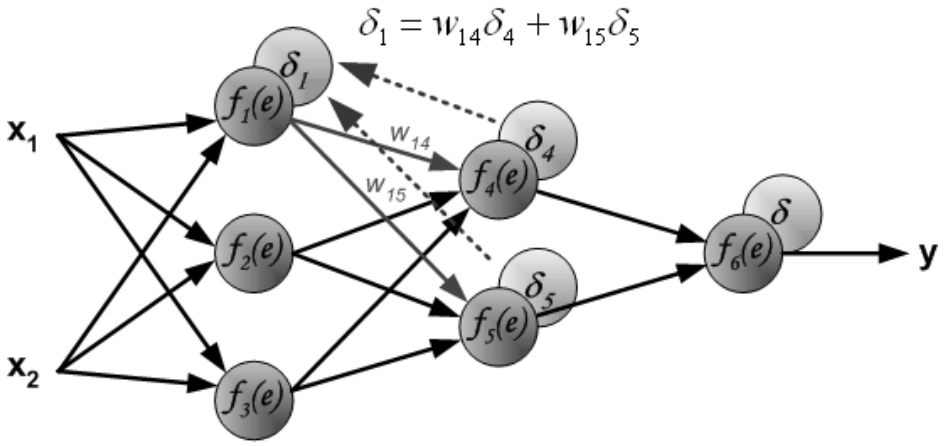
Zrejme najznámejším a najpoužívanejším algoritmom na učenie viacvrstvových sietí s dopredným šírením je algoritmus backpropagation. Jedná sa o zovšeobecnenie delta pravidla pre skryté vrstvy siete. Základom učenia je spätné šírenie chyby od výstupnej vrstvy na vstupnú.

Na začiatku každého cyklu učenia sú sieti prezentované vstupné hodnoty a sieť následne spočíta k nim zodpovedajúce výstupy. Výstupné hodnoty zo siete sú následne porovnané s požadovanými hodnotami výstupov a je spočítaná chyba na jednotlivých výstupných

neurónoch podľa vzorca: $\delta_i = y_i - f_i(x)$, kde y_i je požadovaný výstup na i-tom neuróne a $f_i(x)$ je výstup z daného neurónu. Po zistení veľkosti chyby je možné spočítať úpravy váh pre neuróny výstupnej vrstvy. Výpočet chyby pre neuróny v skrytej vrstve však nie je možný, pretože požadované hodnoty pre ne nie sú známe. Preto je nutné chybu na výstupoch siete spätne šíriť smerom na vstupy. Pri šírení chyby je nutné zohľadniť váhy na spojeniach, ktorými sa chyba šíri. Proces šírenia chyby je schematicky zobrazený na Obr. 21 a Obr. 22. Plné čiary predstavujú ováhované spojenia medzi neurónmi, prerušované čiary predstavujú šírenie chyby.



Obr. 21 Spätné šírenie chyby z výstupu na skrytú vrstvu



Obr. 22 Spätné šírenie chyby medzi skrytými vrstvami

Po výpočte chyby pre všetky neuróny v sieti, ktoré majú váhy schopné učenia je možné spočítať prírastky váh podľa vzťahu:

$$w_{ni} = w_{ni} + \alpha\delta_n \frac{\partial f_n(e)}{\partial e} x_{ni}$$

Kde index *n* určuje neurón v sieti, *i* určuje váhu v neuróne, α predstavuje mieru učenia, x_{ni} veľkosť vstupu do daného neurónu a váhy. Vzťah $\frac{\partial f_n(e)}{\partial e}$ vyjadruje deriváciu aktivačnej funkcie príslušného neurónu vzhľadom k jeho vstupom.

Existujú dva základné prístupy k algoritmu backpropagation: prírastkový a dávkový. Pri prírastkovom učení sú hodnoty váh upravované po každej dvojici trénovacích vektorov. Pri dávkovom, sú sieti prezentované všetky dvojice z trénovacej množiny, gradienty spočítané pre každú dvojicu sú sumované a po predložení všetkých vzorov sú váhy upravené na základe tejto sumy. Všeobecne je možné povedať, že dávkové učenie potrebuje viac iterácií učiaceho algoritmu ku konvergencii, avšak je výhodné pre viaceré pokročilejšie algoritmy učenia.

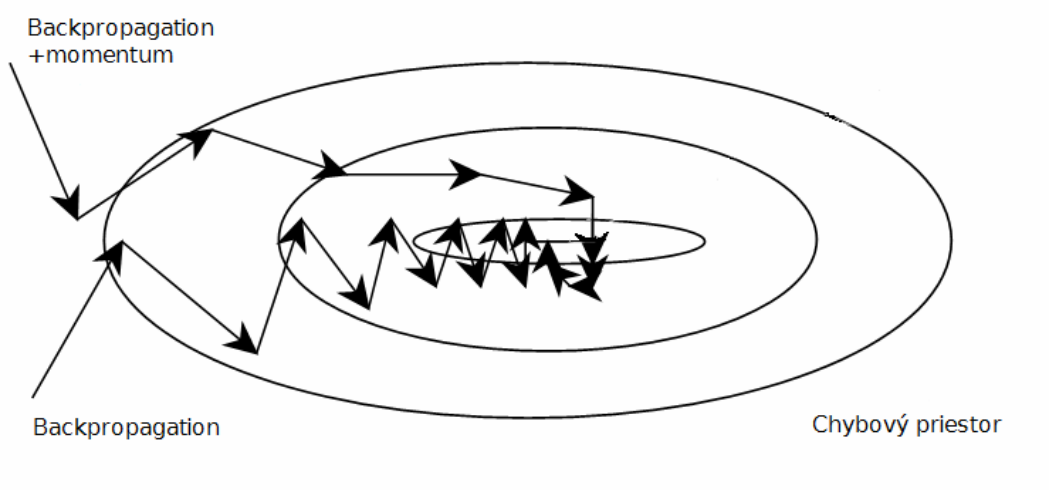
Pre detailné odvodenie algoritmu backpropagation viď. [2].

Pri použití miery učenia, ktorá je veľmi malá je väčšia pravdepodobnosť, že algoritmus dokáže nájsť minimum. Rýchlosť konvergencie je však príliš malá. Naopak pri použití príliš veľkých hodnôt riskujeme prekročenie minima, alebo osciláciu okolo neho. Najčastejšie používaná technika na prekonanie tohto problému je relatívne malá miera učenia a použitie parametra momentum (hybnosť). Ten k váhe pridáva časť prírastku váhy v minulom kroku učenia:

$$w_{ni}(t+1) = w_{ni}(t+1) + \alpha\delta_n \frac{\partial f_n(e)}{\partial e} x_{ni} + m\Delta w_{ni}(t)$$

Kde *m* predstavuje veľkosť parametra momentum a $\Delta w_{ni}(t)$ prírastok váhy v minulom kroku. Výsledkom tohto vylepšenia je, že váhy majú tendenciu sledovať smer váh v predchádzajúcich krokoch. To celkovo prispieva k väčším zmenám váh neurónov bez oscilácie. Efekt je možné vidieť na Obr. 23.

Nevýhodou je nutnosť ukladania prírastkov váh pre všetky váhy v sieti duplicitne, teda pre momentálny krok a pre predchádzajúci.



Obr. 23 Vplyv parametra momentum na algoritmus backpropagation

2.1.4 Premenlivá miera učenia

Pri štandardnom algoritme backpropagation je miera učenia konštantná počas celej doby učenia. Výsledky učenia sú tak silne závislé na správne nastavenej veľkosti tohto parametra. Pre praktické použitie nie je vhodné určovať mieru učenia pred samotným učením, naopak optimálna veľkosť miery učenia sa v priebehu učenia postupne mení, ako sa algoritmus pohybuje po chybovom priestore.

Z toho dôvodu je možné zlepšiť konvergenciu algoritmu backpropagation, ak je možné mieru učenia meniť počas učenia siete. Cieľom týchto zmien je vždy použiť čo najväčšie prírastky váh, pričom algoritmus učenia zostane stabilný (nevyskytujú sa oscilácie, zvyšuje sa chyba siete).

Po ukončení jedného kroku učenia je chyba po danom kroku porovnaná s chybou v predchádzajúcom kroku. Ak pomer novej a starej chyby prekročí určitú preddefinovanú hodnotu (napr. 1,04), miera učenia je zmenšená vynásobením číslom menším ako 1 (napr. 0,7). V opačnom prípade dôjde k zvýšeniu miery učenia, vynásobením číslom väčším ako 1 (napr. 1,05).

Týmto spôsobom je možné prispôsobovať mieru učenia momentálnemu stavu siete a udržiavať učiaci algoritmus stabilný.

2.1.5 Delta-Bar-Delta

Delta-Bar-Delta pravidlo bolo navrhnuté za účelom ďalšieho zrýchlenia a skvalitnenia učenia viacvrstvových sietí s dopredným šírením. Činnosť tohto algoritmu je založená na štyroch predpokladoch:

- Každá váha má svoju mieru učenia
- Každá miera učenia sa môže meniť v každej iterácii učenia
- Ak parciálna derivácia chybovej funkcie podľa konkrétnej váhy má rovnaké znamienko ako v predchádzajúcich iteráciach, je príslušná miera učenia zvýšená
- Ak parciálna derivácia chybovej funkcie podľa konkrétnej váhy v predchádzajúcich krokoch menila znamienko, je príslušná miera učenia znížená

Úpravy miery učenia sú obdobné ako v predchádzajúcom prípade (2.1.4). Algoritmus si musí ukladať miery učenia pre všetky váhy v sieti spolu s predchádzajúcimi hodnotami (alebo aspoň znamienkami) derivácií. Delta-Bar-Delta pravidlo sa spravidla využíva len pri dávkovom učení.

2.1.6 Quick propagation

Backpropagation a algoritmy od neho odvodené využívajú na učenie siete parciálne prvé derivácie chybovej funkcie. Pomocou týchto derivácií je možné použiť gradientnú metódu na zmenu váh siete. Ak uskutočníme nekonečné množstvo krokov v smere gradientu je dokázané, že učenie dosiahne lokálne minimum. Empiricky bolo zistené, že pre veľké množstvo problémov je toto lokálne minimum zároveň minimom globálnym, alebo aspoň „dostatočne dobré“ pre väčšinu potrieb.

V praxi je však nemysliteľné použiť nekonečné množstvo krokov učenia, preto je snaha zrýchliť schopnosť algoritmov hľadať minimá. Nanešťastie parciálne prvé derivácie, ktoré sú k dispozícií pri učení algoritmom backpropagation, dávajú len veľmi malú predstavu, aká je maximálna veľkosť kroku v chybovom priestore.

Ako riešenie tohto problému sa najčastejšie používajú dva prístupy. Prvým je snaha o dynamickú zmenu miery učenia založenú na predchádzajúcich výsledkoch (2.1.4, 2.1.5). Druhým prístupom je využitie druhej derivácie chybovej funkcie vzhľadom k váham siete. Pri znalosti druhej derivácie je možné použiť napr. Newtonovu metódu na úpravu váh.

Nanešťastie určenie druhej derivácie je výpočtovo náročný proces, preto sa používajú aproximácie tejto hodnoty.

Učiaci algoritmus quickpropagation sa snaží kombinovať oba prístupy. Jeho základ je totožný s dávkovým backpropagation, len s tým rozdielom, že ukladá parciálnu deriváciu z minulého kroku učenia pre každú váhu spolu s hodnotami upravujúcimi váhy v minulom kroku. Pri výpočte úprav váh algoritmus predpokladá dve vlastnosti chybového priestoru:

1. Závislosť chyby na váhe, pre každú váhu je konkávna parabola
2. Zmena váhy nezmení chybový priestor ostatných váh

Na základe týchto dvoch predpokladov je potom možné vypočítať úpravy váh. Pre každú váhu v sieti spočítame pravdepodobné umiestnenie minima paraboly pomocou vzťahu:

$$\Delta w(t) = \frac{S(t)}{S(t-1) - S(t)} \Delta w(t-1)$$

Kde *S(t)* a *S(t-1)* sú hodnoty prvých parciálnych derivácií chybovej funkcie vzhľadom na váhy siete v tomto a predchádzajúcom kroku. Táto hodnota je samozrejme len hrubým odhadom optimálnej hodnoty váhy, avšak pri postupnom aplikovaní je efektívnejšia ako backpropagation.

Ak by boli hodnoty parciálnych derivácií v dvoch po sebe nasledujúcich krokoch rovnaké a ich znamienka by boli rovnaké, veľkosť kroku by sa limitne blížila k nekonečnu. Obdobný problém by nastal, ak by bola hodnota derivácie väčšia ako v predchádzajúcom kroku, v tom prípade by krok bol v opačnom smere. Obe tieto možnosti sú pri učení siete škodlivé, na ich odstránenie bol do algoritmu zavedený maximálny parameter maximálneho prírastku. Ten určuje maximálny násobok zmeny váhy oproti predchádzajúcemu kroku, jeho obvyklá hodnota je 1,75. Ak by bol tento parameter prekročený, ako zmena váh sa zoberie hodnota parametra krát zmena váh v predchádzajúcom kroku.

Keďže sú v každom kroku učenia potrebné informácie z predchádzajúceho kroku, je nutné proces učenia v prvej iterácií pozmeniť. Pre prvú iteráciu je použitý dávkový backpropagation na získanie počiatočných prvých derivácií a prírastkov váh.

2.1.7 Resilient propagation

Resilient propagation je algoritmus na učenie viacvrstvových neurónových sietí s dopredným šírením. Je to jeden z najlepších algoritmov využívajúcich len prvú deriváciu chybovej funkcie.

Aktivačné funkcie neurónov sú spravidla sigmoidné funkcie. Ich charakteristickým znakom je, že komprimujú neobmedzený vstup na obmedzený výstupný interval. Derivácia týchto funkcií sa blíži k nule, keď sa veľkosť vstupu zvyšuje. Táto vlastnosť môže predstavovať problém pri učení pomocou gradientných metód, pretože malé hodnoty gradientu znamenajú malé prírastky váh, aj keď sú váhy vzdialené od svojej optimálnej hodnoty.

Úlohou resilient propagation je eliminovať nežiadúce vplyvy veľkosti derivácií. Pre určenie prírastku váh sú použité len znamienka derivácií, veľkosť derivácie nemá na prírastky váh žiadnen vplyv. Veľkosť zmeny váh je určovaná samostatnou hodnotou Δ, pričom každá váha v sieti má svoju vlastnú. Veľkosť Δ sa mení na základe vývoja chybovej funkcie siete, obdobne ako miera učenia pri algoritme Delta-Bar-Delta (2.1.5):

$$\Delta_{ij}(t) = \begin{cases} \eta^{+} \cdot \Delta_{ij}(t-1) & ,if \ \frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) > 0 \\ \eta^{-} \cdot \Delta_{ij}(t-1) & ,if \ \frac{\partial E}{\partial w_{ij}}(t-1) \cdot \frac{\partial E}{\partial w_{ij}}(t) < 0 \\ \Delta_{ij}(t-1) & ,else \end{cases}$$

Kde konštanty η^{-} a η^{+} predstavujú zvyšovanie a znižovanie parametra pre zmenu váh. Hodnoty týchto konštánt sú v medziach: $0 < \eta^{-} < 1 < \eta^{+}$

Ak došlo k zmene znamienka derivácie znamená to, že algoritmus prekročil lokálny extrém, je preto nutné veľkosť kroku zmenšiť. V opačnom prípade, ak je znamienko derivácie rovnaké ako v predchádzajúcom kroku, smeruje chybová funkcia k lokálnemu minimu, teda je možné zväčšiť veľkosť kroku. Ak bola niektorá z dvojice derivácií nulová, veľkosť kroku sa nemení. Zmena váh je zhora a zdola ohraničená parametrami Δ_{max} a Δ_{min}, tie zabraňujú príliš veľkým resp. príliš malým prírastkom váh.

Základom pre resilient propagation je dávkový algoritmus backpropagation, rozdiel je len vo výpočte prírastkov váh. Kľúčová časť resilient propagation v pseudokóde:

18 foreach w_{ij} do

19 if $\frac{\partial E}{\partial w_{ij}}(t-1)\cdot\frac{\partial E}{\partial w_{ij}}(t)>0$ then

20 $\Delta_{ij}(t)=\min(\Delta_{ij}(t-1)\cdot\eta^{+},\Delta_{\max})$

21 $\Delta w_{ij}(t)=-sign(\frac{\partial E}{\partial w_{ij}}(t))\cdot\Delta_{ij}(t)$

22 $w_{ij}(t+1)=w_{ij}(t)+\Delta w_{ij}(t)$

23 elseif $\frac{\partial E}{\partial w_{ij}}(t-1)\cdot\frac{\partial E}{\partial w_{ij}}(t)<0$ then

24 $\Delta_{ij}(t)=\max(\Delta_{ij}(t-1)\cdot\eta^{-},\Delta_{\min})$

25 $w_{ij}(t+1)=w_{ij}(t)-\Delta w_{ij}(t-1)$

26 $\frac{\partial E}{\partial w_{ij}}(t)=0$

27 elseif $\frac{\partial E}{\partial w_{ij}}(t-1)\cdot\frac{\partial E}{\partial w_{ij}}(t)=0$ then

28 $\Delta w_{ij}(t)=-sign(\frac{\partial E}{\partial w_{ij}}(t))\cdot\Delta_{ij}(t)$

29 $w_{ij}(t+1)=w_{ij}(t)+\Delta w_{ij}(t)$

30 end;

Funkcia *sign()* nadobúda hodnotu +1 ak je jej argument väčší ako nula, -1 ak je argument menší ako nula, v iných prípadoch je jej hodnota 0. V prípade zmeny znamienka derivácie je prevedená „skok späť“, teda konkrétna váha je vrátená na svoju hodnotu v predchádzajúcom kroku. Zároveň je nastavená hodnota predchádzajúcej derivácie na 0, aby sa predišlo dvojnásobnej penalizácii parametra Δ.

Tento algoritmus sa často označuje ako Rprop^{+}, kde index $^{+}$ označuje použitie sledovania váh z predchádzajúcich krokov učenia.

Vylepšenie algoritmu Rprop^{+}, ktoré mení spôsob postihu za zmenu znamienka derivácie, bolo predstavené v [20]. Toto vylepšenie je postavené na predpoklade, že aj keď derivácia zmenila znamienko, teda prekročila lokálne minimum, nemusí táto zmena znamenať zhoršenie chybovej funkcie siete. Takže rozhodnutie zvrátiť zmenu váhy nie je vždy nutné, naopak môže byť na škodu, ak došlo k zlepšeniu celkovej chyby siete. Z tohto dôvodu je

návrat na staré hodnoty váh podmienený zhoršením chybovej funkcie siete. Takto upravený algoritmus sa označuje ako iRprop$^+$ (improved resilient propagation so sledovaním váh).

Zmena oproti Rprop$^+$ je len v krokoch po zmene znamienka derivácie, tieto kroky sú zapísané v pseudokóde:

```
31 elseif ∂E/∂w_ij(t-1) · ∂E/∂w_ij(t) < 0 then
32   Δ_ij(t) = max(Δ_ij(t-1) · η⁻, Δ_min)
33 if E(t) > E(t-1) then
34       w_ij(t+1) = w_ij(t) - Δw_ij(t-1)
35   ∂E/∂w_ij(t) = 0
```

E(t) a E(t-1) predstavujú chybu siete v tomto a predchádzajúcom kroku. Teda oproti Rprop$^+$ je nutné, aby si algoritmus ukladal hodnoty chyby v predchádzajúcom kroku.

Ďalším vylepšením je algoritmus iRprop$^-$, ktorý už nepoužíva metódu sledovania váh. V tomto prípade nie je nutné ukladať predchádzajúce zmeny váh, pretože zmena váh sa vykonáva v každom prípade.

Algoritmus iRprop$^-$ v pseudokóde:

```
36 foreach w_ij do
37     if ∂E/∂w_ij(t-1) · ∂E/∂w_ij(t) > 0 then
38         Δ_ij(t) = min(Δ_ij(t-1) · η⁺, Δ_max)
39     elseif ∂E/∂w_ij(t-1) · ∂E/∂w_ij(t) < 0 then
40         Δ_ij(t) = max(Δ_ij(t-1) · η⁻, Δ_min)
41     w_ij(t+1) = w_ij(t) - sign(∂E/∂w_ij(t)) · Δ_ij(t)
42 end;
```

2.1.8 Diferenciálna evolúcia

V roku 1995 predstavili Ken Price a Rainer Storm diferenciálnu evolúciu. Je to algoritmus určený na optimalizáciu mnohorozmerných funkcií a patrí do kategórie evolučných stratégií. Pri učení neurónovej siete je optimalizovanou (účelovou) funkciou chybová funkcia siete, pričom menenými parametrami sú váhy jednotlivých neurónov siete.

Na činnosť evolúcie majú vplyv 4 parametre:

- NP – Veľkosť populácie, $NP > 4$
- F – Mutačná konštanta, $F \in [0, 2]$
- CR – Prah kríženia, $CR \in [0,1]$
- G – Počet generácií (evolučných cyklov), $G > 0$

Diferenciálna evolúcia sa v každej generácii snaží vyšľachtiť čo najlepšieho jedinca, pričom vhodnosť jedinca určuje chybová funkcia siete, pre váhy, ktoré sú obsiahnuté v danom jedincovi.

Tvorba nových jedincov je vykonávaná pomocou procesu mutácie a kríženia. Algoritmus postupne vyberá jednotlivých jedincov z populácie, ku každému jedincovi náhodne vyberie dvoch ďalších, rôznych jedincov a jedinca s najmenšou hodnotou chybovej funkcie. Dvaja náhodne vybraní jedinci sa od seba odčítajú, tým sa získa diferenčný vektor. Diferenčný vektor sa vynásobí mutačnou konštantou, ktorá predstavuje mutáciu, takto získaný vektor sa nazýva váhovaný diferenčný vektor. Ten sa pričíta k najlepšiemu jedincovi z populácie, vznikne šumový vektor. Pripraví sa prázdny, skúšobný vektor, ktorý svojou veľkosťou zodpovedá veľkosti jedinca v populácii. Pre každú pozíciu v skúšobnom vektore sa generuje náhodné číslo v intervale [0, 1] a porovnáva sa s prahom kríženia. Ak je náhodné číslo väčšie ako prah kríženia, na príslušnú pozíciu v skúšobnom vektore je vložená hodnota z príslušnej pozície vo vybranom jedincovi, v opačnom prípade z príslušnej pozície v šumovom vektore. Pre takto získaný skúšobný vektor je spočítaná hodnota chybovej funkcie siete.

Do novej populácie pre ďalšiu generáciu sa vyberá z dvojice aktívny jedinec – skúšobný vektor ten, ktorý má nižšiu hodnotu chybovej funkcie.

Algoritmus diferenciálnej evolúcie je najcitlivejší na parameter *F* (mutačná konštanta)[14]. Ak je hodnota tohto parametra príliš veľká, najlepšie riešenia sú ľahko zničené a schopnosť nájsť globálne riešenie je malá. Pri príliš malých hodnotách parametra *F* je diverzita populácie malá, a môže spôsobiť uviaznutie algoritmu v lokálnych minimách. Preto je pri učení neurónovej siete použitá premenlivá hodnota mutačnej konštanty, tá je na začiatku učenia veľká, aby zabezpečila diverzitu populácie, no postupne sa znižuje, aby sa zabránilo strate dobrých riešení. Miera znižovania mutačnej konštanty je daná rovnicou:

$$F = \beta\left(e^{\lambda} - 1\right)$$

Kde:

$$\beta \in [0.2, 0.6]$$

$$\lambda = \frac{G_{\max}}{G_{\max} + G}$$

G_{max} predstavuje maximálny počet generácií a G momentálnu generáciu.

Pseudokód popisujúci činnosť diferenciálnej evolúcie:

```
43 Vytvorenie počiatočnej populácie
44 Výpočet chybových funkcií pre jednotlivých jedincov
45 for i<G begin
46    for j<NP begin
47       Výber j-teho jedinca
48       Vytvorenie skúšobného jedinca pomocou mutácie a kríženia
49       Zápis lepšieho z dvojice jedincov do novej populácie
50    end;
51    Nahradenie starej populácie novou
52    Úprava veľkosti mutačnej konštanty
53    Ak je chyba nulová, alebo dostatočne malá ukonči učenie
54 end;
```

2.1.9 SOMA

SOMA je pomerne nový algoritmus určený na optimalizáciu mnohorozmerných funkcií. Samotný názov je skratkou zo: „Samo organizujúci sa migračný algoritmus“. Základné princípy činnosti algoritmu sú založené na správaní sa inteligentných jedincov, ktorí kooperujú na riešení spoločnej úlohy. Vlastnosť samoorganizácie vyplýva z faktu, že sa jedinci pri hľadaní najlepšieho výsledku navzájom ovplyvňujú. Algoritmus je možné

zaradiť do skupiny tzv. evolučných algoritmov, aj keď v tomto prípade nedochádza k tvorbe nových potomkov, len k ich migrácii. Z toho dôvodu sa jedna iterácia algoritmu nazýva migračné kolo.

Činnosť algoritmu SOMA je ovplyvňovaná skupinou parametrov:

- PathLength – Maximálna dĺžka cesty, akú prejde aktívny jedinec v danom migračnom kole
- Step – Veľkosť kroku pri prehľadávaní priestoru
- PRT – Perturbácia
- NP – Veľkosť populácie

Rovnako ako v prípade diferenciálnej evolúcie je tvorba nových jedincov realizovaná pomocou mutácie a kríženia.

Mutácia je premenovaná na perturbanciu. Dôvodom je skutočnosť, že mutovaný je spôsob, akým sa pohybujú jedinci priestorom možných riešení. Veľkosť tejto mutácie závisí na nastavení parametra PRT. Pre každého jedinca sa pred každým krokom generuje perturbačný vektor, ten má rovnaký rozmer ako jedinci v populácii. Následne sú generované náhodné čísla a porovnávané s parametrom PRT, ak je náhodné číslo menšie je na príslušnú pozíciu v perturbačnom vektore priradená hodnota 1, v ostatných prípadoch hodnota 0.

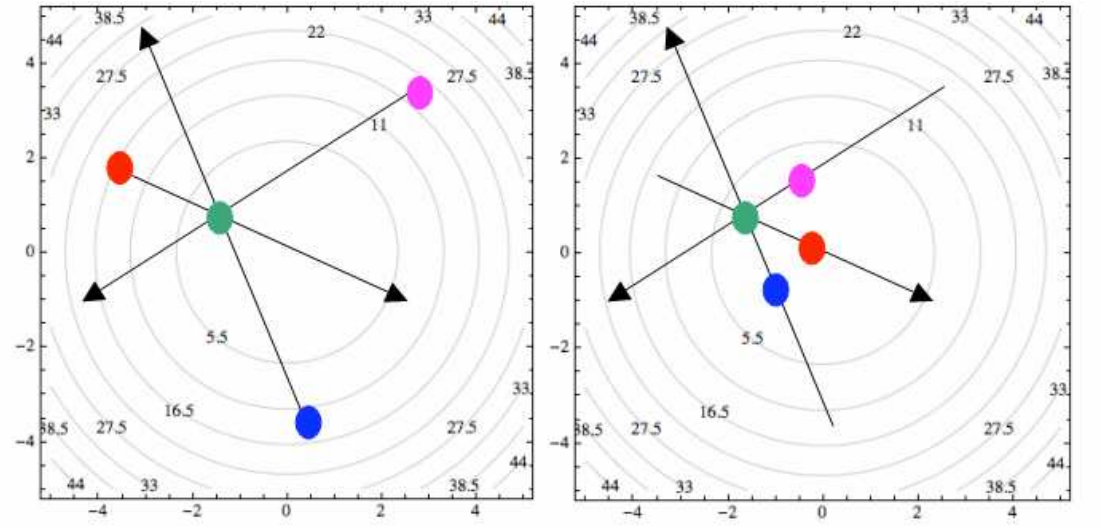
Kríženie predstavuje tvorbu nového potomstva, v tomto prípade sa jedná o pohyb jedincov po diskrétnych krokoch v priestore pomocou dvoch už existujúcich jedincov (rodičov). Počas pohybu si každý jedinec pamätá súradnice, kde našiel najlepšie riešenie, toto riešenie postupuje do ďalšieho migračného kola. Pohyb jedincov sa riadi podľa rovnice:

$$x_{i,j}^{ML+1} = x_{i,j,start}^{ML} + (x_{L,j}^{ML} - x_{i,j,start}^{ML}) \cdot t \cdot PRTVector_j$$

Kde $x_{i,j,start}^{ML}$ je pozícia aktívneho jedinca na počiatku, $x_{L,j}^{ML}$ pozícia jedinca s najlepšou hodnotou účelovej funkcie (vedúci jedinec, leader), $PRTVector_j$ perturbačný vektor a t je hodnota kroku z intervalu [0, PathLength].

Perturbačný vektor ovplyvňuje pohyb jedinca tak, že ak má všetky prvky rovné 1, tak sa aktívny jedinec pohybuje priamo k vedúcemu jedincovi. Ak je však niektorý prvok rovný

0, potom sa daná súradnica nemení a cesta priestorom je odklonená z priameho smeru. To zaručuje, že je prehľadávaná väčšia časť priestoru [8].



Obr. 24 Princíp činnosti SOMA, pred migráciou a po nej

2.2 Učenie bez učiteľa

Na rozdiel od učenia s učiteľom nie sú sieti predkladané výstupné vzory, podľa ktorých by upravovala svoje výstupy, ale sieť samotná rozhoduje, ako organizovať predkladané vstupné dáta.

2.2.1 Hopfieldovo učenie

Učenie Hopfieldovej siete je jednorazový proces. Jeho účelom je znížiť energiu tých stavov (vstupných vzorov), ktoré sa má sieť naučiť. Sieť môže byť následne použitá na obnovenie uloženej hodnoty na základe predloženia porušeného (zašumeného) vstupu.

Po naučení sa na určitú skupinu vstupných vzorov je sieť automaticky schopná rozpoznávať aj ich negatívy, teda opačné hodnoty. Kapacita siete však postačuje len na určitý počet vzorov, udáva sa kapacita a vzor na 15 neurónov [29]. Po prekročení množstva uložených vzorov nastávajú dva problémy:

1. Uložené vzory sa stávajú nestabilnými
2. Vznikajú chybné stabilné stavy (stavy, ktoré nezodpovedajú vstupným vzorom)

Sieť je teda správne naučená, ak je hodnota energetickej funkcie vstupných vzorov lokálnym minimom. Aby bolo možné získať naučené vzory pri použití poškodených vstupov, je výhodné, aby mali vzory medzi sebou čo najväčšiu Hammingovu vzdialenosť.

Učenie Hopfieldovej siete v pseudokóde:

```
55 for i<počet neurónov begin
56     for j<počet neurónov begin
57         wij=0
58         if i ≠ j then
59             foreach p do
60                 wij=pi*pj
61             end;
62     end;
63 end;
```

Kde indexy *i* a *j* predstavujú pozíciu neurónu v sieti, w_{ij} je váha spojenia medzi neurónmi *i* a *j*, p je vstupný vektor z trénovacej množiny.

2.2.2 SOM učenie

Cieľom učenia SOM sietí je zmena váh neurónov pracovnej vrstvy tak, aby rôzne časti vrstvy reagovali podobne na určité vstupné vzory. Učenie je čiastočne motivované spracovaním zmyslových informácií v rôznych častiach mozgovej kôry v ľudskom mozgu.

Stručný popis učenia SOM v pseudokóde:

```
64 Inicializácia váh siete
65 for i<C begin
66     foreach x do
67         Nájdi neurón, s najmenšou vzdialenosťou od x
68         Spočítaj veľkosť susedstva víťazného neurónu
69         Uprav váhy každého neurónu v susedstve, v závislosti na jeho
           vzdialenosti
70     end;
71     Uprav veľkosť susedstva
72 end;
```

Premenná *C* predstavuje počet iterácií učenia, premenná *x* jeden vstupný vektor z trénovacej množiny.

Algoritmus učenia SOM sietí patrí do je tzv. konkurenčné učenie. To znamená, že vždy jeden neurón je určený ako víťaz pre daný vstupný vektor, a jeho váhy sú upravené najviac.

Víťazný neurón je v prípade Kohonenových máp označovaný ako BMU (best matching unit – najviac vyhovujúca jednotka). Na určenie BMU slúži vzdialenosť váh neurónu od vstupného vektora (viď. 1.5). Je teda nutné spočítať vzdialenosti pre všetky neuróny v pracovnej vrstve a z nich vybrať najmenšiu. Následne sú váhy všetkých neurónov v susedstve BMU upravené v závislosti na ich vzdialenosti. Pre úspešné naučenie siete je potrebné, aby sa veľkosť susedstva postupne zmenšovala. Zmenšovanie polomeru susedstva prebieha podľa vzťahu:

$$\sigma(t) = \sigma_0 \cdot e^{\left(-\frac{t}{\lambda}\right)}$$

Kde σ_0 predstavuje polomer susedstva na počiatku učenia, *t* je aktuálna iterácia, λ je časová konštanta:

$$\lambda = \frac{NC}{Ln(\sigma_0)}$$

Kde *NC* je celkový počet iterácií učenia a funkcia *Ln()* vracia hodnotu prirodzeného logaritmu.

Úprava jednotlivých váh neurónov v susedstve BMU prebieha podľa vzťahu:

$$w(t+1) = w(t) + \Theta(t) \cdot L(t) \cdot (x(t) - w(t))$$

Kde *t* je iterácia učenia, *x* vstupný vektor, *w* vektor váh, *L* miera učenia a premenná Θ vyjadruje mieru zmeny váh v závislosti na vzdialenosti od BMU:

$$\Theta(t) = e^{\left(-\frac{d^2}{2\cdot\sigma^2(t)}\right)}$$

$$L(t) = L_0 \cdot e^{\left(-\frac{t}{\lambda}\right)}$$

Na začiatku učenia sú ovplyvňované veľké časti pracovnej vrstvy, keďže polomer susedstva je pomerne veľký (obyčajne polovica väčšieho z rozmerov siete), postupne sa počet ovplyvňovaných neurónov zmenšuje. Taktiež veľkosť úprav váh siete sa s postupom učenia zmenšuje, pretože miera učenia klesá. Týmto spôsobom je možné vytvárať zhluky dát, pričom podobné dáta budú umiestnené blízko seba.

2.2.3 Elastické učenie

Koncept elastického učenia po prvýkrát predstavili Richard Durbin a David Willshaw ako metódu pre hľadanie optimálnych ciest pri riešení problému obchodného cestujúceho. Algoritmus je veľmi podobný učeniu SOM siete a využíva SOM sieť, ktorej pracovná vrstva je v tvare kruhu. Cieľom je namapovať neuróny na mestá umiestnené v dvojrozmernom priestore. Obrazne možno proces učenia prirovnať k umiestňovaniu elastickej pásky na skupinu ihiel, ktoré predstavujú pozíciu miest.

Na počiatku učenia sú váhy siete inicializované tak, aby v priestore vstupných hodnôt tvorili kruh, s rovnakými vzdialenosťami medzi susednými neurónmi. Neuróny sú postupne „priťahované“ mestami a zároveň svojimi susedmi. Tieto dve sily umožňujú, aby cesta viedla cez všetky mestá a zároveň bola dostatočne krátka. Miera, s akou mestá a susedia pôsobia na neuróny, je závislá od vzdialenosti, teda vzdialenejšie mestá budú mať na neurón menší vplyv.

Použitie elastického učenia je možné rozšíriť aj na praktickejšie oblasti ako je problém obchodného cestujúceho. Napríklad plánovanie, logistiku, hľadanie najkratších ciest pri výrobe mikročipov, sekvencovanie génov.

Pre detailné odvodenie algoritmu viď. [26]

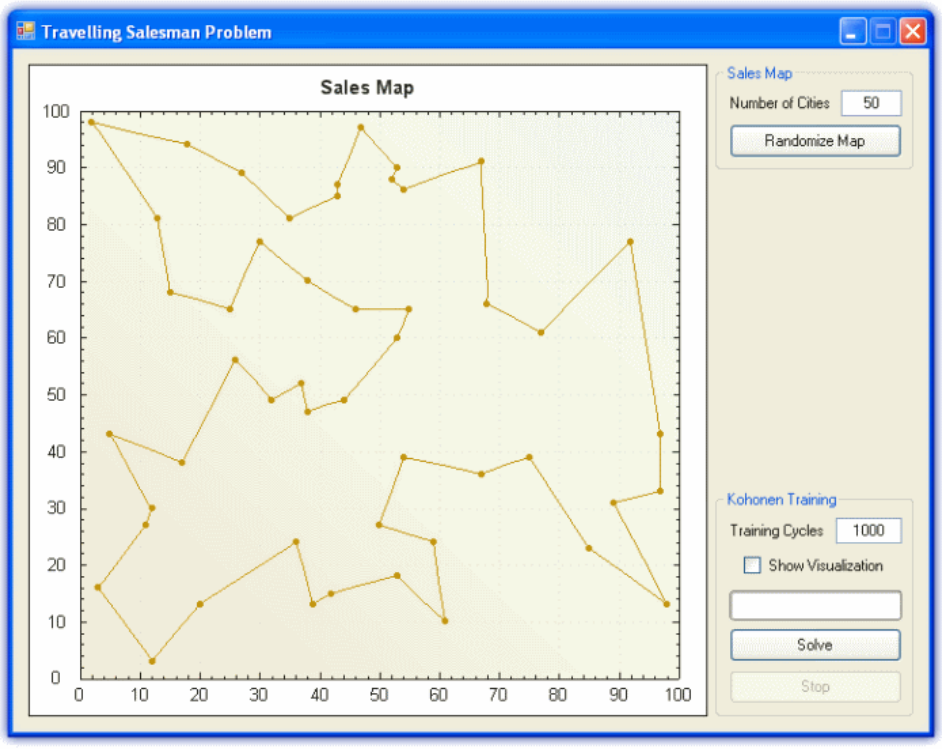
3 POUŽITIE NEURÓNOVÝCH SIETÍ

3.1 Existujúce implementácie v C#

Knižníc a programov pracujúcich s neurónovými sieťami existuje veľké množstvo. Pre túto prácu sú však relevantné len tie, ktoré sú napísané v programovacom jazyku C# a sú voľne dostupné.

3.1.1 NeuronDotNet

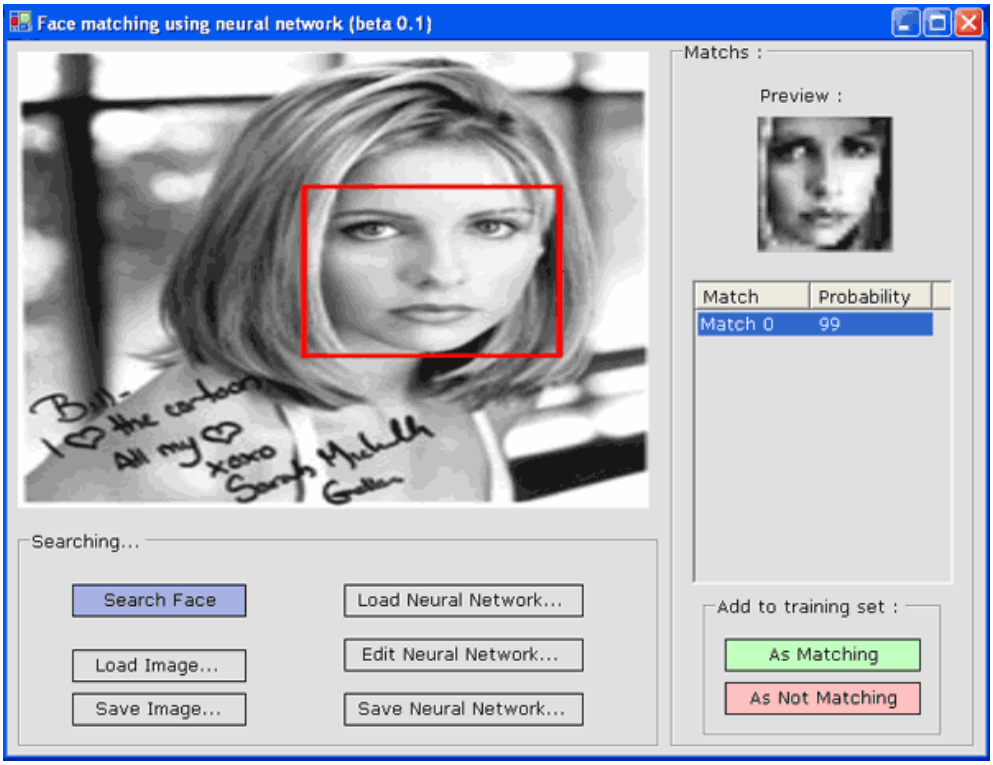
NeuronDotNet vznikol za účelom vytvorenia open source knižnice pre prácu s neurónovými sieťami v prostredí .NET. Vývoj sa začal v roku 2006. S pomocou tejto knižnice je možné vytvárať siete s algoritmom backpropagation a Kohonenove siete. Takto vytvorené siete je možné ďalej používať v rozličných vytváraných aplikáciach. Stránka projektu poskytuje kompletný používateľský manuál, spolu so základnou teóriou pre tvorbu neurónových sietí. Autori poskytli aj niekoľko ukážkových aplikácií, ktoré demonštrujú použitie oboch typov sietí.



Obr. 25 Použitie Kohonenovej mapy na riešenie problému obchodného cestujúceho

3.1.2 C# Neural network library

Tento projekt je dielom Francka Fluereyho. Jeho cieľom je vytvoriť software pre rozoznávanie ľudských tvárí v obraze. S pomocou tejto knižnice je možné vytvoriť sieť, ktorá obsahuje algoritmus backpropagation a neuróny môžu mať jednu z 3 aktivačných funkcii (lineárna, sigmoida, gaussova). Hotové siete je možné ďalej zlepšovať pomocou genetických algoritmov. V ukážkových aplikáciach sú načrtnuté možnosti knižnice. Program obsahuje aj jednoduchý editačný nástroj váh a možnosť ukladať/načítavať siete.

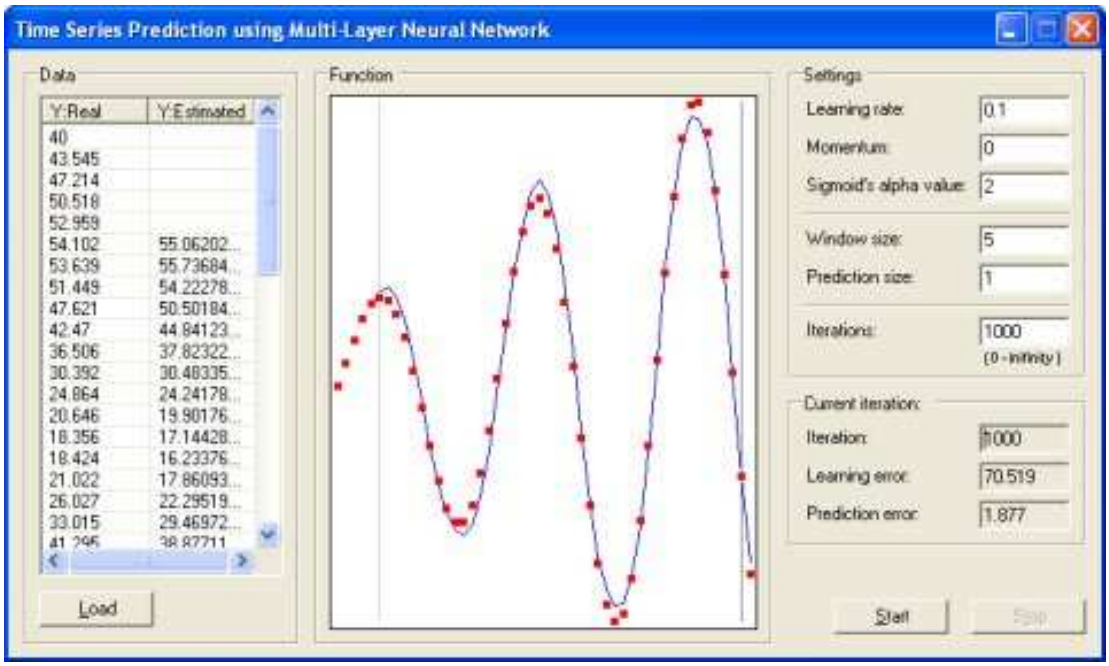


Obr. 26 Demo aplikácia pre detekciu tváre v obraze.

3.1.3 Neural Networks on C#

V roku 2006 vytvoril Andrew Kirillov ďalšiu knižnicu pre prácu s neurónovými sieťami. Jeho prístup bol odlišný oproti predchádzajúcim prípadom. Nevytvoril hotové typy sietí (ako napríklad Kohonenovu, perceptron atď.), v knižnici sa nachádzajú triedy popisujúce jednotlivé prvky siete (neuróny, metódu učenia, aktivačné funkcie, vrstvy a sieť ako celok). Z týchto modulárnych prvkov je ďalej možné vytvoriť ľubovoľnú sieť. Nevýhodou tohto postupu sú počiatočné problémy pri vytváraní jednoduchých typov sietí. To je však vyvážené možnosťou vytvárať nové topológie a jednoduchosťou implementácie odlišných

učiacich algoritmov. Autor ku knižnici dodáva aj kompletnú dokumentáciu. Na domovskej stránke projektu je možné nájsť aj niekoľko demo aplikácií.



Obr. 27 Príklad aplikácie používajúcej „Neural Networks on C#“ pre predikciu časových postupností

3.2 Použitie v súčasnosti

Množstvo neurónových sietí, ktoré sú v dnešnej dobe navrhované sú štatisticky presné, ale ich použitie nie je možné v oblastiach, kde užívatelia predpokladajú, že počítač vyrieši ich problém s absolútnou presnosťou. Bežná presnosť dnešných sietí sa pohybuje v rozmedzí 85-90% [30], nanešťastie veľmi málo aplikácií toleruje takúto mieru nepresnosti.

V súčasnosti sa stále pracuje na zlepšovaní presnosti sietí, avšak aj v tomto stave sú umelé neurónové siete využívané v niektorých odvetviach, kde je 90 percentná presnosť zlepšením oproti súčasnému stavu.

Prednosti využitia neurónových sietí spočívajú hlavne v ich schopnosti identifikovať funkcie z pozorovania a následne tieto funkcie použiť. Táto vlastnosť je zvlášť užitočná v aplikáciach, kde je komplexnosť dát príliš veľká, aby boli funkčné vzťahy zisťované ľuďmi. Aj keď je neurónová sieť schopná vykonávať požadované činnosti, je ťažké zistiť, ako sa daná sieť naučila a dôvody, ktoré stoja za jej výsledkami. Najčastejšie používaným prístupom na zisťovanie dôvodov za správaním siete je sledovanie vplyvu vstupov na jednotlivé neuróny a následne výstupy siete.

Aplikácie neurónových sietí vo väčšine spadajú do jednej z kategórii:

- Aproximácia funkcií, regresná analýza
- Predikcia časových postupností a modelovanie
- Klasifikácia
- Rozpoznávanie vzorov a zmien
- Spracovanie, vyhľadávanie a kompresia dát

II. PRAKTICKÁ ČÁST

4 KNIŽNICA NEURAL NETWORKS

Hlavným cieľom tejto diplomovej práce bolo vytvorenie knižnice v jazyku C# na prácu s neurónovými sieťami. Pomocou tejto knižnice je možná plnohodnotná práca s neurónovými sieťami, teda vytváranie sietí s rôznou topológiou, ich učenie a následné použitie. Výsledná knižnica bola pomenovaná „Neural networks“.

Pri návrhu knižnice bol kladený dôraz na objektový prístup a modulárnosť. To znamená, že triedy popisujúce rovnakú oblasť (napr. aktivačné funkcie) sú plne zameniteľné. Taktiež v prípade budúceho rozširovania činnosti knižnice stačí len vytvoriť triedy odvodené z príslušných vzorových tried. Funkcie, ktorých činnosť je definovaná až v odvodených triedach sa nazývajú abstraktné, v C# je ich možno rozpoznať podľa kľúčového slova *abstract.* Trieda, v ktorej sú deklarácie týchto funkcií, je taktiež označená ako abstraktná.

Funkčnosť celej knižnice je rozdelená do piatich funkčných celkov, tzv. namespace, alebo menných priestorov. Nasledujúce kapitoly sa budú zaoberať ich popisom, popisom jednotlivých tried, možnosťami nastavenia a príkladmi použitia.

4.1 Namespace functions

Tento namespace obsahuje pomocné a podporné funkcie potrebné pre vnútorné fungovanie modelov neurónov, ako aj funkcie užitočné pri získavaní informácií o sieti. Pre využitie týchto funkcií je nutné do programu pridať riadok:

```
73 using NeuralNetworks.Functions;
```

4.1.1 Aktivačné funkcie

Všetky triedy, ktoré reprezentujú aktivačné funkcie neurónov s dopredným šírením sú odvodené od abstraktnej triedy *activationFunction.* Tá deklaruje prototypy funkcií, ktoré musia implementovať triedy od nej odvodené:

```
74 public abstract double output(double arg, double threshold)
```

Funkcia vracia hodnotu aktivačnej funkcie pre danú hodnotu vstupu (*arg*) a prahu (*threshold*). Vstup do funkcie je suma ováhovaných vstupov neurónu.

```
75 public abstract double derivation(double arg)
```

Metóda pre výpočet derivácie aktivačnej funkcie. Argumentom funkcie je suma ováhovaných vstupov do neurónu. V prípade, že nie je možné určiť deriváciu funkcie (napr. skoková funkcia) je vyvolaná výnimka.

```
76 public virtual double derivationFromOutput(double arg)
```

Pre niektoré aktivačné funkcie (napr. sigmoida, hyperbolický tangens) je možné spočítať hodnotu derivácie z jej hodnoty. Tento spôsob je výpočtovo menej náročný, preto sa používa, ak je možné.

```
77 public double Steepness
```

Tento verejne prístupný člen umožňuje získať, alebo nastaviť strmosť aktivačnej funkcie.

Zoznam všetkých tried implementujúcich aktivačné funkcie je v Tab. 3. Podrobnejší popis jednotlivých aktivačných funkcií je v kapitole 1.4.

Tab. 3 Triedy aktivačných funkcií

Trieda	**Aktivačná funkcia**
linearActivationFunction	Lineárna funkcia
perceptronActivationFunction	Skoková funkcia
bipolarPerceptronActivationFunction	Bipolárna skoková funkcia
sigmoidActivationFunction	Sigmoida
bipolarSigmoidActivationFunction	Bipolárna sigmoida
hyperbolicTangensActivationFunction	Hyperbolický tangens
gaussianActivationFunction	Gaussova funkcia
elliotActivationFunction	Elliotova funkcia
bipolarElliotActivationFunction	Bipolárna elliotova funkcia

4.1.2 Vzdialenostné funkcie

Pre činnosť neurónov v Kohonenovej sieti sú potrebné vzdialenostné funkcie. Základnou triedou, od ktorej sú odvodené ostatné, je *distanceFunction.* Tá definuje len jednu funkciu, ktorú musia implementovať odvodené triedy:

```
78 public abstract double distance(double[] weights, double[] sample)
```

Funkcia slúži na výpočet vzdialenosti medzi dvoma vektormi, vektorom váh (*weights*) a vstupným vektorom (*sample*). Návratovou hodnotou je samotná vzdialenosť vektorov.

Triedy vzdialenostných funkcií sú uvedené v Tab. 4, ich podrobnejší popis je možné nájsť v kapitole 1.5.

Tab. 4 Triedy vzdialenostných funkcií

Trieda	Vzdialenostná funkcia
euclideanDistance	Euklidova vzdialenosť
manhattanDistance	Manhattan vzdialenosť

4.1.3 Transformačné funkcie

Transformačné funkcie sú použité v triede vstupnej vrstvy (4.3.2) na transformáciu vstupného vektoru do požadovaného intervalu.

Každá transformačná funkcia musí byť odvodená od triedy *transformationFunction*, a implementovať funkciu:

```
79 public abstract double[] transform(double[] inputs)
```

Táto funkcia vykonáva transformáciu hodnôt. Argument funkcie je vstupný vektor, výstupom je vektor upravených vstupných hodnôt .

Implementovanú transformačnú funkciu reprezentuje trieda *linearTransformation.* Tá vykonáva lineárnu transformáciu hodnôt. Transformácia prebieha podľa vzorca:

$$y = factor \cdot x$$

Teda výstupy sú lineárnou kombináciou vstupov. Veľkosť parametra *factor* je možné získať, alebo zmeniť pomocou prístupovej metódy:

```
80 public double Factor
```

4.1.4 Trieda helper

Táto trieda obsahuje statické funkcie, ktoré sa priamo nepodieľajú na učení a vybavovaní siete, ale zjednodušujú prácu so sieťou a dátami.

Implementované funkcie sú:

```
81 public static double[] shuffle(double[] ar)
82 public static double[][] shuffle(double[][] ar)
```

Tieto dve funkcie slúžia na zamiešanie poradia prvkov v poli. Ak je argumentom matica, mení sa poradie riadkov matice (vektorov), nie však poradie prvkov vo vektore. Návratovou hodnotou je vektor s náhodne zmeneným poradím prvkov.

```
83 public static void shuffle(ref double[][] ar1, ref double[][] ar2)
```

Funkcia je podobná ako predchádzajúce dve, s tým rozdielom, že nemá návratovú hodnotu, ale výsledok ukladá priamo do vstupných argumentov (pomocou parametra *ref*). Účelom je zamiešať poradie riadkov dvoch matíc, ale tak, aby relatívne poradie medzi týmito maticami zostalo zachované. Je užitočná pri zmene poradia predkladania učiacich vzorov pri prírastkovom učení s učiteľom.

```
84 public static double[][][] getWeights(network network)
```

Pre zjednodušenie práce s váhami siete je pomocou tejto funkcie možné získať všetky váhy v trojrozmernej matici. Prvý rozmer reprezentuje index vrstvy v sieti, kde index nula má prvá skrytá vrstva (vstupná vrstva nemá váhy), druhý rozmer je index neurónu vo vrstve a tretí rozmer je index váhy v neuróne. Argumentom funkcie je sieť, ktorej váhy sa majú získať.

```
85 public static void setWeights(double[][][] weights,
   feedForwardNetwork network)
86 public static void setWeights(double[][][] weights, positionNetwork
   network)
```

V súčinnosti s predchádzajúcou funkciou pracuje funkcia na nastavenie váh. Tá má dva argumenty, maticu váh siete a samotnú sieť. Preťažené funkcie fungujú pre siete s dopredným šírením a Kohonenove mapy.

4.2 Namespace neurons

Menný priestor *neurons* obsahuje triedy, ktoré predstavujú neuróny v sieti. Keďže jednotlivé triedy sú rozsiahlejšie ako v predchádzajúcich prípadoch, budú v tejto kapitole uvedené len verejne prístupné metódy, deklarácie abstraktných a virtuálnych funkcií a parametre triedy. Zvyšnú funkciu je možné dohľadať v dokumentácií (viď. 4.6), alebo priamo v zdrojových kódoch, ktoré sú k diplomovej práci priložené.

Namespace sa do programu pridáva príkazom:

```
87 using NeuralNetworks.Neurons;
```

4.2.1 Trieda neuron

Všetky triedy neurónov sú odvodené od triedy *neuron*. Tá implementuje súbor základných vlastností a funkcií spoločných pre všetky modely neurónov.

Verejne prístupné metódy triedy:

```
88 public abstract double output(double[] inputs)
```

Abstraktná funkcia, ktorej návratovou hodnotou je veľkosť výstupu neurónu pre daný vektor vstupov (*inputs*).

```
89 public abstract neuron Copy(int inputs)
```

Táto funkcia je využitá pri konštrukcii vrstiev neurónov, vracia plytkú kópiu neurónu. Pomocou argumentu *inputs* je možné meniť počet vstupov nového neurónu.

```
90 public void randomize(double min, double max, Random rander)
```

Pri inicializácii neurónu je vhodné, aby počiatočné hodnoty váh neboli nastavené na 0, ale na náhodné hodnoty. Tieto náhodné hodnoty je možné ovplyvňovať dvoma parametrami *min* a *max*, ktoré predstavujú minimálnu a maximálnu hodnotu týchto náhodných čísel. Argument *rander* je inštancia generátora náhodných čísel.

```
91 public double LastOutput
```

Táto prístupová metóda umožňuje zistiť hodnotu posledného výstupu z neurónu.

```
92 public double getWeight(int index)
93 public double[] getWeights()
```

Metódy slúžia na získavanie hodnoty váhy, resp. všetkých váh neurónu, vrátane biasu. Argument index je poradie váhy.

```
94 public void setWeight(int index, double newWeight)
95 public void setWeights(double[] newWeights)
```

Nastavovanie váh je možné jednotlivo, alebo pre celý súbor váh naraz. Pri nastavovaní hodnôt všetkých váh sa vytvára plytká kópia vektora *newWeights*, takže tento argument nie je ovplyvňovaný.

```
96 public int getNInputs()
```

Funkcia vracia počet vstupov do neurónu vrátane biasu.

4.2.2 Trieda feedForwardNeuron

Trieda *feedForwardNeuron* reprezentuje neuróny, ktoré sa nachádzajú v sieťach s dopredným šírením. Konštruktor triedy je definovaný:

```
97 public feedForwardNeuron(int inputs, double treshold,
   activationFunction activationFunction)
```

Kde parameter *inputs* je počet vstupov neurónu vrátane biasu, *threshold* je hodnota prahu a *activationFunction* je inštancia aktivačnej funkcie.

Verejne prístupné metódy triedy:

```
98 public double Threshold
```

Prístupová metóda, pomocou ktorej možno získať, alebo nastaviť hodnotu prahu.

```
99 public double derivedOutput()
```

Funkcia vracia hodnotu derivácie aktivačnej funkcie neurónu. Funkcia nemá žiadne argumenty, pretože derivácia sa počíta z posledných vstupov, alebo výstupov neurónu (premenné *lastInput* a *lastOutput*).

4.2.3 Trieda positionNeuron

Neuróny v SOM sieti sú implementované pomocou triedy *positionNeuron.* Neurón namiesto hodnoty výstupu vracia vzdialenosť svojich váh od vstupného vektora. Konštruktor triedy je definovaný nasledovne:

```
100 public positionNeuron(int inputs, distanceFunction distanceFunction)
```

Argument *inputs* je počet vstupov neurónu, *distanceFunction* je inštancia vzdialenostnej funkcie.

4.3 Namespace layers

Neuróny v sieťach sú organizované vo vrstvách. Funkčnosť týchto vrstiev závisí na type neurónov, z ktorých sa skladajú.

Aby bolo možné nasledujúce triedy použiť, je treba do programu pridať riadok:

```
101 using NeuralNetworks.Layers;
```

4.3.1 Trieda layer

Predlohou pre všetky ostatné vrstvy je abstraktná trieda *layer*.

Verejne prístupné funkcie:

```
102 public abstract double[] output(double[] inputs);
```

Táto funkcia vracia výstup z neurónov vo vrstve. Výstup je pole hodnôt zo všetkých neurónov. Vstupy predstavuje vektor *inputs*.

```
103 public virtual void randomize(double min, double max, Random rander)
```

Nastavenie váh v každom neuróne vo vrstve na náhodné hodnoty. Funkcia len volá metódu *randomize* pre každý neurón(viď. 4.2.1), takže argumenty sú zhodné.

```
104 public virtual neuron getNeuron(int index)
105 public virtual neuron[] getNeurons()
```

K neurónom v sieti je možné buď jednotlivo, keď argument *index* určuje pozíciu neurónu vo vrstve, alebo ku všetkým naraz. Funkcie vracajú *neuron*, resp. *neuron[]*, preto je nutné získané objekty pretypovať na príslušný typ.

```
106 public int getSize()
```

Návratovou hodnotou tejto funkcie je počet neurónov v sieti.

4.3.2 Trieda inputLayer

Trieda *inputLayer* predstavuje vstupnú vrstvu do siete. Táto vrstva neobsahuje neuróny, a teda ani váhy schopné učenia, jej funkčnosť zabezpečujú inštancie triedy *transformationFunction.*

Najčastejšie používané aktivačné funkcie majú výstupy v intervale (0; 1) a žiadna hodnota vstupu nemôže spôsobiť prekročenie tohto intervalu. Pri veľmi vysokých hodnotách na vstupe aktivačných funkcií môže dôjsť k tzv. nasýteniu. Funkcia vtedy aj na pomerne veľký nárast vstupu reaguje len malou zmenou výstupu a tým prakticky eliminuje možnosť učenia. Aby sa tento problém odstránil, je vhodné transformovať hodnoty na prijateľnú veľkosť. Takéto zmeny hodnôt zabezpečuje vstupná vrstva, nie je teda nutné dáta pred použitím upravovať. Výstupom z vrstvy sú teda transformované hodnoty vstupov.

Konštruktory triedy:

```
107 public inputLayer(int nNeurons)
108 public inputLayer(int nNeurons, transformationFunction
    transformation)
```

Argument *nNeurons* udáva počet vstupov do vrstvy, *transformation* je inštancia triedy *transformationFunction*. Ak nie je transformačná funkcia určená (použije sa konštruktor s jedným argumentom) vrstva len kopíruje hodnotu vstupu na výstup.

Vzhľadom na to, že vrstva neobsahuje neuróny nemajú funkcie *randomize*, *getNeurons* a *getNeuron* žiadnu úlohu a ako návratovú hodnotu vracajú *null*.

4.3.3 Trieda feedForwardLayer

V sieťach s dopredným šírením sú ako pracovné vrstvy používané inštancie triedy *feedForwardLayer*. Vrstva obsahuje neuróny implementované triedou *feedForwardNeuron*.

Konštruktory triedy:

```
109 public feedForwardLayer(int nNeurons, int nInputs, feedForwardNeuron
    neuronTemplate)
110 public feedForwardLayer(feedForwardNeuron[] neuronList)
```

Argument *nNeurons* udáva počet neurónov vo vrstve, *nInputs* počet vstupov do jednotlivých neurónov, *neuronTemplate* je vzor pre neuróny, ktoré budú vo vrstve. Všetky neuróny budú mať rovnaký počet vstupov, aktivačnú funkciu a veľkosť prahu ako vzorový neurón. V druhom konštruktore je argument *neuronList* pole už inicializovaných neurónov, zvyšné vlastnosti, ako počet neurónov a vstupov, sú z nich následne dopočítané.

4.3.4 Trieda positionLayer

Trieda *positionLayer* zastupuje pracovnú vrstvu v SOM sieťach. Táto vrstva obsahuje neuróny triedy *positionNeuron* a jej vlastnosti priamo súvisia s jej geometriou (šírka, výška).

Konštruktor triedy:

```
111 public positionLayer(int nInputs, int width, int height,
    positionNeuron neuronTemplate)
```

Argument nInputs určuje počet vstupov do danej vrstvy, argumenty *width* a *height* šírku a výšku vrstvy, *neuronTemplate* je, ako v prípade predchádzajúcej triedy, vzor pre neuróny vrstvy. Počet neurónov je daný vzťahom:

$$n = width \cdot height$$

```
112 public int BMU(double[] inputs)
```

Táto funkcia vracia index BMU (viď. 1.8) vo vrstve.

```
113 public int Width
114 public int Height
```

Prístupové metódy na zisťovanie a výšky a šírky vrstvy.

4.4 Namespace networks

Menný priestor *networks* obsahuje triedy, ktoré umožňujú vytvárať samotné neurónové siete. S inštanciami týchto tried je potom možné ďalej pracovať, učiť ich a naučené siete používať.

Pre sprístupnenie menného priestoru je do programu nutné pridať riadok:

```
115 using NeuralNetworks.Networks;
```

4.4.1 Trieda network

Abstraktná trieda *network* je základ pre ostatné architektúry sietí.

Verejné metódy triedy:

```
116 public abstract void addLayer(layer layer);
```

Pridá vrstvu *layer* do siete. Prvá pridaná vrstva je vstupná, posledná je interpretovaná ako výstupná.

```
117 public void setLayer(int position, layer layer)
```

Vloží do zoznamu vrstiev novú vrstvu, *layer*, na pozíciu *position*. Ak táto pozícia neexistuje, teda na túto pozíciu ešte nebola vložená žiadna vrstva, je vyvolaná výnimka.

```
118 public void randomize(double min, double max)
```

Nastavenie váh v sieti na náhodné hodnoty. Funkcia postupne volá *randomize* pre každú vrstvu siete, okrem vstupnej. Ako inštanciu generátora náhodných čísel používa jeden spoločný pre všetky vrstvy.

```
119 public abstract double[] output(double[] inputs);
```

Návratovou hodnotou funkcie je výstup siete pre vstupný vektor *inputs*. Hodnota výstupu je zároveň výstup z poslednej vrstvy siete. Informácia sa postupne šíri cez vrstvy v poradí, v akom boli vložené. To znamená, že výstup jednej vrstvy je vstupom do ďalšej.

```
120 public layer getLayer(int index)
```

Vracia vrstvu na danej pozícii, *index*, v sieti.

```
121 public int NLayers
```

Umožňuje zisťovať počet vrstiev v sieti.

```
122 public int NWLayers
```

Počet pracovných vrstiev v sieti, teda všetkých vrstiev okrem vstupnej.

4.4.2 Trieda feedForwardNetwork

Táto trieda slúži na vytváranie sietí s dopredným šírením. Oproti pôvodnej triede *network* pridáva len jednu verejnú funkciu:

```
123 public double[][] BPOutput(double[] inputs)
```

Táto funkcia vracia maticu výstupov z jednotlivých neurónov v sieti. Tieto hodnoty sú potrebné pri učení gradientnými metódami.

4.4.3 Trieda backPropagationNetwork

Trieda *backPropagationNetwork* je odvodená od siete s dopredným šírením. Jej účelom je zjednodušiť tvorbu novej siete. Verejné metódy sú len konštruktory triedy:

```
124 public backPropagationNetwork(int[] pattern)
125 public backPropagationNetwork(int[] pattern, activationFunction
    actFunction)
```

Argument *pattern* je vektor, ktorý určuje topológiu siete. Číselné hodnoty určujú počty neurónov v príslušnej vrstve. Počet neurónov v prvej a poslednej vrstve je daný rozmermi vstupných a výstupných dát. V skrytých vrstvách je počet neurónov ľubovoľný, jedinou podmienkou je, aby bol väčší ako nula. Napríklad vektor [2, 5, 3, 1] vytvorí sieť s dvoma vstupmi, dvoma skrytými vrstvami s 5 a 3 neurónmi a jedným výstupom.

Argument *actFunction* určuje, akú aktivačnú funkciu budú mať všetky neuróny v sieti, ak nie je uvedený je použitá sigmoida.

4.4.4 Trieda perceptronNetwork

Táto trieda zjednodušuje vytváranie jednovrstvových sietí typu perceptron. Podobne ako pri triede *backPropagationNetwork* je argumentom konštruktora vzor topológie siete, jeho dĺžka je však obmedzená na 2:

```
126 public perceptronNetwork(int[] pattern)
```

4.4.5 Trieda linearOutputBackPropagationNetwork

Veľká väčšina aktivačných funkcií mapuje svoje vstupy do intervalu (0; 1), preto je nutné prispôsobovať veľkosť požadovaných výstupov pri učení. Toto obmedzenie je možné obísť, ak sa vo výstupnej vrstve použije lineárna aktivačná funkcia, ktorej výstupy nie sú takto obmedzené. Trieda *linearOutputBackPropagationNetwork* slúži na zautomatizovanie tvorby takýchto sietí.

Konštruktory triedy sú:

```
127 public linearOutputBackPropagationNetwork(int[] pattern)
128 public linearOutputBackPropagationNetwork(int[] pattern,
    activationFunction actFunction)
129 public linearOutputBackPropagationNetwork(int[] pattern,
    activationFunction actFunction, activationFunction linActFunction)
```

Argumenty *pattern* a *actFunction* majú rovnakú funkciu ako v triede *backPropagationNetwork*. Argument *linActFunction* je lineárna aktivačná funkcia, ktorá bude použitá vo výstupnej vrstve.

4.4.6 Trieda positionNetwork

Trieda *positionNetwork* je implementáciou Kohonenovej siete. Sieť by mala obsahovať len dve vrstvy, vstupnú a pracovnú. A budú do siete pridané ďalšie vrstvy, spôsobí to nemožnosť učenia siete. Konštruktory triedy:

```
130 public positionNetwork()
131 public positionNetwork(int nInputs, int width, int height)
132 public positionNetwork(int nInputs, int width, int height,
    positionNeuron neuronTemplate)
```

Argument *nInputs* určuje počet vstupov do siete, teda rozmer vstupného vektora, argumenty *width* a *height* predstavujú rozmery pracovnej vrstvy a vyplýva z nich počet neurónov v nej, *neuronTemplate* je vzor pre neuróny pracovnej vrstvy.

```
133 public int SLayer
```

Určuje umiestnenie SOM vrstvy v zozname vrstiev siete.

```
134 public double[] getWeights(int nNeuron)
```

Návratovou hodnotou tejto funkcie je pole váh neurónu, na pozícii *nNeuron*.

4.4.7 Trieda hopfieldNetwork

Keďže má Hopfieldova sieť oproti ostatným sieťam značne zjednodušenú štruktúru, trieda *hopfieldNetwork* nie je odvodená od triedy *network*. Sieť neobsahuje inštancie triedy *layer* ani triedy *neuron*, váhy sú uložené v symetrickej matici.

```
135 public hopfieldNetwork(int nInputs)
```

Konštruktor triedy má len jeden argument, tým je počet vstupov do siete. Z tejto hodnoty vychádza aj veľkosť siete.

```
136 public double[] output(double[] input)
```

Funkcia vracia reakciu siete na daný vstupný vektor. Výstupom je súbor výstupov zo všetkých neurónov. Keďže výstup je jednorozmerné pole, interpretácia výstupu do iných tvarov (matica) musí byť vykonaná až po získaní tohto poľa.

```
137 public void setWeight(int index1, int index2, double newWeight)
```

Nastaví váhu v matici váh na hodnotu *newWeight*. Váhy v Hopfieldovej sieti sú vždy asociované so spojením medzi dvoma neurónmi. Argumenty *index1* a *index2* určujú, ktoré dva neuróny to sú a ďalej slúžia ako indexy do matice váh. Ich poradie je zameniteľné, pretože matica váh je symetrická.

```
138 public int NInputs
```

Prístupová metóda, ktorá umožňuje zistiť počet vstupov do siete. Počet neurónov je rovnaký ako počet vstupov.

```
139 public int OutputThreshold
```

Vybavovanie Hopfiedovej nie je jedno rázový proces ako pri ostatných typoch sietí. Kvalita výstupu je závislá na počte iterácií, po ktorých sa výstup siete nezmenil. Príliš nízka hodnota tohto parametra môže spôsobiť, že sieť nedosiahne minimum svojej energetickej funkcie a teda nenájde príslušný uložený vzor. Vysoká hodnota nezhorší kvalitu výstupu, len zbytočne zvýši čas potrebný na výpočet.

4.5 Namespace learning

Predpokladom na využívanie neurónových sietí je ich úspešné naučenie. Na učenie sietí slúžia triedy v mennom priestore *learning*. Všetky učiace algoritmy majú rovnaký predpis konštruktora, kde jediným jeho argumentom je inštancia siete, ktorá sa má učiť. Podrobnejší popis algoritmov je v kapitole 2.

Aby bolo možné tieto triedy využívať je do programu nutné pridať riadok:

```
140 using NeuralNetworks.Learning;
```

4.5.1 Trieda learningEvents

Väčšina učiacich algoritmov (okrem učenia Hopfieldovej siete) má iteračný charakter. Aby bolo možné sledovať priebeh učenia a reagovať naň, používajú všetky triedy udalosť, ktorá je vyvolaná po ukončení každej iterácie. Trieda *learningEvents* definuje metódy potrebné pre implementovanie a používanie tejto udalosti:

```
141 public delegate void iterationEndEventHandler(object sender,
    EventArgs e)
```

Tento delegát určuje predpis pre funkcie, ktoré budú pripojené k udalosti.

```
142 public event iterationEndEventHandler iterationEnd
```

Inštancia udalosti, ktorá oznamuje ukončenie iterácie učenia.

4.5.2 Trieda supervisedLearning

Na učenie sietí s dopredným šírením sa používa skupina algoritmov nazvaných učenie s učiteľom (supervised learning). Všetky takéto algoritmy sú odvodené od abstraktnej triedy *supervisedLearning*.

Verejne prístupné metódy deklarované v triede:

```
143 public abstract double learn(double[][] inputs, double[][] outputs,
    int cycles, double minError)
```

Táto funkcia reprezentuje samotné učenie siete. Argumenty *inputs* a *outputs* predstavujú trénovaciu množinu (súbor vstupov a k nim príslušných výstupov), *cycles* určuje maximálny počet iterácií algoritmu. Argument *minError* definuje veľkosť chyby siete, po dosiahnutí ktorej sa učenie zastaví. Ak je nastavený na záporné číslo, vykoná sa plný počet iterácií, bez ohľadu na veľkosť chyby.

```
144 public double LearningRate
```

Prístupová metóda umožňujúca zisťovanie a nastavovanie miery učenia.

```
145 public List<double> ErrorList
```

Zoznam, v ktorom sú umiestnené chyby siete pre jednotlivé iterácie a je tak možné sledovať vývoj chyby počas učenia.

```
146 public int Iteration
```

Umožňuje zistiť, ktorá iterácia učenia sa práve vykonáva.

4.5.2.1 Trieda perceptronLearning

Táto trieda definuje učiaci algoritmus pre učenie jednovrstvových sietí s dopredným šírením a skokovými aktivačnými funkciami v neurónoch. Ak by mala sieť, ktorá sa má učiť viac ako jednu vrstvu, je vyvolaná výnimka.

Trieda nedeklaruje žiadne nové, verejne prístupné metódy, len implementuje abstraktné funkcie odvodené od triedy *supervisedLearning*.

4.5.2.2 Trieda deltaRuleLearning

Trieda *deltaRuleLearning* implementuje učenie jednovrstvových sietí pomocou gradientnej metódy. Funkčne vychádza z triedy *perceptronLearning*, podmienkou však je, aby aktivačné funkcie neurónov boli derivovateľné.

4.5.2.3 Algoritmus backpropagation

Algoritmus backpropagation a jeho ďalšie modifikácie sú implementované v štyroch triedach (

Tab. 5). Nové, verejne prístupné metódy sú len v triedach, ktoré používajú momentum:

```
147 public double Momentum
```

Táto prístupová metóda umožňuje zisťovať a nastavovať hodnotu parametra učenia momentum.

Tab. 5 Triedy implementujúce algoritmus backPropagation a jeho modifikácie

Trieda	**Algoritmus učenia**
backPropagationLearning	Prírastkový algoritmus
momentumLearning	Prírastkový algoritmus s momentum
batchBackPropagation	Dávkový algoritmus
batchMomentum	Dávkový algoritmus s momentum

4.5.2.4 Trieda variableLearningRate

Implementácia učenia s premenlivou mierou učenia je v triede *variableLearningRate*. Tá vychádza z *batchBackPropagation*, ale navyše obsahuje dve verejné metódy:

```
148 public double LearningRateIncrease
149 public double LearningRateDecrease
```

Tieto dve prístupové metódy nastavujú a zisťujú veľkosť premenných, ktoré určujú veľkosť zmeny miery učenia.

4.5.2.5 Trieda deltaBarDelta

Táto trieda slúži na učenie siete pomocou algoritmu Delta-Bar-Delta. Verejné metódy sú totožné s triedou *variableLearningRate.*

Na počiatku učenia sú hodnoty miery učenia pre všetky neuróny nastavené na jednotnú hodnotu, tú umožňuje meniť prístupová metóda *LearningRate.*

4.5.2.6 Trieda quickPropagation

Trieda *quickPropagation* implementuje učenie pomocou algoritmu quickpropagation. Oproti triede *batchBackPropagation*, z ktorej je odvodená, má len jednu novú verejnú metódu:

```
150 public double GrowFactor
```

Táto prístupová metóda umožňuje získať a nastaviť hodnotu parametra, určujúceho maximálny prírastok váh siete.

4.5.2.7 Algoritmus resilientpropagation

V mennom priestore *Networks* sa nachádzajú tri triedy, ktoré umožňujú učiť sieť s dopredným šírením pomocou algoritmu resilientpropagation. Ich názvy a príslušný algoritmus sú uvedené v Tab. 6.

Všetky tri triedy majú niekoľko prístupových metód na získavanie a zmenu svojich parametrov:

```
151 public double UpdateMax
152 public double UpdateMin
```

Maximálna a minimálna veľkosť zmeny váh.

```
153 public double NegativeChange
154 public double PositiveChange
```

Premenné určujúce zmenšovanie a zväčšovanie prírastku váh.

Tab. 6 Triedy implementujúce algoritmus resilientpropagation a jeho modifikácie

Trieda	Algoritmus učenia
resilientPropagation	rprop+
improvedResilientPropagation	irprop+
resilientPropagation4	irprop-

4.5.2.8 Trieda differentialEvolution

Učenie neurónovej siete pomocou diferenciálnej evolúcie umožňuje trieda *differentialEvolution*. Tá implementuje prístupové metódy, ktoré umožňujú získavať a nastavovať hodnoty parametrov diferenciálnej evolúcie:

```
155 public double MaxValue
156 public double MinValue
```

Minimálna a maximálna veľkosť váh pri tvorbe novej populácie, tieto premenné neobmedzujú celkovú veľkosť váh, len ich tvorbu.

```
157 public int PopulationSize
```

Veľkosť populácie.

```
158 public double CR
```

Prah kríženia.

```
159 public double F
```

Mutačná konštanta.

```
160 public double B
```

Premenná udávajúca mieru poklesu mutačnej konštanty v priebehu učenia.

4.5.2.9 Trieda SOMA

Použitie samoorganizačného migračného algoritmu na učenie neurónových sietí implementuje trieda SOMA. Táto trieda je odvodená od triedy *differentialEvolution*, ďalej

však používa súbor parametrov použitých v tomto algoritme a k nim príslušné prístupové metódy:

```
161 public double PathLength
```

Maximálna dĺžka cesty aktívneho jedinca.

```
162 public double Step
```

Veľkosť jedného kroku.

```
163 public double PRT
```

Veľkosť perturbácie.

4.5.3 Trieda unsupervisedLearning

Algoritmy na učenie sietí bez učiteľa sú odvodené od abstraktnej triedy *unsupervisedLearning*. Jedinou verejnou metódou, ktorú musí každá odvodená trieda implementovať je metóda *learn*:

```
164 public abstract double learn(double[][] inputs, int cycles)
```

Argument *inputs*, predstavuje súbor vstupných hodnôt, *cycles* počet iterácií učenia.

4.5.3.1 Trieda hopfieldLearning

Na učenie Hopfieldových sietí slúži trieda *hopfieldLearning*. Je nutné, aby argumentom konštruktora triedy bola inštancia triedy *hopfieldNetwork*. Keďže učenie prebieha jednorázovo a nie iteračne ako ostatných algoritmoch učenia, je argument *cycles* v podstate zbytočný, jeho prítomnosť však vyžaduje predpis funkcie *learn*. Podobne aj udalosť *iterationEnd* je volaná len jedenkrát pred ukončením samotného učenia.

4.5.3.2 Trieda SOMLearning

Táto trieda je určená na učenie Kohonenových sietí, teda jej konštruktor vyžaduje ako svoj argument inštanciu triedy *positionNetwork*.

Trieda implementuje dve verejné metódy:

```
165 public double Radius
166 public double Maxradius
```

Obe slúžia na zistenie veľkosti a nastavenie polomeru susedstva BMU. *Maxradius* je maximálna veľkosť susedstva, *Radius* aktuálna veľkosť.

4.5.3.3 Trieda elasticLearning

Trieda *elasticLearning* implementuje elastické učenie a je odvodená od predchádzajúcej triedy, takže verejné metódy má rovnaké a nepridáva žiadne nové.

Pre správnu činnosť učenia je nutné, aby jeden rozmer pracovnej vrstvy siete bol rovný 1, teda vrstva sa dá interpretovať ako jednorozmerná.

4.6 Dokumentácia

Dokumentácia celej knižnice je realizovaná pomocou komentárov priamo v zdrojovom kóde.

Tieto komentáre sú priradené každej triede, funkcii a atribútu. Popisujú nielen funkcie, ale aj ich jednotlivé argumenty a význam návratových hodnôt. Po kompilácii projektu sa vygeneruje XML súbor, ktorý je možné použiť na vytvorenie dokumentácie, ktorá je pre ľudí prijateľnejšia.

Dokumentácia pribalená ku knižnici je vo formáte chm, teda skompilované html. Na jej vytvorenie bol použitý program *Sandcastle Help File Builder*. Súčasťou súboru s dokumentáciou sú aj časti z teoretickej a praktickej časti práce, vysvetľujúce základné princípy činnosti neurónových sietí.

5 PRÍKLADY POUŽITIA

Táto kapitola popisuje použitie výslednej knižnice spolu s ukážkami kódu. Aby bolo možné pracovať s knižnicou Neural networks, je nutné pridať referenciu na ňu do projektu. Najjednoduchším spôsobom ako knižnicu pridať do existujúceho projektu v Microsoft Visual Studio, je použiť ponuku menu Project→Add Reference→Browse a následne vybrať umiestnenie knižnice (súbor NeuralNetworks.dll). K jednotlivým triedam je potom možné pristupovať po pridaní príslušného menného priestoru pomocou direktívy *using* (viď. 4).

Pri práci s neurónovými sieťami je treba splniť nasledujúce kroky:

1. Definícia problému, určenie počtu vstupných a výstupných veličín
2. Vytvorenie siete s ohľadom na problém
3. Vytvorenie trénovacej množiny
4. Učenie siete
5. Používanie siete pri riešení problému

5.1 Siete s dopredným šírením

Pri vytváraní siete je možné použiť viacero tried. Ak postačuje vytvoriť sieť s rovnakými aktivačnými funkciami vo všetkých neurónoch je výhodné použiť triedy *backPropagationNetwork* alebo *linearOutputBackPropagationNetwork*. Ak je potrebné vytvoriť sieť s presne špecifikovanými neurónmi je treba použiť triedu *feedForwardNetwork.*

V nasledujúcich ukážkach kódu bude vytvorená sieť, ktorá má topológiu 2-4-1 (2 vstupné neuróny, 4 skryté neuróny, 1 výstupný neurón).

Vytvorenie siete pomocou triedy *backPropagationNetwork*, kde všetky neuróny používajú hyperbolický tangens ako svoju aktivačnú funkciu:

```
167 backPropagationNetwork net = new backPropagationNetwork(new int[] {
    2, 4, 1 },new hyperbolicTangensActivationFunction(0.4));
```

Vytvorenie siete s rôznymi aktivačnými funkciami v každom neuróne je možné pomocou triedy feedForwardNetwork. Najprv je nutné vytvoriť pole príslušných neurónov a z nich

následne skrytú vrstvu. Vstupná a výstupná vrstva sú taktiež vytvárané ručne. Vytvorené vrstvy sú potom pridané do siete.

```
168 feedForwardNeuron[] neurons = new feedForwardNeuron[4];
169 neurons[0] = new feedForwardNeuron(3, 0.2, new
    sigmoidActivationFunction(steepness));
170 neurons[1] = new feedForwardNeuron(3, 0.2, new
    bipolarElliotActivationFunction(steepness));
171 neurons[2] = new feedForwardNeuron(3, 0.2, new
    hyperbolicTangensActivationFunction(steepness));
172 neurons[3] = new feedForwardNeuron(3, 0.2, new
    elliotActivationFunction(steepness));
173 inputLayer iL = new inputLayer(2);
174 feedForwardLayer hL = new feedForwardLayer(neurons);
175 feedForwardLayer oL = new feedForwardLayer(1, 4, new
    feedForwardNeuron(4, 0, new
    bipolarSigmoidActivationFunction(steepness)));
176 feedForwardNetwork siet = new feedForwardNetwork();
177 ackPropagationNetwork siet = new backPropagationNetwork(new int[] {
    2, 4, 1 },new hyperbolicTangensActivationFunction(0.4));
178 siet.addLayer(iL);
179 siet.addLayer(hL);X
180 siet.addLayer(oL);
```

Na učenie siete je potrebné vytvoriť matice vstupných a výstupných hodnôt. V tomto prípade sa bude jednať o hodnoty, ktoré sieť naučia fungovať ako XOR. Matica vstupných hodnôt je označená ako *Lin*, výstupných hodnôt je *Lout*.

```
181 double[][] Lin = new double[4][];
182 Lin[0] = new double[] { 1, 1 };
183 Lin[1] = new double[] { 1, 0 };
184 Lin[2] = new double[] { 0, 1 };
185 Lin[3] = new double[] { 0, 0 };
186 double[][] Lout = new double[4][];
187 Lout[0] = new double[] { 0 };
188 Lout[1] = new double[] { 1 };
189 Lout[2] = new double[] { 1 };
190 Lout[3] = new double[] { 0 };
```

Pred učením je vhodné nastaviť hodnoty váh v nej na náhodné hodnoty. Následne je možné použiť vytvorenú sieť a trénovaciu množinu na učenie. Ako učiaci algoritmus je použitý

quickpropagation, ktorý má nastavený počet iterácií na 10000 a ukončovaciu chybu na 0,001. Výsledná chyba siete sa uloží do premennej *error*.

```
191 net.randomize(0.01, 0.5);
192 quickPropagation bpl = new quickPropagation(siet);
193 double error = bpl.learn(Lin, Lout, 10000, 0.001);
```

S naučenou sieťou je možné ďalej pracovať. Overiť si správnosť činnosti siete je možné predložením jedného vstupného vektoru z trénovacej množiny, výstup siete by sa mal približne zhodovať s príslušným požadovaným výstupom.

```
194 siet.output(Lin[0]);
```

5.2 Hopfieldova sieť

Pred použitím Hopfieldovej siete v programe je nutné vytvoriť inštanciu triedy *hopfieldNetwork*. Argumentom konštruktora triedy je počet neurónov v sieti, ten musí byť rovnaký ako veľkosť vstupného vektora.

```
195 hopfieldNetwork net = new hopfieldNetwork(5);
```

Na rozdiel od učenia s učiteľom stačí pre učenie Hofieldovej siete vytvoriť len maticu vstupných hodnôt. V tomto sú to dva vektory o dĺžke 5:

```
196 double[][] Lin = new double[2][];
197 Lin[0] = new double[] { 1, 1, -1, 1, 1 };
198 Lin[1] = new double[] { -1, -1, 1, -1, -1 };
```

Učenie siete je realizované pomocou triedy *hopfieldLearning*. Keďže sa jedná o učenie, ktoré je vykonané len v jednej iterácii, nie je nutné určovať počet iterácií učenia. V nasledujúcej ukážke je vytvorený inštancia triedy *hopfieldLearning* a sieť je naučená na príslušnú trénovaciu množinu *Lin*.

```
199 hopfieldLearning hnl = new hopfieldLearning(net);
200 hnl.learn(Lin);
```

Pred získavaním výstupu zo siete je vhodné nastaviť veľkosť prahu výstupu. Vstupnými hodnotami do siete sú zašumené vektory, ktorých pôvodné vzory sa majú zo siete získať.

```
201 siet.OutputThreshold = 20;
202 siet.output(new double[] { 1.3, 1.1, 1.2, 1,1 })
```

5.3 SOM

Pre prácu s Kohonenovými mapami je určená trieda *positionNetwork*. Pred vytvorením samotnej siete je nutné vytvoriť vzorový neurón, z ktorých bude zložená pracovná vrstva siete. V tejto ukážke je to neurón s 3 vstupmi a vzdialenostnou funkciou manhattan. Výsledná sieť má rozmery pracovnej vrstvy 40x40 neurónov.

```
203 positionNeuron neuron = new positionNeuron(3, new
    manhattanDistance());
204 positionNetwork net = new positionNetwork(3, 40, 40, neuron);
```

Podobne ako pri Hopfieldovej sieti je trénovacia množina len jedna matica vstupných hodnôt. V tomto prípade je to použitá trénovacia množina prevzatá zo vzorovej aplikácie „zoskupovanie farieb“. Jedná sa teda o 8 farieb reprezentovaných hodnotami červenej, zelenej a modrej farby.

```
205 double[][] Lin = new double[8][];
206 Lin[0] = new double[] { 255, 0, 0 };
207 Lin[1] = new double[] { 0, 255, 0 };
208 Lin[2] = new double[] { 0, 128, 64 };
209 Lin[3] = new double[] { 0, 0, 255 };
210 Lin[4] = new double[] { 0, 0, 128 };
211 Lin[5] = new double[] { 255, 255, 51 };
212 Lin[6] = new double[] { 255, 102, 64 };
213 Lin[7] = new double[] { 255, 0, 255 };
```

Učenie siete, teda inštancia triedy *SOMLearning*, má len jeden riadiaci parameter, tým je počet iterácií učenia. Učenie siete na danú trénovaciu množinu a 5000 iterácií učenia:

```
214 SOMLearning sl = new SOMLearning(net);
215 error = sl.learn(Lin, 5000);
```

Výstup zo siete je index BMU. V nasledujúcej ukážke je získaný index BMU pre červenú farbu (vektor [255, 0, 0]) a do premennej *val* uložená jeho hodnota váh.

```
216 double index = net.output(new double[] { 255, 0, 0 })[0];
217 double[] val = net.getWeights((int)index);
```

Pri vytváraní siete, ktorá má byť učená pomocou elastického učenia, je potrebné dodržať, aby jeden z rozmerov siete bol rovný 1. Teda vytvorí sa sieť s rozmermi 40x1 neurón.

```
218 positionNeuron neuron = new positionNeuron(3, new
    manhattanDistance());
219 positionNetwork net = new positionNetwork(3, 40, 1, neuron);
```

Trénovacia množina je zoznam súradníc bodov na ploche:

```
220 double[][] Lin = new double[4][];
221 Lin[0] = new double[] { 10, 1 };
222 Lin[1] = new double[] { 5, 4 };
223 Lin[2] = new double[] { 6, 8};
224 Lin[3] = new double[] { 4, 2};
```

Učenie siete:

```
225 elasticLearning el = new elasticLearning(net);
226 el.learn(Lin, 5000);
```

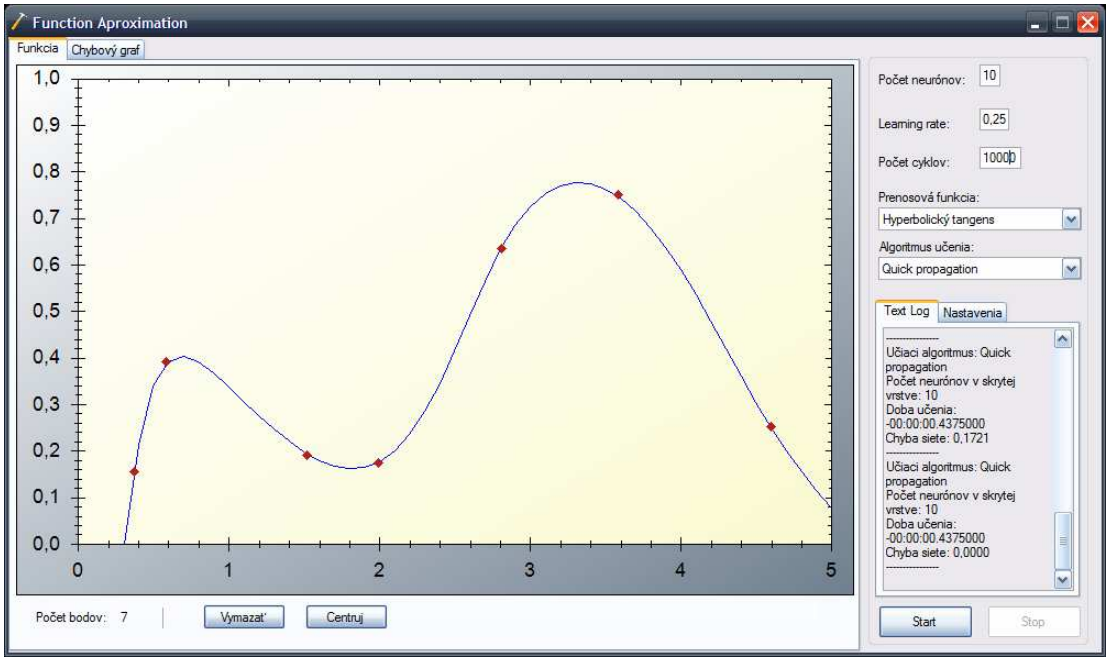
6 VZOROVÉ APLIKÁCIE

Pre demonštráciu činnosti knižnice a spôsobu práce s ňou slúžia vzorové aplikácie.

6.1 Aproximácia funkcií

Častým využitím sietí s dopredným šírením je aproximácia funkcií. Snahou je naučiť sieť tak, aby jej výstup aproximoval funkciu danú trénovacou množinou. Klikaním myšou do zobrazovacej oblasti okna sa vytvárajú body, ktoré budú tvoriť trénovaciu množinu. Vstupnými dátami sú súradnice na osi x, požadovanými výstupmi sú súradnice bodov na osi y. Po stlačení tlačidla „Start“ sa začne učenie siete, ktoré je možné prerušiť stlačením tlačidla „Stop“.

Po ukončení učenia je zobrazená výsledná aproximácia funkcie. Výsledok je získaný postupným predkladaním hodnôt v celom rozsahu osi x sieti.

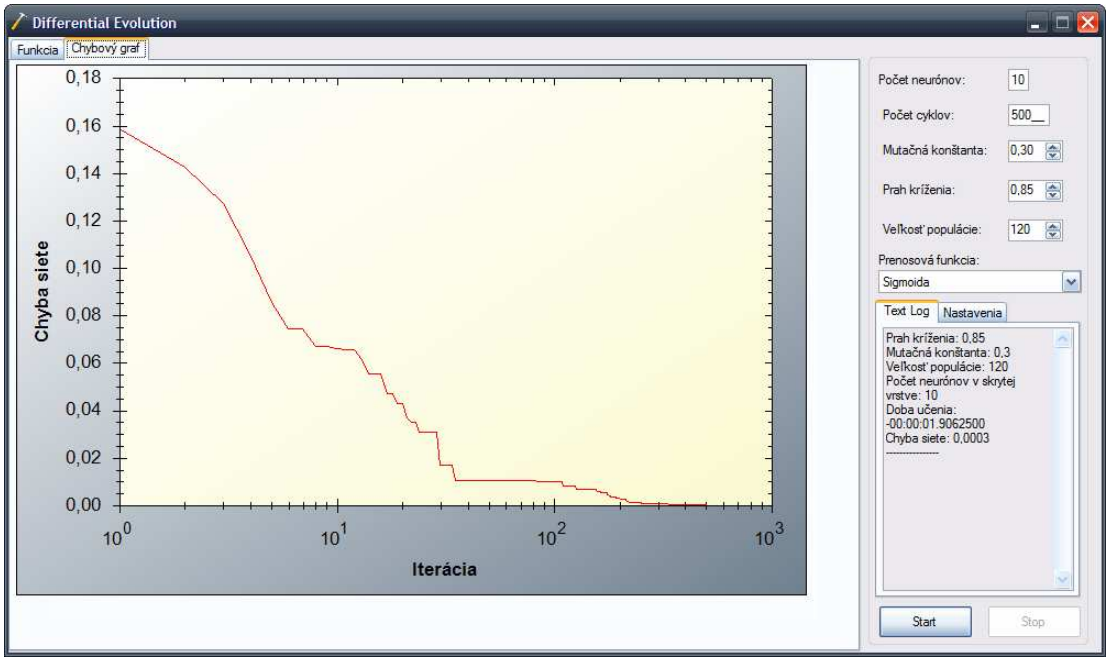


Obr. 28 Aproximácia funkcií

Učenie siete a sieť samotnú je možné ovplyvniť zmenou rôznych parametrov, ich zoznam je v Tab. 7. Záložka „Chybový graf“ zobrazuje priebeh chyby siete počas učenia.

Tab. 7 Možnosti nastavenia siete pri aproximácií funkcií

Názov	Funkcia
Počet neurónov	Počet neurónov v skrytej vrstve siete
Learning rate	Miera učenia
Počet cyklov	Maximálny počet iterácií algoritmu učenia
Prenosová funkcia	Prenosová (aktivačná) funkcia neurónov siete
Algoritmus učenia	Použitý algoritmus učenia
Minimálna hodnota	Minimálna hodnota pri inicializácií váh
Maximálna hodnota	Maximálna hodnota pri inicializácií váh
Maximálna chyba	Hodnota chyby, pri ktorej učenie skončí
Strmosť prenosovej	Strmosť aktivačnej funkcie neurónov



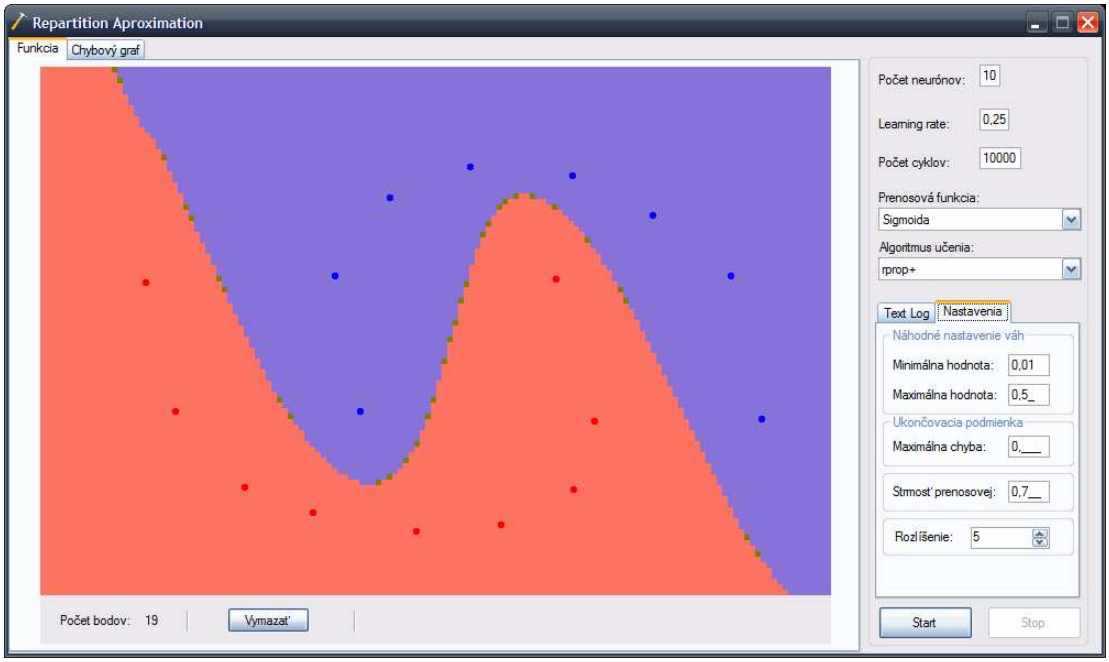
Obr. 29 Vývoj chyby siete počas učenia

6.2 Klasifikácia

Ďalším častým využitím siete s dopredným šírením je klasifikácia.

Trénovaciu množinu je opäť možné vytvoriť použitím myši. Ľavé tlačidlo pridá do trénovacej množiny prvok z množiny s označením 0, pravé prvok z množiny 1. Obe množiny sú farebne odlíšené. Po naučení siete sa zobrazí výsledok, ktorý predstavuje rozdelenie plochy medzi obe množiny. Červená farba predstavuje množinu 0, sú k nej priradené výstupy siete, ktoré sú menšie ako 0,4, modrá farba predstavuje množinu 1, k tej patria výstupy siete väčšie ako 0,6. Výstupy siete v intervale (0,4; 0,6) sú považované za neurčité a sú zobrazené zelenou farbou.

Aplikácia má rovnaké možnosti nastavenia ako pri aproximácii funkcií, len s jednou pridanou možnosťou: „Rozlíšenie“. To určuje hustotu prehľadávania priestoru pri vybavovaní siete.



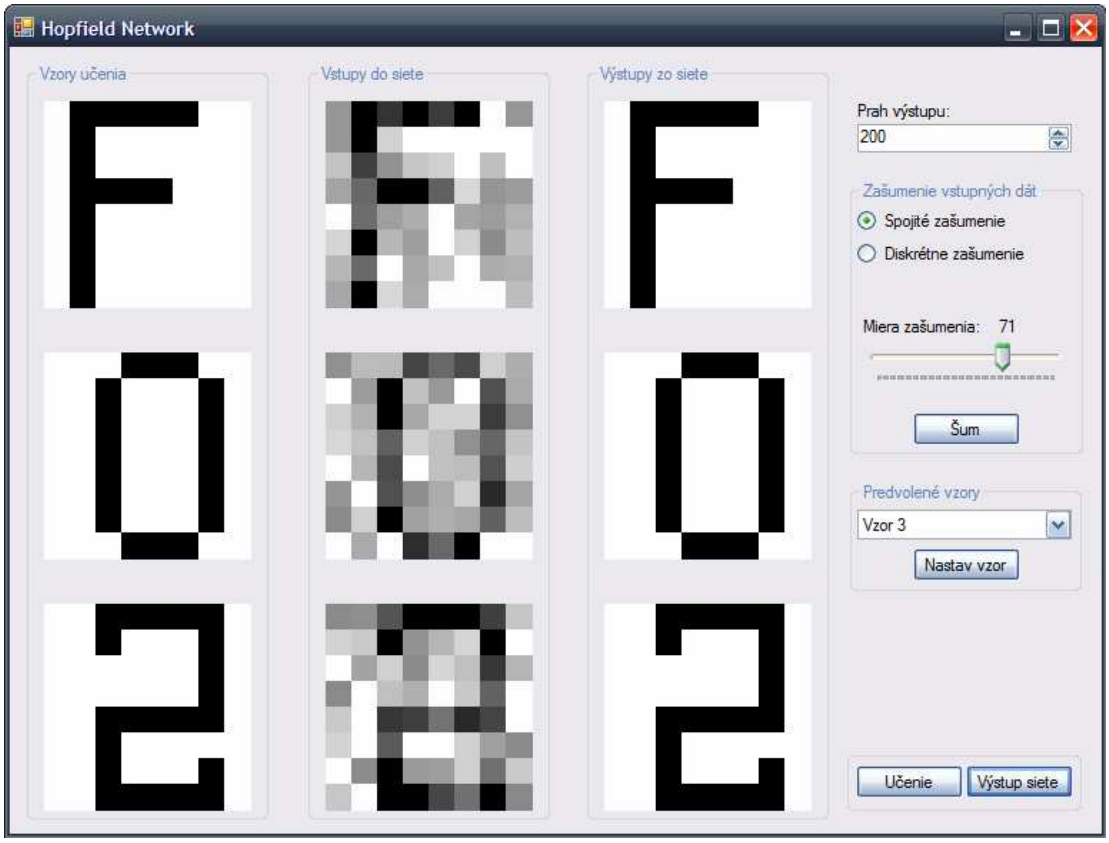
Obr. 30 Klasifikácia

6.3 Hopfieldova sieť

Táto aplikácia slúži na predstavenie možností Hopfieldovej siete. Veľkosť siete je konštantná, 64 neurónov, a nie je možné ju meniť. Aplikácia umožňuje vytvoriť 3 vstupné vzory, na ktoré sa sieť naučí. Tieto vzory sa vytvárajú v prvom stĺpci, pomenovanom „Vzory učenia“, pomocou myši. Pre zjednodušenie sú v aplikácii pripravené 3 sady vzorov, ktoré je možné načítať pomocou tlačidla „Nastav vzor“. Po stlačení tlačidla „Učenie“ sa sieť naučí na dané vstupné vzory. V druhom stĺpci – „Vstupy do siete“ je možné vytvoriť vzory, ktoré budú predložené sieti pri vybavovaní. Po naučení siete sú v tomto stĺpci zobrazené pôvodné trénovacie vzory, tie je možné pozmeniť pomocou myši, alebo tlačidla „Šum“. Na výber sú dva spôsoby zašumenia:

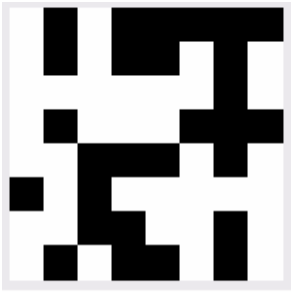
- Spojité (Obr. 33) – hodnota bunky môže nadobúdať hodnotu v obore [0; 1]
- Diskrétne (Obr. 32) – hodnota bunky je buď 1, alebo -1

Mieru zašumenia je možné meniť posuvníkom, ktorého hodnota určuje pravdepodobnosť, s akou dôjde k zmene hodnoty v bunke. Pri spojitom zašumení je potom hodnota bunky daná náhodným číslom z intervalu [0; 1].

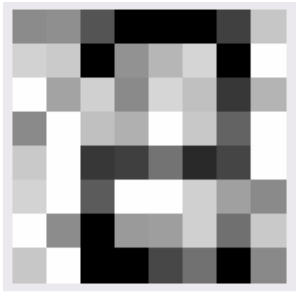


Obr. 31 Hopfieldova sieť

Tlačidlom „Výstup siete“ sa do tretieho stĺpca („Výstupy zo siete“) zakreslia reakcie siete na vzory z druhého stĺpca. Kvalitu výstupu možno meniť pomocou prahu výstupu.



Obr. 32 Diskrétne zašumenie



Obr. 33 Spojité zašumenie

6.4 Zoskupovanie farieb

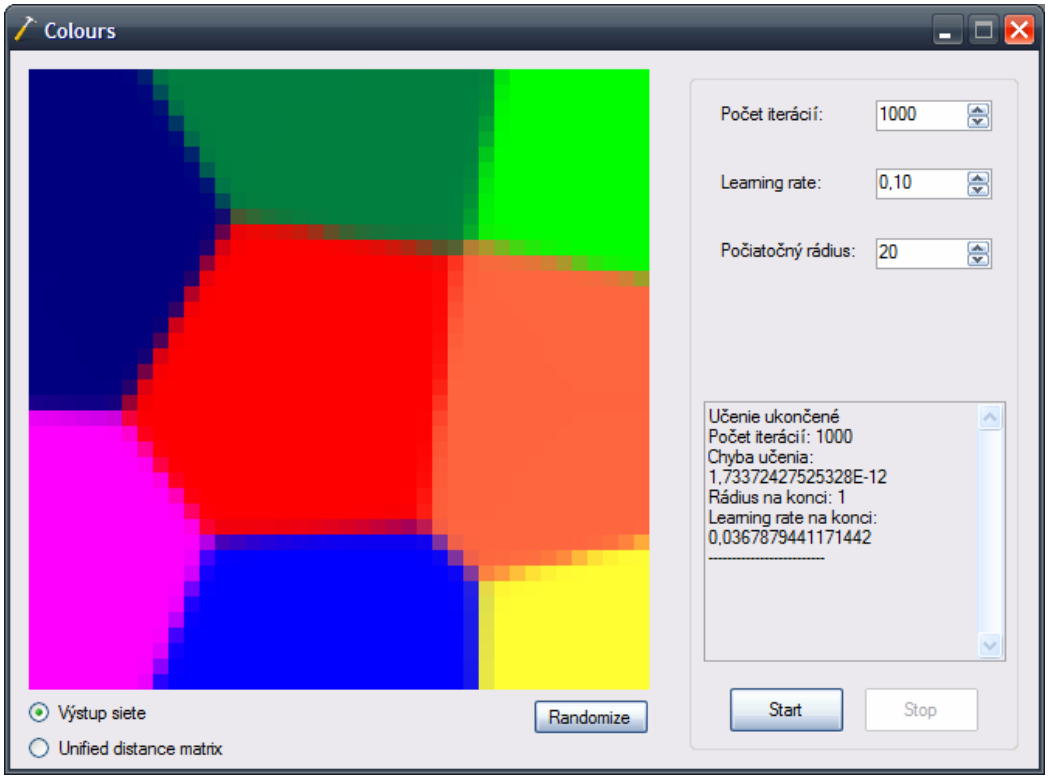
Kohonenova sieť sa v praxi často využíva na vizualizáciu mnohorozmerných dát na dvojrozmernú plochu. Najjednoduchším spôsobom ako demonštrovať túto vlastnosť je vizualizácia farieb. Farby sú prezentované ako trojrozmerné vektory ([R; G; B]). Trénovacou množinou je 8 farieb, ktoré sieť postupne namapuje na svoje váhy a ich výsledné umiestnenie zobrazí.

Ovládanie aplikácie je podobné ako v prípade aproximácie funkcií, pribudli len 4 ovládacie prvky špecifické pre SOM:

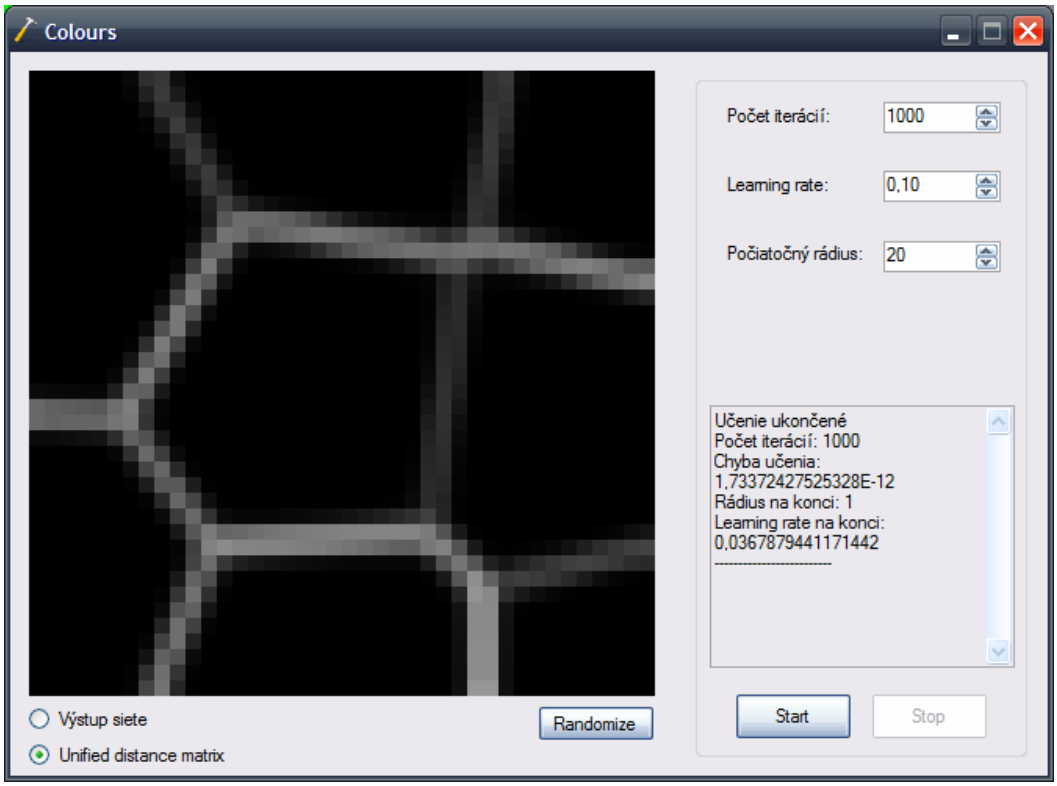
Tab. 8 Možnosti nastavenia pri zoskupovaní farieb

Názov	**Funkcia**
Počiatočný rádius	Polomer veľkosti susedstva BMU na začiatku učenia
Randomize	Nastaví váhy siete na náhodné hodnoty
Výstup siete	Zobrazovanie umiestnenia jednotlivých farieb v sieti (Obr. 34)
Unified distance matrix	Zobrazovanie UMD pre momentálny stav váh siete (Obr. 35)

Na zistenie kvality výstupu siete slúži Unified distance matrix – UMD (voľne preložené ako unifikovaná matica vzdialeností). Tá predstavuje súbor čísel, kde každé predstavuje aritmetický priemer vzdialeností neurónu od jeho susedov. Vizualizácia UMD je na Obr. 35. Čierna farba znamená, že medzi susedmi sú nulové rozdiely, čím väčšie sú rozdiely, tým svetlejšia je farba. Čierne oblasti je možné interpretovať ako zhluky (clustre) a svetlé oblasti ako hranice medzi nimi.



Obr. 34 Zoskupovanie farieb



Obr. 35 Unified distance matrix

6.5 Obchodný cestujúci

Na demonštráciu možností elastického učenia SOM siete je určená aplikácia na riešenie problému obchodného cestujúceho.

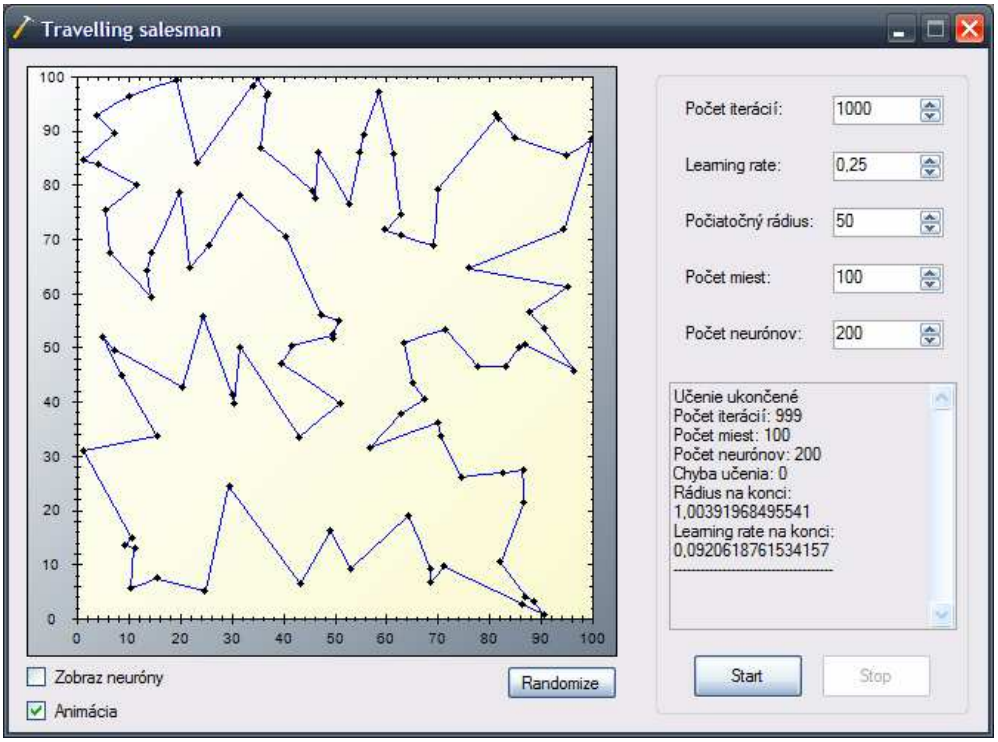
Problém obchodného cestujúceho patrí medzi klasické úlohy v optimalizácii a kombinatorike. Cieľom je nájsť najkratšiu cestu medzi súborom miest tak, aby bolo každé mesto navštívené práve jedenkrát.

Ovládanie vychádza z predchádzajúcej aplikácie. Špecifické nastavenia sú uvedené v Tab. 9.

Tab. 9 Možnosti nastavenia elastického učenia

Názov	**Funkcia**
Počet miest	Počet miest problému
Počet neurónov	Počet neurónov v sieti
Zobraz neuróny	Zobrazenie pozície jednotlivých neurónov na výslednej ceste
Animácia	Zobrazovanie priebehu učenia

Po ukončení učenia je zobrazená nájdená cesta medzi mestami. Predpokladom pre nájdenie optimálnej cesty je, aby počet neurónov bol väčší, prinajmenšom rovnaký, ako počet miest.



Obr. 36 Riešenie problému obchodného cestujúceho

ZÁVĚR

Cieľom tejto práce bolo vytvoriť knižnicu, ktorá umožňuje vytváranie a prácu s umelými neurónovými sieťami v programovacom jazyku C#.

Teoretická časť sa venovala základným popisom danej problematiky. Nachádza sa v nej stručná história vzniku a používania neurónových sietí, spolu s najvplyvnejšími priekopníkmi. Ďalšia časť popisuje analógiu medzi nervovou sústavou živočíchov a umelými neurónovými sieťami. V nadväznosti na predchádzajúcu kapitolu sú popísané modely neurónov, spolu s princípmi ich fungovania. Tieto modely sú základom pre činnosť celých sietí. Ďalšie popisy sa týkajú rôznych topológií sietí implementovaných v praktickej časti.

Podstatná časť teoretickej časti je venovaná učeniu neurónových sietí. Obsahuje základné princípy fungovania učiacich algoritmov, ktoré sú následne implementované v praktickej časti práce.

Posledná kapitola teoretickej časti obsahuje popis niekoľkých existujúcich implementácií neurónových sietí v C# a použitie neurónových sietí v súčasnosti.

Praktická časť je zameraná na popis výsledných programov. Prvá časť obsahuje zoznam všetkých verejne prístupných tried a ich verejných metód. Nasleduje krátka ukážka používania knižnice, pre všetky architektúry sietí. Posledná kapitola praktickej časti sa zameriava na vzorové aplikácie.

Okrem samotného textu je výstupom tejto práce samotná knižnica spolu so vzorovými programami, ktoré demonštrujú činnosť a použitie jednotlivých architektúr sietí. Všetky výstupy sú dodané v binárnej forme aj ako zdrojové kódy.

Súčasťou praktickej časti je aj dokumentácia k jednotlivým funkciám a atribútom tried v knižnici. Dokumentácia je vo formáte chm, teda skompilované html. V nej sú popísané všetky funkcie, vrátane *private* a *protected* členov tried, argumenty funkcií a ich návratové hodnoty. V dokumentácii sú zaradené aj súčasti z tejto práce, hlavne z jej teoretickej časti, pre lepší popis a pochopenie fungovania knižnice.

ZÁVĚR V ANGLIČTINĚ

The aim of this thesis was to create a library that allows construction and work with artificial neural networks in the C # programming language.

The theoretical part focuses on basic description of the problem. First part contains brief history of neural networks, together with people who influenced this field. The next section describes the analogy between the nervous system of animals and artificial neural networks. Following chapter describes the models of neurons, together with the principles of their operation. These models are essential for the activity of entire networks. Further descriptions concern the different network topologies implemented in practical part.

A substantial part of the theoretical part is devoted to learning algorithms of neural networks. It contains its basic principles, which are then implemented in practical work.

The last chapter of the theoretical part contains a description of several existing libraries working with neural networks in C # and the use of neural networks at the present.

The practical part focuses on description of neural networks library. The first section contains a list of all public classes and their public methods. After that is a short demonstration of use of library, for all types of networks. Last chapter of practical part focuses on sample applications.

Main output of this thesis is neural networks library and sample programs that demonstrate the operation and use of networks. All outputs are delivered in binary form, as well as source code.

Part of the practical part is a documentation of the individual functions and classes in the library. Documentation is in chm format, which is compiled html. It describes all features, including private and protected members of classes, functions, arguments and return values. The documentation includes the components of this work, mainly from the theoretical part, for a better description and understanding of the functionality of the library.

SEZNAM POUŽITÉ LITERATURY

[1] ERIC ROBERTS. Neural networks [online]. 2000 [cit. 2009-04-10]. Dostupný z WWW: <http://cse.stanford.edu/class/sophomore-college/projects-00/neural-networks/index.html>.

[2] BOSE, N. K., LIANG, P. Neural network fundamentals, with graphs, algorithms and applications, 1996. 478 s.

[3] Artificial neural network [online]. 2008 [cit. 2009-04-10]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Artificial_neural_network>.

[4] BrainMaker Neural Network Application Examples [online]. 2000 [cit. 2009-04-10]. Dostupný z WWW: <http://www.calsci.com/Applications.html>.

[5] ANDERSON, Dave, MCNEIL, George. Artificial Neural Networks Technology. Data & Analysis Center for Software [online]. 1992 [cit. 2009-04-10].

[6] JIŘÍ, Šíma, ROMAN, Neruda. Teoretické otázky neuronových sítí. Praha : Matfyzpress, 1996. 195 s.

[7] GUIYING, Ning, YONGQUAN, Zhou. A Modified Differential Evolution Algorithm for Optimization Neural Network, 2007. 5 s.

[8] OPLATKOVÁ, Zuzana, OŠMERA, Pavel, ŠEDA, Miloš, VČELAŘ, František, ZELINKA , Ivan. Evoluční výpočetní techniky - principy a aplikace, 2008. 536 s. ISBN 80-7300-218-3.

[9] MICHELE, Estebon. Perceptrons: An Associative Learning Network, 1997. 5 s.

[10] HORNIK, Kurt. Approximation Capabilities of Multilayer Feedforward Networks. Neural Networks, 1991.

[11] HEATON, Jeff. Introduction to Neural Networks with Java, 2005. 378 s. ISBN 9780977320608.

[12] Supervised learning [online]. 2009 [cit. 2009-04-10]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Supervised_learning>.

[13] KROSE, Ben, VAN DER SMAGT, Patrick. An introduction to neural networks, 1996. 135 s.

[14] Differential evolution homepage [online]. 1996 [cit. 2009-04-10]. Dostupný z WWW: <http://www.icsi.berkeley.edu/~storn/code.html#R___>.

[15] TADEUSIEWCZ, Ryszard. Principles of training multi-layer neural network using backpropagation [online]. 1992 [cit. 2009-04-10]. Dostupný z WWW: <http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html>.

[16] DEMUTH, Howard, BEALE, Mark, HAGAN, Martin. Neural network toolbox 6 - User's guide. [s.l.] : [s.n.], 2009. 906 s. ISBN 0-9717321-0-8.

[17] SMITH, Leslie. Six lectures on Back-Propagation [online]. 1996 [cit. 2009-04-10]. Dostupný z WWW: <http://www.cs.stir.ac.uk/~lss/31X7/BPlectures/>.

[18] Neurónové siete Inžiniersky prístup (1. diel) [online]. c1998 [cit. 2009-10-04]. Dostupný z WWW: <http://www.ai-cit.sk/source/publications/books/NS1/html/node1.html>.

[19] FAHLMAN, Scott. An Empirical Study of Learning Speed in Back-Propagation Networks, 1988. 19 s.

[20] IGEL, Christian, HUSKEN, Michael. Improving the Rprop Learning Algorithm, 2000. 7 s.

[21] BUCKLAND, Mat. Kohonen's Self Organizing Feature Maps [online]. 2002 [cit. 2009-04-10]. Dostupný z WWW: <http://www.ai-junkie.com/ann/som/som1.html>.

[22] CHESNUT, Casey. Self Organizing Map AI for Pictures [online]. 2004 [cit. 2009-04-10]. Dostupný z WWW: <http://www.generation5.org/content/2004/aiSomPic.asp>.

[23] Self-organizing map [online]. 2009 [cit. 2009-04-10]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Kohonen>.

[24] SUCHAL, Ján. Komunity a Kohonenove samoorganizačné mapy [online]. 2004 [cit. 2009-04-10]. Dostupný z WWW: <http://johno.jsmf.net/archiv/komunity-a-kohonenove-samoorgranizacne-mapy.html>.

[25] CIMPONERIU, Andrei. Modeling the Development of Ocular Dominance and Orientation Preference Maps in The Primary Visual Cortex with The Elastic Net [online]. 1999 [cit. 2009-04-10]. Dostupný z WWW: <http://camelot.mssm.edu/~andrei/elastic_net_tutorial/elastic_net_tutorial.htm>.

[26] FIESLER, Emile, FIESLER, Beale, BEALE, Russell. Handbook of Neural Computation, 1997. 80 s. ISBN 075030524X.

[27] Memory - the Hopfield Net [online]. c1995 [cit. 2009-04-10]. Dostupný z WWW: <http://www.web-us.com/brain/neur_hopfield.html>.

[28]Hopfield net [online]. 2009 [cit. 2009-04-10]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Hopfield_net>.

[29] Hopfield Neural Network [online]. 2007 [cit. 2009-04-10]. Dostupný z WWW: <http://www.learnartificialneuralnetworks.com/hopfield.html>.

[30] ANDERSON, Dave, MCNEIL, George. Artificial Neural Networks Technology [online]. 1992 [cit. 2009-04-10]. Dostupný z WWW: <https://www.dacs.dtic.mil/techs/neural/neural_ToC.php>.

[31] Neural Network [online]. 2009 [cit. 2009-04-10]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Neural_networks>.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

TLU Threshold logic units

BMU Best matching unit

SOM Self organizing map

SOMA Samoorganizující se migrační algoritmus

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I: POROVNANIE ALGORITMOV UČENIA SIETÍ S DOPREDNÝM ŠÍRENÍM

Všetky testované algoritmy mali za úlohy naučiť sieť s topológiou 2-10-1 na XOR problém. maximálny počet iterácií bol nastavený na 10000, minimálna požadovaná chyba na 0,001, miera učenia 0,25. Veľkosť populácie pre diferenciálnu evolúciu je nastavená na 52, pre SOMU na 26 jedincov.

Učenie s každým algoritmom bolo opakované 50 krát, pred každým opakovaním boli hodnoty váh inicializované na náhodné hodnoty z intervalu [0.001, 0.5].

V tabuľkách je zobrazený počet iterácií, ktorý bol potrebný na dosiahnutie minimálnej požadovanej chyby pre jednotlivé algoritmy a čas, ktorý bol potrebný na výpočet.

Trieda učenia	**Počet iterácií**		
	Minimálny	**Maximálna**	**Priemerný**
backPropagationLearning	4017	8811	6190
momentumLearning	2704	5739	3675
batchBackPropagation	2825	7055	3992
batchMomentum	2060	4131	2719
variableLearningRate	663	1547	996
deltaBarDelta	716	1501	942
quickPropagation	43	1821	217
resilientPropagation	27	69	42
improvedResilientPropagation	30	50	37
resilientPropagation4	30	59	37
differentialEvolution	4	9	7
SOMA	7	22	13

Trieda učenia	Doba trvania [ms]		
	Minimálny	**Maximálna**	**Priemerný**
backPropagationLearning	322	18996	9724
momentumLearning	306	13682	7278
batchBackPropagation	193	9225	4549
batchMomentum	125	7186	3680
variableLearningRate	43	2359	1244
deltaBarDelta	63	3054	1536
quickPropagation	11	356	187
resilientPropagation	4	142	70
improvedResilientPropagation	6	103	54
resilientPropagation4	5	103	51
differentialEvolution	64	2692	1324
SOMA	574	28508	14100