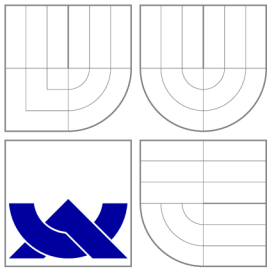
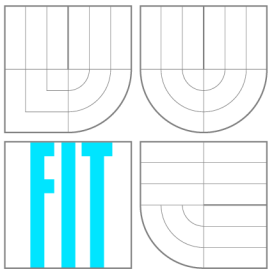


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# AKCELEROVANÉ NEURONOVÉ SÍTĚ NA GRAFICKÉ KARTĚ

ACCELERATED NEURAL NETWORKS ON GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN TOMKO

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTIN KRČMA,

BRNO 2015

# Abstrakt

Tato práce se věnuje implementaci aplikace pro simulaci neuronových sítí a její akceleraci za využití grafického procesoru. Aplikace se zaměřuje především na sítě typu Feedforward a jejich učení algoritmem Backpropagation, podporuje však i jiné typy sítí a umožňuje rozšíření o další učící algoritmy. Aplikace také umožňuje zavést do sítě různé poruchy struktury, což je možné využít pro testování odolnosti neuronových sítí vůči poruchám. Práce je implementována v jazyce C++ za využití OpenCL pro výpočty na GPU. Výsledky akcelerace učení algoritmem Backpropagation byly porovnány s volně dostupnou knihovnou FANN.



# Abstract

This thesis deals with the implementation of an application for artificial neural networks simulation and acceleration using a graphics processing unit. The computation and training of feedforward neural networks using the Backpropagation algorithm are the main focus of this thesis, but the application also supports other network types, and it makes it possible to extend the application with different training algorithms. Next, the application allows us to create neural networks with structural anomalies, and thus, to test the neural network's fault tolerance. The application is implemented in the C++ language, using OpenCL to manage GPU computation. The Backpropagation acceleration results were compared with the free open source library FANN.



# Klíčová slova

Neuronové sítě, strojové učení, akcelerace, GPU, OpenCL, C++

# Keywords

Neural networks, machine learning, acceleration, GPU, OpenCL, C++


# Citace

Martin Tomko: Akcelerované neuronové sítě na grafické kartě, bakalářská práce, Brno, FIT VUT v Brně, 2015

# Akcelerované neuronové sítě na grafické kartě

## Prohlášení

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

.....
Martin Tomko
20. května 2015

## Poděkování

Chcel by som sa poďakovať svojmu vedúcemu, Ing. Martinovi Krčmovi, za ochotný prístup a pomoc pri tvorbe bakalárskej práce.

# Obsah

# Kapitola 1

# Úvod

Neurónové siete sú už niekoľko desaťročí predmetom intenzívneho výskumu, a predstavujú veľmi sľubnú oblasť pre rozvoj umelej inteligencie a strojového učenia.

Jednou z najväčších prekážok pri nasadzovaní neurónových sietí do praxe je však časová náročnosť ich učenia – podľa zložitosti úlohy a siete môže trénovanie na určitú úlohu trvať od niekoľko minút po niekoľko týždňov [14].

Táto práca si preto kladie za cieľ implementovať aplikáciu pre prácu s neurónovými sieťami, ktorá nielen umožní učenie neurónových sietí, simuláciu ich výpočtu a testovanie natrénovaných neurónových sietí, ale výpočty tiež akceleruje využitím výpočetnej sily GPU.

Aplikácia tiež umožní úpravy štruktúry sietí (ako pridané či odobrané spoje medzi neurónmi), čo umožní experimentáciu s upravenými sieťami a testovanie odolnosti sietí voči poruchám.

Stručný popis neurónových sietí a s nimi súvisiace definície obsahuje kapitola 2. Popisu technológie OpenCL[1], ktorá je využívaná pre akceleráciu aplikácie, sa venuje kapitola 3. Kapitoly 4 a 5 sa venujú návrhu a popisu implementácie aplikácie, kým kapitola 6 obsahuje popis a výsledky experimentov, ktoré porovnávajú rýchlosť aplikácie s voľne dostupnou knižnicou FANN[2].

[1] https://www.khronos.org/opencl/
[2] http://leenissen.dk/fann/wp/

# Kapitola 2

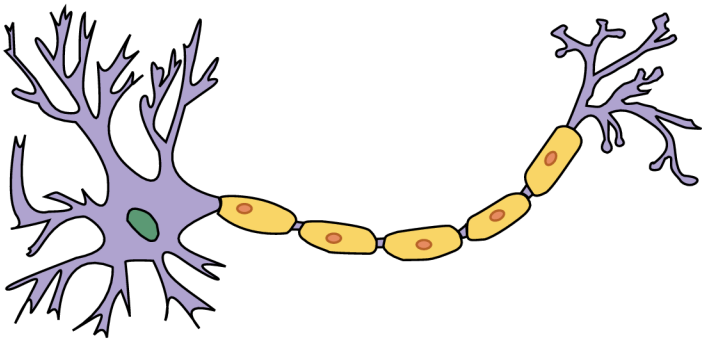
# Neuróny a neurónové siete

Táto kapitola pokrýva základné princípy neurónových sietí, a špeciálne sa venuje typom neurónových sietí, ktorých implementáciou sa táto práca zaoberá.

*Neurónová sieť* (presnejšie *umelá neurónová sieť*) je výpočetný model inšpirovaný ľudským mozgom, pozostávajúci z jednotiek nazývaných neuróny. Podobne ako ľudské neuróny sú tieto umelé neuróny navzájom poprepájané spojmi, ktorými sa navzájom ovplyvňujú. Významným aspektom neurónových sietí je ich *učenie* – prispôsobenie siete na riešenie určitej úlohy, prebiehajúce typicky konfiguráciou spojení medzi neurónmi na základe vhodnej *trénovacej sady*.

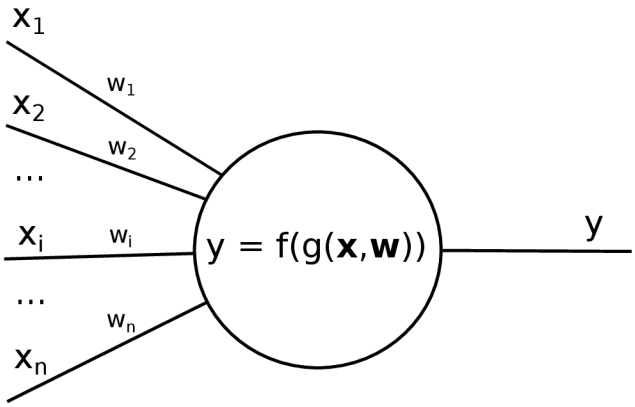
## 2.1 História neurónových sietí

Pôvodnou inšpiráciou umelých neurónových sietí sú biologické neurónové siete, špeciálne ľudské. Dôležitým predpokladom pre uvažovanie o umelých neurónoch bolo teda hlbšie pochopenie ľudskej nervovej sústavy. Veľkým prevratom v chápaní tejto oblasti bola myšlienka, že ľudský mozog sa skladá zo základných funkčných jednotiek nazývaných neuróny (ilustrované na obrázku 2.1). Tento objav potvrdili koncom 19. storočia Santiago Ramón y Cajal a Camillo Golgi [10], ktorí boli v roku 1906 za svoju prácu na štruktúre nervovej sústavy ocenení Nobelovou cenou za fyziológiu alebo medicínu [1].



Obrázok 2.1: Ilustrácia biologického neurónu [4]

Prvý umelý model biologického neurónu zostavili Warren S. McCulloch a Walter Pitts v roku 1943 [11]. V roku 1949 predstavil Donald O. Hebb vo svojej knihe *The Organization of Behavior* [8] princípy učenia neurónových sietí, spočívajúce v úprave sily spoja medzi




Obrázok 2.2: Všeobecná schéma umelého neurónu

dvoma neurónmi na základe opakovaného prenosu informácií medzi nimi. Frank Rosenblatt vynašiel v roku 1958 [12] *perceptron*, model učenia neurónu založený na metóde najväčšieho spádu (gradientnom zostupe). Obmedzenia tohoto modelu demonštrovali Marvin Minsky a Seymour Papert v knihe *Perceptrons* v roku 1969 [13]. Ďalším učiacim sa modelom založenom na metóde najväčšieho spádu bol ADALINE (Adaptive Linear Neuron), ktorý v roku 1960 predstavili Bernard Widrow a Marcian Hoff.

## 2.2 Neurón

Základným stavebným prvkom neurónovej siete je neurón, inšpirovaný biologickým neurónom. Všeobecný neurón má $n$ vstupov $x_i$ a jeden výstup $y$, ktorý môže byť napojený na vstupy ďalších neurónov, alebo môže byť priamo výstupom neurónovej siete. S každým vstupom $x_i$ je asociovaná váha $w_i$, ktorú je možné chápať ako silu daného spoja, t.j. mieru, akou vstup $x_i$ ovplyvňuje výstup neurónu. Vstupy neurónu môžeme súhrnne označiť ako vstupný vektor $\mathbf{x} = (x_1, x_2, ..., x_n)$. Podobne môžeme definovať váhový vektor $\mathbf{w} = (w_1, w_2, ..., w_n)$.

Výpočet neurónu je definovaný dvoma funkciami – Bázovou funkciou $f : \mathbb{R}^{2n} \to \mathbb{R}$, ktorá vstupnému vektoru $\mathbf{x}$ a váhovému vektoru $\mathbf{w}$ priradí reálne číslo $u$, nazývané *potenciál* neurónu, a aktivačnou funkciou $g : \mathbb{R} \to \mathbb{R}$, ktorá ďalej k potenciálu $u$ priradí výstup neurónu. Celý výpočet neurónu je teda možné popísať ako $y = g(f(\mathbf{x}, \mathbf{w}))$, kde $f$, resp. $g$ označuje bázovú, resp. aktivačnú funkciu. Táto situácia je ilustrovaná na obrázku 2.2.

Ako bázová funkcia $f$ sa typicky používa jeden z dvoch typov funkcií – *lineárna* alebo *radiálna* bázová funkcia. Možné použiteľné typy aktivačnej funkcie závisia od použitej bázovej funkcie – výberu týchto funkcií sa venuje nasledujúca sekcia.

### 2.2.1 Lineárna bázová funkcia

Lineárna bázová funkcia (LBF) je definovaná ako

$$u = \sum_{i=1}^{n} w_i x_i, \tag{2.1}$$

tzn. zodpovedá skalárnemu súčinu vektorov $\mathbf{x}$ a $\mathbf{w}$.

Uvedieme niekoľko aktivačných funkcií najčastejšie používaných s lineárnou bázovou funkciou, pričom ich grafy možno nájsť na obrázku 2.2.1.

Jednou z najjednoduchších aktivačných funkcií používaných s touto bázovou funkciou je skoková funkcia s prahom $\theta$:

$$f_{step}(u) = \begin{cases} a & \text{ak } u < \theta \\ b & \text{ak } u > \theta \\ y^{(n-1)} & \text{ak } u = \theta \end{cases}, \tag{2.2}$$

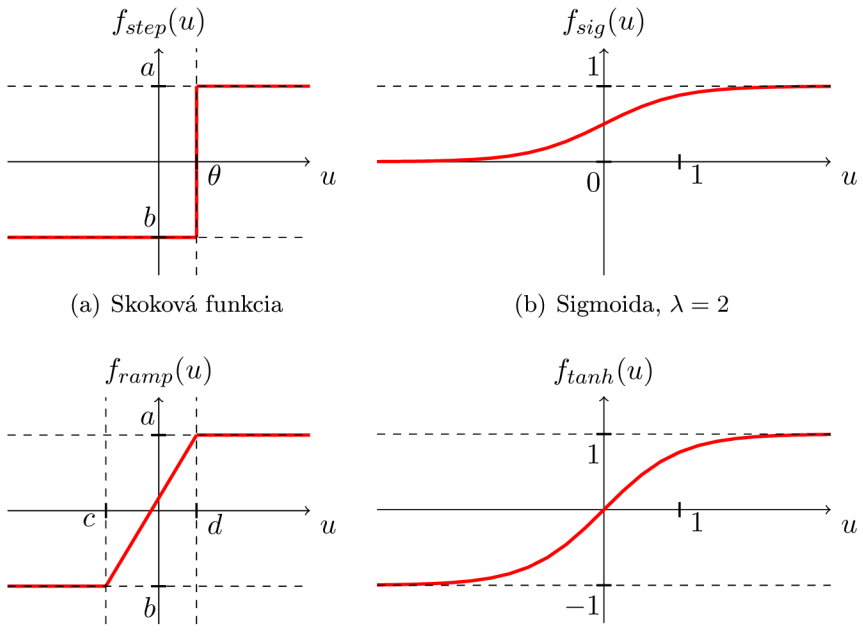
kde $y^{(n-1)}$ reprezentuje pôvodný výstup neurónu (tzn. nedôjde k žiadnej zmene) a $a$, $b$ sú výstupné hodnoty, často definované ako $a = 0$, $b = 1$ (binárny výstup) alebo $a = -1$, $b = 1$ (polárny výstup).

Parameter $\theta$ sa nazýva *prah* aktivačnej funkcie. Pre zjednodušenie výpočtu sa (nielen pri skokovej funkcii) často používa úprava, v ktorej sa do aktivačnej funkcie namiesto $\theta$ dosadí hodnota 0, a medzi vstupy do neurónu sa zaradí spoj s váhou $-\theta$ z imaginárneho neurónu s výstupom pevne nastaveným na hodnotu 1.

V prípade, že potrebujeme využiť spojitú aktivačnú funkciu, jednou z najjednoduchších možností je *rampová* (po častiach lineárna) funkcia:

$$f_{ramp}(u) = \begin{cases} a & \text{ak } u < c \\ b & \text{ak } u > d \\ a + \frac{b-a}{d-c}(u-c) & \text{ak } c \leq u \leq d \end{cases}, \tag{2.3}$$

kde $a$ a $b$ sú minimálna a maximálna hodnota funkcie, a $c$ a $d$ sú hranice intervalu, na ktorom funkcia lineárne rastie z najmenšej hodnoty na najväčšiu.




(a) Skoková funkcia

(b) Sigmoida, $\lambda = 2$

(c) Rampová funkcia

(d) Hyperbolický tangens, $\lambda = 1$

Obrázok 2.3: Grafy aktivačných funkcií používaných s LBF

Ďalšou bežne využívanou aktivačnou funkciou je *sigmoida*, nazývaná tiež *logistická funkcia*:

$$f(u) = a + \frac{b-a}{1+e^{-\lambda u+c}}. \tag{2.4}$$

Niekedy sa ako sigmoida označuje priamo tvar tejto funkcie s parametrami $a = 0$, $b = 1$ a $c = 0$:

$$f_{sig}(u) = \frac{1}{1+e^{-\lambda u}}, \tag{2.5}$$

kde $\lambda$ je vhodne zvolený parameter. Pojem *sigmoidálna funkcia* môže v rôznych zdrojoch označovať aj všeobecnejšiu triedu funkcií, napríklad merateľné funkcie $f$ spĺňajúce $\lim_{x\to-\infty} f(x) = 0$, $\lim_{x\to+\infty} f(x) = 1$ [5]. V tomto texte bude však pojem sigmoida označovať výlučne tvar uvedený v rovnici 2.5.

Poslednou aktivačnou funkciou, ktorú pre lineárnu bázovú funkciu uvedieme, je *hyperbolický tangens*, ktorý tiež umožňujeme prispôsobiť parametrom $\lambda$:

$$f_{tanh}(u) = \tanh(\lambda u). \tag{2.6}$$

Hyperbolický tangens je možné algebraicky popísať ako podiel hyperbolického sínusu a hyperbolického kosínusu, ktoré je ďalej možné popísať pomocou exponenciálnej funkcie:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.7}$$

Za zmienku stojí tiež vzťah medzi sigmoidou a hyperbolickým tangensom, ktorý je možné pomocou algebraickej definície hyperbolického tangensu ľahko dokázať:

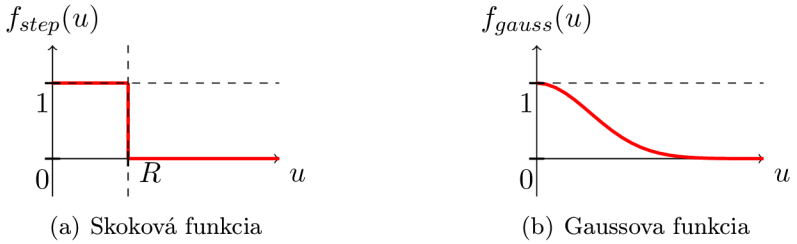
$$\frac{1}{1+e^{-x}} = \frac{1}{2}(1 + \tanh\frac{x}{2}). \tag{2.8}$$

### 2.2.2 Radiálna bázová funkcia

Radiálna bázová funkcia (RBF) je definovaná ako

$$u = \sqrt{\sum_{i=1}^{n}(x_i - w_i)^2}, \tag{2.9}$$

teda zodpovedá euklidovskej vzdialenosti vektoru $\mathbf{x}$ od vektoru $\mathbf{w}$ a dosahuje len nezáporné hodnoty. Rovnako ako pri LBF je možné používať aj nespojité, aj spojité varianty aktivačných funkcií. Ich grafy je možné vidieť na obrázku 2.4.




(a) Skoková funkcia (b) Gaussova funkcia

Obrázok 2.4: Grafy aktivačných funkcií používaných s RBF

Typickou nespojitou aktivačnou funkciou je napríklad skoková funkcia:

$$f_{step}(u) = \begin{cases} a & \text{ak } u \leq R \\ b & \text{ak } u > R \end{cases}, \tag{2.10}$$

pre výstupné hodnoty $a$ a $b$ a prah $R$. Na rozdiel od používania skokovej funkcie pri lineárnej bázovej funkcii však typicky platí $a > b$.

Možnou spojitou aktivačnou funkciou je napríklad nasledovná varianta Gaussovej funkcie:

$$f_{gauss}(u) = e^{-(u/\sigma)^2}, \tag{2.11}$$

s vhodne zvoleným parametrom $\sigma$.

## 2.3 Klasifikácia neurónových sietí

Neurónové siete je možné triediť do skupín na základe rôznych vlastností. Táto podkapitola najprv stručne popíše niektoré z týchto rozdelení a následne vytýči konkrétne typy neurónových sietí, ktorými sa táto práca zaoberá. Detailnejšiemu popisu týchto typov sa budú venovať ďalšie podkapitoly.

Z už spomínaných pojmov sa dajú neurónové siete rozdeliť napríklad podľa použitej bázovej či aktivačnej funkcie (hoci všetky neuróny jednej siete nemusia používať tú istú aktivačnú funkciu), či podľa vstupných a výstupných hodnôt (binárne, 0 alebo 1; bipolárne, -1 alebo 1; spojité; reálne čísla z určitého intervalu). Medzi ďalšie rozdelenia patrí napríklad rozdelenie sietí podľa architektúry, podľa spôsobu učenia, alebo podľa aplikácie.

### 2.3.1 Delenie podľa architektúry

Toto rozdelenie berie do úvahy systém prepojenia medzi jednotlivými neurónmi. Hoci všeobecne tieto spojenia nie sú obmedzené (ľubovoľný neurón môže a nemusí byť prepojený s ľubovoľným iným neurónom, či sebou samým), dajú sa tu vyzdvihnúť aspoň dve významné dichotómie:

- rozlíšenie medzi *vrstvovými* a *nevrstvovými* sieťami, tzn. či sú neuróny rozdelené do disjunktných vrstiev, ktoré sú do seba v poradí zapojené;
- rozlíšenie medzi *rekurentnými* a *nerekurentnými* sieťami, tzn. či sa výstupy v sieti šíria iba jedným smerom (nerekurentné), alebo sa môžu niektoré výstupy šíriť aj proti tomuto smeru (rekurentné).

Ďalej vymenujme niektoré z najbežnejších špeciálnych prípadov:

a) **Plne prepojená sieť** – Z každého neurónu vedie spoj do seba samého, aj do každého iného neurónu. Váhy je možné reprezentovať plnou štvorcovou maticou;

b) **Plne prepojená symetrická sieť** – Plne prepojená sieť s pridanou podmienkou, že $w_{ij} = w_{ji}, \forall i, j \leq n$. Na reprezentáciu váh postačuje trojuholníková matica;

c) **Vrstvová sieť** – Neuróny sú rozdelené do zoradených vrstiev, pričom žiadne neuróny neovplyvňujú neuróny predchádzajúcich vrstiev;

d) **Acyklická sieť** – Vrstvová sieť, v ktorej neexistujú spoje medzi neurónmi rovnakej vrstvy;

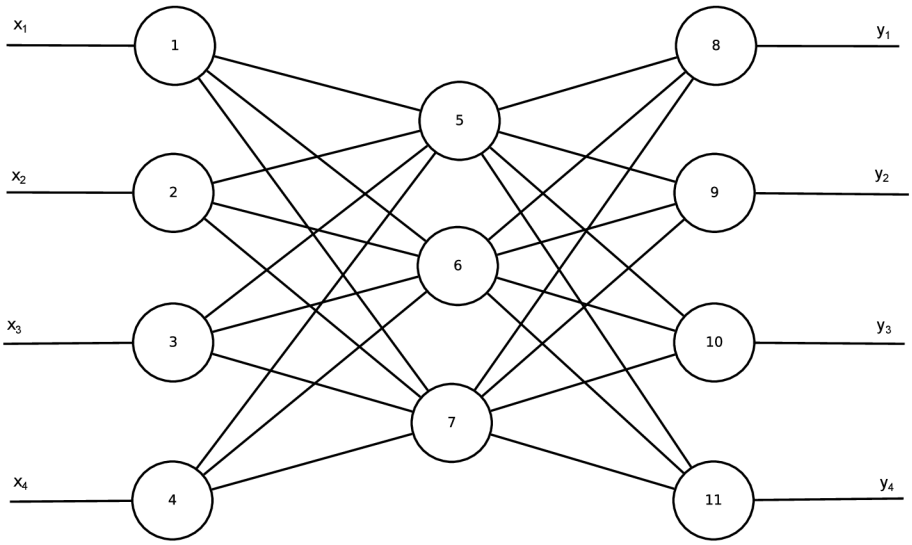
e) **Dopredná sieť** – Acyklická sieť, v ktorej existujú väzby iba medzi susednými sieťami.

Stojí za zmienku, že b je špeciálnym prípadom a, kým e je špeciálnym prípadom d, čo je zase špeciálnym prípadom c.

Modely d a e sa niekedy označujú ako siete typu *Feedforward*, podľa jediného smeru toku informácií v sieti. V tejto práci budeme týmto pojmom označovať úplné dopredné siete využívajúce pre učenie algoritmus Backpropagation, viz. sekcia 2.4.2

## 2.4 Neurónové siete typu Feedforward

Siete typu *Feedforward*, často spájané s učiacim algoritmom Backpropagation, sú (aspoň na začiatku 21. storočia) jedným z najpoužívanejších modelov neurónových sietí (Podľa určitých odhadov sú tieto siete s algoritmom Backpropagation využívané vo viac ako 90% komerčných aplikácií [14]). Neuróny sú v tomto modeli rozdelené do vrstiev – jednej *vstupnej*, 0-n *skrytých* a jednej *výstupnej*. Počet skrytých vrstiev nie je obmedzený, avšak pri sietiach typu Feedforward sa málokedy využívajú viac než dve skryté vrstvy. Týmto sa tieto siete značne odlišujú od tzv. *Deep sietí*, ktoré práve naopak využívajú väčšie počty skrytých vrstiev. Sieť je teda z pohľadu architektúry možné opísať ako doprednú sieť, definovanú v 2.3.1. Jednoduchá Feedforward sieť je ilustrovaná na obrázku 2.5.




Obrázok 2.5: Sieť Feedforward pozostávajúca z 11 neurónov. Neuróny 1-4 tvoria vstupnú vrstvu, neuróny 5-7 tvoria skrytú vrstvu, a neuróny 8-11 tvoria výstupnú vrstvu. Informácie sa po spojoch medzi neurónmi šíria zprava doľava, do vstupnej vrstvy je vedený vstupný vektor $\mathbf{x}$, z výstupnej vrstvy vychádza výstupný vektor $\mathbf{y}$.

Slovo „feedforward“ označuje algoritmus využívaný pre výpočet výstupu siete – vstup je najprv privedený na neuróny vstupnej vrstvy, z ktorej sú signály ďalej predané prvej skrytej vrstve (alebo priamo výstupnej vrstve, v prípade že sieť žiadnu skrytú vrstvu nemá), a výpočet postupne prebieha po jednotlivých vrstvách, kým sa nevypočítajú výstupy neurónov výstupnej vrstvy, ktoré predstavujú výstup siete. Algoritmus Feedforward je bližšie opísaný v sekcii 2.4.2, kým algoritmu Backpropagation sa venuje sekcia 2.4.3.

### 2.4.1 Použité symboly

Sieť typu feedforward pozostáva z neurónov rozdelených do $N_l \geq 2, N_l \in \mathrm{N}$ vrstiev číslovaných od jednotky, pričom vrstva 1 je vstupná vrstva, vrstva $N_l$ je výstupná vrstva, a všetky vrstvy medzi nimi sú skryté vrstvy. Výstup každého neurónu vrstvy $l$ je privedený na vstup každého neurónu vrstvy $l + 1$, pre $l \in \langle 1, N_l - 1 \rangle$.

Počet neurónov vstupnej (resp. výstupnej) vrstvy by mal zodpovedať šírke vstupu (resp. výstupu) riešenej úlohy.

### 2.4.2 Algoritmus Feedforward

Algoritmus Feedforward predstavuje relatívne jednoduchý princíp dopredného výpočtu využívaného v Backpropagation sieťach (a s malou úpravou v Shortcut sieťach):

1. Priveď vstupný vektor $\mathbf{x}$ na vstupnú vrstvu (vrstva 1);
2. Pre každú vrstvu $l := 2 \,..\, N_l$ vypočítaj výstupy neurónov na základe vstupov z predchádzajúcej vrstvy;
3. Výstupná vrstva (vrstva $N_l$) predstavuje výstupný vektor $\mathbf{y}$ siete.

### 2.4.3 Algoritmus Backpropagation

Algoritmus Backpropagation spočíva v postupnej aplikácii vzorov z trénovacej množiny a následnom prispôsobovaní váh siete pre priblíženie výstupov k požadovaným výstupom. Tieto zmeny váh sa šíria postupne od výstupnej vrstvy dozadu. Pre určitý vzor sa váhy opakovane upravujú, kým odchýlka výstupu siete od vzorového výstupu neklesne pod určitú hodnotu. Táto časť algoritmu sa nazýva *vnútorný cyklus*, v kontraste s tzv. *vonkajším cyklom*, ktorý iteruje po jednotlivých vzoroch z trénovacej množiny, kým sieť nedosahuje uspokojujúce výsledky pre všetky vzory, alebo nebol dosiahnutý limit iterácií. Jedna iterácia vonkajšieho cyklu cez celú trénovaciu množinu sa nazýva *epocha*.

Skôr ako prejdeme k detailom úprav váh, uveďme kostru algoritmu Backpropagation:

**Vonkajší cyklus:** Opakovane volaj vnútorný cyklus pre všetky prvky $(\mathbf{x}, \mathbf{t})$ trénovacej množiny, kým sieť nemapuje všetky vzory na správne výstupy, alebo kým nevyprší stanovený limit epoch;

**Vnútorný cyklus:** Kým sieť nemapuje správne daný vzorový vstup $\mathbf{x}$ na jemu zodpovedajúci výstup $\mathbf{t}$, opakuj nasledovné:

1. Priveď vzor $\mathbf{x}$ na vstup siete a vypočítaj pomocou algoritmu Feedforward jej výstup $\mathbf{y}$;
2. Ak je výstup $\mathbf{y}$ dostatočne blízky vzorovému výstupu $\mathbf{t}$, ukonči vnútorný cyklus, inak pokračuj ďalším krokom;
3. Šírením odchýlky postupne upravuj váhy spojov smerujúce do jednotlivých neurónov $j$ vrstvy $l := N_l \,..\, 2$, teda postupujúc od výstupnej vrstvy dozadu až po prvú skrytú vrstvu.

Ďalej uveďme konkrétne výpočty zabezpečujúce úpravu váh. Pre detailnejšie odvodenie využívaných vzorcov viz. napríklad [14].

Chybu siete $E$, tzn. akúsi odchýlku vypočítaného výstupu od požadovanej hodnoty (vzorového výstupného vektora) môžeme definovať napríklad nasledovne:

$$E = \frac{1}{2}\sum_{i=1}^{n}(t_i - o_i)^2, \tag{2.12}$$

kde $n$ je počet neurónov výstupnej vrstvy, $t_i$ je $i$-tý prvok vzorového výstupného vektoru a $o_i$ je výstup $i$-tého neurónu výstupnej vrstvy.

Zmenami váh sa snažíme túto chybu minimalizovať, zmenu konkrétnej váhy teda môžeme odvodiť z parciálnej derivácie chyby podľa tejto váhy:

$$\Delta w_{ij} = -\eta\frac{\partial E}{\partial w_{ij}}, \tag{2.13}$$

kde $\eta$ je vhodne zvolená kladná konštanta nazývaná *koeficient učenia*. Podľa reťazového pravidla pre parciálne derivácie môžeme deriváciu v predchádzajúcom vzorci vyjadriť ako

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial u_j}\frac{\partial u_j}{\partial w_{ij}}, \tag{2.14}$$

kde $u_j$ predstavuje potenciál neurónu $j$, teda cieľového neurónu váhy $w_{ij}$.

Pre lineárnu bázovú funkciu platí

$$\frac{\partial u_j}{\partial w_{ij}} = o_i, \tag{2.15}$$

avšak hodnota $\frac{\partial E}{\partial u_j}$ závisí od použitej aktivačnej funkcie, a od skutočnosti, či sa neurón $j$ nachádza vo výstupnej alebo v skrytej vrstve.

Pre jednoduchosť zaveďme označenie

$$\delta_j \equiv -\frac{\partial E}{\partial u_j}. \tag{2.16}$$

Zmenu váhy $w_{ij}$ môžeme teda vypočítať ako

$$\Delta w_{ij} = \eta\delta_j o_i, \tag{2.17}$$

zostáva nám však ešte odvodiť hodnotu $\delta_j$ pre jednotlivé neuróny $j$.

Parciálnu deriváciu vo vzorci 2.16 opäť rozložíme pomocou reťazového pravidla:

$$\frac{\partial E}{\partial u_j} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial u_j} \tag{2.18}$$

Deriváciu chyby podľa výstupu neurónu $j$ určíme podľa jeho príslušnosti do výstupnej, resp. skrytej vrstvy nasledovne [14]:

$$\frac{\partial E}{\partial o_j} = \begin{cases} o_j - t_j & \text{ak je } j \text{ vo výstupnej vrstve} \\ -\sum\limits_k \delta_k w_{jk} & \text{ak je } j \text{ v skrytej vrstve} \end{cases} \tag{2.19}$$

Vo vzorci pre výstupnú vrstvu označuje $t_j$ $j$-tý prvok požadovaného výstupného vektoru, a vzorec pre skrytú vrstvu obsahuje sumu nad všetkými neurónmi $k$ vrstvy okamžite nasledujúcej za vrstvou neurónu $j$ (tzn. nad všetkými neurónmi $k$, do ktorých vedie z neurónu $j$ spoj).

Ďalej zostáva vo vzorci 2.18 činiteľ $\frac{\partial o_j}{\partial u_j}$, ktorého hodnota závisí na aktivačnej funkcii $f$ daného neurónu, a zodpovedá jej derivácii podľa vstupu, teda $\frac{\partial f(u_j)}{\partial u_j}$. Pre sigmoidu (viz. rovnica 2.5) má táto derivácia hodnotu

$$\frac{\partial o_j}{\partial u_j} = -\lambda o_j(1 - o_j), \tag{2.20}$$

a pre hyperbolický tangens (viz. rovnica 2.6)

$$\frac{\partial o_j}{\partial u_j} = -\lambda(1 - o_j^2). \tag{2.21}$$

Derivácia rampovej funkcie (viz. rovnica 2.3) má nenulovú hodnotu iba na intervale medzi hodnotami $c$ a $d$, preto je pri učení neurónov využívajúcich túto aktivačnú funkciu potrebné dávať pozor, aby sa ich výstupy držali v týchto medziach [17]:

$$\frac{\partial o_j}{\partial u_j} = \begin{cases} \frac{b-a}{d-c} & \text{ak } c \leq o_j \leq d, \\ 0 & \text{ak } o_j < c \text{ alebo } d < o_j. \end{cases} \tag{2.22}$$

Skoková funkcia má deriváciu nulovú na celom definičnom obore (okrem prahu, kde derivácia nie je definovaná), preto Backpropagation nie je vhodným algoritmom pre učenie sietí, ktoré obsahujú neuróny využívajúce túto aktivačnú funkciu.

V $n$-tej iterácii teda určíme novú váhu pričítaním nejakého prírastku alebo úbytku k starej:

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

Zmenu váhy $\Delta w_{ij}^{(n)}$ určíme podľa rovnice 2.17, pričom môžeme tiež zaviesť závislosť od hodnoty zmeny váhy v predchádzajúcom kroku:

$$\Delta w_{ij}^{(n)} = \eta \delta_j o_i + \alpha \Delta w_{ij}^{(n-1)},$$

Kde $\eta$ a $\alpha$ sú kladné konštanty, presnejšie $\eta$ je tzv. *koeficient učenia* (learning rate) a $\alpha$ je tzv. *koeficient zotrvačnosti* (momentum rate), $o_i$ je výstup neurónu $i$, z ktorého vychádza hrana s upravovanou váhou, a $\delta_j$ je hodnota určujúca mieru zmeny váhy smerujúcej do neurónu $j$, definovaná v rovnici 2.16. Hodnota $\delta_j$, ako bolo demonštrované, závisí od aktivačnej funkcie a polohy neurónu $j$. Uvedieme príklad pre sigmoidu, $\lambda = 1$:

a) Neurón $j$ je vo výstupnej vrstve:

$$\delta_j = o_j(1 - o_j)(t_j - o_j),$$

kde $t_j$ je $j$-tý člen vzorového výstupného vektora $\mathbf{t}$;

b) Neurón $j$ je v skrytej vrstve $l$:

$$\delta_j = o_j(1 - o_j) \sum_k \delta_k w_{jk}^{(n)},$$

pre všetky neuróny $k$ vrstvy $l+1$. Všimnime si, že tento vzorec využíva váhy z kroku $n$.

### 2.4.4 Anomálie štruktúry siete

Siete typu Feedforward implementované v tejto práci môžu popri štruktúre definovanej v predchádzajúcich lekciách obsahovať aj určité anomálie štruktúry, resp. (simulované) poruchy siete. Ide o nasledujúce prípady:

- **Neprirodzené spoje** – V klasickej definícii Feedforward siete vedú spoje z daného neurónu výlučne do neurónov nasledujúcej vrstvy, v tejto implementácii však môže užívateľ porušiť túto štruktúru definovaním spoja medzi neurónmi, ktoré by typicky prepojené neboli. Túto skutočnosť treba brať do úvahy pri výpočte potenciálov neurónov;
- **Fixné výstupy neurónov** – Neurónom je možné nastaviť fixnú výstupnú hodnotu, ktorú budú vysielať na výstup bez ohľadu na ich vstupy. V dôsledku je potrebné buď zabrániť ich prepisovaniu pri výpočte, alebo ich po prepise znovu vrátiť na pôvodnú hodnotu;
- **Fixné váhy spojov** – Podobne ako výstupom neurónov je možné váham nastaviť pevnú hodnotu aj váham. Tu je potrebné zabrániť ich zmenám pri učení siete;
- **Odobrané neuróny** – Vybraný neurón je možné zo siete odobrať, čo sa na sieti prejaví tak, že výstup neurónu nebude nijakým spôsobom ovplyvňovať výstupy naňho napojených neurónov, tzn. ide o situáciu identickú s nastavením výstupu neurónu na fixnú nulovú hodnotu;
- **Odobrané spoje** – Je možné tiež zrušiť spoj implicitne vytvorený medzi dvoma neurónmi, čo je podobne ako v predchádzajúcom bode identické s nastavením váhy na fixnú nulovú hodnotu.

Tieto prípadné zmeny štruktúry vyžadujú určité úpravy implementácie algoritmov Feedforward a Backpropagation, preto je im potrebné pri návrhu venovať špeciálnu pozornosť.

## 2.5 Neurónové siete typu Shortcut

Podobne ako je možné siete Feedforward definovať ako úplné dopredné siete, siete typu *Shortcut* sú úplné acyklické siete. Ich neuróny sú teda taktiež rozdelené do vrstiev, ale z jedného neurónu konkrétnej vrstvy vedú spoje do všetkých nasledujúcich, nie iba do susednej, ako je tomu pri sietiach Feedforward.

Na výpočet (resp. učenie) sietí Shortcut sa využíva upravená forma algoritmu Feedforward (resp. Backpropagation). Táto úprava spočíva v prípade Feedforwardu v tom, že bázová funkcia neurónu závisí na neurónoch všetkých predchádzajúcich vrstiev, nie iba okamžite predchádzajúcej. Pri Backpropagation je potrebné upravovať podstatne viac váh, tieto úpravy však prebiehajú rovnakým spôsobom, ako pre siete Feedforward (viz. sekcia 2.4.3) – popísané vzorce sú stále platné, ak správne určíme zdrojový a cieľový neurón spoja, iba nepožadujeme, aby boli zo vzájomne susediacich vrstiev.

## 2.6 Hopfieldove siete

Už na prvý pohľad je zrejmý rozdiel medzi *Hopfieldovými* sieťami a predchádzajúcimi dvoma typmi spočívajúci v architektúre siete – Hopfieldove siete sú plne prepojené symetrické siete,

tzn. všetky neuróny sú vzájomne prepojené, čím tiež odpadá potreba rozdeľovať neuróny do vrstiev, a spoje medzi dvoma danými neurónmi v oboch smeroch majú rovnakú váhu (čiže pre neuróny $i$ a $j$ platí $w_{ij} = w_{ji}$).

Z tejto architektúry taktiež vyplýva, že Hopfieldove siete sú rekurentné. Ich výpočet teda nie je možné vykonávať po vrstvách, ale je potrebné postupne vyhodnocovať výstupy jednotlivých neurónov, kým sa sieť nedostane do stabilného stavu.

Hopfieldovu sieť je možné opísať ako asociatívnu pamäť – jednotlivé prvky trénovacej sady sú v sieti zakódované ako stabilné stavy siete, a pri výpočte bude stav siete pre určitý vstupný vektor konvergovať k najbližšiemu z týchto stabilných stavov.

### 2.6.1 Učenie Hopfieldových sietí

Pre sieť s $n$ neurónmi a trénovacou sadou $T$ pozostávajúcou z $n$-prvkových vektorov $s$ je možné váhu spoja medzi neurónmi $i$ a $j$, $i, j \leq n$, určiť pomocou nasledovného vzorca:

$$w_{ij} = \begin{cases} 0 & \text{ak } i = j \\ \sum\limits_{s \in T} x_i^{(s)} x_j^{(s)} & \text{ak } i \neq j \end{cases},$$

kde $x_i^{(s)}$ je $i$-tý prvok vektoru $s$ trénovacej sady. Výpočet zrejme spĺňa symetriu váh, teda platí $w_{ij} = w_{ji}$, a je ho možné zapísať pomocou matíc ako $\mathbf{W} = \mathbf{S}\mathbf{S}^{\mathrm{T}}$, kde $\mathbf{W}$ je matica váh rozmerov $n \times n$, a

$$\mathbf{S} = \begin{pmatrix} \mathbf{s}_1 & \mathbf{s}_2 & \cdots & \mathbf{s}_i & \cdots & \mathbf{s}_k \end{pmatrix}$$

je matica rozmerov $n \times k$ ($k$ je počet vektorov trénovacej sady), ktorej stĺpce predstavujú vektory $\mathbf{s}_i$ trénovacej sady, a $\mathbf{S}^{\mathrm{T}}$ je transponovaná matica k matici $\mathbf{S}$.

Prah neurónu $i$ sa dá ďalej vypočítať pomocou sumy váh jeho spojov so všetkými ostatnými neurónmi $j$ [2]:

$$\theta_i = \frac{1}{2} \sum_{j=1}^{n} w_{ij} \tag{2.23}$$

### 2.6.2 Simulácia Hopfieldových sietí

Výpočet výstupu Hopfieldovej siete spočíva v postupnom výpočte výstupov jednotlivých neurónov, kým sa stav siete neustáli. Je ho možné opísať nasledujúcimi krokmi:

1. Privedieme na vstup siete vstupný vektor $\mathbf{x}$ (teda nastavíme výstup neurónu $i$ na hodnotu $x_i$, teda $i$-tý prvok vstupného vektora, pre všetky neuróny $i$);
2. Opakovane vykonávame výpočet výstupov jednotlivých neurónov, pričom jednu iteráciu tohoto cyklu nazývame epochou. V prípade, že sa v danej epoche nezmenil výstup žiadneho neurónu (resp. zmeny boli menšie ako určitá predvolená hodnota $\epsilon$), cyklus ukončíme – stav siete sa ustálil;
3. Prečítame výstupy jednotlivých neurónov, ktoré teraz predstavujú výstup siete.

Zostáva popísať, akým spôsobom prebieha vyhodnocovanie výstupov neurónov v rámci jednej epochy. Výstupy vyhodnocujeme po jednom neuróne a nový výstup neurónu je použitý vo výpočtoch ďalších výstupov. Požadujeme, aby boli neuróny na vyhodnotenie vyberané náhodne s rovnomerným rozložením pravdepodobnosti, a aby bol v rámci jednej epochy vyhodnotený každý neurón, aby bolo možné detekovať konvergenciu výpočtu. Jednou

z možností je postupne náhodne vyberať neuróny, kým nebola hodnota každého neurónu vyhodnotená aspoň raz. Táto metóda spĺňa stanovené podmienky [14].

## 2.7 Kohonenove siete

*Kohonenove siete* sú z hľadiska architektúry dvojvrstvové dopredné siete, ktoré využívajú tzv. *súťažívé učenie* – susediace neuróny spolu navzájom súťažia, pričom váhy víťazných neurónov sú zosilňované, kým váhy porazených neurónov sú zoslabené, prípadne ostávajú nezmenené.

Kohonenova sieť pozostáva z dvoch vrstiev – vstupnej vrstvy, a výstupnej vrstvy, nazývanej tiež *Kohonenova vrstva*, ktorá pozostáva z tzv. *Kohonenových neurónov*.

Kohonenove siete sú *topologicky organizované* – je potrebné pre Kohonenovu vrstvu definovať topológiu, ktorá definuje, akým spôsobom sa meria vzdialenosť medzi neurónmi. *Okolím* určitého neurónu $i$ s polomerom $r$ rozumieme všetky neuróny, ktorých vzdialenosť od neurónu $i$ je nanajvýš $r$. Príklady okolí pre štvorcovú topológiu, ktorá bude definovaná o niekoľko odstavcov nižšie, je možné vidieť na obrázku 2.6.

Jedna z najjednoduchších možností je jednorozmerné usporiadanie, v ktorom sú neuróny umiestnené po poradí vedľa seba, pričom susediace neuróny majú medzi sebou vzdialenosť 1 a pre ostatné neuróny táto hodnota zodpovedá počtu neurónov medzi nimi zvýšenému o 1.

Kohonenovu vrstvu je však možné organizovať aj do viacrozmerných topológií. Špeciálne sa budeme venovať dvojrozmernej *štvorcovej* topológii. Neuróny sú v nej rozmiestnené do pravouhlej mriežky (viz. obrázok 2.6), pričom polohu daného neurónu je možné popísať jednoducho napríklad dvoma celočíselnými hodnotami – horizontálnou vzdialenosťou $x$ od ľavého okraja mriežky a vertikálnou vzdialenosťou $y$ od horného okraja mriežky (teda napríklad neurón v ľavom hornom rohu by mal súradnice $x = 1$, $y = 1$). Vzdialenosťou medzi dvoma neurónmi so súradnicami $(x_i, y_i)$ a $(x_j, y_j)$ budeme potom definovať ako $\max\{|x_i - x_j|, |y_i - y_j|\}$ (v iných zdrojoch môže byť vzdialenosť na tejto topológii definovaná inak, napríklad ako $|x_i - x_j| + |y_i - y_j|$).

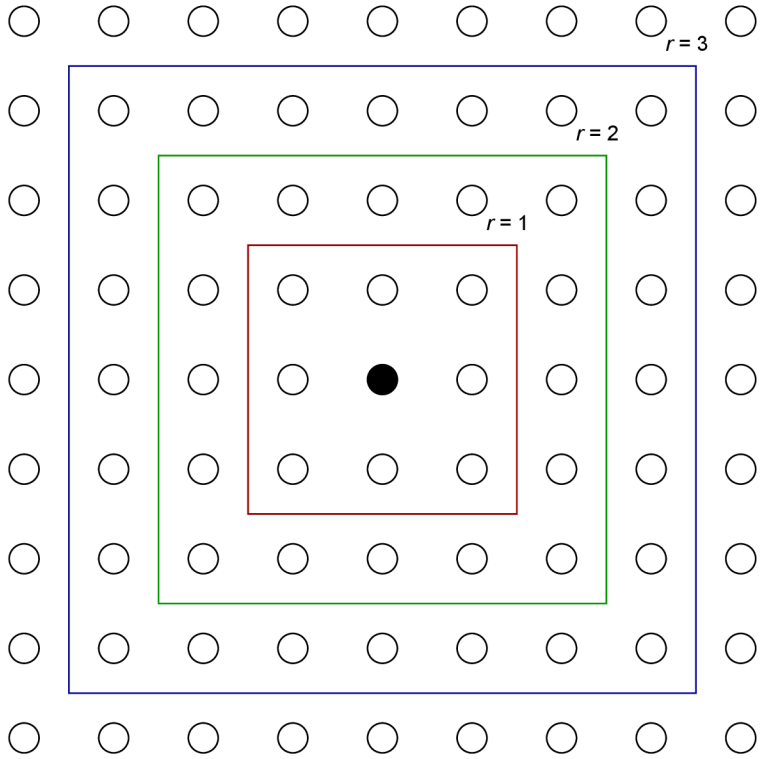
### 2.7.1 Učenie Kohonenových sietí

Jedna epocha učenia prebieha aplikáciou nasledujúceho postupu na všetky dvojice vzorového vstupu a výstupu $(\mathbf{x}, \mathbf{t})$ trénovacej sady:

1. Privedieme vstupný vektor $\mathbf{x}$ na vstup siete;
2. Pre každý neurón $j$ Kohonenovej vrstvy vyhodnotíme vzdialenosť $d_j$ vstupného vektoru od váhového vektoru $\mathbf{w_j} = (w_{1j}, w_{2j}, \ldots, w_{kj})$ prisluhujúceho neurónu $j$, kde $k$ je počet neurónov vstupnej vrstvy, podľa nasledovného vzorca:

$$d_j = \sqrt{\sum_{i=1}^{n} (x_i - w_{ij})^2}, \tag{2.24}$$

kde $x_i$ je $i$-tý prvok vstupného vektoru, a suma je cez všetky neuróny vstupnej vrstvy. Takto vypočítanú vzdialenosť $d_j$ možno chápať ako výstup radiálnej bázovej funkcie, definovanej v rovnici 2.9;




Obrázok 2.6: Ilustrácia štvorcového topologického usporiadania Kohonenovej vrstvy. Červený, zelený a modrý štvorec ohraničujú po rade okolia čierneho neurónu s hodnotami $r = 1$, 2 a 3.

3. Vyberieme Kohonenov neurón s najmenšou vzdialenosťou $d_j$. Tento neurón je víťazným neurónom;

4. Upravíme váhy všetkých neurónov $j$ v okolí víťazného neurónu (vrátane víťazného neurónu) nasledovne:

$$\mathbf{w}_j^{(n+1)} = \mathbf{w}_j^{(n)} + \eta(\mathbf{x}^{(n)} - \mathbf{w}_j^{(n)}), \tag{2.25}$$

kde $n$, resp. $n + 1$ predstavuje stav siete pred, resp. po zmene, a $\eta$ je vhodne zvolený *parameter učenia*. Okolie neurónu tvoria neuróny v okruhu definovanom vopred určeným parametrom $r$ okolo víťazného neurónu a topológiou siete. Stojí za zmienku, že pre $\eta = 0$ by váhy zostali nezmenené, a pre $\eta = 1$ by bol váhový vektor prepísaný vstupným vektorom. Neurónom mimo okolia sa váhy nemenia.

Učenie Kohonenových sietí spočíva v opakovaní týchto epoch, kým sa sieť nedostane do stabilného stavu, t.j. kým nenastane epocha, v ktorej nebolo nutné vykonávať zmeny, alebo boli zmeny menšie ako určitá predvolená požadovaná hodnota $\epsilon$.

Parametre $\eta$ a $r$ je odporúčané postupne znižovať.

# Kapitola 3

# OpenCL

Ako bolo ilustrované v kapitole 2, algoritmy pre simuláciu a učenie uvedených typov neurónových sietí pozostávajú prevažne z jednoduchých maticových a vektorových operácií (súčet a súčin matíc, sčítanie vektorov, násobenie matice konštantou ...), prípadne operácií ako súčet $N$ čísel, či nájdenie najmenšieho z $N$ čísel. Tieto operácie sú relatívne ľahko paralelizovateľné, je preto vhodné akcelerovať ich výpočet využitím GPU. Táto kapitola sa venuje popisu technológie využitej pre tento účel, kým detaily akcelerácie sú bližšie opísané v kapitole 5, špeciálne v sekcii 5.2.

## 3.1 OpenCL

*OpenCL* (Open Computing Language) je otvorený štandard pre paralelné programovanie na rôznych platformách, pôvodne vyvíjaný spoločnosťou Apple Inc. a aktuálne spravovaný konzorciom Khronos Group[1]. Hoci OpenCL umožňuje programovanie rôznych typov zariadení, v tejto práci sa zameriame predovšetkým na grafické procesory.

OpenCL poskytuje abstrakciu nad hardvérom, ktorá programátorovi umožňuje využívať zariadenia ako grafické procesory pre paralelizovanie výpočtov bez potreby znalosti konkrétneho typu zariadenia. Pre tento účel definuje abstraktnú architektúru zariadení, popísanú v sekcii 3.2. OpenCL pozostáva z dvoch častí:

- API pre koordináciu paralelných výpočtov, ktoré môžu všeobecne prebiehať aj na viacerých zariadeniach, a zabezpečuje komunikáciu zariadenia s hostiteľským prostredím;
- Programovacieho jazyka založeného na podmnožine ISO C99, ktorý umožňuje implementáciu výpočtov prebiehajúcich v cieľovom zariadení.

Štandard OpenCL definuje API pre C a C++, existujú však aj rozhrania pre iné jazyky (Python[2], Java[3], ...), udržiavané tretími stranami.

## 3.2 Architektúra OpenCL

Výpočet prebieha v hostiteľskom prostredí a aspoň jednom OpenCL zariadení (*device*). Hostiteľské prostredie riadi komunikuje s OpenCL device a riadi ich výpočet. V kontexte

[1] https://www.khronos.org/opencl/
[2] http://mathema.tician.de/software/pyopencl/
[3] http://www.jocl.org/

tejto práce bude OpenCL device vždy GPU.

Zariadenie je delené na jednu alebo viac *compute units*, ktoré sa ďalej delia na *processing elements*. Hostiteľské prostredie vydáva príkazy, ktoré zariadenie vykonáva, pričom výpočet prebieha na úrovni processing elements.

### 3.2.1 Výpočetný model

OpenCL program pozostáva z dvoch častí – z *kernelov*, ktoré bežia na OpenCL zariadeniach, a hostiteľského programu, ktorý spravuje OpenCL zariadenia a spúšťanie kernelov.

Kernely sú procedúry napísané v jazyku OpenCL pre cieľové zariadenia. Pri zadaní príkazu na spustenie kernelu musí hostiteľský program definovať $N$-rozmerný indexový priestor úlohy ($N = 1, 2$ alebo 3). Kernel je následne vykonaný pre každý bod indexového priestoru. Jedna takáto instancia kernelu sa nazýva *work-item* a je identifikovaná svojim bodom v indexovom priestore. OpenCL umožňuje jednotlivým work-itemom pristupovať k svojim indexom pomocou vstavaných funkcií.

Jednotlivé work-itemy sú ďalej organizované do skupín nazývaných *work-groups*. Tieto skupiny majú rovnako ako work-itemy priradené svoje indexy, a jednolitvé work-itemy majú v rámci skupín priradené lokálne indexy. V rámci jednej work-group zdieľajú work-itemy lokálnu pamäť a môžu využívať dostupné synchronizačné procesy. Výpočty jednotlivých work-itemov danej work-group prebiehajú paralelne na processing elementoch jednej compute unit [15].

### 3.2.2 Pamäťový model

Pamäť dostupná work-itemom pri výkone kernelov je organizovaná do nasledovných štyroch regiónov:

- *globálna pamäť* (*global memory*), ktorá je dostupná všetkým work-itemom vo všetkých skupinách;
- *konštantná pamäť* (*constant memory*), ktorá je spravovaná hostiteľským prostredím, a jej obsah sa počas výkonu kernelu nemení;
- *lokálna pamäť* (*local memory*), spoločná pre work-itemy jednej konkrétnej work-group;
- *súkromná pamäť* (*private memory*), dostupná iba jednému konkrétnemu work-itemu.

Program hostiteľského prostredia vytvára *pamäťové objekty* (*memory objects*) v globálnej pamäti, a vydáva príkazy, ktoré s nimi operujú. OpenCL rozlišuje dva typy pamäťových objektov – *buffer* a *image*. Táto práca využíva výlučne buffery, ktoré predstavujú jednorozmerné pole prvkov, kým image predstavuje viacrozmerné dáta.

## 3.3 Programovanie OpenCL

Pri programovaní OpenCL aplikácie je v hostiteľskom programe potrebné najprv vytvoriť kontext, a pre jednotlivé zariadenia vytvoriť fronty príkazov. API hostiteľského prostredia ďalej umožňuje vytvárať programy pre zariadenia, a pamäťové objekty. Z vytvorených programov sú ďalej vytvárané kernely, ktorých spustenie je spolu s manipuláciou pamäťových objektov možné zadávať prostredníctvom vopred vytvorených front príkazov.

Pre bližšiu ilustráciu princípov OpenCL uveďme kernel pre výpočet súčtu dvoch vektorov:

Zdrojový kód 3.1: Vektorový súčet

```
__kernel void vector_sum(__global float* a,
                         __global float* b,
                         __global float* c) {
  int i = get_global_id(0);

  c[i] = a[i] + b[i];
}
```

# Kapitola 4

# Návrh

Táto kapitola popisuje vysokoúrovňový návrh aplikácie. Detailom implementácie jednotlivých častí, obzvlášť algoritmom simulácie výpočtu a učenia neurónových sietí, sa venuje kapitola 5.

## 4.1 Požiadavky na aplikáciu

Cieľom práce je vytvoriť aplikáciu s jednoduchým textovým rozhraním, ktorá umožní trénovať, simulovať a testovať vybrané typy neurónových sietí. Veľmi dôležitou požiadavkou je tiež rýchlosť výpočtu – pre tento účel by mala byť aplikácia akcelerovaná pomocou GPU.

Ďalej je potrebné umožniť úpravy štruktúry siete, ako pridávanie a odoberanie spojov medzi neurónmi, či nastavenie neurónu alebo spoja na pevnú hodnotu. Táto vlastnosť aplikácie umožní užívateľovi testovať odolnosť sietí voči poruchám.

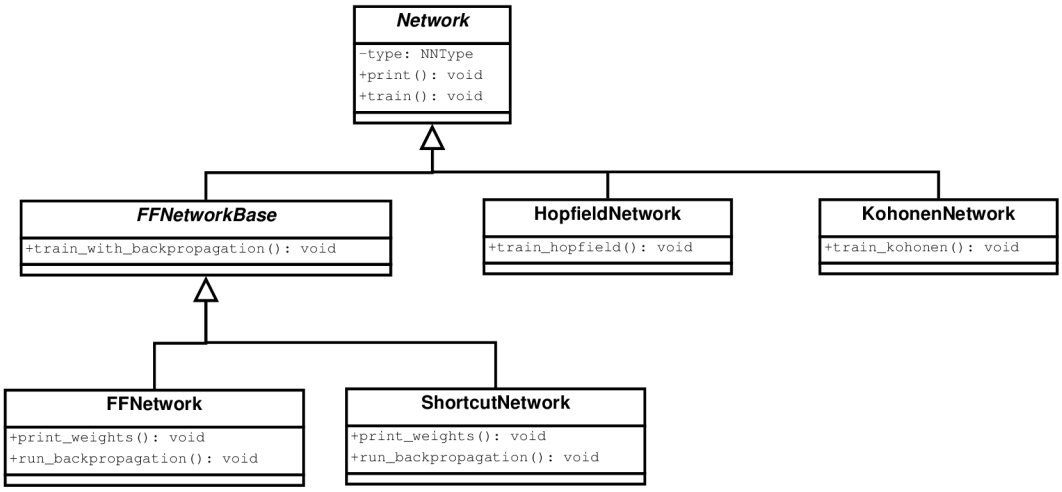
Aplikácia by mala tiež umožniť ľahkú rozšíriteľnosť – predovšetkým možnosť pridávania ďalších typov neurónových sietí, a ďalších algoritmov učenia.

## 4.2 Objektový návrh

Požiadavka na implementáciu podobných operácií (simulácia, učenie, testovanie) s rôznymi typmi neurónových sietí je v objektovom návrhu veľmi prirodzene realizovateľná hierarchiou tried, ktoré jednotlivé typy implementujú. Abstraktná trieda `Network` poskytuje spoločné rozhranie pre požadované operácie (učenie siete, výpočet siete, výpis siete do súboru ...) a atribúty ako je typ siete, pričom jej potomkovia implementujú funkcionalitu špecifickú pre konkrétne typy sietí – `FFNetwork`, `ShortcutNetwork`, `HopfieldNetwork`, `KohonenNetwork`. Vzhľadom na blízky vzťah medzi sieťami typu Feedforward a Shortcut sú ďalej ich spoločné prvky vyčlenené do abstraktnej triedy `FFNetworkBase`, kým jej potomkovia `FFNetwork` a `ShortcutNetwork` implementujú časti, v ktorých sa odlišujú (teda organizácia váh a prístup k nim). Vzťahy medzi spomínanými triedami spolu s vybranými atribútmi pre ilustráciu je možné vidieť na obrázku 4.1.

Pre spracovanie súborov so sieťami je ďalej vhodné vytvoriť modul `Parser`, ktorý má na starosti čítanie jednotlivých direktív vo vstupnom súbore, a vytvorenie im zodpovedajúcej siete (sieť zadaného typu so zadanými rozmermi), prípadne upravovanie vopred vytvorenej siete (nastavenie hodnoty určitej váhy).

Pre spoľahlivé vytváranie sietí je možné vytvoriť abstraktnú továreň, ktorú je možné implementovať pomocou jednej funkcie `createFactory`.




Obrázok 4.1: Triedny diagram jednotlivých typov sietí

Pre správu zdrojov OpenCL a uľahčenie vydávania príkazov zariadeniu môžeme implementovať triedu `OCLCaller`, ktorá sprostredkuje komunikáciu medzi objektom neurónovej siete a OpenCL API. Sieť žiada `OCLCaller` o vytvorenie objektov potrebných pre výpočet (programs, kernels, buffers) a udržuje si na ne odkazy, pomocou ktorých môže ďalej žiadať o vydávanie príkazov.

## 4.3 Hlavné telo programu

Tok programu je z pohľadu hlavného tela relatívne jednoduchý – je ho možné popísať ako tri za sebou nasledujúce kroky:

1. Spracovanie parametrov z príkazového riadku – predovšetkým voľba mien vstupných súborov a módu aplikácie (trénovanie siete, testovanie siete, alebo výpočet so zadanou sieťou);

2. Načítanie siete, prípadne trénovacej sady zo vstupných súborov. Testovaciu sadu, resp. vstupné dáta pre výpočet so sieťou nie je nevyhnutné načítavať a ukladať v tomto kroku, keďže je možné ich načítavať striedavo s výpočtom v kroku 3 (môže to však byť vhodné z hľadiska akcelerácie). Z pohľadu objektového návrhu tento krok spočíva v načítaní a spracovaní súboru s popisom neurónovej siete modulom `Parser` a vytvorení zodpovedajúceho objektu neurónovej siete;

3. Vlastný výpočet programu, jeden z troch módov, ktorého voľba bola zistená v kroku 1:

   a) **Trénovanie siete** – Prebieha učenie siete na základe poskytnutej trénovacej sady, výstupom je popis natrénovanej siete, resp. správa o neúspechu učenia;

   b) **Testovanie siete** – Sieť vykonáva výpočet nad zadanými testovacími vstupmi a svoje výstupy porovnáva so zadanými vzorovými výstupmi. Toto porovnanie spolu s vypočítanou odchýlkou zobrazuje ako výstup programu;

   c) **Výpočet so sieťou** – Načítavanie zadaných vstupov a výpočet s nimi.

Z pohľadu objektového návrhu tento krok spočíva v zavolaní príslušnej metódy objektu siete – `train`, `test` alebo `compute`.

Jadro programu teda predstavuje krok 3, v ktorom sú obsiahnuté algoritmy učenia a simulácie jednotlivých sietí. Kapitola 5 sa bude preto najviac venovať práve implementácii tohoto kroku.

# Kapitola 5

# Implementácia

## 5.1 Použité technológie

Pre implementáciu práce bol zvolený jazyk C++, verzia štandardu C++11, vo forme podporovanej prekladačom gcc verzie 4.8. Na akceleráciu výpočtov pomocou GPU využíva aplikácia OpenCL verzie 1.1, ktorého API pre C++ je definované štandardom OpenCL.

C++ je univerzálny programovací jazyk s podporou pre objektovo orientované programovanie. Štandard C++11 bol zvolený ako relatívne moderná verzia jazyka, ktorá je v čase písania práce vo veľkom rozsahu podporovaná dostupnými prekladačmi. Použitý prekladač (gcc 4.8) však nepodporuje modul pre prácu s regulárnymi výrazmi, ktorý je od C++11 zahrnutý v štandardnej knižnici. Parser sieťových súborov preto využíva implementáciu regulárnych výrazov z knižnice glibc (GNU C Library).

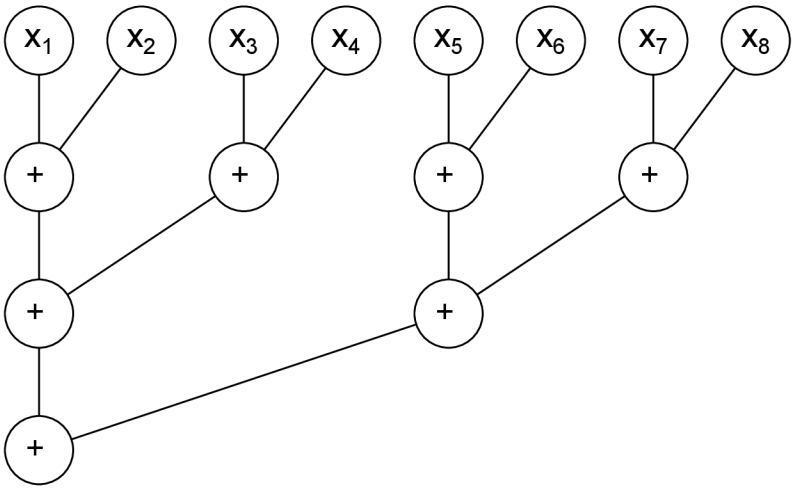
## 5.2 Akcelerácia výpočtov pomocou OpenCL

Základným nástrojom pre akceleráciu výpočtov pomocou GPU je ich paralelizácia. Implementované algoritmy obsahujú mnohé pasáže, v ktorých je treba vykonať určitý výpočet pre každý z určitej množiny objektov (napríklad pre všetky neuróny určitej vrstvy), pričom výsledok pre konkrétny objekt nijakým spôsobom nezávisí na výsledkoch ostatných objektov. Príkladom takejto úlohy je výpočet aktivačných funkcií neurónov jednej vrstvy – výsledok pre konkrétny neurón závisí iba od jeho potenciálu a aktivačnej funkcie.

Tieto typy úloh nevyžadujú žiadnu komunikáciu medzi jednotlivými work-itemami, a ich paralelný výpočet je preto možné realizovať relatívne jednoducho implementáciou kernelu, ktorý vykoná požadovaný výpočet pre jeden objekt, a následným spustením kernelu pre všetky požadované objekty. Na tomto princípe funguje aj implementácia vektorového súčtu popísaná zdrojovým kódom v sekcii 3.3.

Paralelizácia úloh vyžadujúcich komunikáciu medzi jednotlivými work-itemami je o niečo komplikovanejšia. Typickým príkladom, ktorý sa v implementovaných algoritmoch vyskytuje opakovane, je súčet $N$ čísel. Túto úlohu je možné realizovať v $\lceil \log_2 N \rceil$ krokoch rozdelením úlohy ilustrovanom na obrázku 5.1 pre $N = 8$ ($\lceil x \rceil$ predstavuje hornú celú časť $x$, t.j. najmenšie $n \in \mathrm{Z}$ také, že $x \leq n$). Tento postup je založený na princípe *sumy prefixov* [3], a je ho možné zovšeobecniť pre ľubovoľnú binárnu asociatívnu operáciu, napríklad pre vyhľadanie minimálneho prvku poľa, ktoré je potrebné pri učení Kohonenových sietí.

Možná implementácia tohoto výpočtu v OpenCL vyzerá nasledovne:




Obrázok 5.1: Ilustrácia paralelného súčtu ôsmich čísel

Zdrojový kód 5.1: Suma $N$ čísel

```
__kernel void sum(__global float* x) {
  int global_size = get_global_size(0);
  int gid = get_global_id(0);

  for (int i = global_size >> 1; i > 0; i >>= 1) {
    if (gid < i) {
      x[gid] += x[gid+i];
    }
    barrier(CLK_GLOBAL_MEM_FENCE);
  }
}
```

Po vykonaní tohoto kernela nad poľom $N$ čísel bude výsledný súčet v prvom prvku poľa `x`, toto správanie je možné ľahko zmeniť napríklad využitím ďalšieho poľa. Kernel tiež na synchronizáciu využíva vstavanú funkciu `barrier`, a využíva bitový posun pre rýchle delenie dvomi.

Táto implementácia však funguje iba pre $N$ ktoré je mocninou dvojky (teda $N = 2^k, k \in \mathrm{N}_0$) – pre iné rozmery úlohy budú niektoré z čísel pri sčitovaní preskočené. Pre úlohu súčtu je tento problém možné riešiť rozšírením úlohy na najbližšiu mocninu dvojky väčšiu ako $N$, pričom prebytočné prvky poľa inicializujeme na nulové hodnoty. Všeobecnejším riešením je po výpočte k získanému výsledku dodatočne sekvenčne pričítať preskočené hodnoty. Túto úpravu je možné vykonať napríklad pridaním nasledovného úryvku kódu na koniec spomínaného kernelu:

Zdrojový kód 5.2: Úprava kódu pre výpočet sumy $N$ čísel

```
    // Nasledovný kód prebehne iba na jednom work-iteme (s id 0),
    // predstavuje dodatočné pričítanie preskočených hodnôt.
  if (gid == 0) {
    for (int i = global_size; i > 1; i >>= 1) {
      if (i&1) x[0] += x[i-1];
    }
  }
```

Ďalším nástrojom, ktorý umožňuje urýchliť určité výpočty, je využitie hierarchie pamätí. Urýchlenie výpočtu maticového súčinu využitím lokálnej pamäti je popísané v [9].

Pri akcelerácii prostredníctvom OpenCL je taktiež vhodná snaha minimalizovať komunikáciu medzi hostiteľským prostredím a cieľovým zariadením.

## 5.3 Formát súboru so sieťou

Súbory so sieťou pozostávajú s direktív oddelených koncami riadkov. Každý súbor so sieťou musí povinne obsahovať aspoň tri direktívy – definíciu typu siete, definíciu štruktúry siete a definície aktivačných funkcií jednotlivých vrstiev.

Nasledujú ostatné, voliteľné direktívy, ako hodnoty jednotlivých váh a prahov, či zavádzanie porúch štruktúry siete. Formát je detailnejšie popísaný v prílohe B, príklady je možné nájsť na priloženom CD v adresári Tests.

## 5.4 Algoritmus Feedforward

Výpočet výstupu siete pomocou algoritmu Feedforward prebieha nasledovne:

1. Priveď vstupný vektor na neuróny vstupnej vrstvy (t.j. vrstvy 1);
2. Pre vrstvy $i = 2..l_{cnt}$ (vrstvy indexované od 1, $l_{cnt}$ je počet vrstiev, resp. index výstupnej vrstvy):

   Vypočítaj výstupy vrstvy $i$ na základe výstupov vrstvy $i - 1$;
3. Vráť výstupy neurónov výstupnej vrstvy ako výstupný vektor.

Z hľadiska akcelerácie je zaujímavé predovšetkým telo vnútorného cyklu v druhom kroku – proces výpočtu výstupov neurónov $i$-tej vrstvy na základe výstupov predchádzajúcej vrstvy. Všeobecne tento výpočet spočíva vo vyhodnotení bázovej funkcie a následnom vyhodnotení aktivačnej funkcie, treba však rátať s možným výskytom anomálií. Konkrétnejšie, pri výpočte bázovej funkcie treba rátať s neprirodzenými spojmi vedúcimi do vrstvy, pri výpočte aktivačnej funkcie treba separátne rátať mimoriadne nastavené aktivačné funkcie, tzn. neuróny s aktivačnou funkciou odlišnou od zvyšku vrstvy, a po výpočte treba odčiniť prípadné zmeny na neurónoch s fixnou hodnotou výstupu.

Výpočet výstupov danej vrstvy je teda možné zhrnúť v nasledujúcich krokoch, pričom každý z nich je možné vykonávať paralelne pre všetky neuróny:

1. Pre každý neurón vrstvy sčítaj príspevky jednotlivých prirodzených vstupov, tzn. $u := \sum_i x_i w_i$ pre jednotlivé neuróny $i$ predchádzajúcej vrstvy, kde $u$ je potenciál konkrétneho neurónu aktuálnej vrstvy (zatiaľ bez prahu), $x_i$ je výstup neurónu $i$, a $w_i$ je váha príslušného spoja (pre radiálnu bázovú funkciu $u := \sum_i (x_i - w_i)^2$);
2. K medzisúčtom získaným v predchádzajúcom kroku pripočítaj príspevky prípadných neprirodzených vstupov;
3. V prípade, že je bázovou funkciou radiálna, vypočítaj druhú odmocninu z jednotlivých medzisúčtov;
4. Pripočítaj k jednotlivým potenciálom prahy zodpovedajúcich neurónov;

5. Vyhodnoť aktivačné funkcie jednotlivých neurónov;

6. Separátne vyhodnoť aktivačné funkcie neurónov, ktoré ich majú odlišné od zvyšku siete;

7. Odčiň prípadné zmeny neurónov s fixnými výstupmi.

Poznamenajme, že výpočet sigmoidy je možné urýchliť využitím vzorca 2.8 – odpadajú tým výpočetne náročné operácie delenia a výpočtu exponenciálnej funkcie.

## 5.5 Algoritmus Backpropagation

Backpropagation má podstatne komplikovanejšiu štruktúru ako Feedforward. Z hľadiska návrhu je vhodné rozlíšiť jadro algoritmu, pozostávajúce zo šírenia chyby konkrétneho výstupu **y** pri konkrétnom vzorovom výstupe **t**, a vonkajšie telo algoritmu, ktoré opakovane prechádza trénovaciu množinu a volá jadro algoritmu na jednotlivé trénovacie vstupy, kým sieť nemapuje všetky na správne výstupy.

Algoritmus 1 opisuje možnú implementáciu tohto vonkajšieho tela, ktorá využíva tri podprogramy: `Backpropagate`, ktorý reprezentuje šírenie chyby posledného výpočtu a zodpovedajúce úpravy váh a prejavuje sa zmenou vnútorného stavu siete $N$, `Feedforward`, ktorý predstavuje výpočet výstupu siete na základe daného vstupu a zodpovedá algoritmu Feedforward opísanému v predchádzajúcej sekcii, a `Err`, ktorý počíta odchýlku výstupu siete **y** od vzorového výstupu **t** podľa rovnice 2.12.

**Algoritmus 1:** Backpropagation

**input** : Network $N$, training set $S$, epoch limit $limit$, maximal error $\varepsilon$
**output**: Trained network $N$

```
1  epoch ← 0 ;
2  trained ← false ;
3  while not trained and epoch < limit do
4      epoch ← epoch + 1 ;
5      trained ← true ;
6      for (x, t) ∈ S do
7          y ← N.Feedforward(x) ;
8          if Err(y, t) > ε then
9              trained ← false ;
10         end if
11         while Err(y, t) > ε do
12             N.Backpropagate(y, t) ;
13             y ← N.Feedforward(x) ;
14         end while
15     end for
16 end while
```

Trénovacia sada $S$ pozostáva z dvojíc $(x, t)$, kde $x$ je vstupný vektor a $t$ vzorový výstupný vektor.

## 5.6 Úpravy pre siete typu Shortcut

Predchádzajúce algoritmy je možné podobným spôsobom implementovať pre Shortcut siete, je však potrebné zohľadniť rozdielnu organizáciu váh. Obzvlášť významne sa tento rozdiel prejavuje v algoritme Backpropagation pri výpočte $\delta_i$ pre neuróny $i$ skrytých vrstiev – je potrebný prístup k váham vychádzajúcich z danej vrstvy, váhy je však z hľadiska algoritmu Feedforward výhodnejšie organizovať podľa cieľovej vrstvy.

## 5.7 Hopfieldove siete

Učenie Hopfieldových sietí predstavuje úlohu maticového súčinu. Tento výpočet je možné vykonať kernelom rozmerov $N^2$, kde jeden work-item zodpovedá jednému prvku výslednej matice váh.

Keďže výpočet siete vyžaduje náhodný výber neurónov na vyhodnotenie, paralelizácia na tejto úrovni nie je možná. Paralelný výpočet je však možné využiť na výpočet potenciálu vyhodnocovaného neurónu.

## 5.8 Rozširovanie aplikácie

Táto sekcia popisuje kroky, ktoré musí uživateľ vykonať, aby pridal do aplikácie novú funkcionalitu v podobe ďalších typov sietí, aktivačných funkcií, atď.

### 5.8.1 Pridávanie nových typov sietí

Pre pridanie nového typu neurónovej siete je potrebné implementovať triedu, ktorá je potomkom triedy `Network`, a všetky jej virtuálne metódy, ktoré má nový typ sietí podporovať.

Novému typu sietí je tiež potrebné pridať jemu zodpovedajúcu konštantu do `enum class NNType`, spárovať ju s požadovaným názvom typu v kolekciách `NETWORK_TYPES` a `NETWORK_TYPE_NAMES` a pridať volanie konštruktora siete pridávaného typu do továrenskej funkcie `create_network`.

### 5.8.2 Pridávanie nových aktivačných funkcií

Novej aktivačnej funkcii je potrebné priradiť zodpovedajúcu konštantu do `enum class ActType` a spárovať ju s názvom typu v kolekciách `ACTIVATION_TYPES` a `ACT_TYPE_NAMES`.

Do metód štruktúry `ActDef`, t.j. `set_defaults` a `print_param`, je ďalej potrebné pripísať inicializáciu, resp. výpis parametrov novej funkcie. Pre jej úspešné načítanie je potrebné pridať spracovanie jej parametrov do funkcie parsera `get_layer_param`. V prípade, že sa štruktúra požadovaného zápisu parametrov funkcie líši od už existujúcich, je vhodné pridať zodpovedajúci regulárny výraz, prípadne spracovať parametre iným spôsobom.

Tiež je potrebné pridať podporu pre aktivačnú funkciu do požadovaných typov sietí. Pre siete Feedforward a Shortcut je to otázka úpravy metód `set_af_kernel` a `set_delta_af_kernel` triedy `FFNetworkBase` pridaním kódu inicializujúceho kernely pre výpočet, resp. učenie so zodpovedajúcou aktivačnou funkciou.

### 5.8.3 Pridávanie nových bázových funkcií

Pridanie podpory novej bázovej funkcie pre dané siete je o niečo problematickejšie – vyžaduje zásah do formátu sieťového súboru a tým pádom aj do parsera týchto súborov, keďže pôvodný návrh s takýmito zmenami nepočítal.

Kroky výpočtu pre jednotlivé implementované siete sú však natoľko oddelené, že zmena bázovej funkcie pre danú vrstvu by bola len otázkou zmeny kernelu pre výpočet bázovej funkcie priradeného danej vrstve, či už pri simulácii výpočtu siete, alebo pri jej učení.

Špeciálne bázové funkcie pre jednotlivé neuróny v rámci vrstvy by vyžadovali podobné dodatočné výpočty, ako už existujú pre špeciálne aktivačné funkcie.

# Kapitola 6

# Experimenty

Ďalšou časťou práce je meranie rýchlosti aplikácie. Táto kapitola sa venuje popisu jednotlivých experimentov a ich výsledkov. V rámci experimentov bola pomocou funkcie `gettimeofday` meraná doba učenia algoritmom Backpropagation rôznych sietí. Pre porovnanie boli rovnaké experimenty vykonané aj s využitím knižnice FANN[1].

Jednotlivé sekcie sú venované konkrétnym experimentom, a pozostávajú zo stručného popisu neurónovej siete a úlohy a z tabuľky obsahujúcej výsledky meraní. Aplikácia implementovaná v rámci tejto práce je v tabuľkách označovaná menom ann. Číselné hodnoty v tabuľkách pod hlavičkami FANN alebo ann predstavujú priemerný čas výpočtu v sekundách. V hlavičkách tabuliek sa ďalej vyskytujú nasledujúce symboly:

- $\varepsilon$ – požadovaná maximálna chyba,
- $\eta$ – koeficient učenia,
- $\alpha$ – koeficient zotrvačnosti.

V jednotlivých experimentoch sa riešia nasledovné úlohy:

- Siete typu Feedforward pre výpočet $N$-bitovej parity pre $N = 2, 3, 4, 8$. Dvojbitová parita zodpovedá logickej funkcii XOR. Úlohy budú po rade označované XOR, parity3, parity4, parity8;
- Sieť typu Shortcut pre úlohu diabetes zo sady PROBEN1 [16];
- Siete typu Feedforward pre výpočet 8-bitovej parity, pričom počet neurónov prvej skrytej vrstvy je zmenený na $M$ pre $M = 100, 500, 1000, 2000, 5000, 10000$, požadovaná maximálna chyba siete je nastavená na relatívne nízku hodnotu (napríklad $\varepsilon = 0.001$), a maximálny počet epoch učenia je nastavený na hodnotu 10000. Cieľom týchto experimentov je odmerať dobu výpočtu pre veľký počet epoch.

Experimenty prebiehali na notebooku Lenovo ThinkPad T520 s procesorom Intel Core i5-2410M (dual-core, 2.3 GHz) a grafickou kartou NVIDIA Quadro NVS 4200M pod systémom Ubuntu 14.04, 64-bit.

[1] http://leenissen.dk/fann/wp/

## 6.1 XOR

Sieť typu Feedforward so štruktúrou `2x3x1`, úloha XOR, $\eta = 0.7$, $\alpha = 0$:

| $\varepsilon$ | FANN | ann |
|---|---|---|
| 0.2 | 0.283 | 4.558 |
| 0.1 | 0.389 | 4.547 |
| 0.05 | 0.364 | 4.339 |
| 0.01 | 0.371 | 4.364 |

## 6.2 Parity3

Sieť typu Feedforward so štruktúrou `3x8x1`, úloha parity3, $\alpha = 0$:

| $\varepsilon$ | $\eta$ | FANN | ann |
|---|---|---|---|
| 0.2 | 0.7 | 0.384 | 2.303 |
| 0.1 | 0.7 | 0.362 | 2.306 |
| 0.05 | 0.7 | 0.159 | 2.356 |
| 0.01 | 0.7 | 0.125 | 10.070 |
| 0.2 | 0.3 | 0.327 | 2.612 |
| 0.1 | 0.3 | 0.377 | 2.838 |
| 0.05 | 0.3 | 0.126 | 5.427 |

## 6.3 Parity4

Sieť typu Feedforward so štruktúrou `4x8x1`, úloha parity4, $\alpha = 0$:

| $\varepsilon$ | $\eta$ | FANN | ann |
|---|---|---|---|
| 0.2 | 0.7 | 0.365 | 3.232 |
| 0.1 | 0.7 | 0.262 | 3.131 |
| 0.2 | 0.3 | 0.374 | 6.633 |
| 0.1 | 0.3 | 0.355 | 6.992 |

Pre hodnoty $\varepsilon = 0.05, 0.01$ a $\eta = 0.7$ výpočet na FANN nekonverguje pred maximálnym počtom epoch (10000).

## 6.4 Parity8

Sieť typu Feedforward so štruktúrou `8x8x8x1`, úloha parity8, $\varepsilon = 0.1$:

| $\eta$ | $\alpha$ | FANN | ann |
|---|---|---|---|
| 0.1 | 0.9 | 0.766 | 188.573 |
| 0.7 | 0.4 | 0.679 | 176.544 |
| 0.3 | 0.0 | 0.739 | 162.773 |

Výpočet ann v 4 z 15 prípadov nekonvergoval pred maximálnym počtom epoch (10000).

## 6.5 Diabetes

Sieť typu Shortcut so štruktúrou `8x16x8x2`, úloha diabetes zo sady PROBEN1 [16], $\eta = 0.1, \alpha = 0.0$:

| $\varepsilon$ | FANN | ann |
|---|---|---|
| 0.1 | 0.571 | 420.517 |
| 0.04 | 0.670 | 679.158 |

## 6.6 Parity8 s úpravou skrytej vrstvy

Siete typu Feedforward so štruktúrou `8x`$M$`x8x1`, kde $M$ je variabilná hodnota, $\varepsilon = 0.001, \eta = 0.7, \alpha = 0.9$, maximálny počet epoch 10000:

| $M$ | FANN | ann |
|---|---|---|
| 100 | 63.970 | 18626.923 |
| 500 | 312.063 | 19057.879 |
| 1000 | 613.107 | 25484.635 |
| 2000 | 1221.334 | 12063.871 |
| 5000 | 3072.649 | 25051.24 |
| 10000 | 6111.508 | 43866.512 |

# Kapitola 7

# Záver

V práci bola úspešne implementovaná aplikácia pre simuláciu neurónových sietí s požadovanou funkcionalitou, vrátane podpory pre poruchy štruktúry siete. Výsledky experimentov však naznačujú, že dosiahnutá akcelerácia výpočtov stále nepostačuje na prekonanie dostupných implementácií neurónových sietí.

Jedným z najvýznamnejších faktorov negatívne ovplyvňujúcich rýchlosť aplikácie bolo sériové vykonávanie príkazov OpenCL. Jeho dôvodom bolo to, že aplikácia bola vyvíjaná a testovaná na stroji, ktorého GPU má iba 1 compute unit a obmedzené zdroje, čo značne komplikuje paralelné vykonávanie rôznych úloh. Tento spôsob implementácie síce zabezpečuje prenositeľnosť na ľubovoľné OpenCL zariadenie, spôsobuje však potrebu intenzívnejšej komunikácie medzi hostiteľským prostredím a GPU (napríklad pravidelné čítanie chyby výstupu siete z GPU pri Backpropagation), ktorú je prípadne možné vyvážiť obmedzením prispôsobiteľnosti a prehľadnosti aplikácie (napríklad implementáciou celého Backpropagation v jednej úlohe).

V aplikácii stále zostáva priestor na ďalšiu akceleráciu. Napríklad pre algoritmus Backpropagation by bolo možné pridať alternatívny výpočet pre GPU s väčším počtom compute units, využívajúci paralelne bežiace úlohy. Aplikáciu je možné (a relatívne ľahké) rozšíriť o podporu ďalších typov sietí a ďalšie algoritmy učenia, prípadne do už implementovaných algoritmov pridať úpravy pre zlepšenie učenia. Príkladom je detekcia lokálnych miním a vyviaznutie z nich pre Backpropagation [6] [7].

# Literatura


[1] The Nobel Prize in Physiology or Medicine 1906. [online], cit. 2014-05-09.
URL http://www.nobelprize.org/nobel_prizes/medicine/laureates/1906/index.html

[2] Slajdy k predmetu Soft Computing.

[3] Blelloch, G. E.: Prefix Sums and Their Applications. 1991.

[4] Wikimedia Commons: Derived Neuron schema with no labels. [online], 2006, cit. 2015-05-12.
URL https://commons.wikimedia.org/wiki/File:Derived_Neuron_schema_with_no_labels.svg

[5] Costarelli, D.: *Sigmoidal Functions Approximation and Applications*. Dizertační práce, Universita Roma Tre, 2014.

[6] Gori, M.; Tesi, A.: On the Problem of Local Minima in Backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, ročník 14, jan 1992, ISSN 0162-8828.

[7] Hamid, N. A.; aj.: Solving Local Minima Problem in Back Propagation Algorithm Using Adaptive Gain, Adaptive Momentum and Adaptive Learning Rate on Classification Problems. *International Journal of Modern Physics: Conference Series*, ročník 09, 2012, ISSN 2010-1945.

[8] Hebb, D. O.: *The Organization of Behavior*. Wiley & Sons, 1949.

[9] Jonathan Tompson, K. S.: An Introduction to the OpenCL Programming Model. 2012.

[10] Kambi, N.; Jain, N.: Landmark Discoveries in Neurosciences. *Resonance*, ročník 17, nov 2012, ISSN 0971-8044.

[11] McCulloch, W. S.; Pitts, W.: A Logical Calculus of the Ideas Immanent in Nervous Activity. 1943.

[12] Mehrotra, K.; Mohan, C. K.; Ranka, S.: *Elements of Artificial Neural Networks*. 1996, ISBN 0-262-13328-8.

[13] Minsky, M.; Papert, S.: *Perceptrons*. The MIT Press, 1969, ISBN 0-262-63111-3.

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

[15] Khronos OpenCL Working Group: The OpenCL Specification. 2009.
URL https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf

[16] Prechelt, L.: PROBEN1 – A Set of Neural Network Benchmark Problems and Benchmarking Rules. 1994.

[17] Rojas, R.: *Neural Networks - A Systematic Introduction*. Springer, 1996, ISBN 978-3-54060-505-8.

# Příloha A

# Obsah CD

Priložené CD obsahuje:

- ibp-text.pdf – textová správa vo formáte pdf;
- Adresár Code – zdrojové súbory programovej časti, vrátane OpenCL kernelov a súboru Makefile;
- Adresár Tests – Obsahuje testovacie súbory, pomocou ktorých bola meraná rýchlosť aplikácie;
- Adresár Text – zdrojové súbory textovej správy, vrátane obrázkov.

# Príloha B

# Gramatika súborov so sieťou

```
# whitespace, including comments, is ignored.
# non-terminals are written with angle brackets - <, >
# terminals are written with quotation marks - "
# * means 0-N repetitions
# + means 1-N repetitions
# | separates alternatives
# square brackets - [, ] - delimit optional parts
# EOL stands for end of line
# EOF stands for end of file

# NOTE: Every directive is on a single line and contains at least one
#       alphabetic character.

<document>         := <type-definition> EOL <structure> EOL <layer-activation> EOL (<op
<comment>          := "#" (anything but EOL)* EOL

<type-definition>  := "type" "=" <type>
<type>             := "feedforward" | "shortcut" | "Hopfield" | "Kohonen"

<structure>        := "structure" "=" <uint> ("x" <uint>)*
# <uint> == positive integer constant

<layer-activation> := "activation" "=" <act-type> ("x" <act-type>)
                      [";" <act-params> ("x" <act-params>)]
# number of <act-type> must be equal to the number of <act-params>
# omitted parameters are interpreted as default parameters
# corresponding <act-type> and <act-params> must be of corresponding forms:
<act-type>         := "sigmoid" | "step" | "tanh" | "ramp"

# NOTE: the addition of types containing "x"
#       would require changes in the parser,
#       because x's are used to delimit layers.

<act-params>       := "default"
                    | <float>                          # sigmoid, tanh;
```

```
                    | "<" <float> "," <float> ">"                  # step;
                    | "<" <float> "," <float> ">" <float> <float> # ramp;
# <float> == floating point constant
# NOTE: No "theta" parameters - bias is implemented separately

<optional-stmt>    := <weight-def> | <bias> | <activation> | <training>
                    | <epsilon> | <epoch-limit> | <file> | <modification>

# weight, bias, activation: no spaces in the first word!
<weight-def>       := <weight> "=" <float>
<weight>           := "w"<uint>","<uint>
<bias>             := "bias"<uint> "=" <float>
<activation>       := "activation"<uint> "=" <act-type> ; <act-params>
# <act-type> and <act-params> must be of corresponding forms

<training>         := "training" "=" <algorithm> [";" <algorithm-params>]
# <algorithm> and <algorithm-params> must be of corresponding forms:
<algorithm>        := "backpropagation" | "Kohonen"
<algorithm-params> := (<float>)*
<epsilon>          := "epsilon" "=" <float>
<epoch-limit>      := "epoch_limit" "=" <int>
<file>             := ("train_file" | "test_file") "=" <filename>
# <filename> == name of an existing training input / test input file

<modification>     := <stuck> | <remove> | <add>
<stuck>            := "stuck" (<weight> | <neuron>) "=" <float>
<neuron>           := "n"<uint>
<remove>           := "remove" (<neuron> | <weight>)
<add>              := "add" <weight> ["=" <float>]
```