Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Petra Kudová

Neuronové sítě typu RBF pro analýzu dat

Katedra softwarového inženýrství

Vedoucí diplomové práce: *Mgr. Roman Neruda CSc.*

Studijní program: *Informatika*

Ráda bych poděkovala Romanu Nerudovi za vedení diplomové práce, rady a připomínky, Pájovi a Zuzce za první pomoc při srážkách se systémem Bang2, Arnoštovi za řešení technických problémů na Joyceovi.

Zvláštní poděkování patří Mluvícímu žabáčkovi za trpělivost.

Prohlašuji, že jsem svou diplomovou práci napsala samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne: 19. dubna 2001 Petra Kudová

„ … obrovská loterie, z níž vidíme jen výhry … “
(Jostein Gaarder, Tajemství karet)

Obsah

Kapitola 1

Úvod

Výzkum v oblasti umělých neuronových sítí začal ve 40. letech, kdy byl zveřejněn jednoduchý matematický model neuronu. Ačkoliv počáteční motivací bylo modelovat nervové systémy živých organismů, neuronové sítě jsou především univerzálním výpočetním prostředkem. Jejich klíčovými vlastnostmi jsou schopnost učit se na základě impulsů z okolního prostředí a schopnost přizpůsobovat se tomuto prostředí. Nás zajímají především dopředné modely neuronových sítí, které mají navíc schopnost univerzální aproximace reálných funkcí. Nejznámějším zástupcem takových dopředných sítí jsou perceptronové sítě. My se v této práci budeme zabývat neuronovými sítěmi typu RBF, které patří mezi mladší modely dopředných sítí.

Naším cílem je navrhnout a prozkoumat různé metody učení neuronových sítí typu RBF. Při učení budeme často využívat metody z jiných oborů matematiky a informatiky (metody vektorové kvantizace, metody nejmenších čtverců apod.). Budeme zkoumat použití jak klasické gradientní minimalizační metody, tak poměrně mladých a v tomto kontextu méně známých genetických algoritmů.

Dalším úkolem je realizovat softwarový systém, který umožní testování jednotlivých metod. Tyto metody jsou implementovány jako autonomní softwaroví agenti v rámci multiagentního systému Bang2. Pomocí tohoto systému otestujeme vybrané metody a popíšeme jejich chování na konkrétních problémech.

Struktura této práce je následující. V kapitole 2 se seznámíme s modelem neuronové sítě typu RBF a uvedeme problém učení sítě.

Další tři kapitoly se zabývají třemi odlišnými přístupy k učení neuronové sítě typu RBF. Kapitola 3 popisuje použití gradientního učení, které bylo inspirováno algoritmem zpětného šíření (Backpropagation) pro vícevrstvé perceptronové sítě. Navrhneme originální zobecněný algoritmus pro gradientní učení RBF sítí. Jiným přístupem je třífázové učení, se kterým se seznámíme v kapitole 4. Tento přístup rozděluje problém učení na několik dílčích úloh, které pak řeší odděleně různými metodami. Kapitola 5 vysvětluje základní principy genetických algoritmů a možnosti využití těchto algoritmů pro učení neuronových sítí typu RBF.

Následuje popis implementace popsaných metod. Ten najdeme v kapitole 6. Protože je implementace součástí multiagentního systému Bang2, seznámíme se nejprve stručně s tímto systémem. Poté popíšeme vlastní realizaci metod učení a rozhraní agentů, kteří se na něm podílejí.

Kapitola 7 se věnuje experimentům. Budeme řešit klasické aproximační a klasifikační úlohy, používané k testování metod umělé inteligence. Aproximaci ilustrujeme na reálné funkci $sin(x)cos(y)$. Klasifikační úlohou bude problém *dvou spirál*, kde se jedná o klasifikaci dvou množin bodů v rovině. Na závěr vyzkoušíme možnost učení na tréninkové množině obsahující šum.

Uživatelská dokumentace realizovaného softwaru je obsažena v příloze. K diplomové práci je přidáno CD, na kterém najdete text této diplomové práce, realizovaný software a pokyny k instalaci.

Kapitola 2

Neuronové sítě typu RBF

Neuronové sítě typu RBF (krátce RBF sítě) patří mezi nejmladší modely neuronových sítí a jsou alternativou klasických modelů, jako jsou vícevrstvé perceptronové sítě.

Motivaci k jejich studiu najdeme v numerické matematice, konkrétně ve studiu interpolací a aproximací dat. Řešení aproximačního problému se většinou hledá jako lineární kombinace basických funkcí v nějakém konkrétním tvaru, například jako kombinace polynomů. V 80. letech se v této souvislosti dostávají do popředí tzv. *radiální basické funkce*.

Radiální funkce je určená svým středem a její hodnota závisí na vzdálenosti argumentu od tohoto středu. Představíme-li si radiální funkci v dvourozměrném prostoru s euklidovskou metrikou, pak množiny bodů se stejnou funkční hodnotou tvoří kružnice.

Použití radiálních basických funkcí v numerické matematice je následováno vznikem nového modelu neuronové sítě, neuronové sítě typu RBF. Tato síť realizuje lineární kombinaci radiálních basických funkcí. Na řešení aproximace se následně pohlíží jako na problém učení neuronové sítě, což přináší nejen nový pohled na věc, ale především nové metody řešení aproximačních problémů.

Nyní se s tímto modelem seznámíme. Nejprve v části 2.1 popíšeme strukturu a funkci RBF jednotky. Poté v části 2.2 přiblížíme model neuronové sítě typu RBF. Část 2.3 se bude zabývat možnostmi obohacení základního modelu sítě o použití vážených norem a konečně část 2.4 nás uvede do problému učení RBF sítí.

2.1 RBF jednotka

Základní stavební jednotkou RBF sítí jsou tzv. *RBF jednotky.* RBF jednotka realizuje radiální funkci. Má n reálných vstupů $(x_1, \cdots, x_n)$ a jeden reálný výstup y. Každá jednotka je určena n-rozměrným vektorem $\vec{c}$, který nazýváme střed, a může mít další parametr šířku b. Vnitřní potenciál počítá jako euklidovská vzdálenost vstupního vektoru $\vec{x} = (x_1, \cdots, x_n)$ od středu $\vec{c}$ dělenou šířkou b.

$$\xi = \frac{\| \vec{x} - \vec{c} \|}{b} \tag{2.1}$$

Výstup RBF jednotky y je roven hodnotě příslušné radiální funkce φ v bodě ξ. Nejpoužívanější radiální funkcí je Gaussova

$$\varphi(z) = e^{-z^2}. \tag{2.2}$$

V tomto případě RBF jednotka realizuje funkci

$$y(\vec{x}) = e^{-\left(\frac{\|\vec{x}-\vec{c}\|}{b}\right)^2}. \tag{2.3}$$

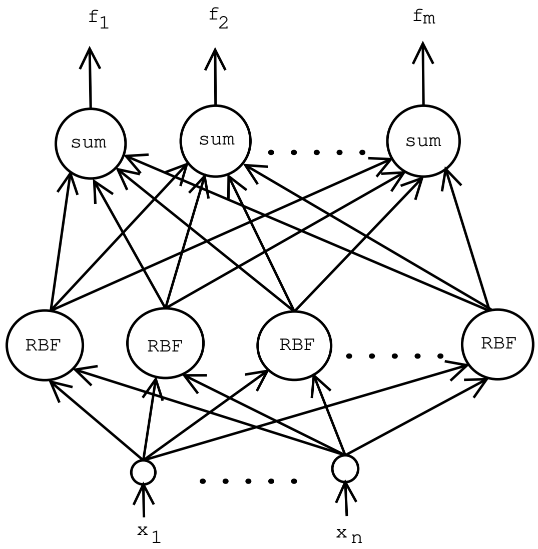
Pro srovnání uveďme funkci klasického perceptronu

$$y(\vec{x}) = \sigma\left(\sum_{i=0}^{n} w_i x_i\right), \qquad kde \quad \sigma(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases} \tag{2.4}$$

RBF jednotka i perceptron realizují nelineární funkci z $\mathbb{R}^n$ do $\mathbb{R}$. Podíváme-li se však na významy těchto funkcí, vidíme, že Gaussova RBF jednotka má významný výstup pouze v okolí svého středu, zatímco perceptron rozděluje vstupní prostor globálně na dva podprostory. Gaussova RBF jednotka se tedy chová v jistém smyslu duálně. Takovéto jednotky, které mají vymezenou lokalitu své působnost, se nazývají *lokálními jednotkami.*

2.2 Model RBF sítě

RBF síť je třívrstvá dopředná neuronová síť (viz obr. 2.2). První vrstva obsahuje n vstupních vstupních jednotek, které slouží pouze k předání vstupních hodnot. Druhá — skrytá — vrstva obsahuje h RBF jednotek. Poslední — výstupní — vrstva je lineární a počet jejích jednotek označme m. RBF síť realizuje reálnou funkci $f : \mathbb{R}^n \to \mathbb{R}^m$ a to tak, že nejprve vrstva RBF jednotek transformuje vstupní vektor $(x_1, \cdots, x_n)$ na vektor $(y_1, \cdots, y_h)$, a pak



Obrázek 2.1: Model RBF sítě

výstupní vrstva transformuje vektor $(y_1, \cdots, y_h)$ na vektor $(f_1, \cdots, f_m)$.

$$f_s(\vec{x}) = \sum_{j=1}^{h} w_{js}\varphi\left(\frac{\| \vec{x} - \vec{c}_j \|}{b_j}\right), \tag{2.5}$$

kde w_{js} je váha spoje od j-té skryté jednotky do s-té výstupní jednotky. Funkce sítě je tedy složením dvou funkcí, první nelineární funkce z $\mathbb{R}^n$ do $\mathbb{R}^h$ a druhé lineární funkce z $\mathbb{R}^h$ do $\mathbb{R}^m$.

Budeme-li se dívat na síť jako na klasifikátor, můžeme říci, že úkolem první vrstvy je transformovat vstupní vektor tak, aby jej bylo možno ohodnotit lineární kombinací jeho nových souřadnic. Tímto problémem se v polovině 60. let zabýval Cover [4, str. 237] a ukázal, že počet skrytých jednotek je vhodné volit větší než je počet vstupních jednotek. To ukazují i výsledky z praxe.

2.3 Vážená norma

Vnitřní potenciál RBF jednotky počítáme jako $\frac{\|\vec{x}-\vec{c}\|}{b}$, kde $\| \vec{x} - \vec{c} \|$ je vzdálenost $\vec{x}$ od $\vec{c}$ počítaná dle euklidovské metriky.

Další možností je využití *vážené normy*, která je definována jako

$$\| \vec{x} \|_C^2 = (\mathbf{C}\vec{x})^T(\mathbf{C}\vec{x}) = \vec{x}^T\mathbf{C}^T\mathbf{C}\vec{x}, \tag{2.6}$$

kde $\mathbf{C}$ je matice $n \times n$ a n je dimenze vstupního vektoru.

Pokud je matice **C** rovna jednotkové matici, dostaneme standardní eukli-dovskou normu.

$$\| \vec{x} \|_C^2 = \| \vec{x} \|^2 \tag{2.7}$$

Použití diagonální matice odpovídá přiřazení váhových koeficientů jednotlivým vstupům.

$$\| \vec{x} \|_C^2 = \sum_{k=1}^{n} c_k^2 x_k^2 \tag{2.8}$$

Obecná matice představuje libovolnou afinní transformaci.

$$\| \vec{x} \|_C^2 = \sum_{k=1}^{n} \sum_{l=1}^{n} a_{kl} x_k x_l, \tag{2.9}$$

kde a_{kl} je kl-tý prvek matice $\mathbf{C}^T\mathbf{C}$.

Matici $\mathbf{C}^T\mathbf{C}$ budeme dále označovat jako $\mathbf{\Sigma^{-1}}$. (Toto označení vychází z analogie s kovarianční maticí vícerozměrného normálního rozdělení. Pro naše účely však není tato analogie důležitá.)

Nahradíme-li euklidovskou metriku právě popsanou váženou normou, RBF síť bude počítat funkci

$$f_s(\vec{x}) = \sum_{j=1}^{h} w_{js} \varphi \left(\frac{\| \vec{x} - \vec{c}_j \|_C}{b_j} \right). \tag{2.10}$$

V tomto případě všechny RBF jednotky používají při výpočtu vzdálenosti stejnou matici C, tedy u všech je prováděna stejná transformace.

V obecném případě může každé jednotce přidat další parametr C a funkce sítě pak má tvar

$$f_s(\vec{x}) = \sum_{j=1}^{h} w_{js} \varphi \left(\frac{\| \vec{x} - \vec{c}_j \|_{C_j}}{b_j} \right). \tag{2.11}$$

Použití vážené normy je vhodné zvláště v případě, že jednotlivé složky vstupního vektoru jsou různého charakteru.

2.4 Učení RBF sítí

Pod pojmem *„učení neuronové sítě"* rozumíme nastavení všech parametrů sítě tak, aby síť realizovala požadovanou funkci. Tato funkce je obvykle zadána množinou uspořádaných dvojic $(\vec{x}, \vec{d})$, kde $\vec{x} = (x_1, \cdots, x_n)$ představuje hodnoty vstupů a $\vec{d} = (d_1, \cdots, d_n)$ je vektor odpovídajících výstupů. Tato dvojice se nazývá *tréninkový vzor* a tato množina *tréninková množina*. Učení, při kterém je k dispozici tréninková množina, nazýváme *učení s učitelem*.

Samotné učení pak typicky probíhá jako minimalizace *chybové funkce*, která je dána sumou čtverců odchylek výstupů sítě od požadovaných výstupů.

$$E_1 = \frac{1}{2}\sum_{t=1}^{k} \parallel \vec{d}^{(t)} - \vec{f}^{(t)} \parallel^2 = \frac{1}{2}\sum_{t=1}^{k}\sum_{s=1}^{m}(d_s^{(t)} - f_s^{(t)})^2, \tag{2.12}$$

kde k je počet vzorů v tréninkové množině, $\vec{f}^{(t)}$ je výstup sítě po předložení vzoru $\vec{x}^{(t)}$ a $\vec{d}^{(t)}$ je požadovaný výstup.

2.4.1 Regularizace

Často klademe na zobrazení realizované sítí další požadavky. Naučená síť by měla dávat rozumné výsledky i pro vstupy, které nejsou obsaženy v tréninkové množině. Také situace, kdy funkce sítě prochází všemi body z tréninkové množiny není vždy ideální, neboť v praxi tréninková množina často obsahuje šum.

Chris Bishop se v [1] zabývá řešením tohoto problému. K chybové funkci přidává další člen E_R :

$$E = E_1 + \lambda E_R \tag{2.13}$$

E_R je zvolen tak, aby měl velké hodnoty pro funkce s velkou oscilací. Parametr λ určuje míru vlivu tohoto regularizačního členu na chybovou funkci. Bishop volí E_R tak, že penalizuje funkce s velkou druhou derivací.

$$E_R = \frac{1}{2}\sum_{t=1}^{k}\sum_{s=1}^{m}\sum_{i=1}^{n}\left(\frac{\partial^2 f_s^t}{\partial x_i^2}\right)^2 \tag{2.14}$$

Kapitola 3

Gradientní učení RBF sítě

Pro učení vícevrstvých sítí perceptronů se používá algoritmus zpětného šíření (Back Propagation), který využívá gradientního algoritmu. Výpočet derivací je založen na zpětné propagaci chyby směrem od výstupní vrstvy ke vstupní.

RBF síť je také vícevrstvou neuronovou sítí, mohli bychom tedy pro její učení použít modifikaci tohoto algoritmu. Protože však RBF sítě mají pouze jednu skrytou vrstvu, nedochází tu k žádnému šíření chyby jako u vícevrstvých perceptronů.

Učení RBF sítí pomocí gradientního algoritmu uvádí Haykin v [4] pro RBF síť, jejímiž parametry jsou středy, matice norem a váhy. My si ještě přidáme jeden parametr a to šířku.

3.1 Gradientní algoritmus

Uvažujeme-li pevnou tréninkovou množinu, pak funkce E_1 vyjadřuje chybu na této množině v závislosti na konfiguraci sítě. Úkolem učení je nalézt takovou konfiguraci, pro kterou je chyba sítě E_1 minimální.

Algoritmus zpětného šíření pro vícevrstvé sítě perceptronů hledá pomocí gradientního algoritmu minimum funkce E_1 v prostoru parametrů.

Algoritmus 3.1.1 popisuje použití gradientního algoritmu pro RBF síť. Počáteční konfiguraci sítě zvolíme náhodně. V každém kroku algoritmu pak upravíme všechny parametry podle

$$p^{\tau+1} = p^{\tau} - \varepsilon \frac{\partial E_1}{\partial p}, \tag{3.1}$$

kde τ označuje diskrétní čas výpočtu, p je konkrétní parametr a $\varepsilon \in (0, 1)$ určuje rychlost učení. Tím posuneme konfiguraci ve směru tečného vektoru funkce E_1 dolů o ε. Získáme tak novou konfiguraci, pro niž je hodnota chybové funkce menší než chyba v předchozím bodě ($E_1^{\tau+1} \leq E_1^{\tau}$). To opakujeme dokud se nedostaneme do lokálního minima funkce.

Algoritmus 3.1.1:

1. $\tau := 0$
   Nastav náhodně $\vec{c}_k(0)$, $b_k(0)$, $\mathbf{\Sigma_k^{-1}}(0)$ a $w_{ks}(0)$ pro $s = 1, \cdots, m$ a $k = 1, \cdots, h$

2. $\tau := \tau + 1$

3. Spočti pro $s = 1, \cdots, m$ a $k = 1, \cdots, h$

$$\begin{array}{rclcrcl} \Delta\vec{c}_k & = & -\varepsilon\frac{\partial E_1}{\partial \vec{c}_k} & & \Delta b_k & = & -\varepsilon\frac{\partial E_1}{\partial b_k} \\ \Delta\mathbf{\Sigma_k^{-1}} & = & -\varepsilon\frac{\partial E_1}{\partial \mathbf{\Sigma_k^{-1}}} & & \Delta w_{ks} & = & -\varepsilon\frac{\partial E_1}{\partial w_{ks}} \end{array}$$

4. Uprav hodnoty parametrů pro $s = 1, \cdots, m$ a $k = 1, \cdots, h$

$$\begin{array}{rclcrcl} \vec{c}_k(\tau) & = & \vec{c}_k(\tau-1) + \Delta\vec{c}_k & & b_k(\tau) & = & b_k(\tau-1) + \Delta b_k \\ \mathbf{\Sigma_k^{-1}}(\tau) & = & \mathbf{\Sigma_k^{-1}}(\tau-1) + \Delta\mathbf{\Sigma_k^{-1}} & & w_{ks}(\tau) & = & w_{ks}(\tau-1) + \Delta w_{ks} \end{array}$$

5. Spočti chybu sítě, je-li dotatečně malá, skonči.

6. Přejdi k bodu 2.

Algoritmus 3.1.1 představuje tzv. *akumulované učení*, kdy k adaptaci dochází po celém průchodu tréninkovou množinou. Hodnota chybové funkce závisí na všech tréninkových vzorech a k výpočtu hodnoty jejích derivací potřebujeme projít celou tréninkovou množinu. Nahradíme-li funkci E_1 parciální chybovou funkcí $E_1^t = \frac{1}{2}\sum_{s=1}^{m}(d_s^{(t)} - f_s^{(t)})^2$, jejíž hodnota závisí pouze na tréninkovém vzoru $\vec{x}^{(t)}$, můžeme adaptaci provádět po každém předložení tréninkového vzoru a snížíme nároky na paměť. Tento algoritmus však už neminimalizuje původní chybovou funkci a může zpomalovat konvergenci.

3.2 Jiné varianty gradientního algoritmu

Základní algoritmus 3.1.1 není moc efektivní. Při malé rychlosti učeni ε je výpočet pomalý, při větší rychlosti často diverguje. Jedna z možností vylepšení gradientního algoritmu je použití tzv. *momentu*.

Při výpočtu změny váhy ve směru gradientu uvažujeme i předchozí změnu vah.

$$\Delta P(\tau) = -\varepsilon \frac{\partial E_1}{\partial P} + \alpha \Delta P(\tau - 1), \tag{3.2}$$

kde P představuje příslušný z parametrů $\vec{c}_k$, b_k, $\mathbf{\Sigma_k^{-1}}$ nebo $w_k s$ a $0 < \alpha < 1$ je parametru momentu.

Jinou možností je *metoda sdružených gradientů* nebo *Newtonova metoda*, které jsou popsané v [4, str. 215-217].

3.3 Derivace chybové funkce

Pro výpočet gradientního algoritmu potřebujeme vyjádřit derivace chybové funkce podle všech hledaných parametrů, t.j. vah, středů, šířek a matice normy.

Derivaci chybové funkce

$$E_1 = \frac{1}{2} \sum_{t=1}^{N} \sum_{s=1}^{m} (e_s^t)^2, \qquad e_s^t = (f_s(\vec{x}^t) - d_s^t)^2 \tag{3.3}$$

podle jednotlivých parametrů snadno vyjádříme použitím pravidel o derivaci součtu a složené funkce. Označme $\varphi_k\left(\vec{x}^t\right) = \varphi\left(\frac{\|\vec{x}^t - \vec{c}_k\|_{\mathbf{C_k}}}{b_k}\right)$ a $\varphi_k'\left(\vec{x}^t\right) = \varphi'\left(\frac{\|\vec{x}^t - \vec{c}_k\|_{\mathbf{C_k}}}{b_k}\right)$.

$$\frac{\partial E_1}{\partial w_{kq}} = \sum_{t=1}^{N} e_q^t \varphi_k\left(\vec{x}^t\right) \tag{3.4}$$

$$\frac{\partial E_1}{\partial \vec{c}_k} = -\frac{\mathbf{\Sigma_k^{-1}}}{b_k} \sum_{t=1}^{N} \frac{[\vec{x}^t - \vec{c}^k]}{\| \vec{x}^t - \vec{c}^k \|_{C_k}} \varphi_k'\left(\vec{x}^t\right) \sum_{s=1}^{m} e_s^t w_{ks} \tag{3.5}$$

$$\frac{\partial E_1}{\partial b_k} = -\frac{1}{b_k^2} \sum_{t=1}^{N} \| \vec{x}^t - \vec{c}^k \|_{C_k} \varphi_k'\left(\vec{x}^t\right) \sum_{s=1}^{m} e_s^t w_{ks} \tag{3.6}$$

$$\frac{\partial E_1}{\partial \mathbf{\Sigma_k^{-1}}} = \frac{1}{2b_k} \sum_{t=1}^{N} \frac{[\vec{x}^t - \vec{c}^k][\vec{x}^t - \vec{c}^k]^T}{\| \vec{x}^t - \vec{c}^k \|_{C_k}} \varphi_k'\left(\vec{x}^t\right) \sum_{s=1}^{m} e_s^t w_{ks} \tag{3.7}$$

3.4 Výpočet hodnot derivací chybové funkce

V algoritmu 3.1.1 potřebujeme spočítat hodnoty $\frac{\partial E_1}{\partial w_{kq}}$, $\frac{\partial E_1}{\partial \vec{c}_k}$, $\frac{\partial E_1}{\partial b_k}$ a $\frac{\partial E_1}{\partial \mathbf{\Sigma_k^{-1}}}$ dle vztahu (3.4–3.7).

K tomu je třeba jeden průchod tréninkovou množinou. Po předložení aktuálního tréninkového vzoru síti nejprve pro každou jednotku k spočteme hodnotu vnitřního potenciálu ξ_k , výstup dané jednotky y_k a hodnotu derivace y'_k. Poté přejdeme k výstupní vrstvě a spočteme hodnoty jejích výstupů a hodnoty chyb vůči tomuto tréninkovému vzoru. Poté se sítí opět vracíme a na základě spočtených hodnot vyhodnotíme $\frac{\partial E_1^{(t)}}{\partial w_{ij}}$, $\frac{\partial E_1^{(t)}}{\partial \vec{c}_i}$, $\frac{\partial E_1^{(t)}}{\partial b_i}$ a $\frac{\partial E_1^{(t)}}{\partial \mathbf{\Sigma_i^{-1}}}$. Přesný postup je popsán algoritmem 3.4.1.

Algoritmus 3.4.1:
Vstup: RBF síť s n vstupními, h skrytými, m výstupními neurony. Tréninková množina $\vec{x}^t; t = 1, \cdots, N$
Výstup: Hodnoty $\frac{\partial E_1^{(t)}}{\partial w_{ij}}$, $\frac{\partial E_1^{(t)}}{\partial \vec{c}_i}$, $\frac{\partial E_1^{(t)}}{\partial b_i}$ a $\frac{\partial E_1^{(t)}}{\partial \mathbf{\Sigma^{-1}}_i}$.
Značení:

$$A_k^t = \frac{[\vec{x}^t - \vec{c}^k]}{\| \vec{x}^t - \vec{c}^k \|_{C_k}} \quad B_k^t = \| \vec{x}^t - \vec{c}^k \|_{C_k} \quad C_k^t = \frac{[\vec{x}^t - \vec{c}^k][\vec{x}^t - \vec{c}^k]^T}{\| \vec{x}^t - \vec{c}^k \|_{C_k}} \tag{3.8}$$

1. $\frac{\partial E_1^{(t)}}{\partial w_{ij}} := 0$, $\frac{\partial E_1^{(t)}}{\partial \vec{c}_i} := \vec{0}$, $\frac{\partial E_1^{(t)}}{\partial b_i} := 0$ a $\frac{\partial E_1^{(t)}}{\partial \mathbf{\Sigma_i^{-1}}} := \mathbf{0}$

2. $t := 1$

3. Pro všechna s: $e_s := 0$

4. Pro každou jednotku k

   - spočti potenciál $\xi_k(\vec{x}^t)$ a členy A_k^t, B_k^t a C_k^t
   - spočti $\varphi_k(\vec{x}^t)$ a $\varphi'_k(\vec{x}^t)$
   - pro všechna s: $e_s := e_s + w_{ks}\varphi_k(\vec{x}^t)$

5. Pro všechna s: v e_s máme výstup s-té jednotky, odečteme od něj d_s^t a získáme odchylku s-té jednotky od požadovaného výstupu.
   $\frac{\partial E_1}{\partial w_{ij}} = \frac{\partial E_1}{\partial w_{ij}} + e_j^t \varphi_i(\vec{x}^t)$

6. Pro každou jednotku k

   - spočti $D = \varphi'_k(\vec{x}^t) \sum_{s=1}^m e_s^t w_{ks}$
   - $\frac{\partial E_1}{\partial \vec{c}_k} = \frac{\partial E_1}{\partial \vec{c}_k} + A_k^t D$
   - $\frac{\partial E_1}{\partial b_k} = \frac{\partial E_1}{\partial b_k} + B_k^t D$
   - $\frac{\partial E_1}{\partial \mathbf{\Sigma_k^{-1}}} = \frac{\partial E_1}{\partial \mathbf{\Sigma_k^{-1}}} + C_k^t D$

7. Pokud $t < N$, $t := t + 1$ a přejdi k bodu 3.

8. Pro každou jednotku k

   - $\frac{\partial E_1}{\partial \vec{c}_k} = -\frac{\mathbf{\Sigma^{-1}}_k}{b_k} \frac{\partial E_1}{\partial \vec{c}_k}$
   - $\frac{\partial E_1}{\partial b_k} = -\frac{1}{b_k^2} \frac{\partial E_1}{\partial b_k}$
   - $\frac{\partial E_1}{\partial \mathbf{\Sigma_k^{-1}}} = \frac{1}{b_k} \frac{\partial E_1}{\partial \mathbf{\Sigma_k^{-1}}}$

3.4.1 Gaussova funkce

Vzorce pro výpočet derivace (3.4–3.7) předpokládají pouze, že RBF jednotky počítají nějakou obecnou radiální funkci φ. Podívejme se nyní na nejrozšířenější případ, RBF jednotky s Gaussovo funkcí, t.j. $\varphi(z) = e^{-z^2}$.

$$\frac{\partial E_1}{\partial w_{kq}} = \sum_{t=1}^{N} e_q^t e^{-\left(\frac{\|\vec{x}^t - \vec{c}_k\|_{C_k}}{b_k}\right)^2} \tag{3.9}$$

$$\frac{\partial E_1}{\partial \vec{c}_k} = \frac{2}{b_k^2} \mathbf{\Sigma^{-1}}_k \sum_{t=1}^{N} [\vec{x}^t - \vec{c}_k] \sum_{s=1}^{m} w_{ks} e_s^t e^{-\left(\frac{\|\vec{x}^t - \vec{c}_k\|_{C_k}}{b_k}\right)^2} \tag{3.10}$$

$$\frac{\partial E_1}{\partial b_k} = \frac{2}{b_k^3} \sum_{t=1}^{N} (\| \vec{x}^t - \vec{c}_k \|_{C_k})^2 \sum_{s=1}^{m} w_{ks} e_s^t e^{-\left(\frac{\|\vec{x}^t - \vec{c}_k\|_{C_k}}{b_k}\right)^2} \tag{3.11}$$

$$\frac{\partial E_1}{\partial \mathbf{\Sigma^{-1}}_k} = -\frac{1}{b_k^2} \sum_{t=1}^{N} [\vec{x}^t - \vec{c}^k][\vec{x}^t - \vec{c}^k]^T \sum_{s=1}^{m} w_{ks} e_s^t e^{-\left(\frac{\|\vec{x}^t - \vec{c}_k\|_{C_k}}{b_k}\right)^2} \tag{3.12}$$

Postup výpočtu je stejný jako v algoritmu 3.4.1, ale výpočet se zjednodušší. Srovnáme-li vzorečky (3.4 – 3.7) se vzorečky (3.9 – 3.12), vidíme, že místo členu $\varphi_k'(\vec{x}^t) \sum_{s=1}^{m} e_s^t w_{ks}$ (viz také krok 6 algoritmu 3.4.1) nyní potřebujeme počítat jednodušší člen

$$\sum_{s=1}^{m} w_{ks} e_s^t e^{-\left(\frac{\|\vec{x}^t - \vec{c}_k\|_{C_k}}{b_k}\right)^2}.$$

Tedy odpadá výpočet derivace $\varphi_k'(\vec{x}^t)$ a navíc jednotlivé sčítance sumy počítáme již v kroku 5. Můžeme je tedy rovnou sčítat a výpočet tohoto členu v kroku 6 odpadá.

Gaussova funkce má navíc tu výhodu, že svůj parametr umocňuje. Vždy stačí spočítat jen hodnotu potenciálu na druhou a tedy při vyhodnocení vzdálenosti nemusíme odmocňovat.

3.5 Poznámky

Uvedli jsme gradientní algoritmus pro učení obecné RBF sítě. Uvažovali jsme RBF síť, která má RBF jednotky určené středem, šířkou a maticí normy. Algoritmus lze samozřejmě použít pro RBF síť, jíž některý z těchto parametrů chybí. Například uvažujeme-li RBF síť, jejíž jednotky nemají adaptovatelnou šířku, v příslušných vzorcích nahradíme šířku jedničkou. Stejně tak RBF síť používající euklidovskou normu odpovídá RBF síti, kde matice norem jsou jednotkové.

Další možností je RBF síť, která používá pro všechny jednotky jednotnou normu. Norma pak není parametrem RBF jednotky, ale RBF sítě. Adaptuje se stejně jako ostatní parametry, potřebnou derivaci spočteme jako

$$\frac{\partial E_1}{\partial \mathbf{\Sigma}^{-1}} = \sum_{s=1}^{m} \frac{1}{2b_s} \sum_{t=1}^{N} \frac{[\vec{x}^t - \vec{c}^s][\vec{x}^t - \vec{c}^s]^T}{\| \vec{x}^t - \vec{c}^s \|_C} \varphi_k' \left(\vec{x}^t\right) \sum_{s=1}^{m} e_s^t w_{ks} \ . \tag{3.13}$$

Srovnáme-li vzorec (3.13) a (3.7), vidíme že tuto derivaci můžeme spočítat jako sumu derivací (3.7).

Kapitola 4

Třífázové učení

Třífázové učení využívá skutečnosti, že parametry RBF sítě lze rozdělit do tří skupin (středy, šířky, váhy) podle významu. První skupinu tvoří středy RBF jednotek, které určují středy radiálních oblastí významných výstupů těchto jednotek. Druhou skupinou jsou další parametry, které RBF jednotka může mít, a to šířky a matice norem. Tyto parametry ovlivňují velikost příslušné radiální oblasti. Třetí skupina obsahuje hodnoty vah spojů mezi skrytou a výstupní vrstvou. Každou skupinu učíme v samostatné fázi metodou, která nejlépe odpovídá charekteru daných parametrů.

Sítí postupujeme směrem dopředu. Nejprve určíme skrytou vrstvu. Naším cílem je nastavit parametry RBF jednotek tak, aby jejich oblasti působnosti pokrývali celý vstupní prostor a zároveň, aby se navzájem příliš nepřekrývali.

Učení RBF jednotek je navíc rozděleno do dvou fází. V první fázi rozmístíme středy RBF jednotek a tedy středy odpovídajících radiálních oblastí po vstupním prostoru tak, aby co nejlépe aproximovaly hustotu vstupních dat. Vycházíme z předpokladu, že vyskytují-li se v nějaké oblasti vstupního prostoru vstupní vektory spíše vyjímečně, postačí nám pokrýt tuto oblast několika málo RBF jednotkami. Naopak tam, kde je hustota vstupních vzorů vysoká, bude zřejmě potřeba více RBF jednotek. V této fázi probíhá učení bez učitele, tedy z tréninkové množiny používáme pouze vstupní části tréninkových vzorů.

V druhé fázi určíme zbývající parametry RBF jednotek, které typicky ovlivňují velikost radiálních oblastí.

V třetí — závěrečné — fázi zbývá nastavit váhy spojů mezi skrytou a výstupní vrstvou, což jsou váhy lineární kombinace.

První fází a různými metodami vektorové kvantizace se budeme zabývat v části 4.1. V části 4.2 popíšeme způsoby nastavení šířek a matic vážených norem. Nastavením vah se zabývá část 4.3.

4.1 První fáze — samoorganizace

Úkolem první fáze je rozmístit h středů ve vstupním prostoru tak, aby co nejlépe aproximovaly hustotu výskytu vzorů. Z tréninkové množiny nás nyní zajímají pouze vstupní části tréninkových vzorů. V této kapitole proto budeme pojmem tréninková množina rozumět množinu $T = \{\vec{x}^t; t = 1, \cdots, k\}$. Takovéto učení, kde máme k dispozici pouze vstupní části vzorů, se nazývá *učení bez učitele* nebo *samoorganizace.*

V praxi se často dává přednost rychlým jednoduchým metodám před časově náročným hledáním optimálního řešení. Osvědčilo se *rovnoměrné rozmístění* středů po vstupním prostoru, výhodné zejména v případě, kdy vstupní data jsou také rozmístěna rovnoměrně. Další možností je náhodné vybrání h *tréninkových vzorů.*

Pokud hledáme takové rozmístění, které opravdu zachycuje rozložení vzorů z tréninkové množiny, použijeme některý z algoritmů *vektrové kvantizace.*

Problém *vektorové kvantizace* vznikl v souvislosti se zpracováním a aproximací signálů. Úkolem je aproximovat hustotu výskytu reálných vstupních vektorů pomocí konečného počtu reprezentantů $\vec{c}_1, \cdots, \vec{c}_h$. Každému vektoru $\vec{x} \in \mathbb{R}^n$ pak přísluší reprezentant $\vec{c}_c$, který je mu nejblíže:

$$c = \mathrm{argmin}_{i=1,..h}\{\| \vec{x}^t - \vec{c}_i \|\}. \tag{4.1}$$

Kvalita řešení je dána chybovou funkcí

$$E = \int \| \vec{x} - \vec{c}_c \|^2 p(\vec{x})d\vec{x}, \tag{4.2}$$

kde $p(\vec{x})$ je pravděpodobnost výskytu vektoru $\vec{x}$. V našem případě přesně neznáme hustotu pravděpodobnosti, ale máme k dispozici tréninkovou množinu. Chybovou funkci proto budeme počítat jako

$$E = \frac{1}{k}\sum_{t=1}^{k} \| \vec{x}^t - \vec{c}_c \|^2 . \tag{4.3}$$

Metody vektorové kvantizace se snaží minimalizovat tuto funkci. Vzhledem k tomu, že index c závisí jak na vzorech $\vec{x}$ tak na jednotkách $\vec{c}$, není triviální vyjádřit derivaci chyby vzhledem k vektorům $\vec{c}$. Obvyklé gradientní metody proto nejsou vhodné a používají se různé heuristické iterativní algoritmy.

4.1.1 Lloydův algoritmus

Základní algoritmus řešící problém vektorové kvantizace, se nazývá **Lloydův algoritmus**. Na začátku inicializujeme reprezentanty $\vec{c}_j$ náhodně. Pak procházíme tréninkovou množinu a ke každému vstupu $\vec{x}_t$ přiřadíme reprezentanta $\vec{c}_c$ dle (4.1). Pro každého reprezentanta spočteme těžiště množiny všech vstupů, jimž byl tento reprezentant přiřazen

$$\vec{t}_j = \frac{1}{|T_j|} \sum_{\vec{x}_j \in T_j} \vec{x}_j \tag{4.4}$$

$$T_j = \{x^t; j = \text{argmin}_{i=1,\cdots,h} \parallel \vec{x}^t - \vec{c}_i \parallel\} \tag{4.5}$$

a nahradíme hodnotu $\vec{c}_j$ hodnotou $\vec{t}_j$. Spočteme chybu a přejdeme k dalšímu průchodu tréninkovou množinou. Výpočet opakujeme dokud chyba klesá nebo není dostatečně malá.

Algoritmus 4.1.1:
Vstup: $T = \{\vec{x}^t; t = 1, \cdots, k\}$, počet reprezentantů h, kritérium ukončení výpočtu.
Výstup: $\vec{c}_j; j = 1, \cdots, h$

1. Rozmísti $\vec{c}_j$ náhodně.
2. Ke každému vektoru $\vec{x}^t \in T$ přiřaď nejbližšího reprezentanta $\vec{c}_c$, zařaď $\vec{x}^t$ do T_c.
3. Pro $j = 1, \cdots, h$ spočti těžiště $\vec{t}_j$ množiny T_j, nahraď $\vec{c}_c$ hodnotou $\vec{t}_j$
4. Spočti chybu, pokud je splněno dané kritérium, skonči.
5. Přejdi k bodu 2.

4.1.2 Kohonenovo učení

Dalším algoritmem je tzv. **Kohoneno učení** (v literatuře často označované jako *k-means clustering*). Jedná se o modifikaci Lloydova algoritmu, kdy k adaptaci dochází po každém předložení tréninkového vzoru.

Algoritmus lze reprezentovat jednoduchou samoorganizační neuronovou sítí. Vstupní vrstva je tvořena n neurony a slouží pouze k přenosu vstupních hodnot $\vec{x} \in \mathbb{R}^n$. Druhá, výstupní vrstva se skládá z h jednotek, které představují hledané reprezentanty. Každé výstupní jednotce přísluší váhový vektor $\vec{c}_j = (c_1, \cdots, c_h)$, který určuje její umístění ve vstupním prostoru. Výstupy sítě nabývají hodnot $\{0, 1\}$ a to tak, že výstupní hodnotu 1 má vždy právě jeden neuron. Takový neuron se nazývá *aktivní*. Po předložení tréninkového vzoru $\vec{x}^t$, si každý neuron spočte svoji vzdálenost od tohoto vstupu $\| \vec{x}^t - \vec{c} \|$. Poté proběhne kompetice, kterou vyhrává neuron j, pro který platí

$$j = \mathrm{argmin}_{i=1,\cdots,h}\{\| \vec{x}^t - \vec{c}_i \|\}. \tag{4.6}$$

Tento neuron se stává aktivním.

Učení této jednoduché sítě vychází z Lloydova algoritmu. Procházíme tréninkovou množinu, každý vzor předložíme síti a výherce kompetice j posuneme směrem k danému vstupu

$$\vec{c}_j^{\,t} = \vec{c}_j^{\,t-1} + \theta(\vec{x}^{\,t-1} - \vec{c}_j^{\,t-1}) \tag{4.7}$$

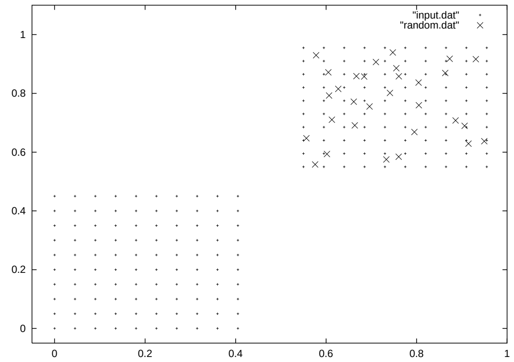
Parametr $0 < \theta \leq 1$ určuje míru změny, na počátku učení je dobré jej volit blízký jedné, během učení jej snižovat až k nule.

Algoritmus 4.1.2:
Vstup: $T = \{\vec{x}^t; t = 1, \cdots, k\}$, počet reprezentantů h, kritérium ukončení výpočtu.
Výstup: $\vec{c}_j; j = 1, \cdots, h$

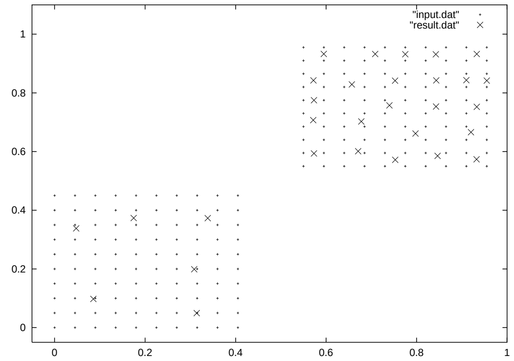
1. Rozmísti $\vec{c}_j$ náhodně.
2. Ke každému vektoru $\vec{x}^t \in T$ urči výherce kompetice c a uprav $\vec{c}_c$ podle vzorce (4.7).
3. Spočti chybu, pokud je splněno dané kritérium, skonči.
4. Přejdi k bodu 2.

Výše uvedené algoritmy však řeší problém vektorové kvantizace jen tehdy, jsou-li vstupní hodnoty rozmístěny víceméně rovnoměrně v jedné konvexní oblasti. Podívejme se na obrázek 4.1.2. Vstupní data se vyskytují ve dvou čtercových oblastech, reprezentanty jsme při inicializaci umístili pouze do horní oblasti. Algoritmus 4.1.2 najde řešení zobrazené na obrázku 4.1.2. Jakmile je do dolní oblasti přitaženo několik nejbližších reprezentantů z horní oblasti, tito reprezentanti vyhrají všechny kompetice pro vstupy z dolní oblasti. To znamená, že žádný další reprezentat sem nebude přesunut. V nejhorším případě se může stát, že se v datech vyskytne oblast s pouze jedním reprezentantem, ačkoliv hustota výskytu vzorů v této oblasti může být vysoká.

K odstranění tohoto problému se používají různé heuristiky. Jednou z nich je tzv. *radiální růst*. Začneme s reprezentanty blízkými nule a tréninkovou množinou $T' = \{\beta\vec{x}^t; \vec{x}^t \in T\}$, kde T je původní tréninková množina a β



Obrázek 4.1: Kohonenovo učení - počáteční rozmístění reprezentantů



Obrázek 4.2: Kohonenovo učení - výsledné rozmístění reprezentantů

je parametr blízký nule. Na začátku učení jsou tedy všechny vzory blízké všem reprezentantům. V průběhu učení zvyšujeme číslo β a nakonec učíme původní data. Toto řešení je však časově náročné. Další řešení navržené Desienem(1988) je tzv. *metoda lokální paměti.* Vychází z předpokladu, že každý reprezentant by při stejné pravděpodobnosti měl zvítězit zhruba $1/h$-krát. Je-li tedy četnost výher nějakého reprezentanta příliš vysoká, vyřadíme ho na nějakou dobu ze soutěže. V samoorganizační síti Kohonenova učení přiřadíme každému neuronu dva parametry, relativní četnost výher f a práh b. Učení pak probíhá tak, že po předložení tréninkového vzoru $\vec{x}^t$ určíme vítězný neuron c a upravíme parametry f a b podle vzorce:

$$f_c^t = (1-\beta)f_c^{t-1} + \beta \tag{4.8}$$

$$f_j^t = (1-\beta)f_j^{t-1}, j \neq c \tag{4.9}$$

$$b_j^t = \gamma(\frac{1}{h} - f_j^t) \tag{4.10}$$

Po té proběhne druhá kompetice

$$c = \text{argmin}_{i=1,..h}\{\| \vec{x}^t - \vec{w}_i \| - b_i\} \tag{4.11}$$

a vítěznému neuronu c změníme váhy dle vzorce (4.7).

Výše uvedené algoritmy lze najít v [9]. Tam je také popsán složitější model samoorganizační sítě *Kohonenova samoorganizační mapa*, která vychází z *Kohonenova učení*. Její neurony jsou však navíc uspořádány do nějaké topologické struktury, která určuje, které neurony spolu sousedí. Učení pak probíhá podobně jako Kohonenovo učení, ale spolu s vítěznými neuronem jsou navíc posunovány i všechny neurony z vítězova okolí.

Různými modifikacemi Lloydova algoritmu a dalšími metodami se zabývá [3], spíše však z hlediska kódování. Pro určení středů RBF jednotek postačí popsané metody, poslední z nich — *Kohonenovo učení* —je zřejmě nejpoužívanější.

4.2 Druhá fáze — nastavení šířek

Úkolem této fáze je nastavit zbývající parametry RBF jednotek, což jsou v našem případě šířky a matice vážených norem.

V [9] je popsán postup pro RBF sítě, jejichž RBF jednotky realizují Gaussovu funkci a mají kromě středu pouze jeden parametr, a to šířku. Zaměříme se nejprve na tento případ.

Podíváme-li se na vzorec pro výpočet odezvy RBF jednotky s Gaussovo funkcí

$$y(\vec{x}) = e^{-\left(\frac{\|x-c\|}{b}\right)^2},$$

vidíme, že tato jednotka dává významné výstupy v radiální oblasti kolem středu $\vec{c}$, přičemž šířka b určuje velikost této oblasti.

Šířky proto nesmí být moc malé, aby ve vstupním prostoru nevznikly oblasti, kde nebude mít žádná z jednotek významnou odezvu. Na druhou stranu nesmí být moc velké, protože pak se oblasti působnosti navzájem překrývají a RBF jednotky ztrácí svůj lokální charakter.

Chybová funkce $E(b_1, \cdots, b_h)$ nabývá minima, právě když jsou tyto dva požadavky vyvážené. Parametr P udává míru překrytí sousedních oblastí.

$$E(b_1, \cdots, b_h) = \frac{1}{2}\sum_{r=1}^{h}\left[\sum_{s=1}^{h} e^{-\left(\frac{\|c_s-c_r\|}{b_r}\right)^2}\left(\frac{\| c_s - c_r \|}{b_r}\right)^2 - P\right]^2 \tag{4.12}$$

V praxi se většinou minimalizace této funkce nahrazuje jednoduchou heuristikou. Šířka RBF jednotek se nastavuje úměrně průměru vzdáleností několika nejbližších sousedů.

V případě naší RBF sítě nepočítáme vzdálenosti pomocí euklidovské metriky, ale pomocí vážených norem. Vztah (4.12) proto zobecníme pro náš případ.

$$E(b_1, \cdots, b_h; \mathbf{\Sigma_1^{-1}}, \cdots, \mathbf{\Sigma_h^{-1}}) = \frac{1}{2} \sum_{r=1}^{h} \left[\sum_{s=1}^{h} e^{-\left(\frac{\| c_s - c_r \|_{\mathbf{C_i}}}{b_r}\right)^2} \left(\frac{\| c_s - c_r \|_{\mathbf{C_i}}}{b_r} \right)^2 - P \right]^2 \tag{4.13}$$

Minimum této funkce budeme hledat jednoduchou gradientní metodu obdobně jako v 3.1.

Algoritmus 4.2.1:

1. Nastav hodnoty $b_1, \cdots, b_h$ a $\mathbf{\Sigma_1^{-1}}, \cdots, \mathbf{\Sigma_h^{-1}}$ náhodně.
2. Pokud je hodnota chyby spočtená dle (4.13) dostatečně malá, skonči.
3. Uprav hodnoty parametrů podle

$$b_i = b_i - \varepsilon \frac{\partial E}{\partial b_i} \tag{4.14}$$

$$\mathbf{\Sigma^{-1}}_i = \mathbf{\Sigma_i^{-1}} - \varepsilon \frac{\partial E}{\partial \mathbf{\Sigma_i^{-1}}} \tag{4.15}$$

pro $i = 1, \ldots, h$.

4. Přejdi ke kroku 2.

Středy RBF jednotek už známe, zbývá vyjádřit hodnoty derivací

$$\frac{\partial E}{\partial b_k} = 2 \left[\sum_{s=1}^{h} e^{-\xi_{sk}^2} \xi_{sk}^2 - P \right] \left[\sum_{s=1}^{h} e^{-\xi_{sk}^2} \left(\xi_{sk}^2 - 1 \right) \frac{\| \vec{c}_s - \vec{c}_k \|_{C_k}^2}{b_k^3} \right]$$

$$\frac{\partial E}{\partial \mathbf{\Sigma_k^{-1}}} = \frac{1}{b_k^2} \left[\sum_{s=1}^{h} e^{-\xi_{sk}^2} \xi_{sk}^2 - P \right] \left[\sum_{s=1}^{h} e^{-\xi_{sk}^2} [\vec{c}_s - \vec{c}_k][\vec{c}_s - \vec{c}_k]^T (1 - \xi_{sk}^2) \right]$$

$$\xi_{sk} = \frac{\| c_s - c_k \|_{C_k}}{b_k}$$

Následují vzorce pro obecnou RBF funkci φ

$$E = \frac{1}{2} \sum_{r=1}^{h} \left[\sum_{s=1}^{h} \varphi \left(\frac{\| c_s - c_r \|_{C_r}}{b_r} \right) \left(\frac{\| c_s - c_r \|_{C_r}}{b_r} \right)^2 - P \right]^2 \tag{4.16}$$

$$\frac{\partial E}{\partial b_k} = -\left[\sum_{s=1}^{h} \varphi\left(\xi_{sr}\right) \xi_{sr}^2 - P\right] \sum_{s=1}^{h} \frac{\| c_s - c_r \|_{C_r}^2}{b_r^3} \left[\varphi'\left(\xi_{sr}\right) \xi_{sr} + 2\varphi\left(\xi_{sr}\right)\right]$$

$$\frac{\partial E}{\partial \mathbf{\Sigma^{-1}}_k} = \frac{1}{2b_r^2} \left[\sum_{s=1}^{h} \varphi\left(\xi_{sr}\right) \xi_{sr}^2 - P\right] \sum_{s=1}^{h} \frac{[\vec{c}_s - \vec{c}_k][\vec{c}_s - \vec{c}_k]}{\| \vec{c}_s - \vec{c}_k \|_{C_k}} \left[\varphi'\left(\xi_{sr}\right) \xi_{sr} + 2\varphi\left(\xi_{sr}\right)\right]$$

4.3 Třetí fáze, problém nejmenších čtverců

Do třetí fáze vstupujeme s RBF sítí, která má pevně určenou skrytou vrstvu. Zbývá nám nastavit hodnoty vah spojů vedoucích do výstupní vrstvy. Požadujeme, aby funkce sítě odpovídala funkci zadané tréninkovou množinou

$$T = \{(\vec{x}^t, \vec{d}^t); \quad t = 1, \cdots, k\}.$$

K tomuto problému budeme přistupovat dvěma odlišnými způsoby. V prvním z nich zformuluje soustavu rovnic, kde neznámými budou hledané váhy, a tuto soustavu budeme řešit. V druhém případě (viz str. 26) budeme minimalizovat chybovou funkci sítě. K tomu použijeme buď gradientní metodu nebo hledání minima položením prvních derivací minimalizované funkce rovných nule.

V obou případech bude třeba vyřešit soustavu lineárních rovnic, která nemusí a ve většině případů nebude mít jednoznačné řešení. Budeme tedy hledat řešení, které je optimální ve smyslu nejlepších čtverců. Numerické metody nejmenších čtverců popíšeme v části 4.3.1

Nyní přistoupíme k prvnímu způsobu řešení. Chceme-li, aby síť pro vstupy $\vec{x}$ z tréninkové množiny dávala odpovídající výstupy $\vec{d}$, musí platit

$$f_s(\vec{x}^t) = \sum_{j=0}^{h} w_{js} y_j(\vec{x}^t) = d_s^t \tag{4.17}$$

pro $s = 1, \cdots, m$, kde m je počet výstupních neuronů, d_s je s-tá složka vektoru $\vec{d}^t$ a $y_j(\vec{x}^t)$ je výstup j-té RBF jednotky po předložení vstupu $\vec{x}^t$.

Protože výstupy RBF jednotek už dokážeme spočítat, dostali jsme soustavu lineárních rovnic s neznámými w_{js}. Tu můžeme přepsat pomocí součinu matic jako

$$\begin{bmatrix} y_1(\vec{x}^1) & \dots & y_h(\vec{x}^1) \\ y_1(\vec{x}^2) & \dots & y_h(\vec{x}^2) \\ \vdots & \ddots & \vdots \\ y_1(\vec{x}^k) & \dots & y_h(\vec{x}^k) \end{bmatrix} \begin{bmatrix} w_{11} & \dots & w_{1m} \\ w_{21} & \dots & w_{2m} \\ \vdots & \ddots & \vdots \\ w_{h1} & \dots & w_{hm} \end{bmatrix} = \begin{bmatrix} d_1^1 & \dots & d_m^1 \\ d_1^2 & \dots & d_m^2 \\ \vdots & \ddots & \vdots \\ d_1^k & \dots & d_m^k \end{bmatrix} \tag{4.18}$$

Matici $k \times h$ výstupů RBF jednotek označme P, matici $h \times m$ vah W a matici $k \times m$ požadovaných výstupů D. Vztah (4.18) můžeme přepsat jako

$$\mathbf{PW} = \mathbf{D}. \tag{4.19}$$

Obvykle máme více tréninkových vzorů než RBF jednotek a dostaneme tak přeurčenou soustavu rovnic. V tom případě hledáme takové řešení, jehož chyba je nejmenší. K tomu použijeme některou z metod nejmenších čtverců. Těmto metodám bude věnována část 4.3.1.

Nyní se podívejme na druhý přístup k hledání optimálních vah. Požadujeme, aby funkce sítě co nejvíce odpovídala zadané tréninkové množině. To znamená, že chceme minimalizovat funkci (2.12):

$$E_1 = \frac{1}{2}\sum_{t=1}^{k} \| \vec{d}^{(t)} - \vec{f}^{(t)} \|^2 = \frac{1}{2}\sum_{t=1}^{k}\sum_{s=1}^{m}(d_s^{(t)} - f_s^{(t)})^2 \quad ,$$

která je součtem čterců odchylek skutečných výstupů sítě od požadovaných výstupů.

V 2.4.1 jsme uvedli problém regularizace, kdy na funkci sítě klademe další požadavky, konkrétně chceme, aby příliš neoscilovala. Chybová funkce je pak rozšířena o regularizační člen

$$E(\mathbf{W}) = \frac{1}{2}\sum_{t=1}^{k}\sum_{s=1}^{m}(d_s^{(t)} - f_s^{(t)})^2 + \lambda\frac{1}{2}\sum_{t=1}^{k}\sum_{s=1}^{m}\sum_{i=1}^{n}\left(\frac{\partial^2 f_s^{(t)}}{\partial x_i^2}\right)^2 \quad . \tag{4.20}$$

Minimalizaci funkce E (4.20) lze opět provádět dvěma způsoby.

První možnost je vyjádřit si derivace E podle w_{ij} a použít klasickou gradientní metodu (viz 3.1). Tato metoda bývá obecně označována jako *LMS* algoritmus (Least-Mean-Squares).

Druhou možností je využití skutečnosti, že funkce nabývá lokálního extrému právě tehdy, když jsou její první derivace rovny nule. První derivace chybové funkce $\frac{\partial E}{\partial w_{ij}}$ položíme rovny nule a pro $q = 1, \dots, h$ a $r = 1, \dots, m$ dostaneme

$$\sum_{t=1}^{k}(d_r^{(t)} y_q(\vec{x}^{(t)})) - \sum_{j=1}^{h} w_{jr} \sum_{t=1}^{k}\left(y_j(\vec{x}^{(t)}) y_q(\vec{x}^{(t)}) + \lambda \sum_{i=1}^{n}\left(\frac{\partial^2 y_j^{(t)}}{\partial x_i^2}\frac{\partial^2 y_q^{(t)}}{\partial x_i^2}\right)\right) = 0 \tag{4.21}$$

kde $y_j(\vec{x}^t)$ je výstup j-té RBF jednotky po předložení vstupu $\vec{x}^t$. Použijeme opět maticovou notaci

$$\mathbf{\Phi W} = \mathbf{D} \ , \tag{4.22}$$

kde $\mathbf{D} \in \mathbb{R}^{h\times m}$, $\mathbf{\Phi} \in \mathbb{R}^{h\times h}$ a

$$\mathbf{D}_{ij} = \sum_{t=1}^{k} d_j^{(t)} y_i(\vec{x}^{(t)}), \tag{4.23}$$

$$\mathbf{\Phi}_{qr} = \sum_{t=1}^{k} \left(y_q(\vec{x}^{(t)}) y_r(\vec{x}^{(t)}) + \lambda \sum_{i=1}^{n} \left(\frac{\partial^2 y_q^{(t)}}{\partial x_i^2} \frac{\partial^2 y_r^{(t)}}{\partial x_i^2} \right) \right). \tag{4.24}$$

Opět nemáme zaručeno, že k matici $\mathbf{\Phi}$ existuje inverzní matice. Budeme tedy hledat takové řešení, jehož chyba je nejmenší ve smyslu nejmenších čtverců.

4.3.1 Problém nejmenších čtverců

Máme zadány matice $\mathbf{A}$ a $\mathbf{B}$ a hledáme takovou matici $\mathbf{X}$, že pro každou matici $\mathbf{X}'$ platí

$$||\mathbf{B} - \mathbf{A}\mathbf{X}'|| \geq ||\mathbf{B} - \mathbf{A}\mathbf{X}|| \tag{4.25}$$

Pokud neexistuje inverzní matice k matici A použijeme k řešení jednu z následujících metod.

Moore-Penroseova pseudoinverse
Řešení spočteme

$$\mathbf{X} = \mathbf{A}^{+}\mathbf{B}, \tag{4.26}$$

kde $\mathbf{A}^{+}$ je *pseudoinverze* matice A. *Pseudoinverzí* matice $\mathbf{A}$ nazýváme takovou matici $\mathbf{A}^{+}$, pro kterou platí

1. $\mathbf{A}\mathbf{A}^{+}\mathbf{A} = \mathbf{A}$	3. $(\mathbf{A}\mathbf{A}^{+})^T = \mathbf{A}\mathbf{A}^{+}$
2. $\mathbf{A}^{+}\mathbf{A}\mathbf{A}^{+} = \mathbf{A}^{+}$	4. $(\mathbf{A}^{+}\mathbf{A})^T = \mathbf{A}^{+}\mathbf{A}$.

Existuje-li k matici $\mathbf{A}$ inversní matice, je rovna její pseudoinversi. Pokud je matice $(\mathbf{A}^T\mathbf{A})$ regulární, pak matice $(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$ splňuje podmínky 1-4. Tato matice bývá označována jako *Moore-Penroseova pseudoinverse*.

SVD rozklad
Pro každou matici $\mathbf{A} \in \mathbb{R}^{m\times n}$ existují matice $\mathbf{U}$, $\mathbf{V}$ a $\mathbf{\Sigma}$ tak, že

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{4.27}$$

a $\mathbf{U}^T\mathbf{U} = \mathbf{V}^T\mathbf{V} = \mathbf{V}\mathbf{V}^T = \mathbf{I}^n$ a $\mathbf{\Sigma}$ je diagonální matice. Prvky diagonály $(\sigma_1, \cdots, \sigma_n)$ se nazývají *singulární čísla matice* $\mathbf{A}$. Pokud je hodnost matice $\mathbf{A}$ rovna r, pak $\sigma_{r+1}, \cdots, \sigma_n$ jsou rovny nule.

Dosadíme-li za $\mathbf{A}$ její SVD rozklad dostáváme

$$\mathbf{U\Sigma V}^T\mathbf{X} = \mathbf{B} \tag{4.28}$$

a tedy

$$\mathbf{X} = \mathbf{V\Sigma}^+\mathbf{U}^T\mathbf{B} \tag{4.29}$$

kde $\mathbf{\Sigma}^+ = diag(\sigma_i^+)$ a $\sigma_i^+ = 1/\sigma_i$, pro $\sigma_i \neq 0$ a $\sigma_i^+ = 0$, pro $\sigma_i = 0$.

G.H.Golub a C.Reinsch [10, str. 134-151] navrhli algoritmus na výpočet SVD rozkladu. Nejprve upravíme matici na bidiagonální tvar a po té se provádí vlastní rozklad. Redukce na bidiagonální tvar se provádí pomocí Householderových transformací [8, str. 52-57]

$$\mathbf{P}^{(k)} = \mathbf{I} - 2\vec{x}^{(k)}\vec{x}^{(k)T} \tag{4.30}$$

$$\mathbf{Q}^{(k)} = \mathbf{I} - 2\vec{y}^{(k)}\vec{y}^{(k)T} \tag{4.31}$$

tak, aby $\mathbf{P}^{(n)}\cdots\mathbf{P}^{(1)}\mathbf{A}\mathbf{Q}^{(1)}\cdots\mathbf{Q}^{(n-2)} = \mathbf{J}^0$ byla v horním bidiagonálním tvaru. Matice $\mathbf{J}^0$ má stejná singulární čísla jako matice $\mathbf{A}$. Tedy je-li

$$\mathbf{J}^0 = \mathbf{G\Sigma H}^T \tag{4.32}$$

pak

$$\mathbf{A} = \mathbf{PG\Sigma H}^T\mathbf{Q}^T \tag{4.33}$$

tak, že $\mathbf{U} = \mathbf{PQ}$, $\mathbf{V} = \mathbf{QH}$ a $\mathbf{P} = \mathbf{P}^1\cdots\mathbf{P}^n$ a $\mathbf{Q} = \mathbf{Q}^1\cdots\mathbf{Q}^{n-2}$.

SVD rozklad bidiaogonální matice probíhá jako iterativní diagonalizace $\mathbf{J}^0 \to \mathbf{J}^1\cdots \to \mathbf{\Sigma}$, kde $\mathbf{J}^{i+1} = \mathbf{S}^{iT}\mathbf{J}^i\mathbf{T}^i$ a $\mathbf{S}^i$ a $\mathbf{T}^i$ jsou transformační matice složené s Givensových rotací [8, str. 49-51].

QR rozklad

B.Businger a G.H.Golub [10, str. 111-118] navrhli řešení problému nejmenších čtverců pomocí Householderových transformací.

Máme matici $\mathbf{A} \in \mathbb{R}^{m\times n}$, kde $m \leq n$ a hodnost matice $\mathbf{A}$ je n, a matici $\mathbf{B}$. Hledáme $\mathbf{X}$ tak, aby pro každou matici $\mathbf{X}'$ platilo

$$\| \mathbf{B} - \mathbf{AX}' \| \geq \| \mathbf{B} - \mathbf{AX} \| \tag{4.34}$$

Platí, že $\| \mathbf{B} - \mathbf{AX} \| = \| \mathbf{QB} - \mathbf{QAX} \|$, pokud $\mathbf{Q}^T\mathbf{Q} = I$.

Zvolíme $\mathbf{Q}$ tak, aby

$$\mathbf{QA} = \mathbf{R} = \begin{pmatrix} \tilde{\mathbf{R}} \\ \mathbf{0} \end{pmatrix}, \tag{4.35}$$

kde $\tilde{\mathbf{R}}$ je horní trojúhelníková matice. Pak $\mathbf{X} = \mathbf{R}^{-1}(\mathbf{QB})$.

Dekompozici lze realizovat pomocí Householderových transformací

$$\mathbf{P}^k = \mathbf{I} - \beta_k \vec{u}^k \vec{u}^{kT} \qquad k = 1, \cdots, n \tag{4.36}$$

Vektory $\vec{u}^k$ a parametry β_k určíme tak, aby k-tá transformace vynulovala v k-tém sloupci všechny prvky pod diagonálou.

$$\begin{aligned}
\sigma_k &= \left(\sum_{i=k}^{m} (a_{ik}^k)^2 \right) & (4.37)\\
\beta_k &= \big(\sigma_k (\sigma_k + |a_{kk}^k|) \big) & (4.38)\\
u_i^k &= 0, \quad i < k & (4.39)\\
u_k^k &= sgn(a_{kk}^k) \left(\sigma_k + |a_{kk}^k| \right) & (4.40)\\
u_i^k &= a_{ik}^k, \quad i > k & (4.41)
\end{aligned}$$

Praktický výpočet probíhá iterativně. Po určení prvního řešení $\tilde{\mathbf{X}}$, položíme $\mathbf{X} = \tilde{\mathbf{X}} + \mathbf{E}$. Potom $\| \mathbf{B} - \mathbf{AX} \| = \| \mathbf{R} - \mathbf{AE} \|$, kde $\mathbf{R} = \mathbf{B} - \mathbf{A}\tilde{\mathbf{X}}$. Výpočet korekce $\mathbf{E}$ je snadný, neboť rozklad matice $\mathbf{A}$ už máme.

Kapitola 5

Genetické algoritmy

V této kapitole se budeme zabývat genetickým učením neuronových sítí typu RBF. Nejprve se v části 5.1 seznámíme s obecným genetickým algoritmem. Část 5.2 popisuje použití tohoto algoritmu pro učení RBF sítí, část 5.3 pak kanonickou verzi genetického učení RBF sítí. V části 5.4 navrhneme drobné vylepšení genetického algoritmu, týkající se významu účelové funkce a selekce. V části 5.5 zmíníme další možnosti využití popsaných genetických algoritmů. V poslední části 5.6 pak navrhneme použití genetických algoritmů na problém vektorové kvantizace.

5.1 Genetický algoritmus

Za počátek historie *genetických algoritmů* jsou považována 70. léta tohoto století a práce Johna Hollanda. *Genetický algoritmus* je stochastická prohledávací metoda inspirovaná evolučními principy jako jsou přirozený výběr, křížení a mutace.

Jedná se o alternativní metodou k řešení problémů optimalizace. Tedy takových problémů, které lze převést na hledání maxima funkce v závislosti na jejích parametrech. V terminologii genetických algoritmů tuto funkci nazýváme *účelovou funkcí.*

Genetický algoritmus pak hledá v prostoru všech přípustných parametrů takové hodnoty parametrů, pro něž je hodnota účelové funkce maximální. Každé přípustné nastavení parametrů považujeme za potencionální řešení problému. Algoritmus pracuje vždy z množinou potencionálních řešení. Každé řešení je zakódováno do konečného řetězce znaků (bitů, čísel), který je označován jako *jedinec.* Množinu jedinců pak nazýváme *populace.*

Výpočet probíhá iterativně a je popsán algoritmem 5.1.1. V každé iteraci sestrojíme novou populaci, tzv. *novou generaci*, pomocí genetických operátorů - selekce, křížení a mutace.

Selekce slouží k výběru jedince do nové populace v závislosti na hodnotě účelové funkce. Modeluje darwinistický přirozený výběr tak, že vybírá jedince náhodně s pravděpodobností přímo úměrnou odpovídající hodnotě účelové funkce.

Operátor křížení vytváří dva nové jedince kombinací dvou vybraných jedinců. Nejpoužívanějším typem křížení je tzv. *jednobodové křížení*, které se realizuje tak, že řetězce představující jedince se na náhodném místě rozdělí a jejich části se vymění.

Operátor mutace vnáší do populace náhodné změny. Realizuje se obvykle tak, že náhodně zvolíme pozici ve vybraném jedinci a danou jednotku (bit, znak,číslo) náhodně změníme.

Výpočet probíhá tak dlouho, dokud se v populaci neobjeví jedinec s dostatečně velkou hodnotou účelové funkce.

Algoritmus 5.1.1:
Vstup: pravděpodobnost mutace p_m, pravděpodobnost křížení p_k, počet jedinců v populaci N, kritérium ukončení výpočtu
Výstup: řešení daného problému

1. Start : Vytvoř počáteční populaci N jedinců $P(0) = \{I_1, \cdots, I_N\}$.
$i := 0$

2. Výpočet účelové funkce : Ke každému jedinci spočti odpovídající hodnotu účelové funkce.

3. Test : Je-li maximum účelové funkce v dané populaci dostatečně velké, skonči a vrať řešení odpovídající jedinci s maximální hodnotou účelové funkce.

4. Nová populace : Vytvoř prázdnou populaci $P(i+1)$ a opakuj následující postup, dokud populace nebude obsahovat N jedinců.

- **i) Selekce** : Vyber dva jedince z populace $P(i)$ v závislosti na jejich účelové funkci.
- **ii) Křížení** : S pravděpodobností p_k vytvoř dva nové jedince aplikací operace křížení na vybrané jedince. Pokud nedojde ke křížení, noví jedinci jsou kopií vybraných jedinců.
- **iii) Mutace** : Na každého z nových jedinců aplikuj s pravděpodobností p_m operaci mutace.
- **iv) Vložení** : Vlož nové jedince do populace $P(i + 1)$.

5. **Nová generace** : $i := i + 1$
Přejdi k bodu 2.

Chceme-li aplikovat algoritmus 5.1.1 na nějakou konkrétní úlohu, musíme určit

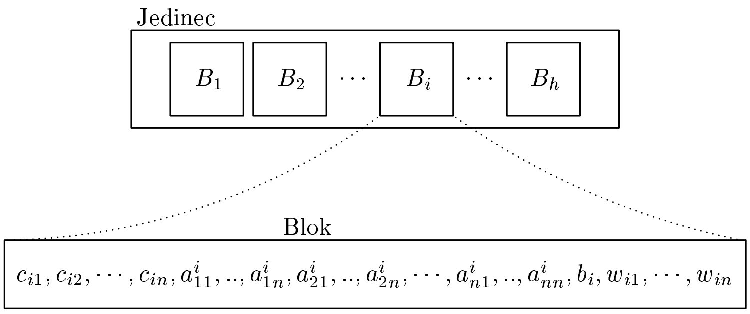
1. způsob zakódování potencionálního řešení do řetězce bitů resp. čísel, znaků.
2. způsob vytváření počáteční populace.
3. účelovou funkci
4. genetické operátory (popis realizace operátorů nad řetězci kódu)
5. parametry genetického algoritmu (velikost populace, pravděpodobnosti operátorů)

Genetické algoritmy se používají typicky v případech, kdy parametrický prostor je příliš velký pro úplné prohledávání a proto se spokojíme s přibližným řešením. Jeho síla spočívá v tzv. *implicitním paralelismu*, kdy populace jedinců paralelně prochází parametrický prostor a navzájem se ovlivňuje a modifikuje. Populace jedinců tak najde řešení rychleji, než kdyby prostor prohledávali izolovaně. Rychlost učení však vždy závisí na konkrétní aplikaci.

5.2 Učení RBF sítí pomocí GA

Učení neuronové sítě můžeme formulovat jako úlohu nalezení minima chybové funkce v závislosti na parametrech sítě. K použití genetického algoritmu na tuto úlohu potřebujeme zakódovat parametry sítě a dokázat ohodnotit takového jedince odpovídající hodnotou účelové funkce.

Zakódování parametrů
Připomeňme si model RBF sítě z části 2.2. Vidíme, že síť je určena svými RBF jednotkami a váhami spojů od těchto jednotek k výstupním jednotkám. Každá jednotka je pak určena svými parametry - středem, případně šířkou a maticí normy. Nabízí se tedy jednoduchá možnost zakódování parametrů sítě do řetězce h bloků, kde h je počet RBF jednotek a každý blok obsahuje zřetězení parametrů dané jednotky a vah spojů vedoucí z této jednotky do výstupní vrstvy. Zakódování je zřejmé z obrázku 5.2.Určení počtu RBF jednotek může být součástí učení. Délka jedince pak není konstantní.



Obrázek 5.1: Jedinec reprezentující neuronovou síť typu RBF. Blok reprezentující i-tou RBF jednotku, c_{ij} označuje souřadnici středu, a^i_{jk} prvek matice Σ_i^{-1}, b_i šířku a w_{ij} hodnotu váhy vedoucí k j-té výstupní jednotce

Počáteční populace

První populaci vygenerujeme náhodně. Neznáme-li počet RBF jednotek, zvolíme nějakou horní mez a každému jedinci určíme počet náhodně. Středy volíme náhodně ve vstupním prostoru (t.j oblast výskytu tréninkových vzorů).

Účelová funkce

Hodnotu účelové funkce $\mathcal{F}$ pro daného jedince spočteme na základě hodnoty chybové funkce sítě (viz část 2.4) určené tímto jedincem.

$$\mathcal{F}(I_i) = C - \sum_{t=1}^{k} \parallel \vec{d}^{(t)} - \vec{f}^{(t)} \parallel^2, \tag{5.1}$$

kde $C \in \mathbb{R}$ je odhad maximální chyby, případně maximální chyba v dané populaci. V případě, že délka jedince není pevně dána, přidáme k účelové funkci další člen, který bude penalizovat jedince s velkým počtem RBF jednotek.

Selekce

Použijeme tzv. *ruletovou selekci*. Představme si, že jedince vybíráme pomocí ruletového kola, které je rozdělené na výseče odpovídající jednotlivým jedincům. Velikost výseče odpovídá pravděpodobnosti p_i výběru daného jedince i. Tuto pravděpodobnost určíme znormováním hodnot účelové funkce.

$$p_i = \frac{\mathcal{F}(I_i)}{\sum_{j=1}^{N} \mathcal{F}(I_j)} \tag{5.2}$$

Selekci je navíc vhodné zkombinovat s tzv. *elitářským mechanismem*. Elita je tvořená určitým počtem (např. 3-5%) nejlepších jedinců z dané populace, kteří jsou reprodukováni do nové generace beze změny. Tak zajistíme, že maximum hodnot účelové funkce neklesá.

Křížení
Použijeme jednobodové křížení. Vybrané jedince budeme označovat jako *otce* a *matku*. Náhodně zvolíme body křížení pro otce i matku tak, aby ležely na hranici bloků. V případě pevné délky jedince volíme bod křížení u otce i matky na stejné pozici v řetězci. Nové jedince pak vytvoříme spojením první části kódu otce s druhou částí kódu matky a naopak.

Mutace
U vybraného jedince náhodně zvolíme jeden blok a jeho hodnoty nebo některé z jeho hodnot náhodně změníme.

5.3 Kanonický genetický algoritmus

Nyní se podívejme na možnost zrychlení předešlého algoritmu. Neruda popisuje v [7] tzv. *kanonický genetický algoritmus* pro učení RBF sítí. Klíčová myšlenka urychlení spočívá v omezení prohledáváného prostoru odstraněním redundancí.

Posloupnost všech parametrů sítě

$$\begin{aligned} P &= \{\mathcal{B}_1, \cdots, \mathcal{B}_h\} \\ \mathcal{B}_i &= c_{i1}, \cdots, c_{in}, b_i, a_{11}^i, \cdots, a_{nn}^i \quad i = 1, \cdots, h \end{aligned} \tag{5.3}$$

(viz kód sítě 5.2) nazveme *parametrizací sítě*. Na množině všech parametrizací zavedeme ekvivalenci tak, že dvě parametrizace prohlásíme za *funkčně ekvivalentní* právě tehdy, určují-li sítě realizující stejnou funkci. Prohledávání pak lze omezit na takový podprostor parametrického prostoru, který obsahuje právě jednoho zástupce každé třídy ekvivalence.

Nejjednodušším typem funkční ekvivalence je tzv. *záměnná ekvivalence*, t.j. permutace skrytých jednotek (permutace sčítanců sumy). V [9] je uveden důkaz věty, která říká, že každé dvě funkčně ekvivalentní RBF sítě s Gaussovou funkcí a metrikou indukovanou skalárním součinem jsou záměnně ekvivalentní.

Parametrizaci $P = \{\mathcal{B}_1, \cdots, \mathcal{B}_h\}$ nazveme *kanonickou*, jestliže platí

$$\mathcal{B}_i \prec \mathcal{B}_{i+1},$$

kde $i = 1, \cdots, h-1$ a $\prec$ je lexikografické uspořádání. Protože třída ekvivalence obsahuje právě takové parametrizace, které se vzájemně liší pouze permutací svých bloků, leží v každé třídě ekvivalence právě jedna kanonická reprezentace.

Nyní potřebujeme upravit náš genetický algoritmus tak, aby hledal řešení pouze mezi kanonickými reprezentacemi.

Kód jedince zůstává stejný jako v části 5.2, ale navíc platí, že $\mathcal{B}_i \prec \mathcal{B}_i + 1$ pro $i = 1, \cdots, h{-}1$. Proto musíme při vytváření počáteční populace generovat pouze jedince reprezentující kanonické parametrizace.

Dále musíme upravit genetické operátory, které vytvářejí nové jedince.

Křížením z 5.2 otce $I_1 = \{\mathcal{B}_1, \cdots, \mathcal{B}_{h_1}\}$ a matky $I_2 = \{\mathcal{D}_1, \cdots, \mathcal{D}_{h_2}\}$ vzniknou dva noví jedinci

$$
\begin{aligned}
J_1 &= \{\mathcal{B}_1, \cdots, \mathcal{B}_s, \mathcal{D}_{r+1}, \cdots, \mathcal{D}_{h_2-r+s}\} \\
J_2 &= \{\mathcal{D}_1, \cdots, \mathcal{D}_r, \mathcal{B}_{s+1}, \cdots, \mathcal{B}_{h_1-s+r}.\}
\end{aligned}
$$

Protože platí $\mathcal{B}_s \prec \mathcal{B}_{s+1}$ a $\mathcal{D}_r \prec \mathcal{D}_{r+1}$ platí vždy alespoň jedna z nerovností

$$
\begin{aligned}
\mathcal{B}_s &\prec \mathcal{D}_{r+1} \\
\mathcal{D}_r &\prec \mathcal{B}_{s+1}.
\end{aligned}
$$

Výsledkem kanonického křížení bude pouze jeden jedinec, a to jedinec J_1 pokud $\mathcal{B}_s \prec \mathcal{D}_{r+1}$, jinak jedinec J_2.

Mutace uvedená v části 5.2 vybere náhodně jeden blok $\mathcal{B}_i$ a jeho hodnotu náhodně změní. Kanonická mutace pracuje stejně, jen náhodná změna probíhá v mezích daných sousedními bloky $\mathcal{B}_{i-1}$ a $\mathcal{B}_{i+1}$. T.j. hodnotu bloku $\mathcal{B}_i$ nahradíme hodnotou $\mathcal{B}_i'$, pro kterou platí $\mathcal{B}_{i-1} \prec \mathcal{B}_i' \prec \mathcal{B}_{i+1}$.

Takto upravený genetický algoritmus pracuje nad $1/k!$ původního parametrického prostoru a dá se tedy očekávat, že řešení nalezne dříve.

5.4 Modifikace selekce a účelové funkce pro učení neuronových sítí

V části 5.2 popisujeme genetické operátory pro učení neuronových sítí. Připomeňme si výpočet hodnoty účelové funkce pro jedince reprezentujícího neuronovou síť (5.1).

$$
\mathcal{F}(I_i) = C - \sum_{t=1}^{k} \parallel \vec{d}^{(t)} - \vec{f}^{(t)} \parallel^2,
$$

kde C je odhad maximální chyby sítě.

V praxi však volba této konstanty není jednoduchá. Jeli tato konstanta podstatně větší než hodnoty chyb příslušející k jedincům v populaci, ohodnocení jedinců v této populaci se příliš neliší a selekce pak neplní svůj účel.

Pokud zvolíme hodnotu C příliš malou, v populaci se může vyskytnout větší množství jedinců ze zápornou hodnotou účelové funkce. Protože záporná hodnota účelové funkce se neuvažuje, v praxi to znamená nulovou hodnotu. V populaci má pak skupina jedinců stejné (nulové) ohodnocení, ačkoliv jim odpovídající hodnoty chybových funkcí se liší.

Dá se očekávat, že jedinci v počátečních a několika dalších generacích budou odpovídat spíše špatně naučeným sítím a jejich chyba bude většinou vysoká. Naopak po delším běhu algoritmu budou převažovat lepší řešení, s malou chybou. Proto se v praxi často volí konstanta C jako maximální chyba v dané populaci. Tím odstraníme výše uvedené problémy. Nevýhodou je však to, že účelová funkce pak ohodnocuje jedince pouze vzhledem k dané populaci. Nelze podle ní tedy usuzovat nic o vývoji výpočtu a rozhodovat o jeho ukončení.

Tento problém odstraníme jednoduchou úpravou genetického algoritmu. Jedinec bude ohodnocen pouze hodnotou chybové funkce odpovídající sítě.

$$\mathcal{F}'(I_i) = \sum_{t=1}^{k} \parallel \vec{d}^{(t)} - \vec{f}^{(t)} \parallel^2 \tag{5.4}$$

Tedy čím menší ohodnocené, tím lepší řešení. Algoritmus tedy bude hledat místo maxima účelové funkce její minimum. Skončí po nalezení jedince z dostatečně malým ohodnocením a tento jedinec pak představuje řešení. Zbývá upravit operátor selekce tak, aby větší pravděpodobnost výběru měli jedinci s nižším ohodnocením. Tady opět využijeme hodnotu maximální chyby, ale v tomto případě stačí maximální chyba v dané populaci a tu známe. Pravděpodobnost výběru jedince i spočteme jako

$$p_i = \frac{(1+\varepsilon)C - \mathcal{F}(I_i)}{N(1+\varepsilon)C - \sum_{j=1}^{N} \mathcal{F}(I_j)}, \tag{5.5}$$

kde N je počet jedinců v populaci, C je maximální hodnota účelové funkce v dané populaci a ε je kladné číslo blízké nule. Konstanta ε zajišťuje, že i pravděpodobnost výběru nejhoršího jedince bude nenulová.

5.5 Modifikace genetického učení

Učení RBF sítě popsané v části 5.2 předpokládá učení všech parametrů RBF sítě. Pokud síť některé z parametrů vůbec nemá nebo jejich hodnoty určovat nechceme, jednoduše je z kódu jedince vypustíme.

Genetický algoritmus pak můžeme kombinovat s klasickými metodami. Například lze genetickým algoritmem učit pouze parametry skryté vrstvy (počet jednotek, středy, šířky, matice norem). Při hodnocení jedince pak nejprve dopočteme neurčené váhy nějakou klasickou metodou (například určitým menším počtem iterací gradientního algoritmu) a ohodnocení určíme na základě chybové funkce doučené sítě.

5.6 Řešení problému vektorové kvantizace

V kapitole 4 jsme popsali tří fázovou metodu učení RBF sítí. Toto učení sestává ze tří podúloh. Jednou z nich je řešení problému vektorové kvantizace.

Protože minimalizace chybové funkce (4.3) není triviální, metody řešení vektorové kvantizace bývají založené na různých heuristikách a většinou nabízejí pouze přibližné řešení.

Nabízí se tedy možnost, řešit problém vektorové kvantizace pomocí genetického algoritmu.

Způsob kódování
Řešení problému vektorové kvantizace je tvořeno množinou h vektorů. Jedinec bude tvořen zřetězením souřadnich těchto vektorů.

$$J = \{w_{11}, w_{12}, \cdots, w_{1n}, w_{21}, \cdots, w_{2n}, \cdots, w_{h1}, w_{h2}, \cdots, w_{hn}\} \tag{5.6}$$

Protože na pořadí vektorů nezáleží, můžeme použít kanonický algoritmus. V tomto případě musí platit

$$\vec{w}_1 \prec \vec{w}_2 \prec \cdots \prec \vec{w}_h, \tag{5.7}$$

kde $\prec$ je uspořádání v lexikografickém pořadí.

Počáteční populace
Počáteční populaci vytvoříme náhodně, v případě kanonického algoritmu vektory před zakódováním setřídíme.

Účelová funkce $\mathcal{F}$
Stejně jako u učení neuronových sití 5.4 budeme hodnotit jedince hodnotou chybové funkce

$$\mathcal{F}(J_i) = E_{VQ}(\vec{w}_1, \cdots, \vec{w}_h), \tag{5.8}$$

kde E_{VQ} je chybovová funkce z vektorové kvantizace (4.3). Čím menší ohodnocení, tím lepší řešení jedinec reprezentuje.

Selekce
Použijeme selekci popsanou v části 5.4.

Křížení
Použijeme jednobodové křížení. Křížením jedinců

$$\begin{aligned} J_1 &= \{w_{11}, \cdots, w_{kl}, \cdots, w_{hn}\} \\ J_2 &= \{w'_{11}, \cdots, w'_{kl}, \cdots, w'_{hn}\} \end{aligned}$$

získáme nové jedince

$$J'_1 = \{w_{11}, \cdots, w'_{kl}, \cdots, w'_{hn}\}$$
$$J'_2 = \{w'_{11}, \cdots, w_{kl}, \cdots, w_{hn}\},$$

kde souřadnice k a l jsou zvoleny náhodně.

V případě kanonického algoritmu budeme bod křížení k volit pouze na hranici bloků odpovídajících jednotlivým vektorům

$$\begin{aligned} J_1 &= \{\vec{w}_1, \cdots, \vec{w}_k \cdots, \vec{w}_h\} \\ J_2 &= \{\vec{w'}_1, \cdots, \vec{w'}_k \cdots, \vec{w'}_h\} \end{aligned}$$

a výsledkem budou pouze jeden jedinec

$$J'_1 = \{w_1, \cdots, w'_k, \cdots, w'_h\} \qquad \vec{w}_{k-1} \prec \vec{w'}_k$$

nebo

$$J'_2 = \{w'_1, \cdots, w_k, \cdots, w_h\} \qquad \vec{w'}_{k-1} \prec \vec{w}_k$$

Mutace
Náhodně vybereme jednu souřadnici a tu náhodně změníme. V případě kanonického algoritmu náhodně vybere blok, který změníme náhodně s ohledem na zachování uspořádání.

Výše uvedený algoritmus představuje alternativu metodám pro řešení vektorové kvantizace uvedených v části 4.1. V případě, že přesně neznáme počet reprezentantů, můžeme tyto genetické operátory upravit tak, aby pracovaly s jedinci různé délky reprezentujícími řešení lišící se počtem reprezentantů. Do ohodnocení takovýchto jedinců je pak třeba zahrnout penalizaci příliš dlouhých jedinců.

Kapitola 6

Implementace

V této kapitole se budeme věnovat implementaci jednotlivých metod učení RBF sítí.

Implementace je realizována v systému Bang2. Proto se nejprve seznámíme s tímto systémem a jeho základními principy. V části 6.1.4 popíšeme implementaci genetických algoritmů, která je součástí tohoto systému.

Část 6.2 pak popisuje samotnou implementaci učení RBF sítí pomocí softwarových agentů.

6.1 Systém BANG2

6.1.1 Základní rysy systému

Bang2 [5] je distribuovaný multiagentní systém určený k tvorbě hybridních modelů umělé inteligence.

Nejprve je nutné říci, co je to *autonomní softwarový agent*. Ačkoliv se jedná o poměrně vžitý pojem, jednotná definice neexistuje. Stan Franklin se tímto problémem zabývá v [2]. Pro naše účely stačí říci, že autonomní softwarový agent je program, který je umístěný v nějakém prostředí a je součástí tohoto prostředí. Pracuje na svém úkolu, dokáže přijímat impulsy ze svého prostředí a sám na ně reagovat. Dokáže si samostatně zjistit potřebné informace, např. dotazem zaslaným ostatním agentům, uživateli apod. Agent je obvykle mobilní, t.j. má možnost přesouvat se v počítačové síti.

Bang2 představuje prostor pro existenci autonomních softwarových agentů, zajišťuje jejich vzájemnou komunikaci a slouží jako rozhraní vůči okolnímu prostředí. Umožňuje distribuci agentů v počítačové síti a jejich paralelní spouštění.

Samotný agent v systému Bang2 realizuje konkrétní metodu umělé inteligence (např. neuronovou síť), numerický výpočet nebo nabízí nějakou speciální službu (přístup k souboru, výstup do programu Gnuplot, apod.).

Komunikace mezi agenty umožňuje jejich spolupráci na řešení problému. Touto spoluprací lze pak realizovat různé hybridní metody, kombinovat klasické principy umělé inteligence a vytvářet složitější modely. Příkladem může být neuronová síť, jejíž váhy jsou nastavovány pomocí gradientního algoritmu a její topologii určuje genetický algoritmus, jehož parametry určuje jiný genetický algoritmus.

6.1.2 Komunikace mezi agenty

Jak už bylo řečeno, důležitou vlastností systému je možnost komunikace mezi agenty.

Agenti spolu komunikují vlastním jazykem. Zprávy zasílané mezi agenty jsou srozumitelné i pro člověka, jejich syntaxe vychází z jazyka XML. Typy zpráv jsou podmnožinou standarndního komunikačního jazyka agentů ACL.

Samotná komunikace pak může probíhat jak synchronně tak asynchronně. Podrobné informace lze najít v [12, oddíl HowTos, stránka call].

Pro zvýšení efektivity je implementováno také binární rozhraní. Toto rozhraní má své klady i zápory. Kladem je především zvýšení rychlosti v případech, kde si agenti potřebují předat větší datovou strukturu. Použitím binárního rozhraní odpadá převod této struktury na XML řetězec a zpět. Pokud se oba agenti nacházejí na stejném stroji, předává se pouze ukazatel. Nevýhodou je, že agent, který taková data příjímá, musí být obeznámem s jejich strukturou. Dokumentace binárního rozhraní je v [12, oddíl HowTos, stránka fastcall].

Nyní se podívejme na syntaxi zpráv předávaných mezi agenty. Jedná se o XML řetězce. Kořenový element určuje typ zprávy. K dispozici jsou zprávy typů *request* (požadavek na provedení příkazu), *inform* (zpráva), *query* (dotaz), *ok* (odpověď v případě, že nedošlo k chybě) a *ugh* (odpověď po výskytu chyby). Zbytek zprávy obsahuje příkazy, informace, apod.

Uveďme nyní typy zpráv, se kterými se budeme setkávat při popisu jednotlivých agentů v části 6.2.

- **request** — XML řetězec ve tvaru `<request>` … `</request>` představuje požadavek na provedení nějakého příkazu.

  Např. `<request><learn/></request>`,
  `<request><openfile filename='data1.dat'/></request>`.

  Pokud agent rozumí zprávě a reaguje na zprávu tvaru
  `<request><příkaz/></request>`, budeme o něm stručně říkat, že *rozumí příkazu* `<příkaz/>`.

- **request set var** — zpráva ve tvaru `<request><set><var>` *jméno proměnné* `</var><value type='typ proměnné'>` *hodnota* `</value></set></request>` slouží k nastavení proměnné agenta na nějakou hodnotu.

  Tedy například `<request><set><var><trainsetname/></var><value type='string'> <string>` sin2.dat `</string></value></set></request>`. Pokud agent rozumí takovéto zprávě, budeme o něm říkat, že umožňuje přístup k proměnné `trainsetname` pro zápis.

- **query get var** — dotaz na hodnotu proměnné se formuluje jako `<query><get><var>` *jméno proměnné* `</var></get></request>`.

  Např. `<query><get><var><error/></var></get></request>` zaslaný agentovi spravujícímu neuronovou síť může znamenat dotaz na hodnotu chybové funkce této sítě.

  Agent, který rozumí zprávě tohoto typu, umožňuje přístup k dané proměnné pro čtení.

Ještě uveďme formát zprávy spojené s binárním rozhraním. Příkladem budiž agent představující tréninkovou množinu. Taková tréninková množina je schopna předat tréninkový vektor pomocí binárního rozhraní. Agent, který chce k této tréninkové množině přistupovat, musí nejprve požádat o tuto službu pomocí zprávy `<query><bin><request><getvector/></request></bin></query>`. Poté obdrží potřebné údaje a další komunikace probíhá přes binární rozhraní.

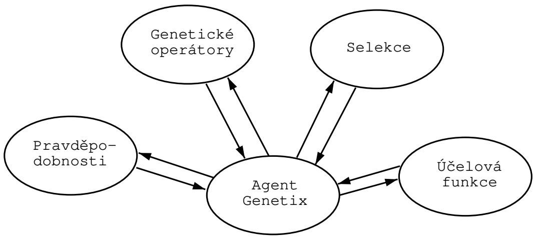
6.1.3 Implementace agenta

Agent je implementován jako třída v jazyce C++ odvozená od abstraktní třídy Agent, která definuje základní chování a metody nutné ke komunikaci s ostatními agenty a prostředím. Autor nového agenta buďto předefinuje funkci ProcessMsg, která zajišťuje zpracování příchozích zpráv, nebo napíše sadu *triggerů* (funkcí asociovaných s konkrétní zprávou), jejichž správné spouštění po příchodu zprávy zajištuje funkce RunTriggers. Další detaily a podrobnější vysvětlení lze najít v [5] nebo v [12, oddíl HowTos, stránka agent] a [12, oddíl HowTos, stránka triggers].

6.1.4 Genetické algoritmy

Bang2 obsahuje agenta *Genetix* (jménem *George*), který zajišťuje obecný genetický algoritmus, nezávislý na právě řešeném problému.

Na konkrétním problému závisí především struktura a kódování jedinců. Proto *Genetix* realizuje pouze jádro genetického algoritmu nezávislé na struktuře jedinců. Všechny specializované operace, jako jsou genetické operátory nebo ohodnocení účelovou funkcí, zadává jiným agentům. Tito agenti už znají strukturu jedinců pro konkrétní úlohu.



Obrázek 6.1: Implementace genetického algoritmu

Obrázek 6.1 znázorňuje agenty spolupracující na genetickém algoritmu. V centru je agent *Genetix*, který výpočet řídí a synchronizuje. Při vytváření nové populace požádá agenta specializovaného na selekci o výběr jedinců. Tyto jedince pak předá agentovi realizujícímu genetické operátory a dostane zpět nové jedince. Ty pak předá dalšímu agentovi, který je dokáže ohodnotit účelovou funkcí.

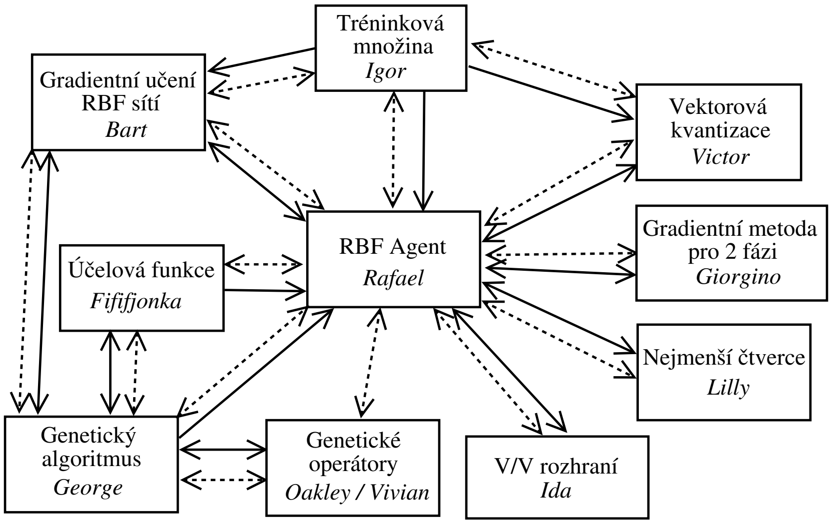
Také je možné připojit agenta, který bude určovat pravděpodobnosti genetických operátorů.

Chceme-li tedy aplikovat genetický algoritmus na náš vlastní problém, stačí naprogramovat agenty realizující výpočet účelové funkce, generování nových jedinců a genetické operátory nad našimi jedinci. Poté můžeme použít připraveného agenta *Genetix*.

6.2 Implementace učení RBF sítí

V části 2.4 jsme popsali řadu metod pro učení RBF sítí. Nyní chceme realizovat systém, přesněji komponentu systému Bang2, který bude realizovat uvedené metody. Budeme používat sítě s Gaussovskými jednotkami, všechny metody však implementujeme tak, aby byly co nejméně závislé na typu použité radiální funkce a umožňovaly v budoucnu rozšíření pro práci s obecnou radiální basickou funkcí.

Nejprve si položme otázku, zda by měl být tento systém opravdu pouze jednou komponentou, jedním agentem. Existuje několik důvodů proti. Za prvé učení RBF sítě obsahuje několik podproblémů, které jsou obecného charakteru, například vektorovou kvantizaci nebo řešení soustavy lineárních rovnic metodou nejmenších čtverců. Bude tedy užitečné mít vždy samostané agenty na řešení těchto podproblémů. Dále chceme mít otevřenou cestu pro nové kombinace metod, nové metody apod. Proto se zdá výhodné realizovat jednotlivé metody odděleně a spojovat je podle potřeby.



Obrázek 6.2: Agenti zapojení do učení RBF sítě. Čárkovaná šipka znározňuje tok XML zpráv, plná šipka přenos binárních dat.

Skutečná situace je zobrazena na obrázku 6.2. Hlavní roli tu hraje *RBF agent*, který představuje správce RBF sítě, umí komunikovat s uživatelem a udržuje potřebné informace o konfiguraci sítě a učení. Sám však neovládá žádný ze způsobů učení. Ale dokáže si najít agenta, který je na danou metodu specializován a provede ji za něj. Většina ostatních agentů na obrázku tedy představuje jednotlivé učící metody.

Ještě uveďme několik speciálních agentů. Agent *Tréninková množina* umožňuje přístup k souboru s tréninkovou množinou a během učení se k němu podle potřeby připojují ostatní agenti. Na obrázku není znázorněn agent *TclShell* realizující grafické uživatelské prostředí pomocí knihovny Tcl/Tk. V případě, že je tento agent přítomen, připojují se k němu všichni agenti a umožňují tak uživateli konfiguraci prostřednictvím grafického prostředí. Dále je možno připojit agenta jménem *Plotter*, který nabízí grafické výstupy pomocí Gnuplotu. Agenti *Tréninková množina*, *TclShell* a *Plotter* jsou součástí systému Bang2.

V následujích kapitolách popíšeme jednotlivé agenty v tomto pořadí: *RBF agenta* a *V/V rozhraní* (6.2.1), abstraktního agenta, definujícího rozhraní všem iteračním agentům (6.2.2), agenty realizující třífázové učení (6.2.3), agenta pro gradientní učení (6.2.4) a agenty spolupracující na realizaci genetických algoritmů (6.2.5).

Každý agent v systému Bang2 je identifikován jménem, proto budeme v dalším textu jednotlivé agenty označovat jmény. Tato jména jsou uvedena na obrázku 6.2, který slouží k orientaci mezi agenty. V případě nejasností je dobré se k němu vrátit.

U každého agenta také uvedeme jméno webové stránky, na které lze najít informace o daném agentovi a seznam zpráv, které přijímá. Stránky jsou součástí on-line dokumantace [12, oddíl Man].

6.2.1 RBF Agent a V/V rozhraní

Rafael (rbf.html)
RBF agent jménem *Rafael* (viz obr. 6.2) slouží především jako správce RBF sítě a jejího učení. Ovládá některé primitivní způsoby nastavení parametrů sítě, ale opravdové učení parametrů přenechává ostatním agentům.

Kromě RBF sítě spravuje tento agent konfiguraci učení. Konfigurací tu rozumíme seznam všech možných informací, jako je jméno tréninkové množiny, typ učící metody, parametry této metody apod. Tuto konfiguraci může uživatel modifikovat pomocí grafického rozhraní nebo zadat v XML formátu v souboru. Pro parametry konfigurace, které nebyly nikým zadány, použije *Rafael* hodnoty, které sám považuje za optimální.

Uveďme seznam příkazů, na které *Rafael* reaguje:

1. `<loadconfig/>` — načtení konfiguraci sítě a učení ze souboru.

2. `<learn/>` — aplikace určené metody učení na RBF síť. Rafael si najde agenty, které k její realizaci použije a případně je nakonfiguruje. Poté proběhne samotné učení.

3. `<load><net/></load>` — načtení sítě ze souboru

4. `<save><net/></save>` — převedení sítě do XML formátu a uložení do souboru

5. `<nettest/>` — otevření uživatelského okénka, které umožňuje zadávat hodnoty vstupů sítě a zobrazovat odezvu sítě. Okénko je realizováno agentem *V/V rozhraní*.

6. `<gnuplot/>` — zobrazení funkce RBF sítě pomocí programu Gnuplot. Provádí se pouze pro sítě s jedním vstupem a výstupem.

7. `<sampleToFile/>` — navzorkování funkce sítě do souboru.

8. `<evalFile/>` — pro vstupní části vzorů z testovací množiny se spočtou výstupy sítě a vše se uloží do zadaného souboru. (Testovací množina se připojuje na místo tréninkové množiny.)

Rafael zpřístupňuje pro čtení proměnnou `error`, která představuje chybu naučené sítě vůči tréninkové množině, jež je právě připojena.

Pomocí binárního rozhraní lze zpracovat dotaz `<query><var><response/> </var></query>`, který pro dané hodnoty vstupů sítě vrátí hodnoty výstupů.

Dále lze pomocí přístupu k proměnným nastavovat všechny položky konfigurace jak sítě tak učení.

Ida (netinterface.html)
Ida (viz obr. 6.2) umožňuje uživatelům pohodlné testování sítě pomocí grafického uživatelského rozhraní. Umí se připojit k jakémukoli agentovi, který má určitý počet vstupů a určitý počet výstupů, a dokáže k zadaným hodnotám vstupů určit hodnoty výstupů.To je každý takový agent, který zpřístupňuje proměnné `inputs`,`outputs` a `response` pro čtení. Ida si pak dotazem na hodnotu těchto proměnných zjistí počet vstupů a výstupů. Poté vygeneruje grafické okno, kde je možné zadat hodnoty vstupů a nechat si zobrazit odpovídající výstupy.

6.2.2 Iterační algoritmus

Iterační algoritmus
Protože většina učících metod je iteračního charakteru, existuje abstraktní agent *Iterační algoritmus.* Tento agent definuje jednotné rozhraní pro všechny iterační agenty, tedy agenty realizující nějaký iterační výpočet.

Běh výpočtu lze ovládat těmito příkazy:

1. `<start/>` — spuštění výpočtu.
2. `<stop/>` — dočasné pozastavení výpočtu.
3. `<continue/>` — pokračování pozastaveného výpočtu.
4. `<end/>` — ukončení výpočtu.

Příkaz je vždy přijat po ukončení právě prováděné iterace, tedy výpočet nemůže být pozastaven nebo ukončen v průběhu iterace. Výpočet může také skončit sám po splnění kritéria ukončení (`stopcriterion`). Jako kritérium lze uvést počet iterací, které mají proběhnout(`iters`), nebo mez, pod kterou má klesnout hodnota chyby (`eps`).

Informace o stavu výpočtu jsou přístupné pomocí proměnných:

1. `error` — aktuální hodnota chybové funkce, pouze pro čtení.
2. `iter` — číslo právě provedené iterace, pouze pro čtení.

Agent může být připojen ke grafickému rozhraní. Během výpočtu pak zobrazuje hodnotu chyby, číslo iterace a vykresluje graf chybové funkce. Proměnná `sample` určuje, po kolika iteracích se sleduje hodnota chyby a obnovují se zobrazované informace. Proměnnou `scale` se násobí hodnota chybové funkce před vykreslením do grafu, lze jí tedy ovlivnit měřítko grafu. (V okně grafu jsou viditelné pouze hodnoty v intervalu $\langle 0, 1 \rangle$). Obě proměnné jsou přístupné pro čtení i zápis.

Příkazem `<confwin/>` otevřeme grafické okno s parametry daného výpočtu, pokud nějaké existují.

Dále je možné průběžně ukládat informace o chybové funkci do souboru. Nastavením promměnné `wlog` určíme periodu, se kterou se bude do souboru zapisovat aktuální iterace a chyba. Proměnná `wlogfilename` určuje jméno souboru.

Každý agent, který je potomkem *Iteračního algoritmu,* rozumí uvedeným příkazům a umožňuje přístup k uvedeným proměnným. Proto vždy pouze uvedeme, že se jedná o *Iterační algoritmus* a nebudeme znovu popisovat základní chování takového agenta.

6.2.3 Třífázové učení

Třífázové učení popsané v kapitole 4 se skládá z kroků, v nichž postupně řešíme dílčí úlohy — vektorovou kvantizaci, nastavení šířek a určení vah.

Prvním krokem je nalezení středů RBF jednotek. *RBF agent* umí jen přímočaré metody, tedy rovnoměrné rozmístění středů ve vstupním prostoru a vybrání náhodných vzorků z tréninkové množiny.

Pokud je po něm požadováno použití nějaké metody vektorové kvantizace, požádá o to agenta pro vektorovou kvantizaci jménem *Victor*. Předá mu potřebné informace, tedy odkaz na tréninkovou množinu, požadovaný počet středů a případné další parametry. Po skončení výpočtu od něj převezme výsledek a nastaví středy své RBF sítě na odpovídající hodnoty.

Jinou možností je nahrazení vektorové kvantizace genetickým algoritmem. O tom více v kapitole 6.2.5.

Pokud mají RBF jednotky nějaké další adaptovatelné parametry, nastavují se v druhé fázi. *RBF agent* má přehled o tom, zda takové parametry existují a jako metodou je chceme učit. Opět je výběr mezi jednoduchou heuristikou (nastavení šířek úměrně vzdálenosti několika nejbližších sousedů), kterou *RBF agent* realizuje sám, nebo časově náročnější minimalizací chybové funkce (4.12). Tuto minimalizi ovládá agent jménem *Giorgino*. *RBF agent* mu předá skrytou vrstvu s nastavenými středy a *Giorgino* mu dopočte hodnoty ostatních parametrů.

Zbývá poslední fáze, nastavení vah výstupních jednotek. *RBF agent* zformuluje pomocí tréninkové množiny soustavu rovnic a předá známé matice agentovi jménem *Lilly*. Může si zvolit, jakou metodu nejmenších čtverců má *Lilly* použít. *Lilly* pak provede samotný výpočet a vrátí *RBF agentovi* hledanou matici, tedy matici vah.

Druhou možností pro hledání vah je použití gradientního algoritmu. To popíšeme v podkapitole 6.2.4.

	metoda	agent
1. fáze	rovnoměrné rozmístění středů	Rafael
	náhodné tréninkové vzory	Rafael
	vektorová kvantizace	Victor
	genetický algoritmus	George, Vivian
2. fáze	heursitika q sousedů	Rafael
	gradientní mimimalizace	Giorgino
3. fáze	nejmenší čtverce	Lilly
	gradientní algoritmus	Bart

Tabulka 6.1: Přehled agentů realizujících třífázového učení.

V tabulce 6.1 je přehled metod a agentů používaných v třífázovém učení. Nyní se podívejme podrobněji na jednotlivé agenty.

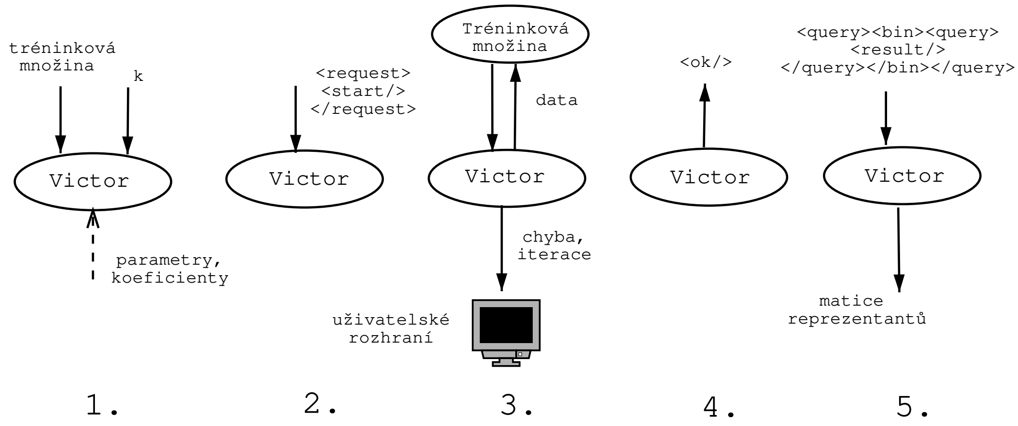
Victor (vq.html)

Victor je agent, který dokáže řešit úlohu vektorové kvantizace. Tedy k množině bodů v n-rozměrném prostoru najde k reprezentantů.

Tuto úlohu umí řešit pomocí Lloydova algoritmu, Kohonenova učení a Kohonenova učení s radiálním růstem, respektive lokální pamětí. Všechny tyto algoritmy mají stejný charakter, proto sdílejí jednoho agenta, a jsou iterativní, proto je *Victor* potomkem *Iteračního algoritmu*.

Obrázek 6.3 zobrazuje typický průběh komunikace *Victora* s uživatelem. Uživatelem rozumíme takový objekt (osobu, agenta), který po *Victorovi* požaduje nějakou službu. V našem případě to bude *RBF agent*. Ten *Victorovi* nejprve předá odkaz na tréninkovou množinu a požadovaný počet reprezentantů. Může také specifikovat, kterou metodu použít, a určit její parametry. Poté spustí výpočet a po jeho skončení získá výsledek.

Typ metody a případné další parametry lze předat příkazem `<config>`



Obrázek 6.3: Průběh komunikace s Victorem. 1. Inicializace výpočtu. 2. Start. 3. Výpočet běží. 4. Výpočet skončil. 5. Předání výsledků.

konfigurace `</config>`, kde *konfigurace* obsahuje XML řetězec určující dané parametry. Pro parametry, které nejsou v konfiguraci obsaženy, se použijí přednastavené hodnoty.

Úloha se zadá pomocí přístupu k proměnným `trainset` a `k`. Zápis do proměnné `trainset` umožňuje specifikovat tréninkovou množinu, a to buď jménem souboru s tréninkovou množinou nebo jménem agenta spravujícího nějakou tréninkovou množinu. Proměnná `k` určuje požadovaný počet reprezentantů.

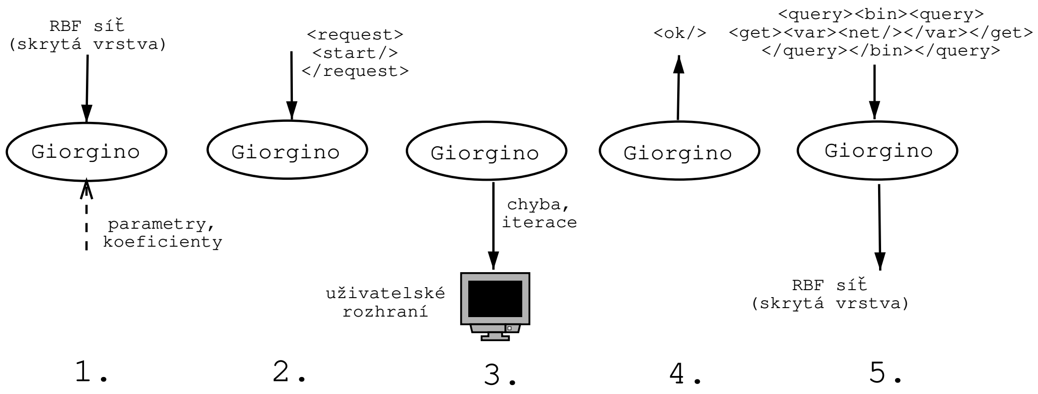
Binární rozhraní umožňuje předání výsledku pomocí `<query><result/> </query>`.

Giorgino (gradient.html)

Agent *Giorgino* je specializován na nalezení vhodných šířek a matic norem RBF jednotek. Hodnoty hledá pomocí minimalizace chybové funkce (4.12) jednoduchým gradientním algoritmem. Gradientní algoritmus má iterativní charakter, tedy *Giorgino* je potomkem *Iteračního algoritmu.*

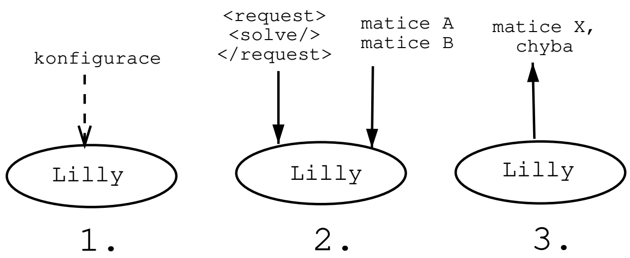
Na obrázku 6.4 je znázorněn typický průběh komunikace *Giorgina* se svým uživatelem, nyní *RBF agentem. RBF agent* nejprve předá skrytou vrstvu RBF sítě. Dále může pomocí příkazu `<config/>` předat hodnoty parametrů a koeficientů gradientní metody a specifikovat, které parametry chce naučit. Poté spustí výpočet a po jeho skončení převezme skrytou vrstvu RBF sítě, doplněnou o hodnoty hledaných parametrů.

Předání RBF sítě před a po výpočtu probíhá přes binární rozhraní zpracování příkazu `<set><var><net/></var></set>` a dotazu `<get><var><net/> </var></get>`.



Obrázek 6.4: Průběh komunikace: 1. Předání zadání 2. Start. 3. Výpočet běží. 4. Výpočet skončil. 5. Předání výsledků.

Lilly (leastsquares.html)
Agent jménem *Lilly* hledá řešení soustavy $\mathbf{AX} = \mathbf{B}$ ve smyslu nejmenších čterců. Řešení umí najít pomocí SVD rozkladu, QR rozkladu nebo Moore-Penroseovy pseudoinverze (viz kapitola 4.3).



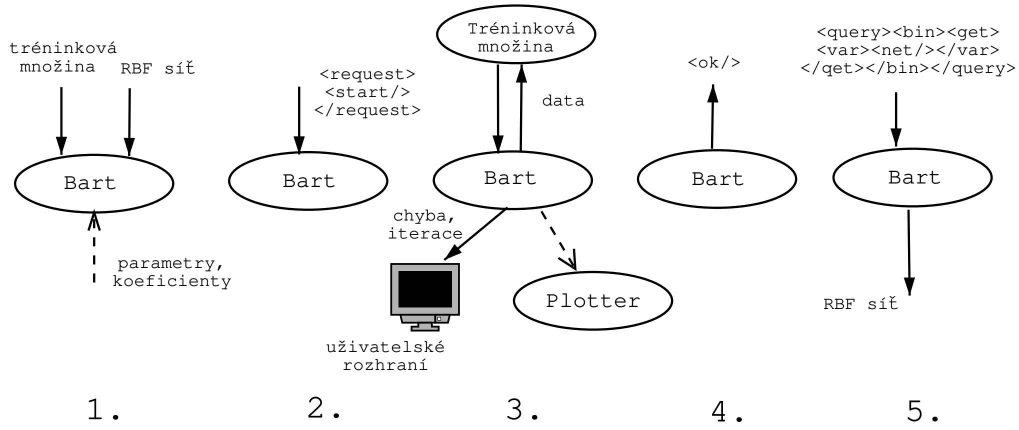
Obrázek 6.5: Průběh komunikace: 1. Konfigurace 2. Žádost o řešení problému 3. Předání výsledku

Uživatel, který požaduje řešení problému, nejprve příkazem `<config/>` určí metodu, kterou má Lilly použít. Neučiní-li tak, použije se SVD rozklad, který *Lilly* považuje za nejsilnější metodu. Poté požádá o binární rozhraní a provedení příkazu `<solve/>`, přes binární rozhraní předá objekt obsahujcí zadání, t.j. matice $\mathbf{A}$ a $\mathbf{B}$. *Lilly* spočte řešení a spolu s hodnotou chyby a případnou chybovou hláškou je předá uživateli, opět přes binární rozhraní.

6.2.4 Gradientní učení

Dalším možným přístupem k učení RBF sítí je gradientní metoda popsaná v kapitole 3. *RBF agent* použije v tomto případě služeb agenta jménem *Bart*.

Bart (back.html)
Bart je agent realizující gradientní algoritmus pro učení RBF sítí. Je schopen učit všechny parametry, které síť obsahuje, nebo pouze vybrané z nich. Je tedy možné použít gradientní algoritmus jak pro učení celé RBF sítě, tak pro poslední krok třífázového učení, kde budeme učit pouze váhy.



Obrázek 6.6: Průběh komunikace: 1. Konfigurace 2. Start výpočtu 3. Bart počítá 4. Konec 5. Předání výsledku

Podívejme se na obrázek 6.6. *RBF agent* nejprve předá *Bartovi* zadání, tedy RBF síť (proměnná `rbfnet`) a odkaz na tréninkovou množinu (proměnná `trainset`). Poté pomocí příkazu `<config/>` specifikuje seznam parametrů, které se mají učit, koeficienty gradientního algoritmu, způsob inicializace a další parametry. Dále lze určit, jak často a kam se má ukládat mezivýsledek, pro případ výpadku hardwaru.

Lze učit všechny skupiny parametrů RBF sítě, pak není nutné předávat *Bartovi* celou RBF síť (rozuměj objekt obsahující všechny parametry sítě), ale stačí pouze předat konfiguraci sítě, kde specifikujeme počet RBF jednotek a určíme, zda mají mít RBF jednotky nastavitelnou šířku a matici normy.

Pokud je nějaká skupina parametrů známá, lze předat *Bartovi* RBF síť, která má tuto skupinu parametrů nastavenou a vynechat tuto skupinu v seznamu parametrů, které se mají učit.

Gradientní metoda začíná s náhodnými hodnotami parametrů sítě. *Bart* k počátečnímu nastavení používá dvě strategie. První je klasická, kdy středy jsou náhodně rozmístěny po vstupním prostoru, šířky a matice norem na hodnoty kolem jedničky a váhy na náhodné malé hodnoty.

Druhá strategie vznikla během experimentů s gradientní metodou a osvědčila se. Středy jsou náhodně vybrané body z tréninkové množiny, šířky jsou rovny jedné, matice norem jsou jednotkové matice a váhy jsou nulové. Síť tak začíná na konstantní nulové funkci, která je dostatečně hladká, a teprve během výpočtu se přizpůsobuje učené funkci. Nezačíná tedy z žádnými nežádoucími oscilacemi.

Inicializaci sítě lze samozřejmě přeskočit. To předpokládá, že *Bart* na počátku opravdu obdrží již inicializovanou RBF síť.

Při učení RBF sítě s jedním vstupem a jedním výstupem lze během výpočtu sledovat funkci sítě zobrazenou programem Gnuplot. *Bart* k tomu využívá agenta *Plotter*. Příkazy `<gnuplot><[on|off]/></gnuplot>` lze zapnout nebo vypnout toto zobrazování. Režie těchto výstupů je však poměrně vysoká a zobrazování je vhodné pouze pro malé funkce a malé tréninkové množiny.

Bart se dá také využít ke zjištění chybové funkce v genetickém algoritmu. Genetické algoritmy budou popsány v části 6.2.5. *Bart* může být připojen k *Georgovi* (viz obr. 6.2) jako účelová funkce. Pak je schopen přijmout dotaz na binární zpracování žádosti `<query><error/></query>`. Přes binární rozhraní pak obdrží RBF síť s nastavenou skrytou vrstvou. Na tuto síť aplikuje určitý počet iterací gradientího algoritmu a výslednou hodnotu chybové funkce vrátí jako hodnotu účelové funkce.

Vraťme se nyní k třífázovému učení z minulé kapitoly. Nyní už je zřejmé, co má *RBF agent* udělat, chce-li k hledání vah v třetí fázi použít gradientní metodu. Předá *Bartovi* síť s naučenou skrytou vrstvou a nakonfiguruje si *Barta* tak, aby doučil pouze její váhy.

6.2.5 Genetické algoritmy

V kapitole 6.1.4 jsme zmínili, že Bang2 obsahuje agenty pro realizaci obecného genetického algoritmu. Nyní chceme tohoto agenta použít pro učení RBF sítí genetickými algoritmy. Musíme tedy k tomuto obecnému algoritmu připojit agenta, který bude realizovat genetické operátory na jedincích představujících RBF sítě, a agenta, který bude schopen takovéto jedince ohodnotit.

Součástí obecného genetické algoritmu v systému Bang2 je agent *George* (Genetix), realizující jádro genetického algoritmu, a agent jménem *Sally* realizující ruletovou selekci. Protože *Rafael* byl prvním uživatelm těchto agentů, bylo třeba je modifikovat a rozšířit.

Popíšeme tedy nejprve tyto dva agenty. Poté se podíváme na agenty *Oakleyho, Vivian a Fififjonku* realizující různé genetické operátory a účelovou funkci pro účely učení RBF sítí. Přehled agentů zapojených do genetického učení je v tabulce 6.2.

Úloha	Genetické operátory	Chybová funkce
Učení všech parametrů RBF sítě	Oakley	Fififjonka
Učení parametrů skryté vrstvy	Oakley	Bart
Hledání středů (vektorová kvantizace)	Vivian	Vivian

Tabulka 6.2: Přehled agentů podílejících se na genetickém učení.

George(genetix.html)
Agent *George* (viz Genetix, kapitola 6.1.4) realizuje vlastní genetický algoritmus. Protože tento algoritmus je iterativního charakteru, byl modifikován tak, aby byl potomkem *Iteračního algoritmu* a měl stejné rozhraní jako ostatní iterační agenti.

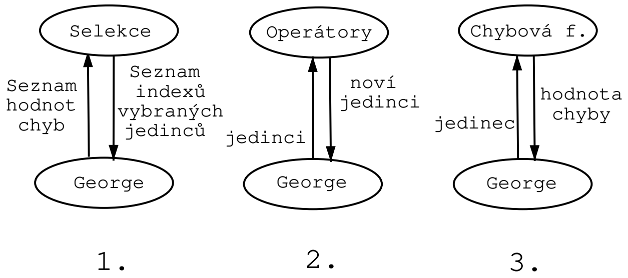
Pro realizaci modifikace genetického algoritmu popsané v části 5.4, kdy účelovou funkci nahradíme chybou funkce a hledáme její minimum, je *George* upraven tak, že umí pracovat ve dvou módech. Buďto pracuje s klasickou účelovou funkcí, nebo s chybou. Mód si nastaví sám. Umí-li agent, který je k němu připojen, spočítat účelovou funkci, *George* pracuje v klasickém módu. V opačném případě se jej zkusí zeptat, zda umí alespoň spočítat chybu. Pokud je odpověď kladná, přepne se do druhého módu.

Dále byla *Georgovi* přidána schopnost zálohovat populaci do souboru. Populace se ukládá jako posloupnost XML řetězců odpovídajících jednotlivým jedincům. Protože však *George* neví nic o struktuře jedinců, se kterými pracuje (neboť tu potřebují znát pouze operátory), neumí už z tohoto souboru populaci obnovit. Nemá totiž ani schopnost vytvářet nové jedince, to za něj vždy dělá agent představující genetické operátory. Ten už je specializován na daný typ jedinců a na něj tedy byla přesunuta schopnost tyto jedince ze souboru obnovit.

Práce s *Georgem* probíhá jako u každého *Iteračního algoritmu*. Před spuštěním výpočtu je možné příkazem `<config/>` nastavit velikost populace, velikost elity a frekvenci zálohování populace. Příkaz `<setservant id="AGENT" job="[fitness|operators|select]"/>` umožňuje připojit jednotlivé specializované agenty. Pokud tak nikdo neučiní, *George* se pokusí najít tyto agenty sám.

Poté je spuštěn výpočet. Během něho je zobrazována chyba a iterace. Pokud se tyto informace ukládají do souboru, je s nimi navíc zapisována průměrná chyba a chyba nejhoršího jedince, délka nejlepšího jedince a průměrná délka jedinců v populaci. Po skončení výpočtu je jako výsledek předán nejlepší jedinec přes binární rozhraní (`<get><var><result/></var></get>`).

Obrázek 6.7 znázorňuje postup *George* při vytváření nové populace.



Obrázek 6.7: 1. George získá od agenta realizujícího selekci vybrané jedince 2. Předá vybrané jedince genetickým operátorům, dostane nové jedince 3. Nové jedince předá agentovi realizujícímu chybovou funkci, která pro ně spočte hodnotu chyby

Sally(gaselect.html)

Agent *Sally* realizuje ruletovou selekci. Žadatel o provedení selekce předá *Sally* přes binární rozhraní vektor hodnot účelové funkce a obdrží požadovaný počet vybraných jedinců.

I *Sally* byla rozšířena o práci s chybou místo účelové funkce. Žádost o provedení selekce obsahuje i specifikaci, o který případ se jedná. V případě, že *Sally* obdrží chybové hodnoty, provede výpočet pravděpodobností tak, aby jedinci s malou chybou měli velkou pravděpodobnost výběru.

```
<query><bin><query><select type="[error|fitness]"></query></bin>
</query>
```

Oakley(rbfgeop.html)

Agent *Oakley* ovládá genetické operátory pro učení RBF sítí. Umí vytvářet jedince reprezentující RBF síť nebo její skrytou vrstvu a aplikovat na ně operátor mutace a křížení.

Než budeme po *Oakleym* požadovat dané operace, musíme mu dát informace o tom, jak požadovaná RBF síť vypadá, tedy počet skrytých jednotek a informace o tom, jaké mají RBF jednotky parametry. Počet jednotek nemusíme specifikovat, tedy bude se také učit a *Oakley* bude vytvářet jedince různých délek. Dále můžeme určit, zda se jedná o kanonickou nebo klasickou verzi genetického algoritmu (viz kaptila 5.3). Lze zadat i omezení pro generování náhodných hodnot jednotlivých parametrů. Vše lze zadat příkazem `<config/>`. Chybějící informace si *Oakley* nahradí svými přednastavenými hodnotami.

Samotné operace pak probíhají přes binární rozhraní. *Oakley* tedy dokáže přes binární rozhraní zpracovat příkaz `<operator num="INDEX"/>`, kde `INDEX` určuje, o který operátor se jedná.

Vivian(vqgeop.html)
Agent *Vivian* realizuje genetické operátory pro operace s množinami vektorů pro účely vektorové kvantizace (viz část 5.6). Chování je podobné jako u *Oakleyho*. Navíc k ní lze připojit *Tréninkovou množinu* a používat ji i jako chybovou funkci.

Fififjonka(fitness.html)
Agent *Fififjonka* umí spočítat hodnotu chybové funkce pro jedince předstvujícího RBF síť.

Práce s tímto agentem je jednoduchá. Je třeba pouze připojit tréninkovou množinu, případně příkazem `<config><error lambda="FLOAT"/></config>` zadat hodnotu regularizačního členu (viz 2.4.1).

Komunikace pak probíhá přes binární rozhraní zpracováním zprávy `<query>` `<error/></query>`.

Má-li *RBF agent* použít k učení své sítě genetické algoritmy, měl by o to požádat *George*.

Chce-li učit celou síť, připojí jako genetické operátory *Oakleyho*, jako chybovou funkci *Fififjonku* a jako selekci *Sally*. Jednotlivé agenty si může pomocí příkazu `<config/>` nakonfigurovat podle svých představ. Například *Oakleymu* může sdělit, zda chce učit i počet skrytých jednotek a *Oakley* má vytvářet jedince různých délek, nebo zda použít kanonické operace. Pak pošle *Georgovi* příkaz ke spuštění výpočtu, případně zadá kritérium ukončení a počká si na výsledek.

Jinou možností je použít genetické učení pouze pro určení parametrů skryté vrstvy. Pak nakonfiguruje *Oakleyho* tak, aby vytvářel pouze jedince představující skrytou vrstvu. Jako účelovou funkci připojí *Barta*. Jedinci představující skrytou vrstvu jsou ohodnocováni *Bartem* tak, že je nejprve na určitý počet iterací spuštěn genetický algoritmus, který jedinci dopočte hodnoty vah, poté se spočte hodnota chybové funkce takto naučené sítě a použije se jako hodnota účelové funkce.

Po skončení genetického algoritmu jsou váhy vítězné sítě doučeny gradientním algoritmem.

Posledním použitím genetického algoritmu je první krok třífázového učení. *Rafael* připojí k *Georgovi* agenta *Vivian*, a to jak na místo operátorů, tak na místo účelové funkce. *Vivian* předá tréninkovou množinu, na jejímž základě je počítána účelová funkce. Výsledkem genetického učení je pak množina vektorů, které *Rafael* použije k nastavení středů.

Kapitola 7

Výsledky experimentů

V následující kapitolách se seznámíme s výsledky experimentů, které byly realizovány pomocí softwarového systému popsaného v předešlé kapitole.

Experimenty byly provedeny na Linuxovém clusteru Joyce v Ústavu informatiky Akademie věd. Výpočty probíhaly na jednotlivých uzlech s procesory Celeron 533 MHz a pamětí 384 MB.

Pro účely experimentů jsme zvolili úlohy různých velikostí tréninkové množiny i sítě. Řešili jsme jak problém aproximace funkce, tak problém klasifikace.

Optimální počty RBF jednotek pro danou úlohu se liší v závislosti na použité metodě. Abychom však měli možnost srovnání, učíme vždy všemi metodami stejně velké sítě.

Termínem *chyba* sítě v následujících kapitolách rozumíme hodnotu chybové funkce

$$E = \sum_{t=1}^{k} \parallel \vec{d}^{(t)} - \vec{f}^{(t)} \parallel^2,$$

kde $\vec{f}^{(t)}$ představuje vektor výstupů sítě po předložení t-tého tréninkového vzoru a $\vec{d}^{(t)}$ vektor požadovaných výstupů.

Každému experimentu odpovídá jedna kapitola, která je navíc rozdělena do podkapitol popisujících použití gradientního učení, třífázového učení a genetických algoritmů. Nakonec vždy tyto rozdílné přístupy porovnáme na základě hodnot chybové funkce a časové náročnosti.

7.1 Aproximace funkce sin(x)cos(y) I.

První úlohou je aproximovat funkci $sin(x)cos(y)$. Funkce je zadána tréninkovou množinou 30×30 tréninkovým vzorů, které jsou výsledkem pravidelného vzorkování funce na intervalu $[0; 2\pi] \times [0; 2\pi]$.

V následujících třech podkapitolách popíšeme postupně aplikaci gradientního učení, třífázového učení a genetických algoritmů na tuto úlohu. Ve všech případech učíme síť o 50 jednotkách.

7.1.1 Gradientní algoritmus

V prvním případě jsme použili gradientní algoritmus. Učená síť měla 50 jednotek, které měly adaptovatelné všechny parametry, včetně šířek a matic vah.

Výpočet jsme nechali běžet 10 000 iterací, hodnotu chybové funkce jsme zjišťovali po každé sté iteraci. Průměrný čas potřebný k provedení 100 iterací byl 101.5 s.

Experiment jsme provedli desetkrát. V prvních pěti případech jsme použili náhodné rozmístění středů a náhodné hodnoty vah blízké nule, náhodné hodnoty šířek blízké jedné a náhodné hodnoty matic vah blízké jednotkovým maticím. Dalších pět případů použilo jako středy náhodné vzorky z tréninkové množiny, nulové váhy, jednotkové šířky a matice vah. Druhá možnost nastavení vznikla během ladění a osvědčila se u některých testovacích úloh. V tomto případě dopadly obě skupiny zhruba stejně.

Obrázek 7.1 zobrazuje pokles chybové funkce během prvních 5 000 iterací. Zobrazen je průměr, minimum a maximum chybové funkce v provedených výpočtech.

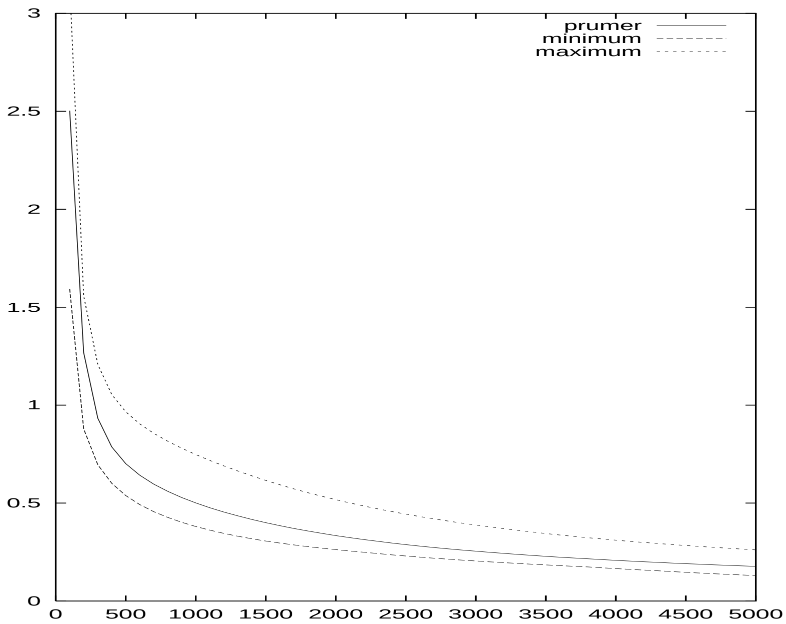
Během dalších 5 000 iterací je pokles chyby daleko pomalejší. Hodnoty jsou shrnuty v tabulce 7.1.

iterace	průměr	minimum	maximum
5000	0.1766	0.1297	0.2612
7500	0.1307	0.0820	0.1959
10000	0.1023	0.0562	0.1552

Tabulka 7.1: Hodnota chybové funkce v daném počtu iterací.

V tabulce 7.2 je uveden počet iterací potřebných k poklesu chybové funkce pod dané ε .

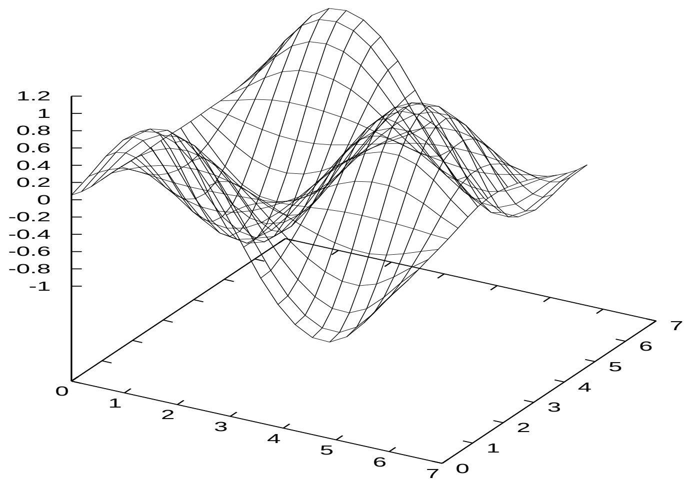
Gradientní algoritmus si s úlohou poradil. Funkce naučené sítě je na obrázku 7.2, průběh její chybové funkce na vstupním prostoru je znázorněn na obrázku 7.3.



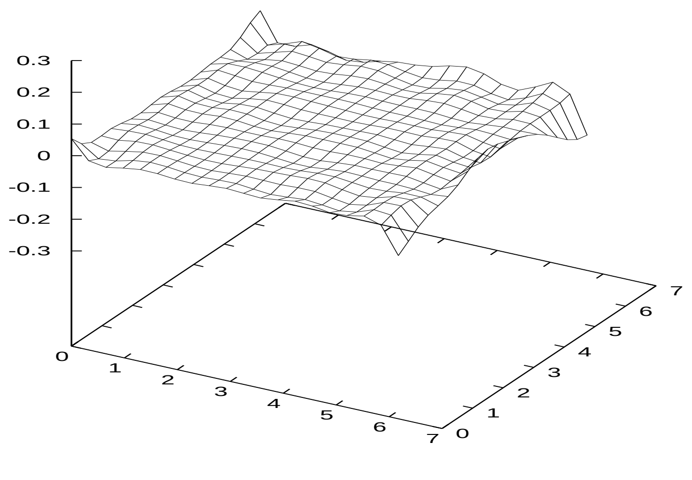
Obrázek 7.1: Graf hodnot chyby sítě v závislosti na iteraci. Pro učení sítě o 50 jednotkách gradientním algoritmem.

ε	průměr	minimum	maximum
1	300	200	500
0.5	1100	600	2200
0.2	4300	3200	7400
0.1	–	6500	–

Tabulka 7.2: Počet iterací potřebných k poklesu chybové funkce pod dané ε.



Obrázek 7.2: Funkce sítě naučené gradientní metodou.



Obrázek 7.3: Chybová funkce pro výsledek gradientního učení.

7.1.2 Tříf ázové učení

šířka	vážená norma	druhá fáze	chyba sítě	čas
ne	ne	-	6.26	2s
ano	ne	Q sousedů, q=1, c=3	0.16	3s
		Q sousedů, q=1, c=3.5	0.08	3s
		Q sousedů, q=1, c=4	0.33	3s
ano	ne	gradientní metoda, P = 2, 200 iterací	12.69	5s
		gradientní metoda, P = 4, 200 iterací	5.65	5s
		gradientní metoda, P = 6, 200 iterací	2.41	5s
		gradientní metoda, P = 10, 200 iterací	0.45	5s
ano	diagonální	gradientní metoda, P = 2, 200 iterací	5.47	8s
		gradientní metoda, P = 4, 300 iterací	13.38	11s
		gradientní metoda, P = 6, 300 iterací	0.65	11s
		gradientní metoda, P = 10, 1000 iterací	0.067	27s
ano	ano	gradientní metoda, P = 2, 200 iterací	2.16	15s
		gradientní metoda, P = 4, 200 iterací	13.12	15s
		gradientní metoda, P = 6, 200 iterací	1.15	15s
		gradientní metoda, P = 10, 500 iterací	0.09	30s

Tabulka 7.3: Třífázové učení.

V druhém případě jsme použili třífázové učení. Protože vstupní vzory jsou rozmístěny rovnoměrně, použijeme rovnoměrné rozmístění RBF jednotek.

Experiment jsme provedli několikrát s různými parametry jednotek a různými způsoby jejich nastavení.

K nastavení vah jsme použili SVD rozklad, který je nejsilnější z metod nejmenších čtverců. RQ rozklad a Moore-Penroseova pseudoinverze dávají méně přesné řešení a v některých případech selžou.

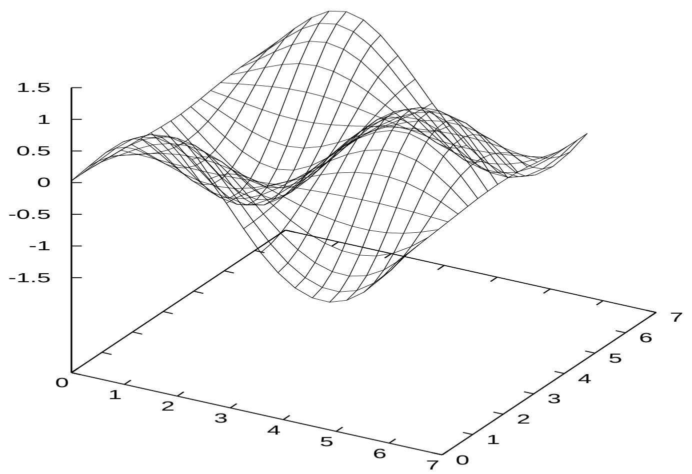
Výsledky provedených experimentů jsou uspořádány do tabulky 7.3. Uvádíme vždy s jakými parametry jednotek byl experiment prováděn a jakou metodou byly tyto parametry nastaveny. U použití průměru vzdáleností q nejbližších sousedů uvádíme koeficient úměrnosti c, u gradientní metody počet iterací a parametr P (viz kapitola 4.12). Dále následuje výsledná chyba sítě a celkový čas výpočtu zaokrouhlený na celé sekundy.

Průměrný čas 100 iterací gradientní metody je pro sítě, kde adaptujeme jen šířky, 1.06 s, pro sítě s adaptovatelnými šířkami a diagonálními maticemi norem 2.41 s a pro sítě s adaptovatelnými šířkami a obecnými maticemi norem 5.12 s.

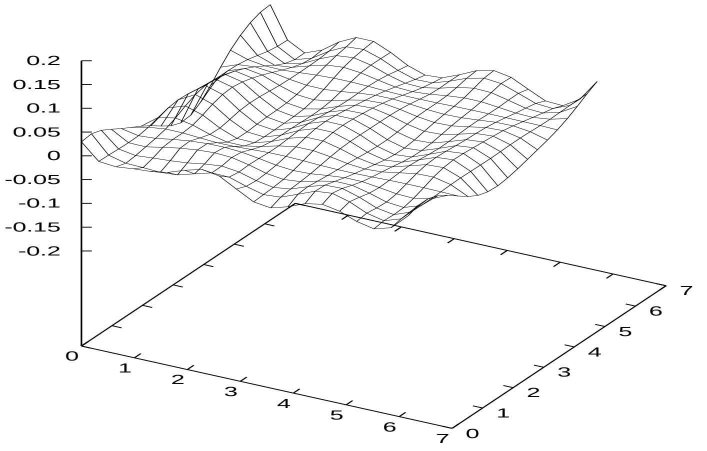
Průměrný čas druhé fáze při použití heuristiky *q sousedů* je 0.004 s. Třetí fáze trvá v průměru 3.35 s.

V tomto případě se ukázalo použití parametru šířky a heuristiky *q sousedů* jako rychlejší a postačující. Minimalizací chybové funkce v druhé fázi a přidáním vážené normy jsme žádné podstatně lepší výsledky nezískali.

Funkce naučené sítě je na obrázku 7.4 a průběh její chybové funkce je znázorněn na obrázku 7.5.



Obrázek 7.4: Funkce sítě naučené třífázovou metodou.



Obrázek 7.5: Chybová funkce na tréninkové množině pro síť naučenou třífázovou metodou.

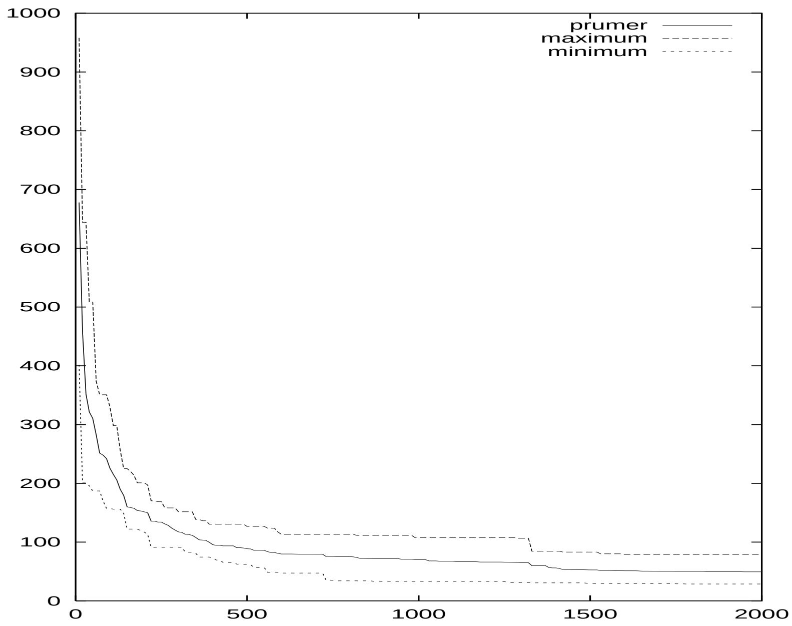
7.1.3 Genetické algoritmy

Poslední metodou použitou v prvním experimentu jsou genetické algoritmy. Nejprve jsme zkusili učit jedince s konstantní délkou, tedy opět sítě o 50 jednotkách. Populace měla 50 jedinců s elitou velikosti 2.

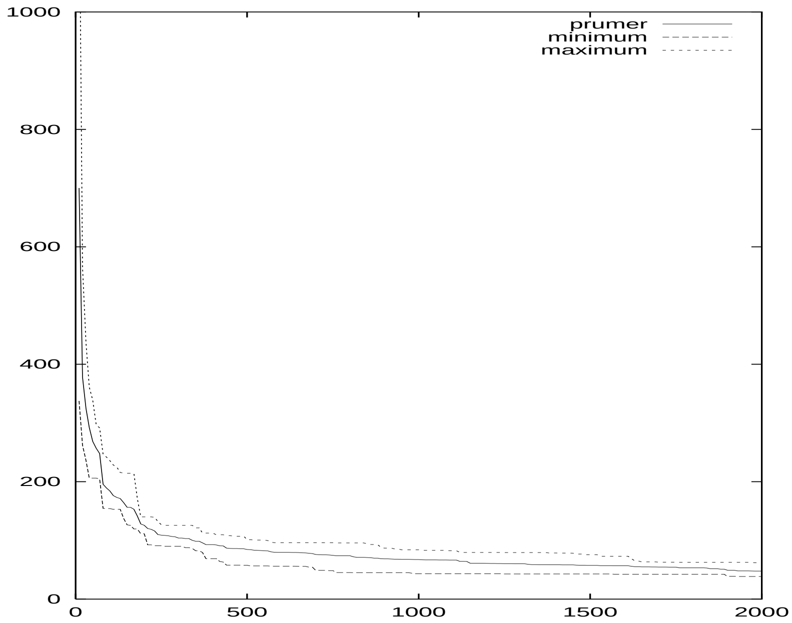
Experiment jsme provedli pětkrát s klasickým algoritmem a pětkrát s kanonickým algoritmem. Všechny výpočty jsme ukončili po 2500. generaci.

Průměrný čas potřebný na 100 generací byl 22 min 8 s.

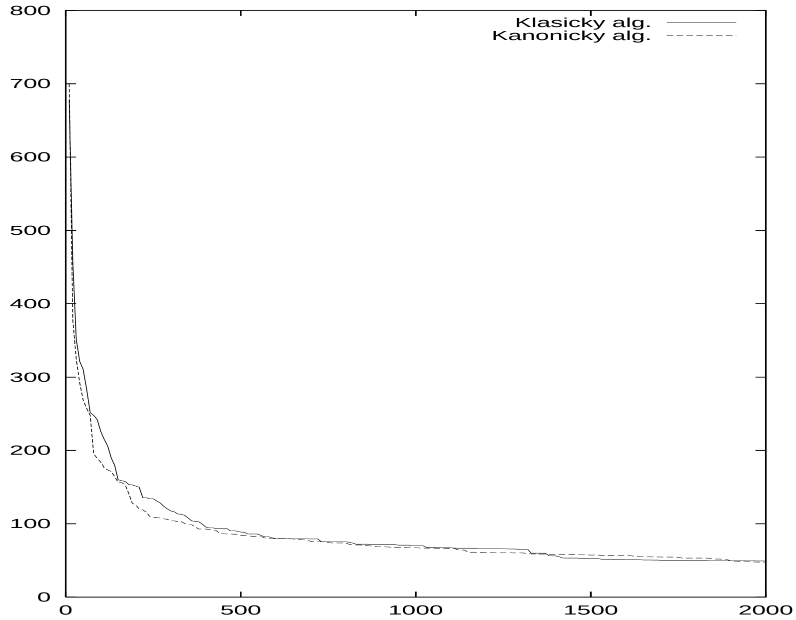
Výsledky byly poměrně špatné. Při použití klasického algoritmu síť dosáhla chyby v průměru 45.6, minimální chyba byla 28.53 a maximální 78.84. Průběh chybové funkce je vidět na obrázku 7.6. U kanonického algoritmu byla průměrná chyba v 2500. generaci 44.87, minimální 35.44 a maximální 60.24. Její průběh během výpočtu je zřejmý z obrázku 7.7. Kanonický algoritmus tedy dopadl jen nepatrně lépe. Když však porovnáme minimální a maximální chybu, zdá se, že u kanonického algoritmu jsou menší rozdíly mezi jednotlivými výpočty. Srovnání průběhu průměru chybové funkce kanonického a klasického algoritmu je na obrázku 7.8.



Obrázek 7.6: Graf hodnot chyby sítě v závislosti na generaci. Pro učení sítě o 50 jednotkách klasickým genetickým algoritmem.



Obrázek 7.7: Graf hodnot chyby sítě v závislosti na generaci. Pro učení sítě o 50 jednotkách kanonickým genetickým algoritmem.



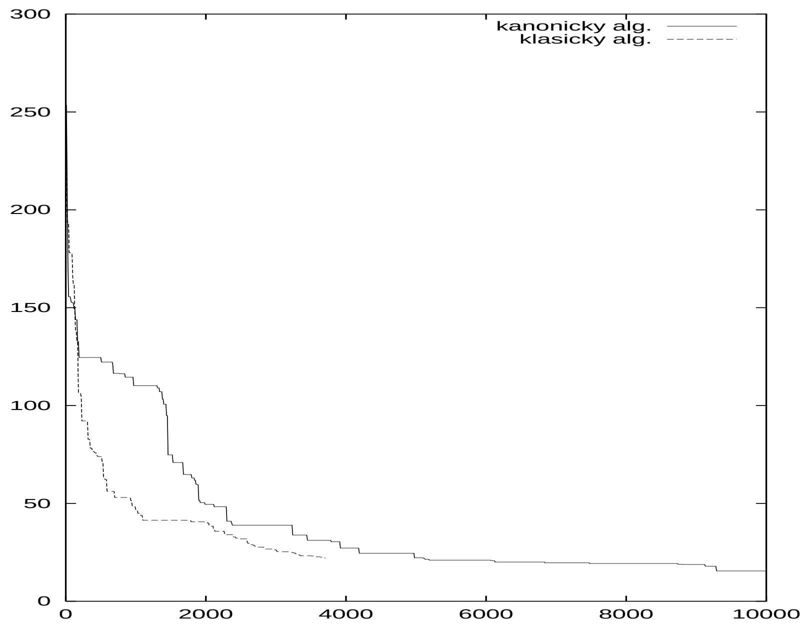
Obrázek 7.8: Srovnání poklesu chybové funkce u kanonického a klasického algoritmu.

Dále jsme zkusili genetický algoritmus s proměnlivou délkou jedince. Jako maximální délku jsme zadali 50 jednotek. Provedli jsme opět deset výpočtů, z toho pět kanonickým algoritmem a pět klasickým algoritmem.

Výpočty běžely různě rychle v závislosti na velikosti jedinců v jednotlivých populacích. Výpočty jsme ukončili po 5 hodinách 59 minutách. Počet generací, které proběhli v jednotlivých výpočtech, a jim odpovídající výsledné chyby jsou zachyceny v tabulce 7.4.

V kanonickém algoritmu se prosadili kratší jedinci, proto běžel rychleji než klasický algoritmus a nakonec dosáhl lepších hodnot. Průběh průměrné chybové funkce obou algoritmů je zobrazen na obrázku 7.9

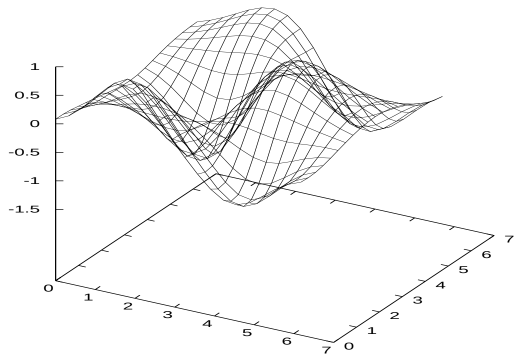
Ani tento experiment nedopadl nejlépe, zvláště když si uvědomíme poměrně velké rozdíly mezi jednotlivými výpočty. Kanonický algoritmus byl v tomto případě úspěšnější a to zejména díky využití kratších jedinců. Funkce sítě s chybou 14.02 (medián výsledných chyb kanonických výpočtů) je znázorněna na obrázku 7.10 a průběh její chybové funkce na vstupním prostoru znázorňuje obr. 7.11.



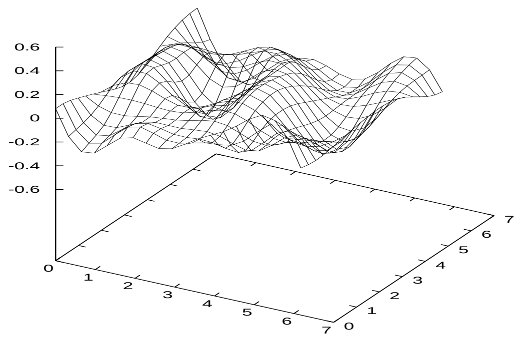
Obrázek 7.9: Vývoj chybové funkce u výpočtu s proměnlivým počtem jedinců.

	Klasický algoritmus			Kanonický algoritmus		
	Iterace	Chyba	Délka	Iterace	Chyba	Délka
1	2330	27.18	22	8230	21.50	14
2	1860	23.62	31	10260	16.18	9
3	2520	21.66	25	12900	9.43	11
4	3710	22.10	32	9300	11.43	11
5	1570	20.69	36	15530	14.02	10

Tabulka 7.4: Situace spuštěných výpočtů v čase 5h 49min



Obrázek 7.10: Funkce sítě naučené genetickým kanonickým algoritmem, síť má 10 jednotek a chybu 14.02.



Obrázek 7.11: Chybová funkce sítě z obrázku 7.10.

7.1.4 Shrnutí

ε	tři fáze	gradientní metoda	genetické alg.
1	3s	5min 5s	–
0.5	3s	18min 37s	–
0.2	3s	1hod 12min 45s	–
0.1	3s	–	–

Tabulka 7.5: Srovnání uvedených metod. Čas potřebný k překonání prahu ε.

Na uvedené úloze jsme vyzkoušeli tři naprosto odlišné přístupy k učení RBF sítí. Výsledky jsou shrnuty v tabulce 7.5. U tří fázové metody se jedná o rovnoměrné umístění středů, nastavení šířek úměrně vzdálenosti q sousedů a použití metody nejmenších čtverců. U gradientní metody adaptujeme všechny parametry jednotek a váhy.

Nejhůře dopadly genetické algoritmy, pro něž jsou sítě o 50 jednotkách příliš velké a které lépe pracují s menšími sítěmi.

Minimální chyba dosažitelná tříkrokovou a gradientní metodou je zhruba stejná. Gradientní metoda je však podstatně pomalejší.

7.2 Aproximace funkce sin(x)cos(y) II.

Další úlohou je opět aproximace funkce $sin(x)cos(y)$. Tentokrát se však zaměříme na menší sítě a menší tréninkové množiny.

Úkolem je pomocí RBF sítě s 16 skrytými jednotkami aproximovat funkci $sin(x)cos(y)$ na základě tréninkové množiny obsahující 10×10 tréninkových vzorů. Tréninkové vzory jsou z intervalu $[0; 2\pi] \times [0; 2\pi]$ a jsou rozmístěny rovnoměrně.

Kapitolu opět rozdělíme na tři části odpovídající různým metodám učení.

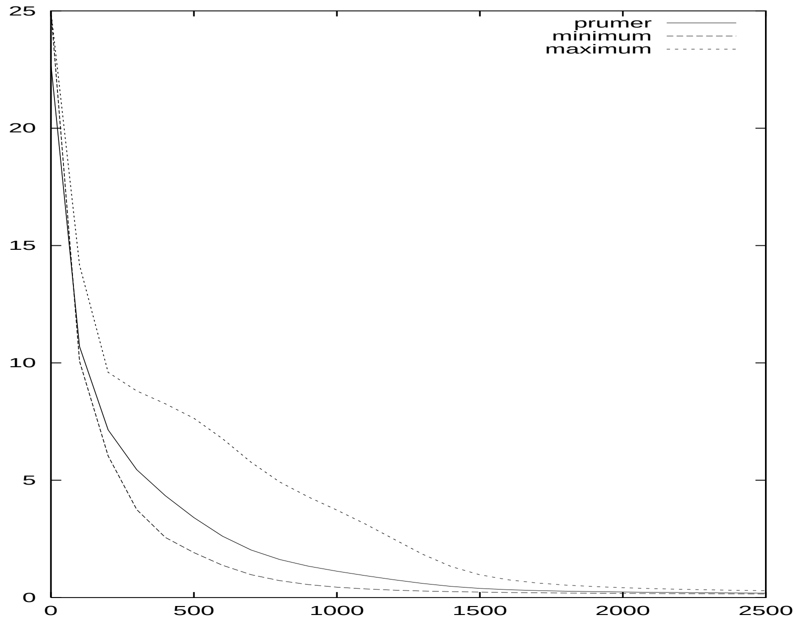
7.2.1 Gradientní algoritmus

První metodou, kterou jsme na řešení úlohy použili, je gradientní algoritmus. Učili jsme sítě s 16 jednotkami. Jednotky měly nastavitelné šířky a matice norem.

Provedli jsme pět výpočtů, všechny běžely 100 000 iterací. Hodnotu chybové funkce jsme počítali každých 100 iterací. Průměrný čas potřebný na 100 iterací byl 3.75 s.

Graf na obrázku 7.12 zobrazuje průběh minima, maxima a průměru chybové funkce realizovaných výpočtů pro prvních 2 500 iterací. Další klesání chybové funkce je pozvolnější a je zřejmé z tabulky 7.6.

V tabulce 7.7 pak vidíme počty iterací, které byly potřeba na dosažení chybové funkce menší než dané ε.



Obrázek 7.12: Graf hodnot chyby sítě v závislosti na iteraci. Pro učení sítě o 16 jednotkách gradientním algoritmem.

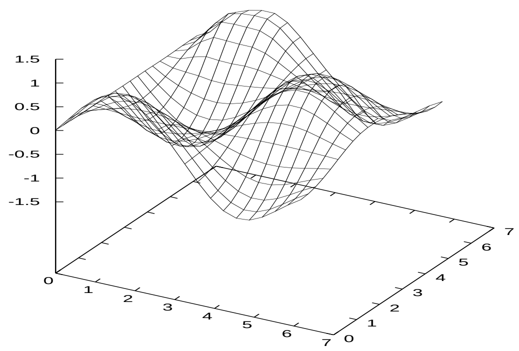
iterace	průměr	mimimum	maximum
10 000	0.097376	0.061401	0.189288
50 000	0.040119	0.010892	0.098053
100 000	0.020647	0.005789	0.045989

Tabulka 7.6: Hodnota chybové funkce

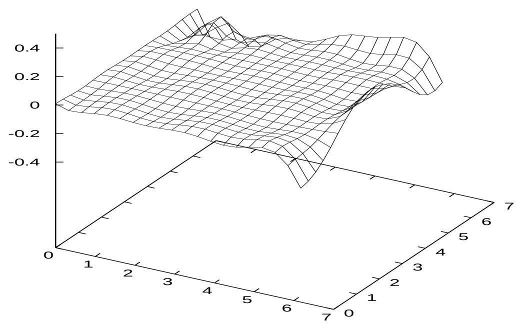
Gradientní metoda byla poměrně úspěšná, funkce naučené sítě je znázorněna na obrázku 7.13 a průběh chybové funkce na vstupním prostoru na obrázku 7.14.

	průměr	minimum	maximum
10.0	200	200	200
1.0	1100	700	1500
0.1	9300	5000	48600
0.05	41600	12600	85000
0.01	–	54700	–

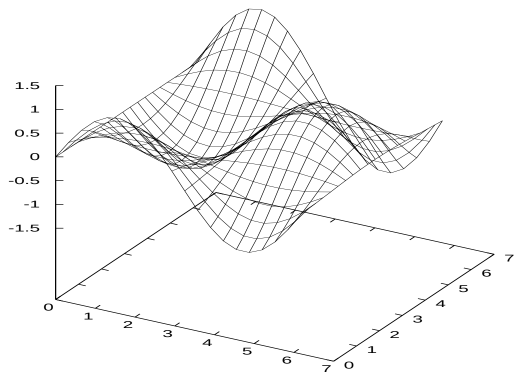
Tabulka 7.7: Počet iterací potřebných k dosažení prahové hodnoty ε.



Obrázek 7.13: Funkce sítě naučené gradientním algoritmem.



Obrázek 7.14: Chybová funkce na tréninkové množině.



Obrázek 7.15: Funkce sítě naučené třífázovou metodou.

7.2.2 Třífázové učení

Druhou použitou metodou je třífázové učení. Učili jsme RBF sítě s 16 jednotkami.

Vzhledem k rovnoměrnému rozmístění vstupů jsme použili jako první krok rovnoměrné rozmístění RBF jednotek po vstupním prostoru.

V druhé fázi jsme zkusili jak jednoduchou heuristiku *q sousedů*, minimalizaci chybové funkce gradientní metodou, tak úplné vynechání této fáze.

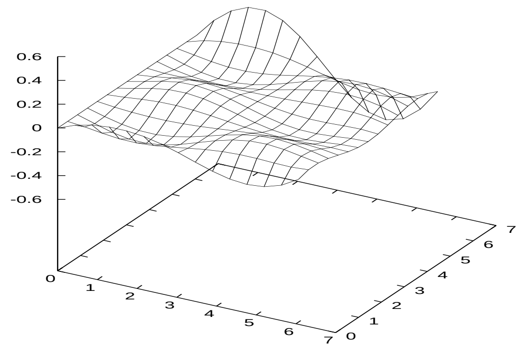
Nakonec jsme doučili váhy sítě použitím metody nejmenších čtverců, využívající SVD rozklad.

V tabulce 7.8 je přehled použitých metod, výsledných chyb a časů potřebných k výpočtu. Průměrný čas k provedení 100 iterací gradientní metody byl pro sítě s šířkami 0.0011 s, pro sítě s šířkami a diagonálními maticemi norem 0.0027 s a pro sítě s šířkami a obecnými maticemi norem 0.0272 s. Při použití heuristiky *q sousedů* trval druhý krok v průměru 0.0003s, Čas potřebný k dopočtení hodnot vah byl průměrně 0.5s.

Třífázové učení je tedy pro malé úlohy velice rychlé. Funkce naučené sítě je znázorněna na obrázku 7.15 a průběh její chybové funkce na 7.16.

šířka	vážená norma	druhá fáze	chyba sítě	čas
ne	ne	-	5.799	1s
ano	ne	Q sousedů, q=1, c=3	0.342	1s
		Q sousedů, q=1, c=3.5	0.560	1s
		Q sousedů, q=1, c=4	0.747	1s
ano	ne	gradientní metoda, P = 2, 500 iterací	2.920	1s
		gradientní metoda, P = 4, 500 iterací	1.391	1s
		gradientní metoda, P = 6, 500 iterací	1.264	1s
		gradientní metoda, P = 10, 500 iterací	1.264	1s
ano	diagonální	gradientní metoda, P = 2, 500 iterací	2.802	2s
		gradientní metoda, P = 4, 500 iterací	3.946	2s
		gradientní metoda, P = 6, 500 iterací	0.214	2s
		gradientní metoda, P = 10, 2000 iterací	0.214	5s
ano	ano	gradientní metoda, P = 2, 400 iterací	1.290	2s
		gradientní metoda, P = 4, 400 iterací	2.742	2s
		gradientní metoda, P = 6, 400 iterací	0.969	2s
		gradientní metoda, P = 10, 1000 iterací	1.181	6s

Tabulka 7.8: Třífázové učení.



Obrázek 7.16: Chybová funkce sítě z obr. 7.15.

7.2.3 Genetické algoritmy

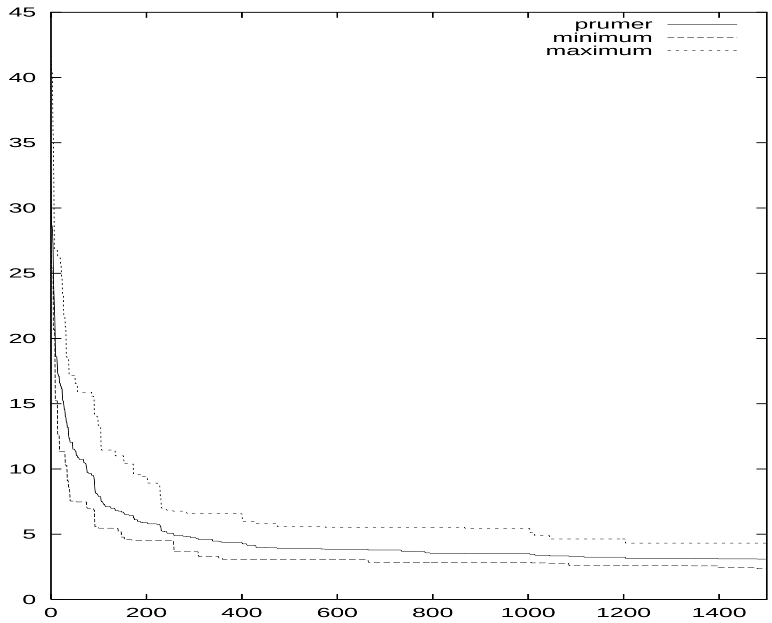
Třetí metodou použitou na aproximaci funkce $sin(x)cos(y)$ byly genetické algoritmy.

Použili jsme populaci 50 jedinců a pustili jsme pětkrát klasický algoritmus a pětkrát kanonický algoritmus. Do elity byli v obou případech vybíráni dva nejlepší jedinci. Všichni jedinci měli stejnou délku, 16 jednotek.

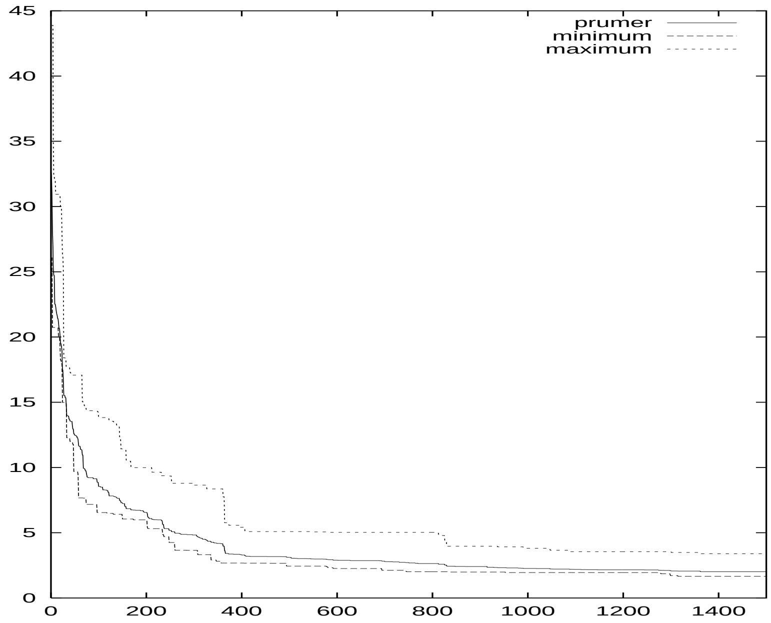
Oba výpočty jsme nechali běžet 5 000 generací, průměrný čas potřebný na 100 generací byl v obou případech 38.45 s.

Na obrázcích 7.17 a 7.18 je vidět průběh minima, maxima a průměru chybové funkce v závislosti na generaci. Srovnání průměrného výpočtu kanonického a klasického algoritmu je pak na obrázku 7.19, kde vidíme, že kanonický výpočet si vedl o něco lépe.

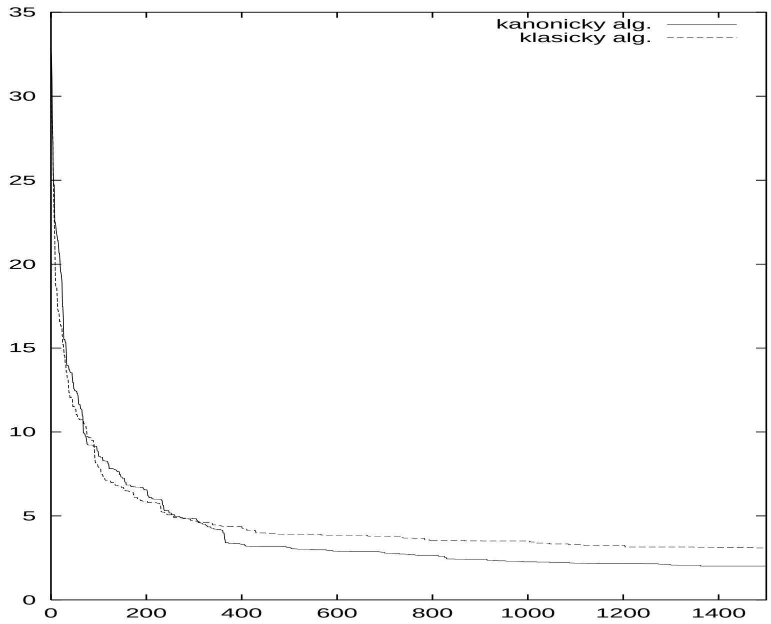
Funkce sítě naučené kanonickým genetickým algoritmem je na obrázku 7.20, obrázek 7.21 znázorňuje průběh její chybové funkce.



Obrázek 7.17: Graf hodnot chyby sítě v závislosti na generaci. Pro učení sítě o 16 jednotkách nekanonickým genetickým algoritmem.



Obrázek 7.18: Graf hodnot chyby sítě v závislosti na generaci. Pro učení sítě o 16 jednotkách kanonickým genetickým algoritmem.



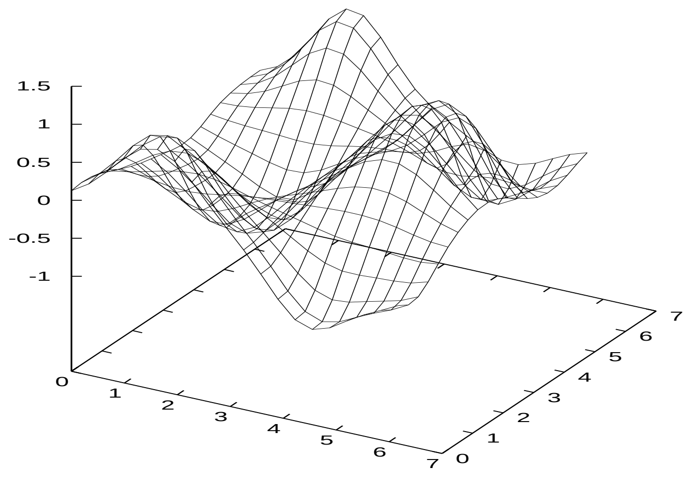
Obrázek 7.19: Srovnání kanonického a klasického genetického algoritmu.

	počet generací	průměr	minimum	maximum
klasický algoritmus	2000	2.833	1.412	4.125
	3000	2.482	1.412	3.422
	5000	2.070	1.396	2.809
kanonický algoritmus	2000	1.658	1.455	2.275
	3000	1.298	1.192	1.747
	5000	1.064	0.822	1.640

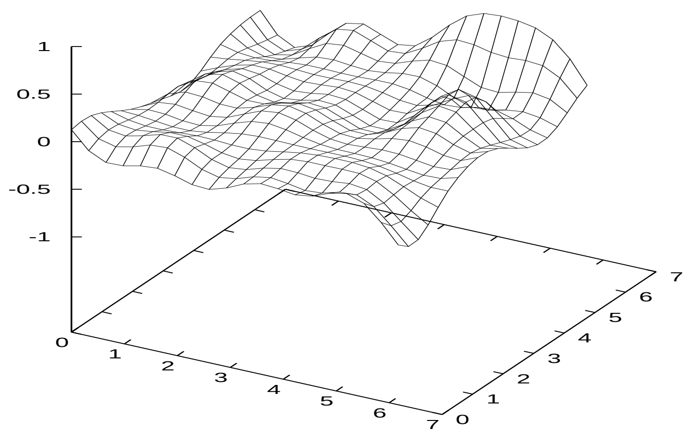
Tabulka 7.9: Průběh chyby v pozdějších generacích.

	Kanonický alg.			Klasický alg.		
ε	průměr	minimum	maximum	průměr	minimum	maximum
20	20	19	27	9	9	32
10	68	48	168	76	34	173
5	260	234	813	258	148	1014
2	1511	832	2342	–	1660	–
1	–	3734	–	–	–	–

Tabulka 7.10: Počet generací nutných pro pokles chyby pod dané ε.



Obrázek 7.20: Funkce sítě naučené kanonickým genetickým algoritmem.

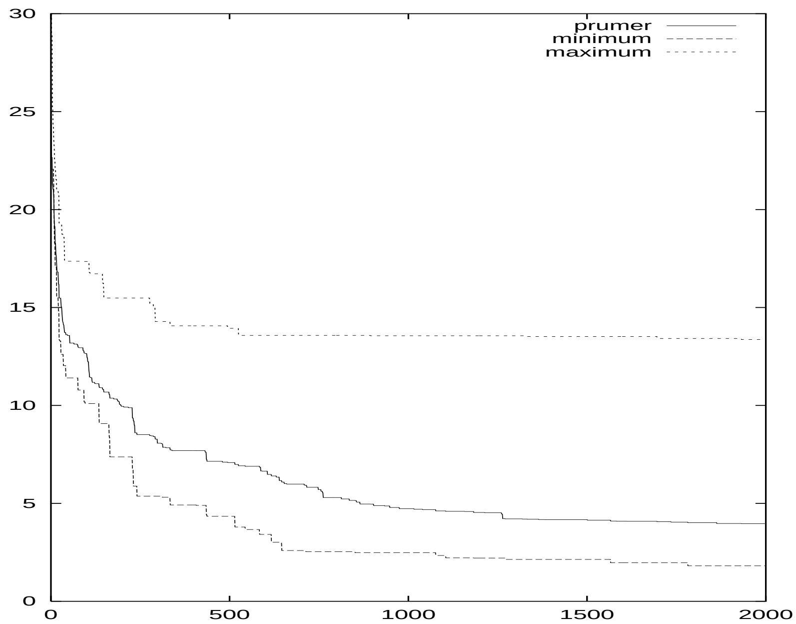


Obrázek 7.21: Chybová funkce sítě z obr. 7.20.

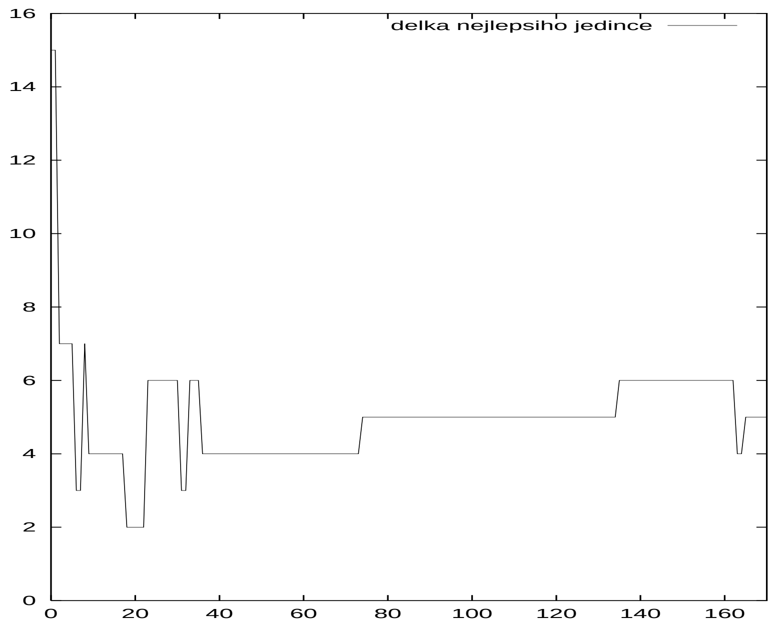
Dalších pět výpočtů jsme provedli s kanonickým algoritmem s proměnlivou délkou jedince. Výpočet běžel 5 000 generací a průměr výsledných chyb je 3.655. Minimální dosažená chyba byla 1.4947 a maximální 13.3641. Tyto hodnoty už jsou přijatelné.

Průměrný čas na výpočet 100 generací je 8.34 s, je však závislý na délce jedinců v dané populaci, proto se během výpočtu mění.

Průběh chybové funkce je znázorněn na obrázku 7.22. Na obrázku 7.23 pak vidíme jak se mění délka nejlepšího jedince v průběhu výpočtu. Na začátku výpočtu jsou sítě v populaci spíše náhodné a menší sítě mají spíše menší hodnotu chybové funkce. Proto délka nejdříve prudce klesne na malé hodnoty. Během výpočtu se pak pomalu prosazují i delší jedinci, ale nárůst délky je velice pozvolný.



Obrázek 7.22: Chybová funkce, kanonický genetický algoritmus s proměnlivou délkou jedince.



Obrázek 7.23: Délka nejlepšího jedince v průběhu prvních iterací.

ε	průměr	minimum	maximum
20	9	9	23
10	196	135	–
5	865	334	–
2	–	1566	–

Tabulka 7.11: Počet generací nutných pro pokles chyby pod dané ε. Kanonický algoritmus s proměnlivou délkou jedince.

7.2.4 Shrnutí

Úlohu aproximace funkce $sin(x)cos(y)$ jsme řešili třemi různými způsoby. Všechny metody si s úlohou dokázaly poradit s přijatelnými výsledky. Opět nejrychlejší je třífázová metoda, nejpomalejší jsou dle očekávání genetické algoritmy.

Nejpřesnějšího řešení jsme dosáhli pomocí gradientní metody, avšak za cenu vyšších časových nároků.

Tabulka 7.12 obsahuje časové srovnání uvedených metod. U třífázové metody se jedná o použití rovnoměrného rozmístění středů a použití metody nejmenších čtverců. Přičemž druhý krok v prvních dvou případech řešíme pomocí q nejbližších sousedů, kde $q = 1$ a koeficient úměrnosti je roven 3. Gradientní metoda adaptuje všechny parametry. U genetického algoritmu se

ε	tři fáze	gradientní metoda	genetické alg.
2	1 s	30 s	9 min 40 s
0.5	1 s	52 s	–
0.25	2 s	1 min 15 s	–
0.1	–	5 min 48 s	–
0.05	–	26 min	–

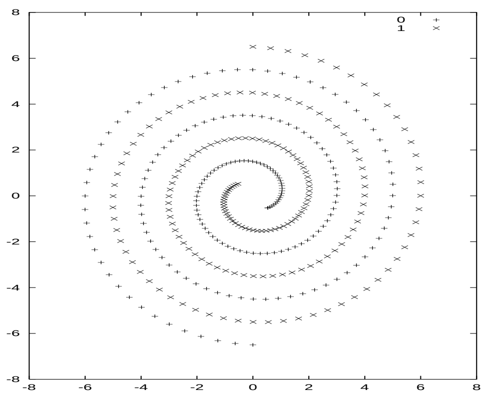
Tabulka 7.12: Srovnání uvedených metod. Čas potřebný k překonání prahu ε.

jedná o kanonickou verzi s populací 50 jedinců, elitou 2 a konstantní délkou jedinců.

7.3 Problém dvou spirál

Dalším prováděným experimentem byla klasifikační úloha, tzv. *problém dvou spirál*. Tréninková data představují dvě množiny bodů a úkolem sítě je separovat tyto dvě množiny. To znamená, že chceme síť naučit tak, aby byla o předloženém vstupu schopna rozhodnout, do které skupiny patří.

Tréninková množina obsahuje 372 vzorků (obr. 7.24). Pro účely učení ohodnotíme jednu množinu bodů hodnotou nula a druhou hodnotou jedna. V ideálním případě by pak naučená síť měla po předložení souřadnic bodu z jedné z těchto množin vrátit buď hodnotu nula nebo jedna, podle toho, zda bod patří do první nebo druhé množiny.



Obrázek 7.24: Tréninková data — dvě spirály.

U naučené sítě budeme reálné výstupy interpretovat tak, že hodnoty z intervalu $(-0.4, 0.4)$ budou odpovídat první množině, hodnoty $(0.6; 1.4)$ druhé množině.

Z povahy tréninkových dat předpokládáme, že problém bude vyžadovat poměrně velké množství RBF jednotek. Použijeme tedy síť se 150 skrytými jednotkami. Tuto síť budeme učit gradientním algoritmem a třífázovým učením. V obou případech vyzkoušíme jak sítě s euklidovskou tak váženou normou, abychom ukázali rozdíl a případně přínos použití vážené normy.

7.3.1 Gradientní metoda

První použitou metodou na problém klasifikace dvou spirál byla gradientní metoda.

Učili jsme síť se 150 jednotkami. Výpočet jsme pustili pětkrát s použitím euklidovské normy a pětkrát s použitím vážené normy, tedy v druhém případě měly jednotky adaptovatelnou matici normy.

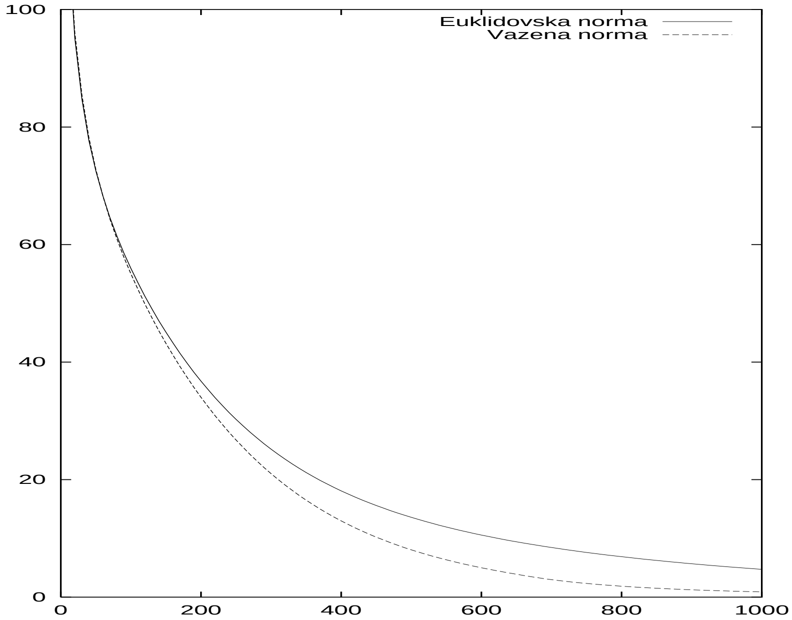
Všechny výpočty jsme nechaly spočítat 10 000 iterací, přičemž hodnota chybové funkce byla zjišťována při každé desáté iteraci. Výpočet, při kterém se adaptovaly i matice vah, běžel asi čtyřikrát pomaleji. Průměrný čas potřebný na 100 iterací bez adaptace norem byl 36.4 s, pro 100 iterací s adaptací norem 133.4 s.

Průměrná počáteční chyba v obou případech byla zhruba 180. Výsledné hodnoty chybové funkce jsou zobrazeny v tabulce 7.13. Průběh výpočtu je zřejmý z obrázku 7.25. Chybová funkce klesá rychleji pro výpočet s adaptovatelnou maticí vah. Tento výpočet je však pomalejší, pro objektivní porovnání tedy slouží tabulka 7.14, kde je uveden počet iterací a odpovídající čas potřebný k poklesu chybové funkce pod dané ε.

Euklidovská norma			Vážená norma		
průměr	minimum	maximum	průměr	minimum	maximum
0.057	0.042	0.075	0.012	0.010	0.014

Tabulka 7.13: Výsledná hodnota chybové funkce (po 10 000 iteracích).

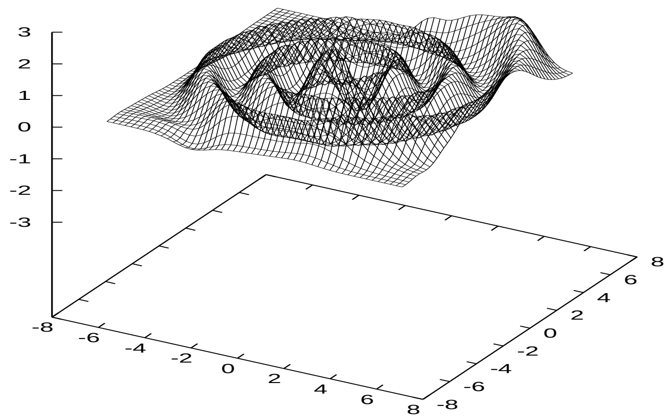
RBF síť naučená gradientním algoritmem problém klasifikace dvou spirál vyřešila. Na obrázku 7.26 je reálná funkce sítě. Obrázek 7.27 pak zobrazuje ohodnocení bodů v rovině naučenou sítí.



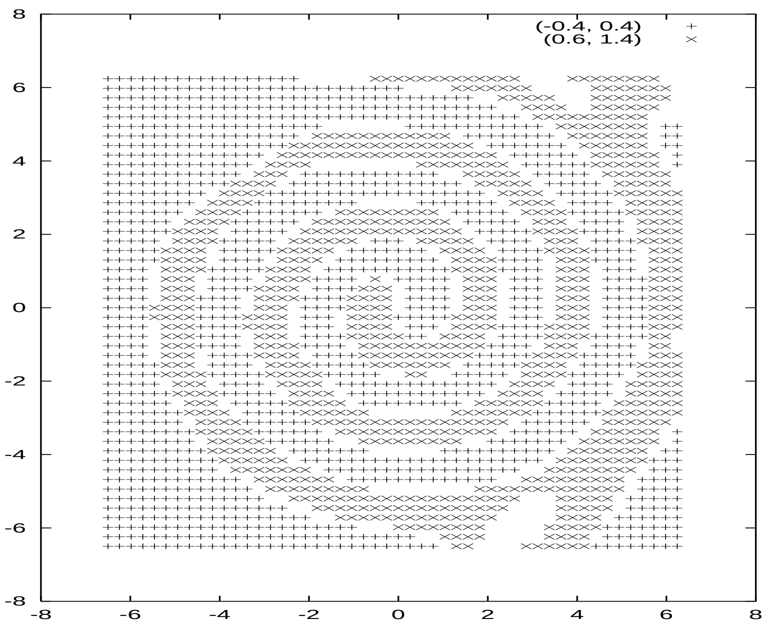
Obrázek 7.25: Porovnání poklesu chybové funkce pro sítě s váženými normami a euklidovskými normami.

	Euklidovská norma		Vážená norma	
ε	iterace	čas	iterace	čas
100	20	7.27s	20	26.68s
10	630	3min 49s	460	10min 14s
1	1990	12min 4s	970	21min 34s
0.5	2590	15min 43s	1220	27min 8s
0.1	6530	39min 37s	2430	54min 2s
0.05	–	–	3610	1hod 20min 16s

Tabulka 7.14: Průměrný počet iterací a čas potřebný k poklesu chybové funkce pod dané ε.



Obrázek 7.26: Funkce sítě naučené gradientní metodou.



Obrázek 7.27: Klasifikace bodů v rovině sítí naučenou gradientní metodou.

7.3.2 Třífázové učení

Druhou metodu, kterou jsme aplikovali na problém dvou spirál, bylo třífázové učení. Protože tréninková data jsou ve vstupním prostoru uspořádána na spirálách, je rozmístění středů RBF jednotek zajímavým problémem. Zaměříme se tedy na první fázi a metody vektorové kvantizace.

Pro určení jednotek vybereme vhodnou metodu a parametry pro další kroky a doučíme zbylé parametry. Opět budeme uvažovat jak síť s euklidovskou normou tak s váženou normou. Sítě budou mít 150 RBF jednotek.

Nalezení vhodných středů jsme řešili pomocí metod vektorové kvantizace, Lloydova algoritmu, Kohonenova učení, Kohonenova učení s lokální pamětí a genetických algoritmů.

U Lloydova algoritmu dochází při jedné iteraci k průchodu celou tréninkovou množinou, naproti tomu u Kohonenova učení adaptujeme zvolenou jednotku po načtení každého tréninkového vzoru. Proto nebudeme nyní používat pojem iterace, ale průchod tréninkovou množinou.

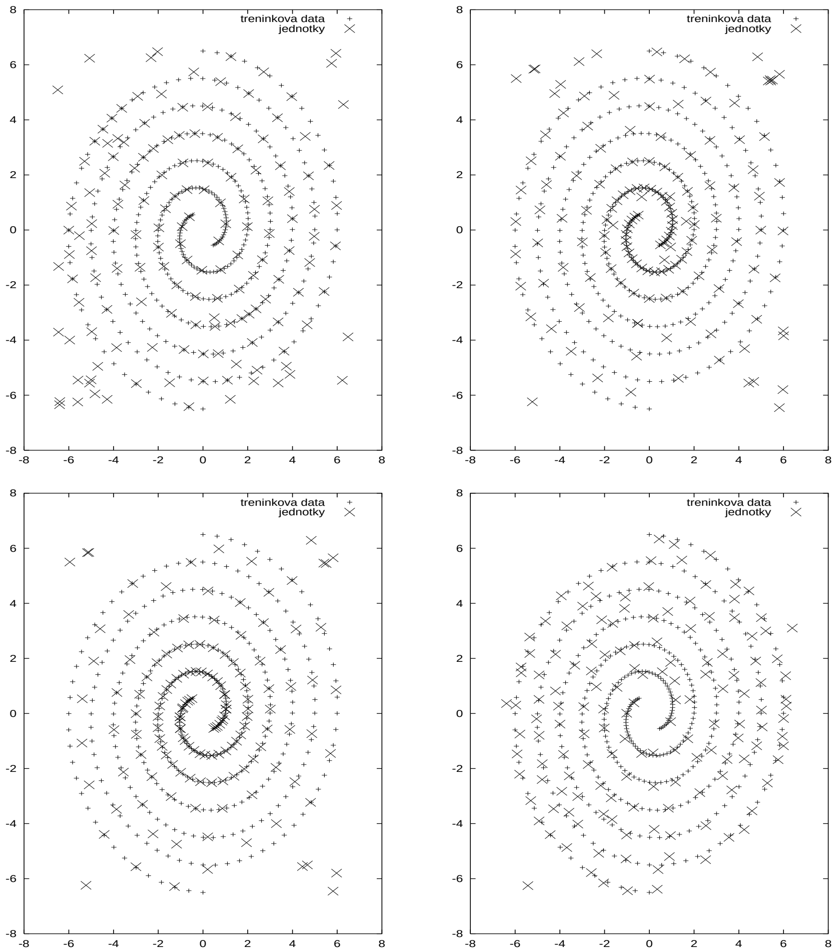
Pro každou z metod jsme výpočet provedli pětkrát. Lloydův algoritmus potřebuje k průchodu tréninkovou množinou v průměru 0.08 s. U Kohonenova učení trvá jeden průchod tréninkovou množinou zhruba 0.17 s, při použití lokální paměti 0.33 s. Genetické algoritmy jsme nechali běžet 2 500 iterací, populace měla 50 jedinců, elita dva jedince. Při výpočtu každé generace bylo tedy potřeba projít tréninkovou množinu 50krát. Průměrný čas potřebný na výpočet 100 generací byl 10 min 25 s.

Tabulka 7.15 obsahuje výsledné chyby, počet potřebných průchodů tréninkovou množinou a čas pro jednotlivé metody. Nejhůře dopadl Lloydův algoritmus, nejlépe Kohonenovo učení s lokální pamětí. Protože počet jednotek je poměrně vysoký, nejsou rozdíly ve výsledné hodnotě chybové funkce příliš velké.

Na obrázku 7.28 je vidět výsledné rozmístění reprezentantů. Ani použití lokální paměti nedokázalo přitáhnout všechny jednotky. Je to způsobeno velkým počtem jednotek v oblasti spirál, tedy některá z těchto jednotek má vždy dostatečně malou četnost výhry, aby kompetici vyhrála.

	chyba	počet průchodů t.m.	čas
Lloydův alg.	0.1296	8	1 s
Kohonenovo učení	0.1066	50	9 s
Lokální pamět	0.0940	200	1 min 6 s
Genetický alg.	0.1183	50 × 2 500	4 hod 31 min

Tabulka 7.15: Přehled použitých metod vektorové kvantizace.

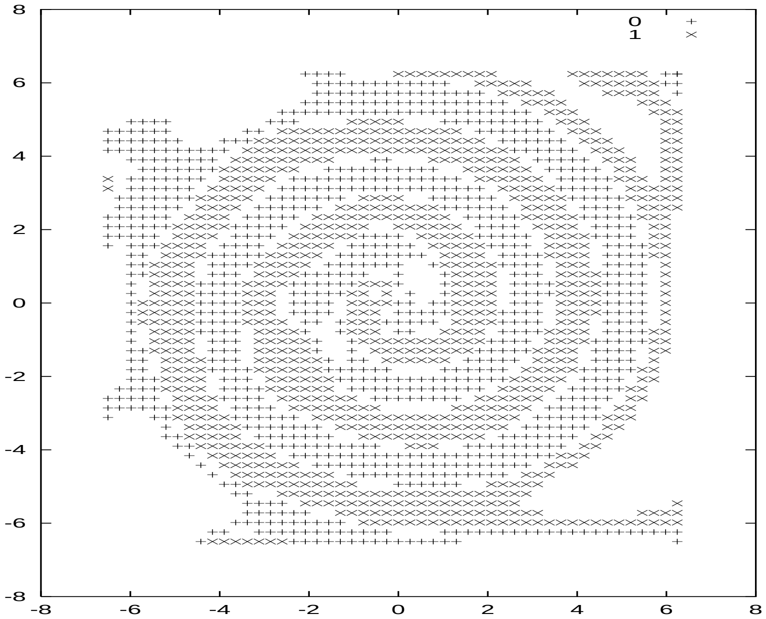


Obrázek 7.28: Lloydův algoritmus (vlevo nahoře). Kohonenovo učení (vlevo nahoře). Lokální paměť (vlevo dole). Genetický algoritmus (vpravo dole).

Středy jsme nastavili pomocí Kohonenova učení s lokální pamětí. Pro druhý krok jsme použili minimalizaci chybové funkce s parametrem $P = 5$. V obou případech pomocí 200 iterací gradientní metody. Váhy jsme doučili pomocí metody nejmenších čtverců využívající SVD rozklad.

Výsledná hodnota chybové funkce je 1.101 při použití euklidovské normy a 0.051 pro sítě používající vážené normy.

Klasifikace bodů v rovině pomocí sítě s váženou normou naučenou třífázovou metodou je znázorněna na obrázku 7.29.



Obrázek 7.29: Výstup sítě s váženou normou, naučené třífázovou metodu.

7.3.3 Shrnutí

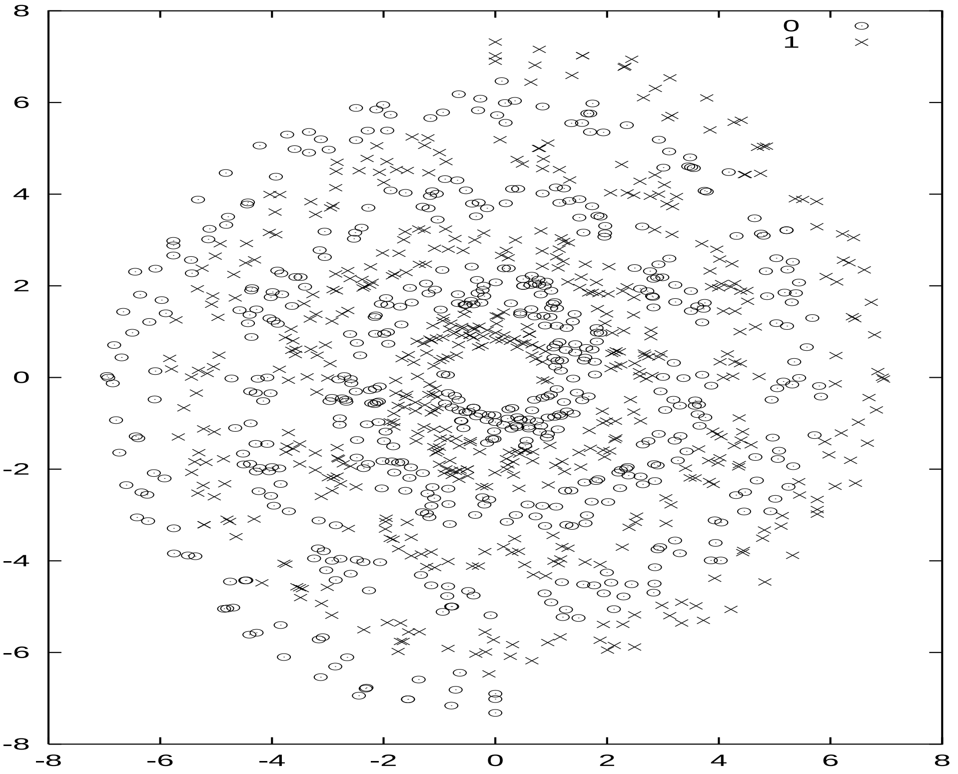
Řešili jsme klasifikační problém pomocí dvou metod učení, gradientního učení a třífázového učení. Genetické algoritmy se pro tento případ nedají efektivně využít, neboť sítě o 150 jednotkách jsou pro ně příliš velké.

Uvažovali jsme síť s euklidovskou normou i použití vážených norem. V obou případech nalezl gradientní algoritmus řešení s menší chybou. Třífázové učení si však poradí s problémem během několika minut, zatímco gradientní algoritmus potřebuje zhruba hodinu.

Obě metody nalezly lepší řešení pro síť, která používá vážené normy. Jedná se o dvourozměrný vstupní prostor a tak víme, že RBF jednotka s euklidovskou normou dává významné výstupy na kruhové oblasti kolem svého středu. Vážená norma umožňuje transformovat tuto oblast do tvaru elipsy, která může být v případě obecné matice normy ještě pootočena. Lepší výsledky v případě použití vážené normy lze vysvětlit tím, že vstupní vzory uspořádané ve spirálách je jednodušší pokrýt elipsami než kružnicemi.

7.4 Problém dvou spirál s přidaným šumem

Posledním experimentem je problém dvou spirál obohacený o šum. Tréninkových dat jsme tentokrát vygenerovali více. Data neleží jen na spirálách, ale také v jejich okolí. Tréninková množina obsahuje 1116 tréninkových vzorů a je zobrazena na 7.30.

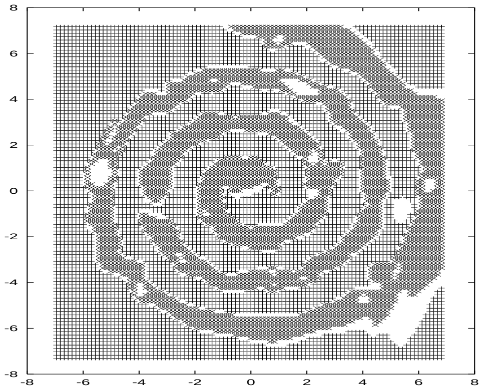


Obrázek 7.30: Tréninková množina se šumem.

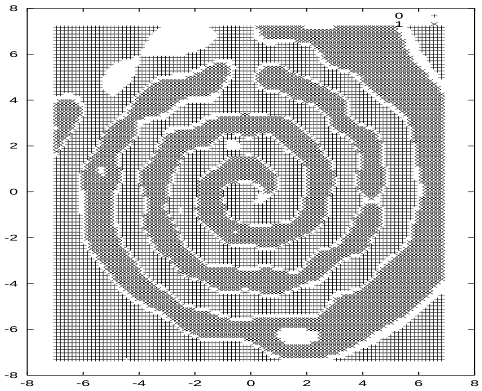
K řešení tohoto problému jsme použili gradientní algoritmus a třífázovou metodu. Učili jsme síť s 200 skrytými jednotkami a váženou normou.

Výpočet gradientní metodou jsme pustili na 10 000 iterací, pětkrát. Průměrná časová náročnost 100 iterací byla 9 min 40 s. Výsledná průměrná chyba je 14.56, minimální 13.49 a maximální 15.87. Na obrázku 7.31 vidíme klasifikaci vstupního prostoru pomocí naučené sítě. Můžeme říci, že výsledek je uspokojivý.

Třífázové učení bylo složeno s Kohonenova učení s lokální pamětí, gradientní metody s parametrem $P = 4$ a SVD rozkladu. Výpočet trval 4 min 20 s a výsledná hodnota chybové funkce je 48.87. Klasifikace vstupního prostoru pomocí naučené sítě je znázorněna na obrázku 7.32.



Obrázek 7.31: Výstup sítě s váženou normou, naučené gradientní metodou na zašuměná data.



Obrázek 7.32: Výstup sítě naučené třífázovým učením.

7.5 Závěr

Převedli jsme několik úloh, jež lze řešit pomocí RBF sítí. Uvedené problémy jsme řešili pomocí tří základních přístupů — gradientních algoritmů, třífázového učení a genetických algoritmů.

Genetické algoritmy měly největší časové nároky a v testovaném počtu generací nepřinesly lepší výsledky než ostatní metody. V některých případech jsou však výsledky genetických algoritmů postačující a do budoucna lze uvažovat o jejich paralelizaci a tedy o výrazném zlepšení časové náročnosti. Pak mohou být genetické algoritmy dobrou alternativou ostatním metodám.

Gradientní algoritmus je druhý v časové náročnosti. Pokud však provedeme dostatečný počet iterací, dosáhneme ve většině případů poměrně přesné řešení. Během experimentů jsme se nesetkali s žádným výrazným uvíznutím v lokálním minimu. Důvodem může být fakt, že chybová funkce je konvexní vůči parametrům vah a nekonvexní pouze vůči parametrům RBF jednotek (středy, šířky, váhy). Protože váhy určují lineární kombinaci výstupní vrstvy, dá se říci, že jejich optimální nastavení s ohledem na skrytou vrstvu je důležitější, než optimální nastavení ostatních parametrů.

Třífázová metoda je jednoznačně nejrychlejší metodou. Během experimentů se ukázalo, že u třífázové metody je výhodnější použít větší počet jednotek, než např. u gradientního učení. To může být způsobeno tím, že nastavení RBF jednotek během prvních dvou kroků nevyužívá informace o požadovaných výstupech. Je totiž zřejmé, že při gradientním učení a genetickém učení mají RBF jednotky tendenci umístit se tam, kde má požadovaná funkce podstatné výstupy. Jinak řečeno v oblasti, kde má požadovaná funkce nulový výstup nebo nevýznamný výstup, žádné RBF jednotky nepotřebujeme nebo jich postačí méně.

U všech metod lze pozorovat větší hodnoty chybové funkce na okrajích tréninkové množiny, což je pochopitelné, neboť to jsou oblasti, kde máme nejméně informací o učené funkci.

Ukázali jsme, že RBF sítě se dají využít k řešení jak aproximačních tak klasifikačních problémů. Volba metody učení, velikost skryté vrstvy a jejích parametrů je závislá na velikosti a typu problému, na dostupných výpočetních možnostech a požadované přesnosti řešení.

Kapitola 8

Závěr

V první — teoretické — části práce jsme se seznámili s různými přístupy k učení neuronových sítí typu RBF. Uvedli jsme model této neuronové sítě rozšířený o použití vážených norem.

Inspirovali jsme se algoritmem zpětného šíření pro vícevrstvé neuronové sítě a odvodili obecný gradientní algoritmus pro neuronové sítě typu RBF. Tento algoritmus je nezávislý na použité radiální funkci, umí adaptovat všechny parametry, včetně šířek a matic vážených norem. Navrhli jsme algoritmus pro výpočet derivací chybové funkce podle jednotlivých parametrů, které jsou během gradientního algoritmu potřeba. Ten jsme pak optimalizovali pro Gaussovu funkci.

Dále jse se zabývali třífázovým učením. To bylo nutné obohatit o možnost adaptace matic vážených norem. Gradientní algoritmus hledající vhodné hodnoty šířek, jsme tedy upravili tak, aby byl schopen adaptovat i tyto matice.

Posledním zkoumaným přístupem bylo použití genetického algoritmu. Kladli jsme důraz na srovnání kanonické a klasické verze tohoto algoritmu. Vyzkoušeli jsme jak pevné délky jedinců, tak použití volné délky jedince k nalezení vhodného počtu RBF jednotek. Genetické algoritmy jsme využili nejenom při učení všech parametrů sítě, ale navrhli jsme jejich začlenění do třífázového učení, kde slouží k hledání vhodných hodnot středů RBF jednotek.

Druhou částí práce je realizace softwarových komponent, které umožňují testování navržených metod. Tyto komponety jsou softwarovými agenty multiagentního systému Bang2. Využití agentů umožňuje snadné rozšiřování systému o nové metody nebo nové kombinace stávajících metod.

Třetí část práce se věnuje testování realizovaných metod. Uskutečnili jsme několik experimentů, na kterých jsme předvedli jednotlivé metody a srovnali jejich výsledky. Ukázali jsme schopnot RBF sítí řešit jak aproximační tak klasifikační úlohy.

Do budoucna se počítá především s možností použití libovolné radiální basické funkce. Tato funkce bude realizována pomocí dalšího agenta. Již v současnosti je možné vytvořit RBF síť používající obecnou radiální funkci. Učit tuto síť je však zatím možné pouze pomocí těch metod, které neobsahují gradientní metody. Gradientní metody využívají znalost derivace konkrétní radiální funkce. Protože Gaussova funkce je snadno derivovatelná a navíc hodnotu potenciálu vždy umocňuje, optimalizovali jsme gradientí algoritmy pro tuto funkci. Jsou však napsány tak, aby jejich rozšíření o spolupráci s agentem realizujícím konkrétní radiální funkci nebylo problémem.

Dále předpokládáme, že bude vytvořen agent, který bude realizovat rozhraní k nějakému pokročilému numerickému softwaru (např. knihovně LAPACK), který realizuje větší škálu metod. Pokud bude mít uživatel takový software nainstalován, bude moci použít tohoto agenta namísto současného agenta implementujícího metodu nejmenších čtverců.

Plánuje se také rozšíření genetických algoritmů o další sady operátorů využívající jak jiné druhy mutace a křížení, tak jiné možnosti kódování jedninců představujících neuronové sítě.

Dalším chystaným vylepšením genetických algoritmů je jejich paralelizace. Ta je u genetických algoritmů, na rozdíl od ostatních metod, velice snadno realizovatelná. Implementace pomocí několika softwarových agentů nám budoucí paralelizaci výrazně usnadní.

Realizovaný systém je tedy otevřený a rozšiřitelný. Doufáme, že se bude nadále vyvíjet a stane se odrazovým můstkem pro tvorbu různých hybridních metod učení RBF sítí, případně součástí rozsáhlých modelů umělé inteligence.

Literatura

[1] Chris Bishop. Improving the Generalization Properties of RBF Neural Networks. Neural Computation, 3:579-588, 1991.

[2] Stan Franklin, Art Graesser. "Is it an Agent, or just a Program": A Taxonomy for Autonomous Agents", Intelligent Agents III, str. 21-35, 1997

[3] Rober M. Gray. Vector Quantization. IEEE ASSP Magazine, 1:4-29, 1984

[4] Simon Haykin. Neural Networks. Macmillan College Publishing Company, New York, 1994.

[5] Pavel Krušina, Roman Neruda, Zuzana Petrová. Soft Computing Agents and Bang2. Technical report No. 819. Institute of Computer Science. Praque, 2000.

[6] John Moody, Christian Darken. Learning with Localized Receptive Fields. Proceedings of the Connectionist Models Summer School. San Mateo: Morgan Kaufmann, 1989, 1988.

[7] Roman Neruda. Functional Equivalence and Genetic Learning of RBF Networks. Praque, 1997. Institute of Computer Science.

[8] Milan Práger. Numerická matematika I. SPN, Praha 1981

[9] Jiří Šíma, Roman Neruda. Teoretické otázky neuronových sítí. Matfyzpress, 1996.

[10] J.H.Wilkinson, C.Reinsch. Handbook for Automatic Computation. Volume II, Linear Algebra. Springer, Berlin, 1971.

[11] Jan Zítko. Úvod do numerické matematiky, 2. svazek. SPN, Praha 1977

[12] On-line dokumentace na `http://bang2.sourceforge.net/`.

Příloha A

Uživatelská dokumentace

V první řadě je nutné zdrůraznit, že tato uživatelská dokumentace není a nemá být jediným zdrojem informací o používání realizovaného softwaru a systému Bang2. Jejím účelem je pouze seznámit čtenáře s nejnutnějšími základy použití systému Bang2 k testování učení RBF sítí. Hlavním cílem je popsat formát konfiguračního souboru.

Návod na instalaci a spuštění systému lze najít na přiloženém CD. Popis grafického prostředí na stránkách jednotlivých agentů v [12, oddíl Man]. Především doporučujeme stránku RBF agenta [12, oddíl Man, stránka rbf] a uživatelskou dokumentaci systému Bang2 [12, oddíl HowTo, User Guide].

Nejprve popíšeme nejjednodušší způsob použití systému Bang2 k učení RBF sítě.

1. Vytvoříme konfigurační soubor, ve kterém specifikujeme požadovanou konfiguraci sítě, jméno souboru s tréninkovou množinou a typ metody učení.

2. Po spuštění systému Bang2 se objeví grafické okno s tlačítky odpovídajícími jménům jednotlivým agentů. Vybereme tlačítko se jménem RBF agenta — *Rafael*.

3. Tím otevřeme *Rafaelovo* grafické okno. Načteme připravenou konfiguraci (tlačítko [LoadConfig]). Implicitně se konfigurace načítá ze souboru `rbf.xml` v adresáři `cfg`. Jiné jméno konfiguračního souboru lze specifikovat v sekci [Files & Saves Config].

4. Načtenou konfiguraci si můžeme prohlédnout a případně upravit (tlačítka [Configure Learning] a [Configure Net]).

5. Nyní jsme připraveni spustit požadovaný výpočet (tlačítko [Learn]). Položky konfigurace, které jsme nespecifikovali, si doplní *Rafael* vlastními hodnotami.

6. Pokud je výpočet iterativního charakteru, otevře se nám grafické okno zobrazující aktuální hodnotu chybové funkce. Takový výpočet ukončíme tlačítkem [End]. Během výpočtu máme možnost upravovat koeficienty učení, pravděpodobnosti genetických operátorů apod.

7. Naučenou síť si můžeme uložit do souboru (tlačítko [Save]). Dále je možné funkci sítě navzorkovat do souboru ([Sample to File]), ohodnotit pomocí sítě vstupní části tréninkové nebo testovací množiny a toto ohodnocení uložit do souboru ([Eval File]), v případě sítě s jedním vstupem a jedním výstupem lze zobrazit graf této sítě ([Gnuplot]).

Nyní se seznámíme se strukturou konfiguračního souboru a významem jednotlivých parametrů. Předpokládáme, že čtenář si přečetl teoretickou část diplomové práce a zná jednotlivé metody.

Konfigurační soubor obsahuje XML element `rbf`. Kterýkoliv podelement nebo jeho atribut obsažený v následujícím výpisu může být vynechán.

```
<rbf>
  <!-- konfigurace RBF sítě -->
  <rbfconfig>
     <neurons input="INT" output="INT" hidden="INT"/>
     <width/>
     <norm type="[none|full|diagonal|
                 diagonal_uniform|full_uniform]"/>
  </rbfconfig>
  <!-- jména používaných souborů -->
  <files>
     <netfile> STRING </netfile>
     <evalfile> STRING </evalfile>
     <samplefile> STRING </samplefile>
     <timelogfile> STRING </timelogfile>
  </files>
  <!-- konfigurace umetody učení -->
  <learning_config>
     <trainset name="STRING"/>
     <regularization lambda="DOUBLE"/>
     [ <three_step> ... </three_step> | <bp> ... </bp>
       | <genetic> .... </genetic> ]
```

```
22:   </learning_config>
23: </rbf>
```

Podelemet `rbfconfig` určuje konfiguraci RBF sítě. Řádek 4 specifikuje počty neuronů v jednotlivých vrstvách. Přítomnost elementu `width` specifikuje možnost adaptace parametru šířka. Řádek 6 a 7 specifikuje použitou normu (`none` je klasická euklidovská norma, `full` respektive `diagonal` použití vážené normy s obecnou respektive diagonální maticí u každé jednotky, `diagonal_uniform` a `full_uniform` znamená použití vážené normy, ale matice je pro všechny jednotky společná).

Podelement `files` (řádek 10) specifikuje jména souborů, pro uložení sítě (ř. 11), pro uložení ohodnocení tréninkové množiny (ř. 12), pro navzorkování sítě (ř. 13) a pro uložení údaje o času výpočtu (ř. 14).

Na řádku 17 začíná specifikace učení. Obsahuje jménou souboru s tréninkovou množinou (ř. 18), hodnotu regularizačního parametru (ř. 19) a typ metody učení (ř. 20).

```
24: <three_step>
25:    <step1>
26:       [ <skip/> | <random_samples/>
27:         | <uniform_samples/> | <vq> ... </vq>
28:         | <genetic>
29:              <genetix> ... </genetix>
30:              <vqgeop cannonical="[yes|no]"/>
31:           </genetic> ]
32:    </step1>
33:    <step2>
34:       [ <skip/>
35:         | <q_neighbours q="INT" coef="DOUBLE"/>
36:         | <gradient overlap="DOUBLE" epsilon="DOUBLE"
37:                     alfa="DOUBLE">
38:              <stopcriterion>
39:                   [ <iteration count="INT"/>
40:                     | <error eps="DOUBLE"/> ]
41:              </stopcriterion>
42:              <inform log="[yes|no]" name="STRING"
43:                      period="INT"/>
44:           </gradient> ]
45:    </step2>
46:    <step3>
47:       [ <skip/>
```

```
48:          | <leastsquares derivate="[yes|no]">
49:             [ <svd/> | <rq/> | <inverse/> ]
50:            </leastsquares>
51:          | <lms> ... </lms> ]
52:     </step3>
53: </three_step>
```

Element `three_step` specifikuje použití třífázového učení. Obsahuje podelementy `step1`, `step2` a `step3` určující jednotlivé kroky. Kterýkoli z těchto tří elementů může obsahovat podelement `skip`, který říká, že daný krok se nemá provádět vůbec.

Pro první krok (ř. 25) můžeme použít rovnoměrné rozmístění středů (`uniform samples`), náhodné vzorky z tréninkové množiny (`random_samples`), metodu vektorové kvantizace (`vq`) nebo genetické učení (`genetic`).

U druhého kroku lze požadovat použití heuristiky *q sousedů* (ř. 35), atribut `q` určuje počet sousedů a atribut `c` koeficient úměrnosti. Druhou možností je použití gradientní minimalizace (ř. 36), můžeme specifikovat parametr překrytí (`overlap`) a parametry gradientní metody (rychlost učení `epsilon`, moment `alfa`). Podelement `stopcriterion` určuje kritérium ukončení výpočtu, lze zadat požadovaný počet iterací nebo hodnotu, pod níž má klesnout chybová funkce; pokud zadáme obě kritéria, výpočet skončí po splnění jednoho z nich. Podelement `inform` určuje periodu, se kterou má bý uživatel informován o průběhu chybové funkce, atribut `log` určuje zda se má tato hodnota zaznamenávat do souboru a atribut `name` určuje jméno tohoto souboru.

Jako třetí krok si vybereme buď `leastsquares` — použití metody nejmenších čtverců (specifikujeme, zda použít SVD rozklad `svd`, Moore-Penroseovu pseudoinverzi `inverse` nebo QR rozklad `rq`), nebo LMS algoritmus — viz str. 95.

```
54: <vq>
55:     <stopcriterion>
56:        [ <iteration count="INT"/>
57:          | <error eps="DOUBLE"/> ]
58:     </stopcriterion>
59:     [ <loyd/>
60:       | <k-means>
61:            <learningrate fi="DOUBLE" delta="DOUBLE"/>
62:            [ <radial alfa="DOUBLE" coef="DOUBLE" iters="INT"/>
63:              | <memory beta="DOUBLE" gamma="DOUBLE"/> ]
64:         </k-means> ]
65:     <inform log="[yes|no]" name="STRING" period="INT"/>
66: </vq>
```

Element `vq` určuje použití vektorové kvantizace. Podelementy `stopcriterion` a `inform` mají stejný význam jako u gradientní metody (viz str. 94). Dále máme výběr mezi použitím Lloydova algoritmu (`loyd`) nebo Kohononova učení (`k-means`). U Kohonenova učení navíc můžeme specifikovat parametr učení (`fi`), upravit klesání tohoto parametru (`delta`), použít jedno z vylepšení — lokální paměť (`memory`) nebo radiální růst (`radial`).

```
67: [ <bp> | <lms> ]
68:    <init> [ <random/> | <heuristic/> | <outside/> ] </init>
69:    <centers   epsilon="DOUBLE" alfa="DOUBLE"/>
70:    <widths    epsilon="DOUBLE" alfa="DOUBLE"/>
71:    <norms     epsilon="DOUBLE" alfa="DOUBLE"/>
72:    <weights   epsilon="DOUBLE" alfa="DOUBLE"/>
73:    <stopcriterion>
74:       [ <iteration count="INT"/>
75:         | <error eps="DOUBLE"/> ]
76:    </stopcriterion>
77:    <gnuplot steps="INT"/>
78:    <save period="INT"><file name="STRING"/></save>
79:    <inform log="[yes|no]" name="STRING" period="INT"/>
80: [ </bp> | </lms> ]
```

Element `bp` nebo `lms` určuje použití gradientního algoritmu. V druhém případě se učí pouze váhy. Řádek 68 určuje způsob inicializace vah, náhodně, jednoduchou heuristikou (středy náhodně z tréninkové množiny, váhy nula, šířky a matice jedna), `outside` říká, že inicializace se má přeskočit, síť bude nastavena mimo agenta realizujícího gradientní výpočet. Řádky 69 – 72 určují rychlost učení a parametr momentu pro jednotlivé parametry (středy, šířky, matice norem, váhy). Samotná přítomnost elementu `centers` říká, že středy jsou adaptovatelné (podobně pro ostatní parametry). Dále je tu `stopcriterion` a `inform`, význam je stejný jako u gradientního algoritmu v druhé fázi (viz str. 94). Řádek 77 říká, že funkce sítě má být průběžně zobrazována pomocí Gnuplotu, atribut `steps` určuje hustotu vykreslení. Toto zobrazování je možné pouze u sítí s jedním vstupem a jedním výstupem. Řádek 78 říká, že mezivýsledek se má zálohovat do souboru a jak často.

```
81: <genetic>
82:    <genetix> ... </genetix>
83:    <rbfgeop cannonical="[yes|no]">
84:       <onlyHidden/>
85:       <difflength max="INT" span="INT"/>
```

```
86:       <load file="STRING"/>
87:       <w_bound low="DOUBLE high="DOUBLE"/>
88:       <d_bound low="DOUBLE high="DOUBLE"/>
89:       <n_bound low="DOUBLE high="DOUBLE"/>
90:    </rbfgeop>
91:    <fitness>
92:       [ <lms> .... </lms>
93:         | <error lambda="DOUBLE"/> ]
94:    </fitness>
95:    <finish> <lms> ... </lms> </finish>
100: </genetic>
```

Element `genetic` specifikuje použití genetického učení. Obsahuje konfiguraci obecné genetiky (ř. 82), konfiguraci genetických operátorů (ř. 83), specifikaci chybové funkce (ř. 91) a případně specifikaci metody, kterou má být vítěz doučen (ř. 95).

Element `rbfgeop` specifikuje genetické operátory učení RBF sítí, atribut `cannonical` říká, zda použít kanonickou nebo klasickou verzi operátorů. Použití elementu `onlyHidden` určuje, že jedinci mají reprezentovat pouze skrytou vrstvu. Element `diffLength` říká, že jedinci můžou být různé délky, atribut `max` pak určuje maximální povolenou délku, atribut `span` rozsah, ve kterém jsou generovány délky jedinců počáteční populace. Řádek 85 říká, že počáteční jedinci nemají být vytvářeni náhodně, ale načítáni ze souboru. Další tři řádky specifikují dolní a horní meze pro váhy (ř. 87), šířky (ř. 88) a matice norem (ř. 89).

Element `fitness` určuje, zda má být hodnota účelové funkce spočtena použitím gradientního algoritmu (pro případ, že učíme pouze skryté vrstvy, `lms`) nebo klasicky průchodem tréninkovou množinou (pro celé sítě, `error`, atribut `lambda` určuje regularizační člen).

```
101:  <genetix popsize="INT" elite="INT"/>
102:     <inform log="[yes|no]" name="STRING" period="INT"/>
103:    <save period="INT"><file name="STRING"/></save>
104:    <rates>
105:       <JMENO_OPERATORU_1 rate="DOUBLE"/>
          ....
106:       <JMENO_OPERATORU_N rate="DOUBLE"/>
107:    </rates>
108:    <stopcriterion>
109:       [ <iteration count="INT"/>
110:         | <error eps="DOUBLE"/> ]
```

```
    </stopcriterion>
</genetix>
```

Element `genetix` je konfigurací agenta realizujícího obecný genetický algoritmus. Atribut `popsize` uručuje velikost populace, `elite` počet jedinců vybíraných do elity. Podelementy `inform` a `stopcriterion` mají stejný význam jako u ostatních iteračních agentů (viz str. 94). Element `save` určuje zálohování populace do souboru.

Element `rates` obsahuje pravděpodobnosti jednotlivých operátorů, v našem případě může obsahovat podelementy `mutation` a `crossover`.