EVOLUČNÍ ALGORITMY A NEURONOVÉ SÍTĚ

URČENO PRO VZDĚLÁVÁNÍ V AKREDITOVANÝCH STUDIJNÍCH PROGRAMECH

EVA VOLNÁ

ČÍSLO OPERAČNÍHO PROGRAMU: CZ.1.07
NÁZEV OPERAČNÍHO PROGRAMU:
VZDĚLÁVÁNÍ PRO KONKURENCESCHOPNOST
OPATŘENÍ: 7.2
ČÍSLO OBLASTI PODPORY: 7.2.2

INOVACE VÝUKY INFORMATICKÝCH PŘEDMĚTŮ VE STUDIJNÍCH PROGRAMECH OSTRAVSKÉ UNIVERZITY

REGISTRAČNÍ ČÍSLO PROJEKTU: CZ.1.07/2.2.00/28.0245

OSTRAVA 2012

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky

Recenzent: Ing. Zuzana Komínková Oplatková, Ph.D.

Název: Evoluční algoritmy a neuronové sítě
Autor: doc. RNDr. PaedDr. Eva Volná, PhD.
Vydání: první, 2012
Počet stran: 152

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

OBSAH

Úvod pro práci s textem pro distanční studium

Cíl předmětu

Seznámit studenty s principy evolučních algoritmů a aplikovat některé evoluční algoritmy zejména v oblasti umělých neuronových sítí. Prohloubit znalosti o umělých neuronových sítích a poskytnout posluchačům představu o neuroevoluci.

Po prostudování textu budete znát:

Tyto učební texty jsou určeny studentům informatiky pro předmět evoluční algoritmy a neuronové sítě. Cílem textu je seznámit studenty se základy evolučních algoritmů, zejména těch, které jsou vhodné pro hledání globálního minima v souvislé oblasti a také seznámit studenty se základními modely umělých neuronových sítích. Závěrečné kapitoly se pak budou věnovat použití evolučních technik při optimalizaci parametrů neuronové sítě.

V textu jsou dodržena následující pravidla:

- je specifikován cíl lekce (tedy co by měl student po jejím absolvování umět, znát, pochopit)
- výklad učiva
- důležité pojmy
- úkoly a otázky k textu
- korespondenční úkoly (mohou být sdruženy po více lekcích)

Úkoly

Vyberte si jeden korespondenční úkol a vypracujte na toto téma semestrální projekt, jehož obhajoba proběhne v dohodnutém termínu. Podrobné informace obdržíte na úvodním tutoriálu.

Pokud máte jakékoliv věcné nebo formální připomínky k textu, kontaktujte autora (eva.volna@osu.cz).

1. Problém globální optimalizace

V této kapitole se dozvíte:

- Jaké heuristiky jsou použitelné při řešení úloh globální optimalizace.
- Jak pracují stochastické algoritmy při řešení problémů globální optimalizace.

Po jejím prostudování byste měli být schopni:

- Formulovat problém globální optimalizace.
- Využít stochastických algoritmů při řešení vybraných úloh globální optimalizace.

Klíčová slova této kapitoly:

Globální optimalizace, stochastický algoritmus, heuristika.

Průvodce studiem

V této kapitole je zformulován problém globální optimalizace a stručně jsou zmíněny základní myšlenky stochastických algoritmů, které se používají pro nalezení globálního minima účelových funkcí. Kapitola byla napsána podle [36].

1.1 Formulace problému globální optimalizace

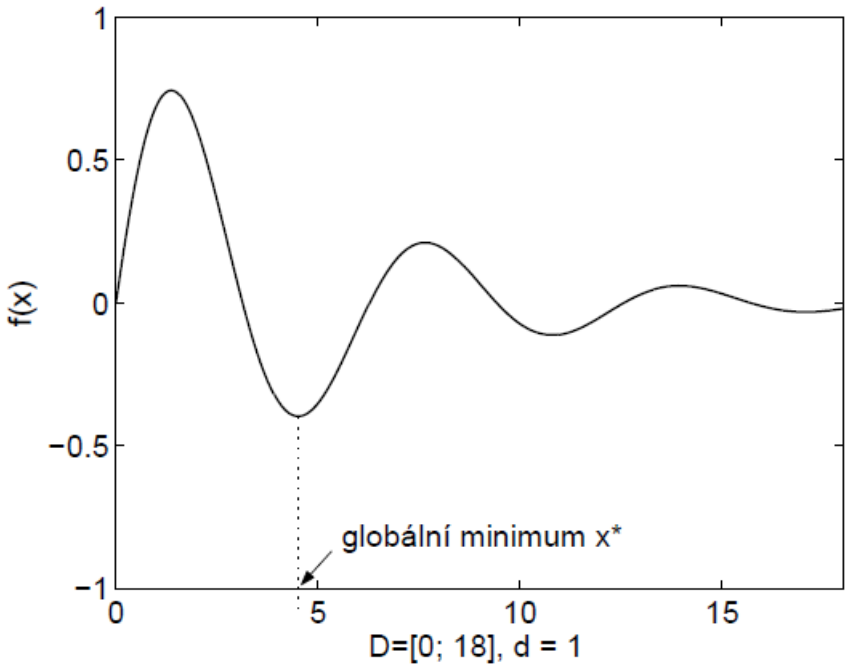
Úlohu nalezení globálního minima můžeme formulovat takto:

Mějme účelovou funkci: $f : D \to R, \quad D \subseteq R^d$.

Máme najít bod $\boldsymbol{x}^* \in D$, pro který platí, že $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})$, pro $\forall\ \boldsymbol{x} \in D$. Nalezení bodu $\boldsymbol{x}^* \in D$ je řešením problému globální optimalizace. Bodu $\boldsymbol{x}^*$ říkáme bod globálního minima (*global minimum point*), definičnímu

oboru *D* se říká doména nebo prohledávaný prostor (*domain, search space*).

Formulace problému globální optimalizace jako nalezení globálního minima není na úkor obecnosti, neboť chceme-li nalézt globální maximum, pak jej nalezneme jako globální minimum funkce $g(x) = -f(x)$. Problém nalezení globálního minima můžeme ilustrovat jednoduchým obrázkem (obr. 1).



Obrázek 1: Nalezení globálního minima

Ze základního kurzu matematické analýzy známe postup, jak nalézt extrémy funkcí, u kterých existuje první a druhá derivace. Zdálo by se, že úloha nalezení globálního minima je velmi jednoduchá. Bohužel tomu tak není. Nalézt obecné řešení takto jednoduše formulovaného problému je obtížné, zvláště když účelová funkce je multimodální (má více lokálních minim), není diferencovatelná, případně má další nepříjemné vlastnosti. Analýza problému globální optimalizace ukazuje, že neexistuje deterministický algoritmus řešící obecnou úlohu globální optimalizace (tj. nalezení dostatečně řešení přesné aproximace $\boldsymbol{x}^*$) v polynomiálním čase, tzn. problém globální optimalizace je NP-obtížný.

Přitom globální optimalizace je úloha, kterou je nutno řešit v mnoha praktických problémech, mnohdy s velmi významným ekonomickým efektem, takže je nutné hledat algoritmy, které jsou pro řešení konkrétních problémů použitelné. Algoritmy pro řešení problému globální optimalizace se podrobně zabývá celá řada monografií, např. Törn a Žilinskas [35], Míka [24], Spall [31] kde je možné najít mnoho užitečných poznatků přesahujících rámec tohoto předmětu.

Pokud hledáme globální minimum v souvislé oblasti D (kartézský součin uzavřených intervalů $\langle a_i, b_i \rangle$), pak globální minimum funkce $f: D \rightarrow R$ je určené vztahem (1):

$$x_{opt} = \arg\min_{x \in D} f(\mathbf{x})$$
$$D = \langle a_1, b_1 \rangle \times \langle a_2, b_2 \rangle \times \cdots \times \langle a_d, b_d \rangle = \prod_{i=1}^{d} \langle a_i, b_i \rangle, \qquad (1)$$
$$a_i < b_i, i = 1,2,\cdots d,$$

a účelovou funkci $f(\boldsymbol{x})$ umíme vyhodnotit s požadovanou přesností v každém bodě $\boldsymbol{x} \in D$. Podmínce (1) se říká *boundary constraints* nebo *box constraints*, protože oblast D je vymezena jako d-rozměrný kvádr. Pro úlohy řešené numericky na počítači nepředstavuje podmínka (1) žádné podstatné omezení, neboť hodnoty a_i; b_i jsou tak omezeny datovými typy užitými pro $\mathbf{x}$ a $f(\mathbf{x})$, tj. většinou reprezentací čísel v pohyblivé čárce.

1.2 Stochastické algoritmy pro globální optimalizaci

Nemožnost nalézt deterministický algoritmus obecně řešící úlohu globální optimalizace vedla k využití stochastických algoritmů, které sice nemohou garantovat nalezení řešení v konečném počtu kroků, ale často pomohou nalézt v přijatelném čase řešení prakticky použitelné.

Stochastické algoritmy pro globální optimalizaci heuristicky prohledávají prostor D. Heuristikou rozumíme postup, ve kterém se využívá náhoda, intuice, analogie a zkušenost. Rozdíl mezi heuristikou a deterministickým algoritmem je v tom, že na rozdíl od deterministického

algoritmu heuristika nezajišťuje nalezení řešení. Heuristiky jsou v praktickém životě zcela samozřejmě užívané postupy, jako příklady můžeme uvést hledání hub, lov ryb na udici, pokus o složení zkoušky ve škole aj.

Většina stochastických algoritmů pro hledání globálního minima v sobě obsahuje zjevně či skrytě proces učení. Inspirace k užití heuristik jsou často odvozeny ze znalostí přírodních nebo sociálních procesů. Např. simulované žíhání je modelem pomalého ochlazování tuhého tělesa, tabu-search modeluje hledání předmětu tak, že v krátkodobé paměti si zapamatovává zakázané kroky vedoucí k již dříve projitým stavům. Popis těchto algoritmů najdete v knize Kvasničky, Pospíchala a Tiňa [21]. Podobné postupy učení najdeme snad ve všech známých stochastických algoritmech s výjimkou slepého náhodného prohledávání.

V posledních desetiletích se s poměrným úspěchem pro hledání globálního minima funkcí užívají stochastické algoritmy zejména evolučního typu. Podrobný popis této problematiky naleznete v knihách Goldberga [11], Michalewicze [25] nebo Bäcka [1, 2] aj.

Kontrolní otázky:

Jak naleznete minimum diferencovatelné funkce analyticky?

Úkoly k zamyšlení:

Kdy jste ve svém životě užili heuristiku? Zkuste ji popsat.

Shrnutí obsahu kapitoly

V této kapitole byl zformulován problém globální optimalizace a představeny základní myšlenky stochastických algoritmů, které se používají pro nalezení globálního minima účelových funkcí.

Pojmy k zapamatování

- globální optimalizace
- stochastický algoritmus
- heuristika

2 Horolezecké algoritmy

V této kapitole se dozvíte:

- Jak pracuje slepý algoritmus.
- Jak pracuje horolezecký algoritmus.
- Jak je zavedeno učení do horolezeckého algoritmu.
- Jak pracuje algoritmus zakázaného prohledávání.

Po jejím prostudování byste měli být schopni:

- Formulovat optimalizační problém pro binární vektor.
- Formulovat operaci mutace v horolezeckém algoritmu.
- Zavést učení do horolezeckého algoritmu.
- Použít zakázaný seznam pro konstrukci okolí.

Klíčová slova této kapitoly:

Slepý algoritmus, horolezecký algoritmus, horolezecký algoritmus s učením, algoritmus zakázaného prohledávání.

Průvodce studiem

V této kapitole uvedeme základní typy stochastických optimalizačních algoritmů, které i když neobsahují evoluční rysy, budou sloužit jako základ pro formulaci evolučních optimalizačních algoritmů. Kapitola je napsaná podle [21].

2.1 Slepý algoritmus

Slepý algoritmus je základní stochastický algoritmus, který opakovaně generuje náhodně řešení z oblasti *D* a zapamatuje si ho jen tehdy, pokud bylo získáno lepší řešení jako to, které už bylo zaznamenáno

v předchozí historii algoritmu. Z důvodů kompatibility tohoto algoritmu s evolučními algoritmy uvedeme jeho implementaci pro binární reprezentaci vektorů [21]. Binární verze funkce (1) pak má tvar: f: $\{0,1\}^k \rightarrow R$. Tato funkce je definována nad množinou binárních vektorů délky k, kdy každému binárnímu vektoru zobrazení f přiřadí reálné číslo z množiny R, tj. $y = f(\boldsymbol{\alpha})$. Analogie optimalizačního problému (1) pro binární vektory má tvar

$$\boldsymbol{\alpha}_{opt} = \arg\min_{\boldsymbol{\alpha}\in\{0,1\}^k} f(\boldsymbol{\alpha}) \tag{2}$$

Globální optimum $\boldsymbol{\alpha_{opt}}$ se nalezne po přezkoušení všech možných binárních vektorů délky k.

Přechod od binárního vektoru $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, ..., \alpha_{kn}) \in \{0,1\}^k$ ke spojitému vektoru $\boldsymbol{x} = (x_1, x_2, ..., x_n) \in D$ může být formálně chápán jako transformace

$$\Gamma : \{0,1\}^{kn} \rightarrow D \qquad \mathbf{x} = \Gamma(\boldsymbol{\alpha}) \tag{3}$$

která zobrazuje množinu binárních vektorů délky kn na body - n-tice reálných čísel z oblasti D. Jinak řečeno, konečná množina (2^{kn}) binárních vektorů délky kn je reprezentovaná pomocí zobrazení Γ body, které mohou být na oblasti D uspořádané do ortogonální mřížky. Minimalizační problém (1) při použití binární reprezentace n-rozměrných vektorů $\boldsymbol{x}$ se tak realizuje na konečné množině diskrétních bodů.

```
procedure Blind_Algoritmus(input: tmax,k,n; output: αfin,ffin);
begin ffin:=∞; t:=0;
      while t<tmax do
      begin t:=t+1;
            α:=randomly generated binary vector of
               the length kn;
            if f(Γ(α))<ffin then
            begin αfin:=α; ffin:=f(Γ(α)) end;
      end;
end;
```

Obrázek 2: Implementace slepého algoritmu

Na obrázku 2 je uvedena Pseudopascalovská implementace slepého algoritmu. Vstupní parametry procedury jsou t_{max} (maximální počet iterací) a konstanty *k* a *n* (délka binárního řetězce jednotlivé proměnné resp. počet proměnných optimalizované funkce *f*). Algoritmus začíná inicializací proměnné f_{fin} (výsledná hodnota nalezeného minima funkce *f*) a *t* (počítadlo iterací). algoritmus se opakuje t_{max}-krát, pak je ukončen a výstupní parametry α_{fin} a f_{fin} obsahují nejlepší hodnoty řešení v binární reprezentaci a příslušnou nejlepší funkční hodnotu.

Jednoduchými úvahami dá se dokázat, že tento jednoduchý stochastický optimalizační algoritmus poskytuje korektní globální minimum optimalizačního problému (1) realizovaného nad ortogonální mřížkou bodů z oblasti *D* za předpokladu, že parametr procedury t_{max} asymptoticky roste do nekonečna

$$\lim_{t_{\max}\to\infty} P\left(t_{\max}\middle|\alpha_{fin}=\alpha_{opt}\right)=1 \qquad (4)$$

kde $P\left(t_{\max}\middle|\alpha_{fin}=\alpha_{opt}\right)$ je pravděpodobnost toho, že slepý algoritmus po t_{max} iteračních krocích poskytne výstupní řešení, které je totožné s přesným řešením (globální minimum).

Obecně můžeme říci, že slepý algoritmus neobsahuje žádnou strategii konstrukce řešení (tj. binárních vektorů délky *kn*) na základě předchozí historie algoritmu. Každé řešení je sestrojené zcela nezávisle (tj. náhodně) od předchozích řešení. Zaznamenává se takové řešení, které v průběhu aktivace procedury poskytuje zatím nejnižší funkční hodnotu. Po ukončení výpočtu je pak zaznamenáno toto řešení.

2.2 Horolezecký algoritmus

Slepý algoritmus může být jednoduše zobecněný na tzv. horolezecký algoritmus (*hill climbing*), kde se hledá nejlepší lokální řešení v určitém okolí a toto řešení je v dalším kroku použité jako "střed" nové oblasti. K formalizaci horolezeckého algoritmu zavedeme některé základní pojmy, které jsou důležité pro jeho jednoduchý popis. Operace *mutace*

transformuje binární vektor α na nový binární vektor α', přičemž stochastičnost toho procesu je určena pravděpodobností P_{mut}

$\alpha' = O_{mut}(\alpha)$,

kde α a α' jsou dva binární vektory stejné délky kn

$\alpha = (\alpha_1, \alpha_2, ..., \alpha_{kn})$ a $\alpha' = (\alpha'_1, \alpha'_2, ..., \alpha'_{kn})$

kde jednotlivé komponenty α'_i, jsou určené takto

$$\alpha_i' = \begin{cases} 1-\alpha_i & \text{pro } random < P_{mut} \\ \alpha_i & \text{jinak} \end{cases} \qquad (5)$$

kde *random* je náhodné číslo z intervalu $\langle 0,1 \rangle$ generované s rovnoměrnou distribucí. Pravděpodobnost P_{mut} určuje stochastičnost operátoru mutace, v limitním případě $P_{mut} \to 0$, pak operátor O_{mut} nemění binární vektor $\lim_{P_{\max} \to 0} O_{mut}(\boldsymbol{\alpha}) = \boldsymbol{\alpha}$.

Základní idea horolezeckého algoritmu spočívá v tom, že vzhledem k určitému zvolenému řešení sestrojíme náhodně předepsaný počet nových řešení tak, že ve zvoleném řešení se náhodně změní bitové proměnné (říkáme, že zvolené řešení je středem oblasti z něj náhodně generovaných řešení). Z této oblasti vybereme nejlepší řešení (tj. s minimální funkční hodnotou nad body z daného okolí), které se použije v následujícím iteračním kroku jako střed nové oblasti. Tento proces se opakuje předepsaný počet-krát, přičemž se zaznamenává nejlepší řešení, které se vyskytlo v průběhu historie algoritmu.

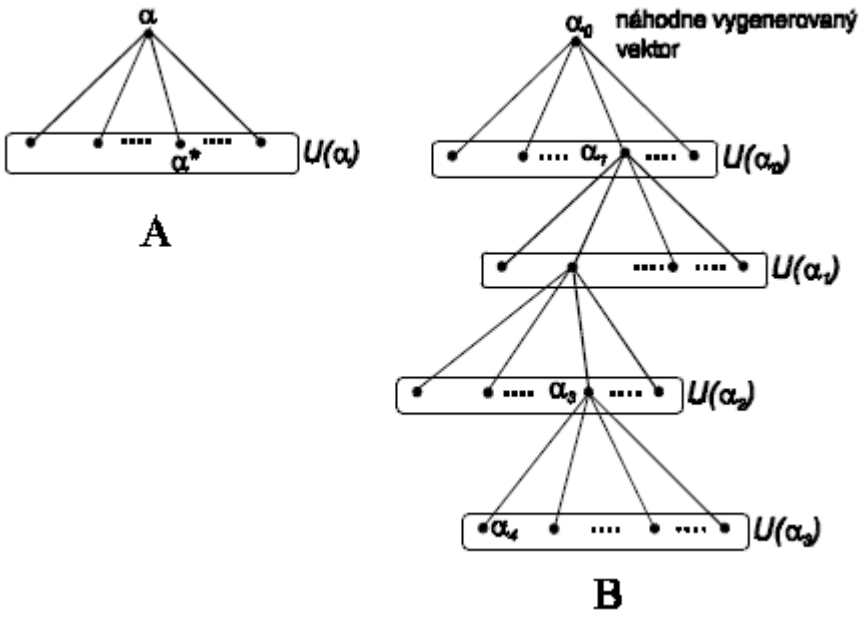
Okolí $U(\alpha)$ binárního vektoru α se sestrojí pomocí vektorů $\alpha' = O_{mut}(\alpha)$

$$U(\alpha) = \{\alpha' = O_{mut}(\alpha)\} \qquad (6)$$

přičemž budeme předpokládat, že kardinalita (počet elementů) se rovná předepsané hodnotě, $|U(\alpha)| = c_0$, kde c_0 je dané kladné celé číslo. Poznamenejme, že v důsledku stochastičnosti aplikace operace mutace na daný binární vektor α má okolí $U(\alpha)$ rovněž stochastický charakter. To, zda nějaký vektor α' patří nebo nepatří do okolí U je určeno jen pravděpodobnostě a ne deterministicky. Nejlepší řešení v okolí $U(\alpha)$ je určeno takto

$$\boldsymbol{\alpha}^* = \arg\min_{\boldsymbol{\alpha}' \in U(\boldsymbol{\alpha})} f\left(\Gamma(\boldsymbol{\alpha}')\right) \qquad (7)$$

V horolezeckém algoritmu se takto získané řešení α^* použije jako "střed" v dalším iteračním kroku algoritmu, viz obr. 3. Implementace horolezeckého algoritmu je uvedena v algoritmu na obr. 4.



Obrázek 3: Schematické znázornění generování okolí binárního vektoru α a nejlepšího řešení α^* v okolí $U(\alpha)$ (diagram A). Počet binárních vektorů v okolí je konstantní, rovná se c_0. Horolezecký algoritmus je znázorněn na diagramu B, tento algoritmus sestává z tvorby posloupností okolí $U(1)$, $U(2)$, $U(3)$, ... Střed okolí $U(i)$ je totožný s nejlepším řešením z předchozího okolí $U(i\text{-}1)$. Algoritmus je inicializován řešením α_0, které je náhodně generované.

```
procedure Hill_Climbing(input:tmax,c0,Pmut; output: ffin,αfin);
begin α:=randomly generated binary vector of the length kn;
      ffin:=∞; t:=0;
      while t<tmax do
      begin t:=t+1;
            α* = arg min_{α'∈U(α)} f(Γ(α))
            if f(Γ(α*))<ffin then begin ffin:=f(Γ(α*)); αfin:=α* end;
            α:=α*;
      end;
end;
```

Obrázek 4: Pseudopascalovská implementace procedury realizující horolezecký algoritmus. Vstupními parametry jsou konstanty t_{max}, c_0 a

P_{mut}, které popisují maximální počet iterací horolezeckého algoritmu, kardinalitu okolí $U(\alpha)$, resp. pravděpodobnost 1-bitové mutace. Algoritmus je inicializován náhodným generováním binárního vektoru α, jehož délka je kn (kde k je délka binární reprezentace reálné proměnné a n je počet proměnných optimalizované funkce f). Binární vektor α^* je nejlepší řešení nalezené v okolí $U(\alpha)$, toto řešení je v následujícím kroku použité jako střed nového okolí. Nejlepší řešení získané v průběhu celé historie je uložené ve výstupních proměnných f_{fin} a α_{fin}.

Analogie (4) ze slepého algoritmu, která říká, že tento jednoduchý algoritmus je asymptoticky schopný najít globální minimum, platí také v horolezeckém algoritmu. V tomto případě se ale kardinalita okolí $c_0 = |U(\alpha)|$ musí asymptoticky zvětšovat do nekonečna

$$\lim_{c_0 \to \infty} P\left(c_0 \middle| \alpha_{fin} = \alpha_{opt}\right) = 1 \qquad (8)$$

Pak je ale zbytečné opakovat iterační kroky horolezeckého algoritmu pro nové lokální optimální řešení, už v rámci jednoho iteračního kroku získáme pro $c_0 \to \propto$ globální řešení.

Jako naznačují jednoduché numerické aplikace, horolezecký algoritmus, i když neobsahuje explicitně evoluční strategii, je poměrně efektivní a robustní stochastický optimalizační algoritmus, který je schopen pro jednoduché úkoly najít globální minimum.

2.3 Horolezecký algoritmus s učením

Horolezecký algoritmus s učením [18, 19] patří mezi jednoduché modifikace standardního horolezeckého algoritmu. Zavedeme dva nové koncepty, které horolezecký algoritmus umožňují modifikovat.

1. *Pravděpodobnostní vektor* $\boldsymbol{w} = (w_1, w_2, \ldots w_{kn}) \in \langle 0,1 \rangle^{kn}$. Jednotlivé jeho komponenty $0 \le w_i \le 1$ určují pravděpodobnosti výskytu proměnné '1 'v dané pozici. Např. pokud $w_i = 0$ (1), pak $\alpha_i = 0$ (1). Pro $0 < w_i < 1$, pak proměnná α_i je náhodně určena vztahem

$$\alpha_i = \begin{cases} 1 & \text{pro } random < w_i \\ 0 & \text{jinak} \end{cases} \qquad (9)$$

kde *random* je náhodné číslo z intervalu ⟨0,1⟩ s rovnoměrnou distribucí. Tento stochastický přístup ke generování bitového vektoru $\boldsymbol{\alpha}$ vyjádříme pomocí funkce $\boldsymbol{\alpha} = R(\mathbf{w})$. Potom, okolí $U(\mathbf{w})$ sestrojené z binárních vektorů náhodně generovaných vzhledem pravděpodobnostním vektoru $\mathbf{w}$ je určeno vztahem:

$$U(\mathbf{w}) = \{\boldsymbol{\alpha} = R(\mathbf{w})\}. \qquad (10)$$

2. *Učení* pravděpodobnostního vektoru $\mathbf{w}$. Nechť $B(\mathbf{w})$ je množina s předepsanou kardinalitou $b = |B(\mathbf{w})|$, která obsahuje b nejlepších řešení z okolí $U(\mathbf{w})$, formálně:

$$B(\mathbf{w}) = \arg\min_{b} \min_{\boldsymbol{\alpha} \in U(\mathbf{w})} f(\Gamma(\boldsymbol{\alpha})). \qquad (11)$$

Pravděpodobnostní vektor je modifikovaný - učený pomocí Hebbova pravidla (známého z teorie neuronových sítí [7, 20]:

$$\mathbf{w} \leftarrow \mathbf{w} + \lambda \sum_{\boldsymbol{\alpha} \in B(\mathbf{w})} (\boldsymbol{\alpha} - \mathbf{w}) \qquad (12)$$

kde λ je koeficient učení (malé kladné číslo) a sumace běží přes b nejlepších řešení z $B(\mathbf{w})$.

```
procedure HCwL(input:time_max, c_0, b, λ; output:α_fin);
begin for i:=1 to n do w_i:=0.5;
      time:=0;
      while time<time_max do
      begin time:=time+1;
            B(w):= arg min_b min_{a∈U(w)} f(Γ(a))
            w:= w+λ Σ_{α∈B(w)} (α−w);
      end;
      α_fin:=best solution of B(w);
end;
```

Obrázek 5: Pseudopascalovská procedura realizující horolezecký algoritmus s učením (HCwL, Hill Climbing with Learning). Algoritmus je inicializován tak, že pravděpodobnosti w_i jsou rovné 1/2. Vnější *while-cyklus* se opakuje $time_{max}$ iterací. V rámci tohoto cyklu se sestrojí množina $B(\mathbf{w})$ pro aktuální pravděpodobnostní vektor $\mathbf{w}$, na základě

znalosti této množiny se adaptuje (učí) pravděpodobnostní vektor. Jako konečné (výstupní) řešení se bere nejlepší řešení z množiny $B(\boldsymbol{w})$ po ukončení iteračního procesu.

Oba tyto nové koncepty (pravděpodobnostní vektor a učení) mohou být jednoduše začleněné do standardního horolezeckého algoritmu. Jmenovitě, místo náhodného generování vektorů okolí pomocí aplikace operátora mutace O_{mut} na fixní binární vektor, nyní jsou vektory okolí generovány pomocí pravděpodobnostního vektoru $\boldsymbol{w}$. Navíc, pravděpodobnostní vektor $\boldsymbol{w}$ je systematicky obnovován pomocí Hebbova učení, které ho posouvá směrem k nejlepším řešením $B(\boldsymbol{w})$ z množiny řešení $U(\boldsymbol{w})$, generované pomocí pravděpodobnostního vektoru $\boldsymbol{w}$, $B(\boldsymbol{w}) \subset U(\boldsymbol{w})$, viz algoritmus na obrázku 5.

Průběh horolezeckého algoritmu s učením a jeho interpretaci podstatně usnadňuje tzv. *parametr nasycení* $\tau(\boldsymbol{w})$ = počet komponent w_i pravděpodobnostního vektoru w, které jsou menší jako w_{eff} nebo jako $(1 - w_{eff})$, kde w_{eff} je malé kladné číslo (např. $w_{eff} = 0.2$). Na začátku algoritmu všechny komponenty pravděpodobnostního vektoru leží v blízkosti 1/2, proto hodnota parametru nasycení je $\tau(\boldsymbol{w}) = 0$. V průběhu historie algoritmu se tento parametr skokově zvyšuje, na závěr historie algoritmu se parametr nasycení rovná počtu komponent pravděpodobnostního vektoru, $\tau(\boldsymbol{w}) = n$. Tato podmínka může být uvažována jako alternativní kritérium pro ukončení horolezeckého algoritmu s učením.

2.4 Algoritmus zakázaného prohledávání (tabu search)

Algoritmus zakázaného prohledávání (*tabu search*) byl navržen koncem 80-tých let Glover [9, 10] jako určité zobecnění horolezeckého algoritmu pro řešení složitých optimalizačních úloh jmenovitě z operačního výzkumu. Základní myšlenka tohoto přístupu je neobyčejně jednoduchá. Vychází z horolezeckého algoritmu, kde se pro dané aktuální řešení generuje pomocí konečnému množiny transformací

některá okolí a funkce se minimalizuje v tomto okolí. Získané lokální řešení se použije jako "střed" nového okolí, ve kterém se lokální optimalizace opakuje; tento proces se opakuje předepsaný počet-krát. V průběhu celé historie algoritmu se zaznamenává nejlepší řešení, které slouží i jako výsledné optimální řešení. Základní nevýhodou tohoto algoritmu je, že se po určitém počtu iteračních kroků vrací k lokálnímu optimálnímu řešení, které se vyskytlo už v jeho předchozím průběhu (problém zacyklení). Glover navrhl jednoduchou heuristiku jak odstranit tento problém. Do horolezeckého algoritmu je zavedena tzv. *krátkodobá paměť*, která si pro určitý krátký interval předcházející historie algoritmu pamatuje inverzní transformace k těm transformacím řešení, které poskytovaly lokální optimální řešení. Tyto inverzní transformace jsou zakázány (tabu) při tvorbě nového okolí pro dané aktuální řešení. Tímto jednoduchým způsobem je možné podstatně omezit výskyt zacyklení. Takto modifikovaný horolezecký algoritmus systematicky prohledává celou oblast řešení, ve které hledáme globální minimum funkce.

Definujme si množinu přípustných transformací

$$S = \{t_1, t_2, ..., t_p\}.$$

Transformace $t \in S$ zobrazuje binární vektor $\alpha \in \{0,1\}^{kn}$ na jiný binární vektor $\alpha' \in \{0,1\}^{kn}$

$$t: \{0,1\}^{kn} \rightarrow \{0,1\}^{kn}$$

pro $\forall t \in S$. Jednoduchá realizace těchto transformací je $t_i(...\alpha_i...) = (...1-\alpha_i...)$ pro $i = 1, 2, ..., p = kn$. Operátor t_i mění v i-té poloze binární hodnotu na její komplement. Obecně jsou transformace z S ohraničené následujícími podmínkami:

1. Nechť $t_1, t_2 \in S$ jsou dvě různé transformace, $t_1 \neq t_2$, pak pro $\forall \alpha \in \{0,1\}^{kn}$ platí $t_1\alpha \neq t_2\alpha$.
2. Pro každou transformaci $t \in S$ existuje taková transformace $t^{-1} \in S$, která je inverzní k t a platí: $t\, t^{-1}\, \alpha = t^{-1}\, t\, \alpha = \alpha$, pro $\forall \alpha \in \{0,1\}^{kn}$.
3. Pro každou dvojici $\alpha_1, \alpha_2 \in \{0,1\}^{kn}$ různých binárních vektorů, $\alpha_1 \neq \alpha_2$, existuje taková posloupnost transformací $t_{i_1}, t_{i_2}, \cdots, t_{i_n} \in S$, že

"výchozí" vektor α_1 je postupně převedený na "konečný" vektor α_2

$$\boldsymbol{\alpha}_1 = \boldsymbol{\beta}'_1 \xrightarrow{t_{i_1}} \boldsymbol{\beta}'_2 \xrightarrow{t_{i_2}} \cdots \xrightarrow{t_{i_n}} \boldsymbol{\alpha}_2 = \boldsymbol{\beta}'_n$$

Okolí $U()$ obsahuje obrazy α vytvořené transformacemi $t \in S$

$$U(\alpha) = \{t\ \alpha;\ \forall\, t \in S\} \qquad (13)$$

Původní horolezecký algoritmus bude nyní modifikován tak, že místo okolí (6) generovaného náhodně pomocí stochastického operátora mutace, použijeme deterministicky definováné okolí (13), generované pomocí přípustných transformací z množiny S. Hlavní omezení této jednoduché modifikace horolezeckého algoritmu je, že po určitém počtu iteračních kroků se výsledná řešení začnou cyklicky opakovat. Po konečnému počtu kroků se tento algoritmus vrátí k řešení, které se už vyskytovalo jako lokální řešení v předchozím iteračním kroku, přičemž nejlepší zaznamenané řešení je obvykle vzdálené od globálního řešení.

Algoritmus zakázaného prohledávání [9, 10] využívá jednoduchou heuristiku, jak pokračovat v hledání globálního minima bez možnosti návratu do lokálního minima, které už bylo zaznamenáno v předchozí historii algoritmu. Hlavní myšlenka této heuristiky je *zakázaný seznam T* (*tabu list*), mající vlastnost krátkodobé paměti, který dočasně obsahuje inverzní transformace k použitým transformacím v předchozích iteracích. Zakázaný seznam transformací $T \subseteq S$, maximální kardinality s, $0 \leq |T| \leq s$, je sestrojený a systematicky obnovovaný v průběhu celého algoritmu. Pokud transformace t patří pro danou iteraci do zakázaného seznamu, $t \in T$, potom se nemůže používat v lokální minimalizaci v rámci okolí aktuálního řešení α. Při inicializaci algoritmu je zakázáný seznam prázdný, po každé iteraci se do zakázaného seznamu dodá transformace, která poskytla lokálně optimální řešení vyrobené z řešení z předchozí iterace. Po s iteracích zakázáný seznam už obsahuje s transformací; ze zakázaného seznamu se vyloučí transformace, která tam byla dodána před s iteracemi. To znamená, že po naplnění zakázaného seznamu (tj. po s iteracích), každé dodání nové transformace je současně doprovázené vyloučením "nejstarší"

transformace (dodané právě před *s* iteracemi), říkáme, že zakázaný seznam se cyklicky obnovuje

$$T : \begin{cases} T \cup \{t^{*-1}\} & \text{pro } |T| < s \\ T \cup \{t^{*-1}\} \setminus \hat{t} & \text{pro } |T| = s \end{cases} \qquad (14)$$

kde t^* je transformace, která vytváří lokální minimum řešení, $\alpha^* = t^*\alpha$ a $\hat{t}$ je "nejstarší" transformace zavedena do zakázaného seznamu právě před *s* iteracemi.

Numerické zkušenosti s algoritmem zakázaného prohledávání ukazují, že velikost *s* zakázaného seznamu je velmi důležitým parametrem pro prohledávání oblasti $\{0,1\}^{kn}$ s možností vymanit se z lokálních minim. Pokud je parametr *s* malý, pak se může vyskytovat zacyklení algoritmu, podobně jako při klasickém horolezeckém algoritmu s okolím sestrojeným podle (6). Zacyklení se sice neopakuje v sousedních dvou krocích, ale řešení se může opakovat po několika krocích. V případě, že parametr *s* má velkou hodnotu, pak při prohledávání oblasti $\{0,1\}^{kn}$ s velkou pravděpodobností "přeskočíme" hluboká údolí minimalizované funkce $f(\alpha)$, tj. vynecháme nalezená lokální minima, které mohou být globálními minimy.

Zakázaný seznam *T* se používá pro konstrukci modifikovaného okolí $U_T(\alpha)$.

$$U_T(\alpha) = \{\alpha' ; \forall t \in S \setminus T : \alpha' = t\alpha\} \qquad (15)$$

jehož kardinalita je $p - s \leq |U_T(\alpha)| \leq p$, přičemž $U_T(\alpha) = U(\alpha)$, pro $T = \emptyset$. Toto okolí obsahuje vektory $\alpha' \in \{0,1\}^{kn}$, které jsou vytvořeny použitím transformací z množiny *S* nepatřících do zakázaného seznamu *T*. Lokální minimalizace se vykonává v modifikovaném okolí $U_T(\alpha)$ s výjimkou tzv. *aspiračního kritéria*. Toto kritérium porušuje restrikci zakázaného seznamu tehdy, pokud existuje taková transformace $t \in S$, že vektor $\alpha' = t\alpha$ poskytuje nižší funkční hodnotu, jako dočasně nejlepší řešení. Implementace algoritmu zakázaného prohledávání je prezentován v algoritmu na obrázku 6.

```
procedure Tabu_Search(input:time_max,s ; output: f_fin,α_fin);
begin α:=randomly generated binary vector of the length kn;
      f_fin:=∞; time:=0; T:=∅;
      while t<time_max do
      begin time:=time+1; f_fin-loc:=∞;
            for t∈S do
            begin α':=tα;
                  if (t∉T and f(Γ(α'))<f_fin-loc) or
                     (f(Γ(α')<f_fin) then
                  begin α*:=α'; t*:=t; f_fin-loc:=f(Γ(α')) end;
            end;
            if f_fin-loc<f_fin then
            begin f_fin:=f_fin-loc; α_fin:=α* end;
            α:=α*;
            if |T|<s then T:=T∪{t*^-1} else T:=(T∪{t*^-1})\{t̂};
      end;
end;
```

Obrázek 6: Pseudopascalovská procedura pro algoritmus zakázaného prohledávání. Vstupními parametry jsou $time_{max}$ a *s*, které určují maximální počet iteračních kroků resp. velikost zakázaného seznamu *T*. Procedura obsahuje dva cykly: vnější *while-cyklus* realizuje iterační kroky algoritmu zakázaného prohledávání, zatímco vnitřní *for cyklus* slouží pro konstrukci okolí $U(\alpha)$. Poznamenejme, že toto okolí není explicitně sestrojené, generují se jen jeho elementy a ty se hned testují, zda jejich funkční hodnota není menší než lokální nejlepší funkční hodnota. Vnější cyklus je ukončen operací obnovy zakázaného seznamu.

Jak už bylo řečeno, algoritmus zakázaného prohledávání je velmi podobný horolezeckému algoritmu. V algoritmu zakázaného prohledávání není okolí řešení sestrojené stochastický způsobem, jako v horolezeckém algoritmu pomocí operátora mutace O_{mut}, ale systematickým deterministický způsobem pomocí přípustných transformací z množiny *S*. Takto realizovaný horolezecký algoritmus má jedno vážné ohraničení, a to cyklický výskyt řešení po určitém počtu iteračních kroků. Základní myšlenka algoritmu zakázaného prohledávání na odstranění tohoto problému zacyklení je zavedení tzv.. zakázaného seznamu, který obsahuje určitý počet inverzních

transformací k těm transformacím, které byly použité v předchozí krátké historii algoritmu. Tento seznam se cyklicky obnovuje tak, že při zavedení inverzní transformace z aktuálního iteračného kroku se z něj odstraní "nejstarší" inverzní transformace. Tento jednoduchý algoritmický trik odstraňuje spolehlivě problém zacyklení horolezeckého algoritmu s deterministický generováním okolí pomocí množiny přípustných transformací.

Kontrolní otázky:

Srovnejte stochastické optimalizační algoritmy uvedené v této kapitole. Na jaké typy problémů byste jednotlivé algoritmy aplikovali?

Úkoly k zamyšlení:

Pokuste se nalézt souvislost mezi stochastickými optimalizačními algoritmy uvedenými v této kapitole a evolučními algoritmy.

Korespondenční úkol:

Vytvořte počítačový program pro realizaci vybraného stochastického optimalizačního algoritmu z této kapitoly.

Shrnutí obsahu kapitoly

V této kapitole byly uvedeny základní typy stochastických optimalizačních algoritmů, tj. slepé prohledávání, horolezecký algoritmus, horolezecký algoritmus s učením a algoritmus zakázaného prohledávání. Tyto algoritmy, i když neobsahují evoluční rysy, představují východislo pro formulaci evolučních optimalizačních algoritmů.

Pojmy k zapamatování

- okolí
- operátor mutace
- parametr nasycení
- pravděpodobnostní vektor
- zakázaný seznam

3 Evoluční algoritmy

V této kapitole se dozvíte:

- Jaké jsou vlastnostmi evolučních algoritmů.
- Jaká je obecná konstrukce evolučních algoritmů.
- Jak pracují genetické algoritmy.

Po jejím prostudování byste měli být schopni:

- Vysvětlit základní pojmy evolučních algoritmů.
- Vysvětlit obecnou konstrukci evolučních algoritmů.
- Používat základní evoluční operátory: selekce, křížení a mutace.

Klíčová slova této kapitoly:

Evoluční algoritmy, genetické algoritmy, genotyp, fenotyp, fitness funkce, evoluční operátory.

Průvodce studiem

Evoluční algoritmy jsou zastřešujícím termínem pro různé přístupy využívající modely evolučních procesů pro účely nemající téměř nic společného s biologií. Snaží se využívat představy o hnacích silách evoluce živé hmoty pro účely optimalizace. Evoluční algoritmy se především používají pro řešení velkých komplexních optimalizačních problémů s mnoha lokálními optimy, protože je zde menší pravděpodobnost, že uváznou v lokálním minimu než u tradičních gradientních metod. Evoluční algoritmy jsou mnohem robustnější než jiné prohledávací algoritmy. Kapitola byla napsána podle [21, 29, 40].

Evoluční algoritmy jsou ve své podstatě jednoduchými modely Darwinovy evoluční teorie vývoje populací. Charakteristické pro ně je

to, že pracují s populací a využívají heuristiky, které určitým způsobem modifikují populaci tak, aby se její vlastnosti zlepšovaly. O některých třídách evolučních algoritmů je dokázáno, že nejlepší jedinci populace se skutečně přibližují k nalezení globálního optima.

Evoluční algoritmy byly a jsou předmětem intenzivního výzkumu a počet publikací z této oblasti je velmi velký. Jedním z hlavních motivů jsou především aplikace v praktických problémech, které jinými metodami nejsou řešitelné. Rozvoj evolučních algoritmů je záležitostí hlavně posledních desetiletí a je podmíněn rozvojem počítačů a pokroky v informatice. Rozsáhlý přehled aplikací evolučních algoritmů je uveden např. v [21, 29].

3.1 Základní pojmy evolučních algoritmů

Evoluční algoritmy přísluší do třídy stochastických prohledávacích algoritmů a pracují s náhodnými změnami navrhovaných řešení. Pokud jsou nová řešení výhodnější, nahrazují řešení předchozí. Důležitým rysem evolučních algoritmů je jejich prohledávací strategie založena na populacích. Populace *genotypů* patří do mnohodimenzionálního prostoru. Genotyp obsahuje *geny* a kóduje obvykle jeden (ale může jich být i více) *fenotyp,* nebo-li kandidáta na řešení úlohy z příslušné domény řešení, např. architekturu umělé neuronové sítě. Během kódování nabývají geny numerických hodnot z odpovídajících domén hodnot tak, aby genotyp mohl být v *dekódovacím* procesu transformován na příslušný *fenotyp*. Celý dekódovací proces by měl být jednoduše proveditelný. Ohodnocení fenotypu determinuje *fitness* korespondujícího genotypu. Evoluční algoritmy preferují genotypy s nejvyšším fitness ohodnocením, jež jsou vytvářeny operátory (např. *mutace*, *křížení*, *inverze* aj.) v průběhu mnoha *generací*. Jedinci v populaci soutěží a navzájem si vyměňují informace tak, že se populace postupně *vyvíjí* směrem ke genotypu, jež koresponduje s největší fitness fenotypu, což odpovídá řešení úlohy. Obecná konstrukce evolučních algoritmů je následující:

1. Generování počáteční populace *G(0)* a nastavení *i=0*;
2. REPEAT
(a) ohodnocení každého jedince v populaci;
(b) výběr rodičů z *G(i)* založený na jejich fitness v *G(i);*
(c) aplikace operátorů na vybrané rodiče a vytvoření potomků, které formují *G(i+1);*.
(d) *i = i + 1;*
3. UNTIL není splněna podmínka ukončení.

Evoluční algoritmy se především používají pro řešení velkých komplexních problémů s mnoha lokálními optimy, protože je zde menší pravděpodobnost, že uváznou v lokálním minimu než u tradičních gradientních metod. Řešení evolučními algoritmy nezávisí na gradientních informacích, a tak jsou především vhodné pro problémy, kde jsou takové informace nedostupné, drahé nebo odhadnuté. Mohou řešit i problémy, které nejsou přímo vyjádřeny přesnou účelovou funkcí. V obecné definici evolučních stochastických optimalizačních algoritmů je jejich síla a někdy i jejich slabost. Pokud však dobře známe typ problému, který máme řešit, dokážeme obvykle také pro jeho řešení vyvinout speciální algoritmy založené na vnitřní znalosti daného problému. Tyto algoritmy jsou téměř vždy efektivnější než naslepo zvolené evoluční algoritmy. Vzhledem k nevyhnutelnému velkému množství ohodnocení fitness funkcí se evolučními algoritmy dají řešit především takové problémy, kdy lze hodnotu funkce vyčíslit dostatečně rychle. Na druhé straně jsou evoluční algoritmy robustní, proto je můžeme použít k optimalizaci jakékoliv neznámé funkce a máme velkou šanci dostat poměrně rozumný výsledek. Evoluční stochastické optimalizační algoritmy se používají pro optimalizaci mnoha-parametrových funkcí s mnoha extrémy, anebo s neznámým gradientem. Použití standardních gradientních metod (např. metoda nejprudšího spádu, sdružených gradientů apod.) i negradientních metod (např. Simplexová metoda) není v těchto případech vhodné, neboť tyto metody často konvergují pouze k extrému nacházejícímu se

blízko počátečního bodu výpočtu a tento extrém již nejsou schopny opustit. Tento nedostatek se obvykle odstraňuje tak, že se opakovaně náhodně zvolí počáteční řešení úlohy a za výsledné inicializační řešení se pak vezme nejlepší z nich. Stochastičnost tohoto postupu spočívá pouze v randomizaci počátečního řešení, následně použitý optimalizační algoritmus je již striktně deterministický. Naproti tomu si stochastické optimalizační algoritmy zachovávají svou „stochastičnost“ v celém průběhu optimalizačního procesu a ne jen při výběru počátečního řešení. Pro tyto algoritmy byly dokázané existenční teorémy, které za jistých předpokladů zabezpečují jejich schopnost nalézt globální extrém, avšak v nekonečném čase [11]. Programování implementace těchto metod je poměrně jednoduchá. Jednou z hlavních podmínek jejich úspěšného použití je vhodná reprezentace proměnných pomocí řetězce znaků (např. bitového řetězce obsahujícího symboly *0* a *1*) a rychlost výpočtu hodnot fitness funkce v daném bodě. Zvlášť tato poslední podmínka limituje úspěšné použití stochastických optimalizačních metod pro optimalizaci složitých problémů, jednoduchost je tak „kompenzována“ náročností na čas výpočtu. Významnou roli má v definici evolučních algoritmů náhodnost výběru chromozómů. Kdybychom do reprodukčního procesu vybírali pouze chromozómy s největší silou (fitness), potom bychom zřejmě podstatně ohraničili doménu, na které hledáme optimální výsledek. Chromozóm s menší silou stále ještě může obsahovat důležitou informaci využitelnou v další evoluci populace. Všechny „částečně urychlující“ heuristiky jsou zde nejen nedostatečné, ale v konečném důsledku i zavádějící, proto se je ani nepokoušíme do evolučních algoritmů zabudovat.

Evoluční algoritmy jsou používané jak pro optimalizaci diskrétních, tak i pro optimalizaci reálných proměnných. Je zřejmé, že fungují pomaleji než jakékoliv jiné heuristické přístupy a pokud nemáme předem zadané podmínky pro globální extrém, nikdy nevíme, zda jsme jej již dosáhli a zda máme optimalizaci zastavit. Mají však i podstatné výhody: A) jsou velmi obecně definované, a proto aplikovatelné téměř na jakýkoliv

problém; B) dokáží se dostat z lokálního extrému. Evoluční proces prohledávání prostoru potenciálních řešení je pak rovnováhou dvou cílů: A) co nejrychleji nalézt nejbližší (většinou lokální) optimum v malém okolí výchozího bodu; B) co nejlépe prohledat prostor všech možných řešení. Jednotlivé evoluční metody se liší právě svým zaměřením k těmto dvěma cílům. Zatím neexistuje jednoznačně přijímaný přístup, jak analyzovat řešený problém tak, abychom určili, který optimalizační algoritmus k jeho řešení použít. V současné době se jedná při používání jednotlivých druhů evolučních algoritmů spíše o zvyk než o důkladnou analýzu problému před volbou algoritmu pro jeho řešení. Evoluční algoritmy mají také velké množství parametrů pro nastavení a mnoho modifikací. Zatím se pro výběr algoritmu obvykle používá metoda pokusu a omylu, stejně tak i pro nastavení jeho parametrů. Existují sice různé „metapřístupy“ při kterých se optimalizuje výběr použité metody včetně nastavení jejích parametrů, ale tyto přístupy jsou velmi výpočetně náročné. Pokud se skutečně nejedná o problém, který se chystáme řešit opakovaně na podobných typech funkcí, je tento přístup neefektivní. V poslední době se stále častěji používají hybridní metody, které přebírají a kombinují několik základních přístupů do jednoho.

Evolučním algoritmům často trvá dlouho přechod od téměř optimálních vnitřních parametrů k optimálním. Tento nedostatek se obvykle nahrazuje spojením evolučního algoritmu s některou z klasických gradientních metod – evoluční algoritmus nalezne přibližné hodnoty optima a gradientní metoda dokončí optimalizaci do globálního optima.

3.2 Genetické algoritmy

Genetické algoritmy byly odvozeny na základě biologické genetiky a teorie evoluce, která ovlivňuje vývoj všeho živého na této planetě. Při vývoji jednotlivých druhů mají geny velký význam. Základem je DNA – deoxyribonukleová kyselina, ve které je zakódován kompletní popis daného jedince. DNA je dlouhý molekulární řetězec tvořený čtyřmi

odlišnými složkami. Uspořádání těchto složek reprezentuje genetický kód. V problematice genetických algoritmů se setkáváme s následujícími pojmy: *Chromozóm* je část DNA, která je stočená do záhybů. *Gen* jsou jednotlivé části chromozómu. Kompletní genetický popis organismu je tzv. *genotyp*. V souvislosti s genotypem se ještě uvádí i tzv. *fenotyp*, který je v podstatě fyzickým popisem genotypu (např. jestliže je v binárním pojetí genotyp „*0101*“, pak fenotyp je jeho dekadická hodnota „*5*“). Geny mohou nabývat pouze jistých hodnot, jejichž obecné označení je *alela*.

Genetické algoritmy pracují v počítačových aplikacích následujícím způsobem [44]: Náhodně je vygenerována množina chromozómů, ze které jsou vybírány dvojice (rodiče) na základě toho, jak dobře splňují kriteriální funkci. Potomci jsou tvořeni křížením svých rodičů, jež následně nahrazují. Stejně jako v přírodě, tak i v technických aplikacích hrají důležitou roli náhodné změny – *mutace*. To se v technice obvykle řeší pomocí generátoru náhodných čísel. Při používání genetických algoritmů v technických aplikacích se používají ještě další pojmy: *Objektivní funkce* – je to funkce, kterou chceme minimalizovat (v případě neuronových sítí je to globální chyba sítě). *Vhodnost* (*fitness*) je číslo, které nám udává vhodnost nového potomka z hlediska kriteriální funkce (je to v podstatě matematický popis životního prostředí daného jedince), obvykle to bývá převrácená hodnota objektivní funkce. Čím je toto číslo větší, tím je i daný jedinec vhodnější pro dané okolní podmínky. *Schéma* je množina genů v chromozómu, které mají jisté specifické hodnoty. V neuronových sítích se takováto skupina chápe jako skupina genů, která je schopna za určitých podmínek vytvořit žádaný efekt. Obsah genů je variabilní. Nejběžnější je obsah v binární podobě, avšak lze použít i dekadický, symbolický i jiný popis.

Vlastní algoritmus genetické optimalizace je cyklus, ve kterém jsou vytvářeni noví potomci (jedinci), tvořící další generaci rodičů. Po každém cyklu testujeme, jestli je splněna podmínka ukončení. Cyklus

se zde nazývá *generace*. Vlastní schéma genetického algoritmu je následující [42]:

1. Navržení genetické struktury.

2. Inicializace.

3. REPEAT

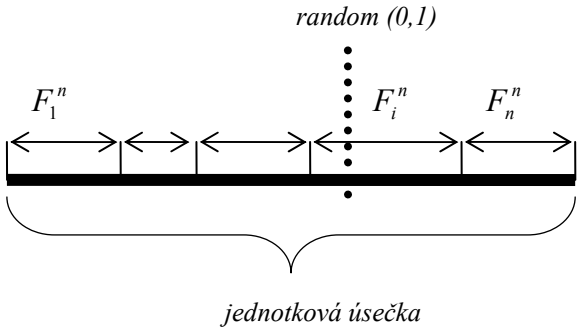
- **(a) Ohodnocení každého jedince v populaci.**
- **(b) Konverze genotypu na fenotyp.**
- **(c) Ohodnocení objektivní funkce.**
- **(d) Konverze objektivní funkce na vhodnost (fitness).**
- **(e) Konverze vhodnosti na selekci rodičů.**
- **(f) Výběr rodičů.**
- **(g) Aplikace operátoru *křížení* na vybrané rodiče a vytvoření potomků, tvořících další generaci rodičů.**
- **(h) Aplikace operátoru *mutace*.**

3. UNTIL není splněna podmínka ukončení.

Z praktických důvodů je vhodné, aby numerické hodnoty fitness byly z otevřeného intervalu (0,1), proto se zavádí tzv. normalizovaná fitness. Normalizovaná fitness F_i^n má pro *i.* jedince z populace obsahující celkem *n* jedinců tvar:

$$F_i^n = \frac{F_i}{\sum_{i=1}^{n} F_i}, \qquad (16)$$

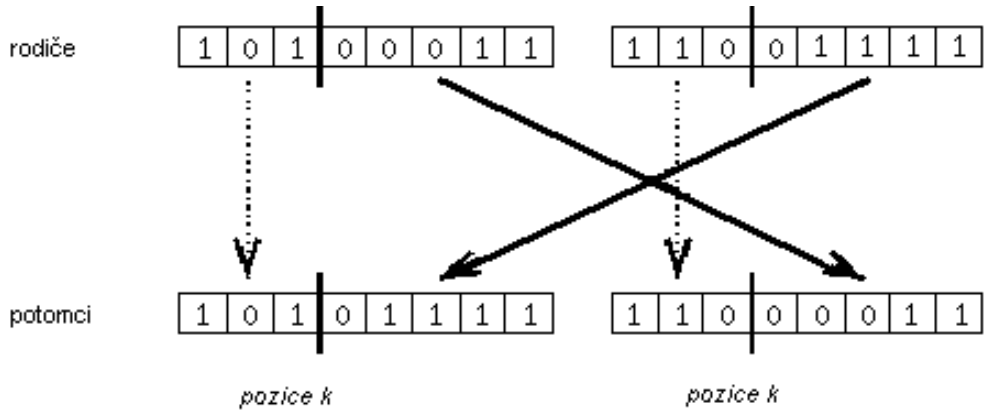
kde F_i je vypočítaná fitness *i.* jedince.



Obrázek 7: Výběr rodičů.

Výběr rodičů pak probíhá následovně [21], viz obrázek 7. Jednotková úsečka je rozdělena na úseky podle velikosti normalizovaných fitness hodnot jedinců z populace. Náhodně vygenerované číslo z intervalu (0,1) představuje polohu na úsečce (je reprezentované hrubou přerušovanou čarou) a podle této polohy určuje chromozóm. Z konstrukce této „rulety“ vyplývá: čím větší je fitness chromozómu, tím větší je i pravděpodobnost jeho výběru.

Aplikace operátoru *křížení* probíhá u dvou vybraných rodičů ve dvou krocích, viz obrázek 8. Nejprve se náhodně stanoví pozice v chromozómu (např. *k*) a potom se vytvoří z původních dvou chromozómů noví dva jedinci tak, že první potomek je tvořen geny na pozici 1 až *k* prvního z rodičů a geny *k+1* až *l* (*l* je délka chromozómu) druhého rodiče. Druhý potomek má pořadí stanoveno opačně, tedy pozice 1 až *k* jsou získané od druhého rodiče, zatímco geny *k+1* až *l* patří prvému z rodičů. Vzniknou tak dva zcela nové chromozomy.



Obrázek 8: Křížení chromozomů

Mutace jsou nezbytnou součástí genetických algoritmů. Díky jim lze mnohdy najít jedince, kteří lépe vyhovují okolním podmínkám a tak zkvalitnit jak genetický proces, tak i množinu jedinců – budoucích rodičů. Při mutaci se prochází jednotlivé geny chromozómu a s určitou velmi malou pravděpodobností se mění jejich hodnota, např v binárním chromozómu se hodnota *1* změní na *0* a opačně.

Kontrolní otázky:

1. Jaké jsou oblasti aplikovatelnosti evolučních algoritmů.
2. Jaká je obecná konstrukce evolučních algoritmů?
3. Jak pracují genetické algoritmy?

Úkoly k zamyšlení:

Zamyslete se, jaké další metody selekce byste mohli v evolučních algoritmech použít.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základními principy evolučních algoritmů. Evoluční algoritmy se především používají pro řešení velkých komplexních optimalizačních problémů s mnoha lokálními optimy, protože je zde menší pravděpodobnost, že uváznou v lokálním minimu než u tradičních gradientních metod. Evoluční algoritmy jsou ve své podstatě jednoduchými modely Darwinovy evoluční teorie vývoje populací. Charakteristické pro ně je to, že pracují s populací a využívají heuristiky, které určitým způsobem modifikují populaci tak, aby se její vlastnosti zlepšovaly.

Pojmy k zapamatování

- populace,
- fenotyp,
- genotyp,
- chromozóm,
- objektivní funkce
- fitness funkce,
- operátory: selekce, křížení, mutace.

4 Optimalizace na bázi částicových hejn

V této kapitole se dozvíte:

- Jak pracuje PSO.
- Jaké má algoritmus SOMA vlastnosti a řídící parametry.
- Jak jsou definovány operátory algoritmu SOMA;

Po jejím prostudování byste měli být schopni:

- Formulovat pojem kolektivní inteligence.
- Vysvětlit základní principy PSO.
- Vysvětlit základní principy SOMA.

Klíčová slova této kapitoly:

Kolektivní inteligence, PSO (Particle Swarm Optimization), SOMA (Self-Organizing Migrating Algorithm)

Průvodce studiem

Algoritmus PSO (algoritmus hejna částic) je evoluční výpočetní technika inspirovaná sociálním chováním ptačích a rybích hejn. PSO využívá populace částic, které prolétají prohledávaným prostorem problému určitou rychlostí určenou individuálně pro každou částici. Pohyb každé částice přirozeně směřuje k řešení blízkému optimu.

Algoritmus SOMA (samo-organizující se migrační algoritmus) byl vytvořený prof. Ivanem Zelinkou [43, 44, 45, 47] z UTB ve Zlíně. Potomci se zde nevytvářejí křížením jako u jiných evolučních algoritmů, ale filozofie algoritmu je založena na pohybu inteligentních jedinců, tj. migraci jedinců. Nejedná se zde o vývoj nových jedinců, ale spíše o přesun původních jedinců na nové pozice.

Kolektivní inteligence je umělá inteligence založená na kolektivním chování decentralizovaných, samoorganizovaných systémů. Takovéto systémy se skládají z populace jedinců, kteří interagují lokálně mezi sebou a se svým okolím. Jedinci se řídí jednoduchými pravidly. Přestože neexistuje centralizovaná struktura řízení, která by určovala, jak se jedinci mají chovat, lokální interakce mezi těmito agenty vedou ke vzniku složitého globálního chování. Inteligenci hejna v přírodě využívají například mravenčí kolonie, hejna ptáků, zvířata ve stádech, bakteriální růst a hejna ryb. Využití této inteligence se nalézá i v oboru robotiky. Většina hejn ji zejména využívá při hledání místa bohatého na potravu.

4.1 PSO

Algoritmus hejna částic (PSO - Particle Swarm Optimization) je evoluční výpočetní technika vyvinuta Eberhartem a Kennedym v roce 1995, která je inspirovaná sociálním chováním ptačích a rybích hejn. Tato metoda má své kořeny jak v umělé inteligenci či sociální psychologii, tak i v počítačových vědách a inženýrství. PSO využívá populace částic, které prolétají prohledávaným prostorem problému určitou rychlostí. V každém kroku algoritmu je tato rychlost pro každou částici určena individuálně, a to podle nejlepší pozice částice a nejlepší pozice částic v jejím okolí v dosavadním průběhu algoritmu. Nejlepší pozice částic se určí za pomocí uživatelem definované fitness funkce. Pohyb každé částice přirozeně směřuje k optimálnímu řešení nebo k řešení blízkému optimu. V angličtině používaný výraz „*swarm intelligence*“ (inteligence roje) vychází z nepravidelného pohybu částic v prohledávaném prostoru, připomínající spíše pohyb komárů než ryb nebo ptáků.

PSO je na inteligenci založená výpočetní technika, která není příliš ovlivňována velikostí nebo nelinearitou problému a dokáže konvergovat k optimálnímu řešení i tam, kde většina analytických metod selhává. Navíc má PSO výhody i oproti jiným podobným optimalizačním

technikám, např. genetickým algoritmům. Je jednodušší na implementaci a nastavuje se u něj méně parametrů simulace. Každá částice si pamatuje svoji předchozí nejlepší hodnotu a nejlepší hodnotu svých sousedů, proto má efektivnější práci s pamětí než genetický algoritmus. Také účinněji udržuje rozmanitost v populaci, protože částice využívají informace od nejlepší částice ke svému zlepšení, kdežto u genetického algoritmu nejslabší řešení zanikají a pouze ty nejlepší zůstávají do další iterace. To vede k populaci jedinců, kteří jsou velmi podobní tomu nejlepšímu [37].

4.1.1 **Základní princip PSO**

Každé možné řešení daného problému lze reprezentovat jako částici, která se pohybuje prohledávaným prostorem. Pozice každé částice je dána vektorem x_i a její pohyb je dán rychlostí v_i.

$$x_i(t) = x_i(t - 1) + v_i(t) \tag{17}$$

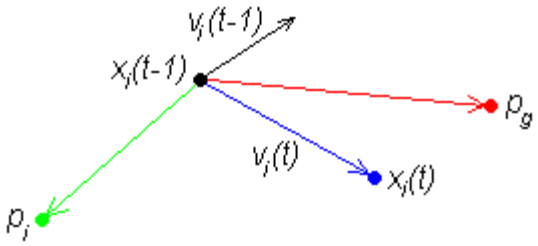
Informace dostupná každé částici je založena na její vlastní zkušenosti a znalosti chování ostatních částic v jejím okolí. Protože důležitost těchto dvou faktorů se může měnit, je vhodné, aby na každý z nich byla aplikována jiná náhodná váha. Rychlost částice se pak určí následovně

$$v_i(t) = v_i(t - 1) + c_1 \cdot rand_1 \cdot (p_i - x_i(t - 1)) + c_2 \cdot rand_2 \cdot (p_g - x_i(t - 1)) \tag{18}$$

kde c_1 a c_2 jsou kladná čísla a $rand_1$ a $rand_2$ jsou náhodná čísla v rozmezí 0-1.

Z rovnice aktualizace rychlosti částice (18) je patrné, že se skládá ze tří hlavních částí. První část $v_i(t - 1)$ se nazývá *setrvačnost*. Představuje snahu částice pokračovat v původním směru pohybu. Tento parametr může být násoben nějakou váhou. Další částí rovnice je přitažlivost k nejlepší nalezené pozici dané částice p_i,hodnota fitness funkce na této pozici se značí p_{best}. Tato přitažlivost je násobena náhodnou váhou $c_1 \cdot rand_1$ a nazývá se *pamětí* částice. Poslední třetí částí rovnice je přitažlivost k nejlepší nalezené pozici jakékoliv částice p_g, odpovídající hodnota fitness funkce se značí g_{best}. Tato přitažlivost je opět násobena

náhodnou váhou $c_2 \cdot rand_2$ a nazývá se *sdílenou informací*, nebo též *společnou znalostí* [37].



Obrázek 9: Vektorové znázornění aktualizace pozice částice [38]

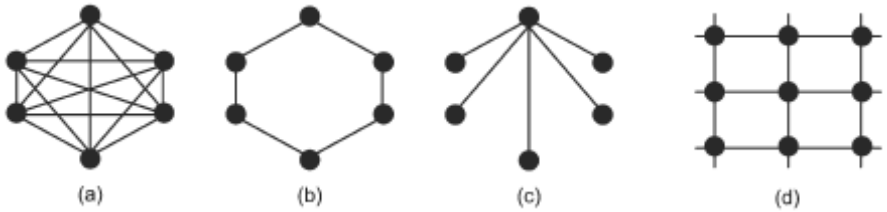
Samotný algoritmus PSO lze shrnout následovně:

1. Inicializace hejna. Každé částici je přiřazena náhodná pozice v prohledávaném prostoru.
2. Pro každou částici je vypočítána hodnota fitness funkce.
3. Porovnání současné hodnoty fitness funkce částice s její p_{best}. Pokud je současná hodnota lepší, je označena za p_{best} a do p_i je uložena současná poloha částice.
4. Nalezení částice s nejlepší fitness funkcí. Tato hodnota je označena za g_{best} a její poloha za p_g.
5. Aktualizace pozic a rychlostí částic dle rovnic (17) a (18).
6. Opakování kroků 2-5 dokud nejsou splněny podmínky ukončení. Tedy dokud není dosažen maximální počet iterací algoritmu nebo není nalezena dostatečně dobrá hodnota fitness funkce.

4.1.2 Topologie PSO

PSO může fungovat se dvěma základními druhy sousedství. Je to buď globální sousedství, při kterém jsou částice přitahovány k nejlepšímu nalezenému řešení z celého roje. To si lze představit jako plně propojenou síť, kde má každá částice přístup ke všem informacím (viz obrázek 10(a)). Druhou možností je lokální sousedství, kde jsou částice přitahovány k nejlepšímu řešení vybíraného pouze z jejich bezprostředních sousedů. U tohoto přístupu existují dvě nejčastější varianty. Jedná se o kruhovou topologii (viz obrázek 10(b)), kde je

každá částice propojena se dvěma sousedy, nebo o centralizovanou topologii (viz obrázek 10(c)). Zde jsou jednotlivé částice od sebe odděleny a veškeré informace jsou shromažďovány v hlavním jedinci.



Obrázek 10: Topologie rojů [38]

Předpokládá se, že plně propojená síť konverguje k řešení rychleji, avšak může uváznout v lokálním optimu. Zatímco přístup s omezeným sousedstvím má větší šanci nalézt optimální řešení, ale pomaleji. Dále se předpokládá, že nejlepších výsledků by měl dosahovat roj s von Neumannovskou topologií (viz obrázek 10(d)).

4.1.3 **Nastavení parametrů**

Při implementaci PSO je třeba mít na zřeteli několik předpokladů, aby byla zajištěna konvergence algoritmu a nedošlo k tzv. *explozi roje*. Mezi tyto předpoklady patří maximální rychlost částic, správné nastavení konstant pro přitažlivost (zrychlení) a nastavení setrvačnosti.

Při každém kroku algoritmu je každé částici vypočítána rychlost pro každý rozměr prostoru řešení. Jelikož je rychlost částice náhodná proměnná, může nabývat jakýchkoliv hodnot a částice se tak může pohybovat chaoticky. Aby k tomu nedocházelo, je vhodné nastavit dolní a horní limit rychlosti částice. Tyto limity je nutno nastavovat empiricky v závislosti na řešeném problému. Pokud by maximální povolená rychlost částice byla příliš vysoká, částice by mohly přeskakovat dobrá řešení. Naopak, pokud by byla příliš nízká, byl by pohyb částic omezen, algoritmus by konvergoval velmi pomalu a nikdy by nemuselo být nalezeno optimální řešení. Výzkum naznačuje, že nejlepším řešením je dynamicky se měnící maximální rychlost (v_{max}).

$$v_{max} = (x_{max} - x_{min}) / N \quad (19)$$

kde N je uživatelem zvolený počet intervalů v daném rozměru, x_{max} a x_{min} je maximální a minimální hodnota souřadnic dosud nalezených částicemi.

Akcelerační konstanty c_1 a c_2 (18) řídí pohyb částic směrem k nejlepší pozici částice, respektive k nejlepší celkové pozici. Nízké hodnoty těchto konstant omezují pohyb částic, zatímco vysoké hodnoty mohou vést k divergenci algoritmu. Bylo provedeno několik experimentů s jedinou částicí v jednorozměrném prostoru [38]. V tomto případě byla akcelerační konstanta uvažovaná jako jediná, protože nejlepší pozice částice i celého roje je stejná. Tedy $\phi = c_1 + c_2$. Pro nízké hodnoty ϕ měla trajektorie charakter sinusoidy. Při vyšších hodnotách se začaly objevovat cyklické trajektorie a při hodnotách vyšších než 4 mířila dráha částice do nekonečna. Po zavedení náhodných vah $rand_1$ a $rand_2$, zamezujících cyklickému opakování dráhy částic, lze obecně říci, že vhodná velikost akcelerační konstanty je právě 4. Tedy $c_1 + c_2 = 4$, neboli $c_1 = c_2 = 2$. Samozřejmě c_1 a c_2 nemusí mít stejnou velikost, opět záleží na řešeném optimalizačním problému.

Zkušenosti ukázaly, že i když jsou maximální rychlost a akcelerační konstanty správně nastaveny, může i tak dojít k explozi roje. V současnosti existují dvě metody snažící se tento problém řešit: omezující faktor a konstanta setrvačnosti. První z metod je omezující faktor, což je konstanta, kterou se násobí pravá strana rovnice aktualizace rychlosti částice (18). Tato konstanta (χ) se používá, pokud je nastaveno $c1 + c_2 > 4$. Vypočítá se jako:

$$\chi = \frac{2}{\left|2 - \phi - \sqrt{\phi^2 - 4\phi}\right|} \quad (20)$$

Výsledná rovnice aktualizace rychlosti se pak zapíše:

$$v_i(t) = \chi \{v_i(t - 1) + c_1 \cdot rand_1 \cdot (p_i - x_i(t - 1)) + c_2 \cdot rand_2 \cdot (p_g - x_i(t - 1))\} \quad (21)$$

Obecně omezující faktor zlepšuje konvergenci částic tím, že tlumí oscilaci částic, jakmile se zaměří na nejlepší pozici v oblasti. Nevýhodou je, že částice nikdy nemusí konvergovat, pokud jejich nejlepší pozice p_i je příliš vzdálena od celkové nejlepší pozice hejna p_g.

Druhá metoda pracuje s parametrem, který násobí pouze setrvačnost $v_i(t - 1)$ a ne celou pravou stranu jako omezující faktor. Tento parametr se značí ϕ_{ic}. Rychlost částice se v tomto případě vypočítá jako:

$$v_i(t)=\phi_{ic}\, v_i(t -1) + c_1 \cdot rand_1 \cdot (p_i - x_i(t -1)) + c_2 \cdot rand_2 \cdot (p_g - x_i(t -1)) \qquad (22)$$

Konstanta setrvačnosti může být implementována jako konstantní, nebo se může měnit v průběhu algoritmu. Tento parametr v podstatě kontroluje detailnost prohledávání prostoru optimalizačního problému. Na počátku algoritmu je tato konstanta nastavena na vyšší číslo (obvykle 0,9), takže částice se pohybují rychleji a tedy rychleji konvergují ke globálnímu optimu. Jakmile je nalezena optimální oblast, tato váha se nastaví na nižší (obvykle 0,4). Tím je zvětšena detailnost prohledávání, což napomáhá nalezení opravdového optima. Nevýhodou je, že jakmile je jednou tato váha snížena, roj ztrácí schopnost prohledávat nové oblasti. To může vést k uváznutí v lokálním optimu.

4.1.4 **Kombinace genetického algoritmu a PSO**

Tento algoritmus kombinuje výhody rojové inteligence a mechanismů přirozeného výběru tak, že zvyšuje počet dobře hodnocených částic, tím že v každém kroku algoritmu snižuje počet špatně ohodnocených částic. Nejen, že lze měnit prohledávané oblasti za pomoci p_{best} a g_{best} parametrů, ale je možné i skákání mezi oblastmi díky mechanismu výběru. To vede ke zvýšení rychlosti konvergence celého algoritmu.
Jedním z možných přístupů jak vylepšit algoritmus PSO, je aplikace reprodukce, která u náhodně zvolených částic mění jak poziční vektor, tak vektor rychlosti.

Například takto [38]:

$$child_1(x) = p \cdot parent_1(x) + (1-p)\, parent_2(x)$$

$$child_1(v) = (parent_1(v) + parent_2(v)) \cdot \frac{|parent_1(v)|}{|parent_1(v) + parent_2(v)|}$$

$$child_2(x) = p \cdot parent_2(x) + (1-p)\, parent_1(x)$$

$$child_2(v) = (parent_1(v) + parent_2(v)) \cdot \frac{|parent_2(v)|}{|parent_1(v) + parent_2(v)|}$$

kde p je náhodné číslo z rozsahu ⟨0-1⟩, $parent_{1,2}(x)$ reprezentuje poziční vektor náhodně zvolených částic, $parent_{1,2}(v)$ představuje odpovídající vektor rychlostí těchto částic a $child_{1,2}(x)$ a $child_{1,2}(v)$ jsou potomci genetického procesu. Takto vzniklými částicemi se pak nahradí částice s nízkou hodnotou fitness funkce.

4.2 Samoorganizující se migrační algoritmus

Činnost algoritmu SOMA je založena vektorových operacích podobně jako jiné populační algoritmy typu genetické algoritmy, diferenciální evoluce a podobně. Algoritmus vytváří nové populace migračním operátorem a speciálním operátorem, který může zastupovat formu mutace. Můžeme jej tedy řadit k evolučním algoritmům. SOMA algoritmus patří také do skupiny memetických či hejnových algoritmů. Potomci se nevytvářejí křížením jako u jiných evolučních algoritmů, ale spíše je filozofie algoritmu založena na pohybu inteligentních jedinců, tj. migraci jedinců. Tito jedinci spolupracují při řešení společného problému. Jedinci prohledávají prostor možných řešení podobně, jako jejich biologické protějšky prohledávají krajinu a hledají např. potravu. Evoluční cyklus se tedy nenazývá generace, ale *migrace*. Nejedná se o vývoj nových jedinců, spíše o přesun původních jedinců na nové pozice. Příklady takového chování, jež byly inspirací pro tento algoritmus, lze nalézt i v reálném světě. Jsou to např. mravenci, včely, predátoři ve smečce hledající potravu apod. Vlastnost samo-organizace u algoritmu SOMA plyne z faktu, že se jedinci ovlivňují navzájem během hledání lepšího řešení, což mnohdy vede k tomu, že v prostoru možných řešení vznikají skupiny jedinců, které se během putování přes

prohledávaný prostor rozpadají či spojují. Jinými slovy si skupina jedinců neboli populace sama organizuje vzájemný pohyb jedinců – proto tedy samo-organizace [44, 45, 47]

4.2.1 **Parametry algoritmu SOMA**

Běh algoritmu SOMA je ovlivňován speciální množinou parametrů, které lze rozdělit na parametry *řídicí* a *ukončovací*. Řídicí parametry jsou takové parametry, které mají vliv na kvalitu běhu algoritmu, zatímco ukončovací parametry představují předem nadefinované podmínky, které běh algoritmu ukončují. Všechny tyto parametry musí být zvoleny uživatelem před začátkem samotného algoritmu. Parametry a jejich doporučené hodnoty pro SOMA algoritmus popisuje následující tabulka (tab. 1)

Tabulka 1: Význam parametrů SOMA algoritmu

Parametr	**Doporučená hodnota**	**Popis**
PathLength	1,1 až 3	Řídicí parametr
Step	0,11 až PathLength	Řídicí parametr
PRT	0 až 1	Řídicí parametr
D	ovlivněno daným problémem	Počet argumentů účelové funkce
PopSize	10 až N	Ukončovací parametr (volí uživatel)
Migrace	10 až M	Ukončovací parametr (volí uživatel)
AcceptedError	libovolně zvolený, případně -1	Ukončovací parametr - velikost chyby, případně chyba není použita

PathLength – určuje, jak daleko se aktivní jedinec zastaví od vedoucího jedince.

- *PathLength* = 1 => aktivní jedinec zastaví na pozici vedoucího jedince (Leader)
- *PathLength* = 2 => aktivní jedinec zastaví za vedoucím jedincem (Leaderem) ve stejné vzdálenosti, ve které od něj startoval
- *PathLength* < 1 => degenerace algoritmu (omezení na hledání lokálních extrémů)

- *PathLength* = 3 => doporučená hodnota parametru pro řešení většiny problémů

Step – určuje velikost skoku jedince. Při řešení problému jednoduché unimodální účelové funkce je možné použít velkou hodnotu parametru pro urychlení chodu algoritmu. Pokud není tvar účelové funkce známý, doporučuje se nastavit hodnotu na 0,11 – prostor možných řešení pak bude prohledáván podrobněji. Parametr *Step* nesmí být celočíselným násobkem parametru *PathLength* – jinak by došlo ke snížení diverzibility populace a proces by tak mohl rychleji skončit v lokálním extrému.

PRT – tzv. pertubace. Podle tohoto parametru se tvoří pertubační vektor (*PRTVector*), který ovlivňuje, zda se aktivní jedinec bude pohybovat přímo k vedoucímu jedinci či ne. Je to velmi důležitý parametr s velkou citlivostí. Optimální hodnota je kolem 0,1. V případě že *PRT* = 1 – dochází k čistě deterministickému prohledávání (stochastická složka je eliminována) a algoritmus je pak omezen pro hledání lokálních extrémů. Perturbace zde představuje jistou formu mutace.

D – počet optimalizovaných proměnných. Závisí přímo na definici problému.

PopSize – je řídicí parametr, který určuje počet jedinců v populaci. Obvykle se doporučuje velikost nastavit na 10krát D. Standardně stačí 30 – 50 jedinců.

Migrace – je parametr, který je ekvivalentní k parametru *Generace* z jiných evolučních algoritmů. V podstatě určuje, kolikrát se populace změní. Je to ukončovací parametr.

AcceptedError – je ukončovací parametr definovaný uživatelem. Určuje maximální možný rozdíl mezi nejhorším a nejlepším jedincem v populaci.

4.2.2 **Princip činnosti algoritmu SOMA**

Vznik SOMA byl inspirován soutěživě-kooperativním chováním inteligentních jedinců řešících společný problém. Chování tohoto typu lze objevit prakticky kdekoli na světě. Jako příklad lze použít chování smečky lovících vlků, včelího úlu, hejna holubů apod. U těchto příkladů je společným úkolem např. hledání potravy, v rámci níž jedinci spolupracují, ale i soutěží. Ve fázi spolupráce si navzájem jednotliví jedinci sdělují, jakou kvalitu hledaného momentálně nalezli a na základě toho se snaží přizpůsobit své další chování. Ve fázi soutěžení, která předchází fázi kooperace, se každý jedinec snaží vyhrát nad ostatními - snaží se nalézt co nejlepší zdroj potravy. Po ukončení fáze soutěže nastane opět fáze spolupráce a jedinci si vymění informace o tom, který z nich má nejlepší zdroj potravy. Ostatní opustí své nalezené zdroje potravy a migrují směrem k jedinci s nejlepším zdrojem potravy a během této migrace se snaží nalézt ještě lepší zdroj. To se opakuje, dokud se všichni nesejdou u nejvydatnějšího zdroje potravy. Na tomto silně zjednodušeném principu funguje i algoritmus SOMA. Následující tabulka (tab. 2) obsahuje paralelismus mezi významem v biologické reprezentaci a terminologií v algoritmu SOMA.

Algoritmus SOMA pracuje v cyklech zvaných *migrační kola*. Ta hrají tutéž úlohu, jakou mají generace v případě genetických algoritmů. Rozdíl mezi migračními koly a generacemi je spíše filozofického charakteru. Během migračních kol nejsou tvořeni noví jedinci, pouze jsou přemisťováni do finální pozice pomocí sekvence pozic vypočítaných vzhledem k pozici *Leadera* - migrují stavovým prostorem možných řešení.

Tabulka 2: Význam biologické terminologie v algoritmu SOMA dle [44, 45, 47].

BIOLOGICKÁ REALITA	IMPLEMENTACE V ALGORITMU
Členové smečky či společenství	Jedinci v populaci – parametr ***PopSize***
Člen smečky s nejlepším zdrojem potravy	Nejlepší jedinec dané migrace – ***Leader***
Potrava	Hodnota účelové funkce
Oblast, ve které dané společenství žije	***Hyper-plocha*** definovaná účelovou funkcí
Pohyb členů smečky v obydlené oblasti	Migrační kola SOMA algoritmu

Jednotlivé kroky algoritmu SOMA:

1 Definice parametrů

před samotným spuštěním algoritmu je nutné nejprve definovat veškeré potřebné parametry– řídicí a ukončovací parametry (*Specimen, Step, PathLength, AcceptedError, Popsize, PRT* a *Migrace*) viz Tabulka 1. Dalším důležitým krokem, je nadefinovat účelovou funkci, která reprezentuje optimalizovaný problém. Řešením pak je nalezení jejího nejlépe globálního extrému, tedy optimální hodnoty parametrů.

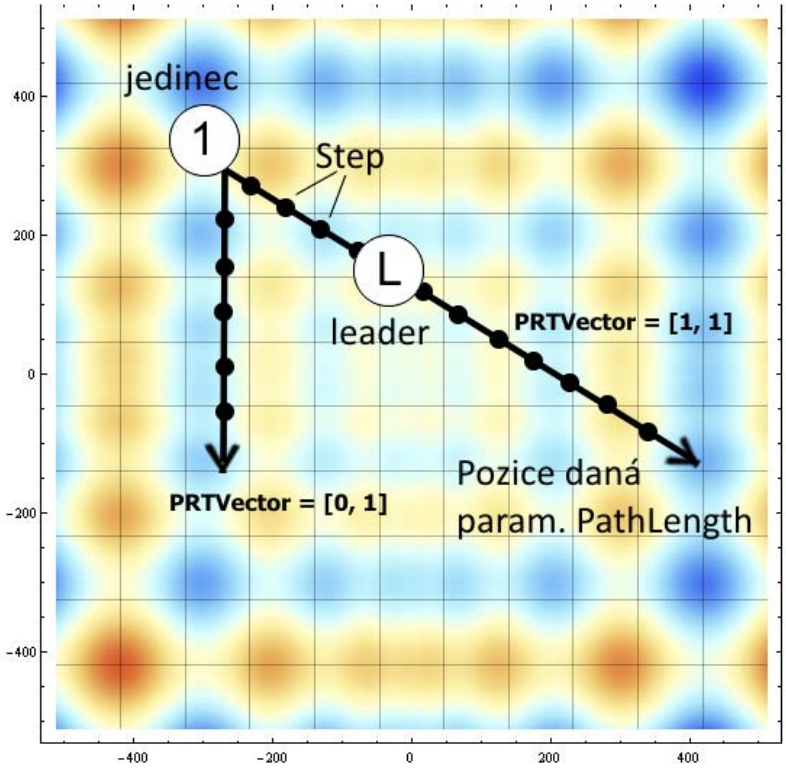
2 Tvorba populace

V tomto kroku vytvoříme počáteční populaci – náhodně vygenerujeme jedince. S využitím parametru *Specimen* a generátoru čísel je pro každý parametr jedince generováno náhodné číslo (jeho hodnota je dána rozsahem *Specimenu*).

3 Migrační kola – migrace

Každého jedince ohodnotíme účelovou funkcí a určíme mezi jedinci *Leadera.* V tuto chvíli dochází k migraci ostatních jedinců směrem k Leaderovi po zvolených krocích (parametr *Step*). Znovu dochází k ohodnocení jedinců účelovou funkcí, a pokud se u některých jedinců změní hodnota k lepšímu – jedinec si ji

zapamatuje. Po skončení migračního kola se všichni jedinci posunou na novou, nejlépe ohodnocenou pozici (obr. 11). *Leader* zůstává na místě. Než daný jedinec započne svou cestu směrem k *Leaderovi*, je vygenerován prázdný *PRTVector* o dimenzi = *D* včetně vygenerované sekvence náhodných čísel z intervalu <0,1> pro každý optimalizovaný parametr. Tato náhodná čísla jsou porovnána s parametrem *PRT*. Jestliže je *n*-té vygenerované číslo větší nežli PRT parametr, pak je *n*-tý parametr *PRTVectoru* nastaven na 0 a v opačném případě na 1. Parametry jedince, které jsou takto nastaveny na 0, se nepřepočítávají (tj. jsou zmraženy) - snižuje se počet stupňů volnosti pohybu jedince. Tento proces nahrazuje operátor mutace, jež obvykle u evolučních algoritmů probíhá. Díky tomu se rapidně zvyšuje robustnost SOMA algoritmu ve smyslu nalezení globálního extrému.



Obrázek 11: Funkce PRTVectoru

4 Zjištění stavu ukončovacích parametrů

Nyní zkontrolujeme, zda je rozdíl mezi *Leaderem* a nejhorším jedincem menší než *AcceptedError*. Stejně tak ověříme, zda došlo

k naplnění počtu migračních kol – parametr *Migrations*. Pokud není splněna ani jedna podmínka, algoritmus se vrací do kroku 3 a pokračuje, jinak se algoritmus zastaví.

5 Stop

Získáváme nejlepšího nalezeného jedince (řešení) po ukončení posledního migračního kola.

4.2.3 **Strategie SOMA**

V dnešní době existuje již několik různých variací algoritmu SOMA. Pro jejich obecné označení se používá výraz *strategie* – takové označení lépe poukazuje na to, že algoritmus využívá kooperace jedinců a geometrického přesouvání populace po hyper-ploše. Strategie lze tedy rozdělit takto [44, 45, 47]:

1 AllToOne

„Všichni k jednomu“ – všichni jedinci migrují k Leaderovi, vyjma jej samotného.

2 AllToAll

„Všichni ke všem“ - tato strategie neobsahuje žádného Leadera, resp. Leaderem se stanou postupně všichni jedinci v populaci. Všichni jedinci migrují ke všem a jediným rozdílem od předchozí *AllToOne* strategie je, že po dokončení migrace aktuálního jedince se daný jedinec vrátí na pozici, kde byl nalezen nejlepší extrém během jeho *PopSize - 1* migračních cest v jednom migračním kole. Tato strategie je náročná na výpočet.

3 AllToAllAdaptive

„Všichni ke všem adaptivně“ – strategie totožná s *AllToAll* s rozdílem, že pokud aktuální migrující jedinec se přesune na novou pozici nalezenou na jeho cestě ihned, nikoli až po migracích všech ostatních jedinců v populaci.

4 AllToOneRand

„Všichni k jednomu náhodně“ – strategie, kde opět existuje Leader a jedinci se snaží k němu přiblížit. Leader však není vybírán podle

nejlepšího ohodnocení, ale náhodným výběrem. Zde je tedy možná určitá modifikace algoritmu, kdy Leader bude vybírán daným algoritmem.

5 Clusters

„Svazky“ – Proces vytváření svazků v SOMA si lze jednoduše představit tak, že jedinci jsou reprezentováni nějakou sub-populací. V každé takové sub-populaci pak probíhá SOMA samostatně za účelem výběru nejlepšího jedince a následné migrace – z toho vyplývá, že svazky se mohou spojovat a rozpadat, čímž je opět podtržen efekt kooperace.

Kontrolní otázky

1. Pokuste se načrtnout blokové schéma, jež odpovídá činnosti algoritmu PSO?
2. Pokuste se načrtnout blokové schéma, jež odpovídá činnosti algoritmu SOMA.

Úkoly k textu:

1. Nalezněte na webu nějaké další aplikace algoritmu PSO.
2. Nalezněte na webu nějaké další aplikace algoritmu SOMA.

Korespondenční úkol:

1. Vytvořte počítačový program pro realizaci algoritmu PSO.
2. Vytvořte počítačový program pro realizaci algoritmu SOMA.

Shrnutí obsahu kapitoly

Kolektivní inteligence je umělá inteligence založená na kolektivním chování decentralizovaných, samoorganizovaných systémů. Takovéto systémy se skládají z populace jedinců, kteří interagují lokálně mezi sebou a se svým okolím. Přestože neexistuje centralizovaná struktura

řízení, která by určovala, jak se jedinci mají chovat, lokální interakce mezi těmito agenty vedou ke vzniku složitého globálního chování. Do skupiny memetických či hejnových algoritmů patří algoritmus PSO (algoritmus hejna částic) i algoritmus SOMA (samo-organizující se migrační algoritmus). Algoritmus PSO využívá populace částic, které prolétají prohledávaným prostorem problému určitou rychlostí určenou individuálně pro každou částici. Pohyb každé částice přirozeně směřuje k řešení blízkému optimu. Filosofie algoritmu SOMA je založena na pohybu inteligentních jedinců, tj. migraci jedinců. Nejedná se zde o vývoj nových jedinců, ale spíše o přesun původních jedinců na nové pozice.

Pojmy k zapamatování

- kolektivní inteligence,
- PSO,
- exploze roje,
- SOMA,
- migrace,
- strategie SOMA.

5 Diferenciální evoluce

V této kapitole se dozvíte:

- Jaké jsou základní pojmy a vlastnosti diferenciální evoluce.
- Jak jsou definovány operátory mutace a křížení v diferenciální evoluci.
- Jak probíhá selekce turnajem.

Po jejím prostudování byste měli být schopni:

- Objasnit jaké jsou způsoby generování nového bodu.
- Objasnit, jak nastavit hodnotu parametrů *F* a *C.*

Klíčová slova této kapitoly:

Diferenciální evoluce, selekce turnajem,

Průvodce studiem

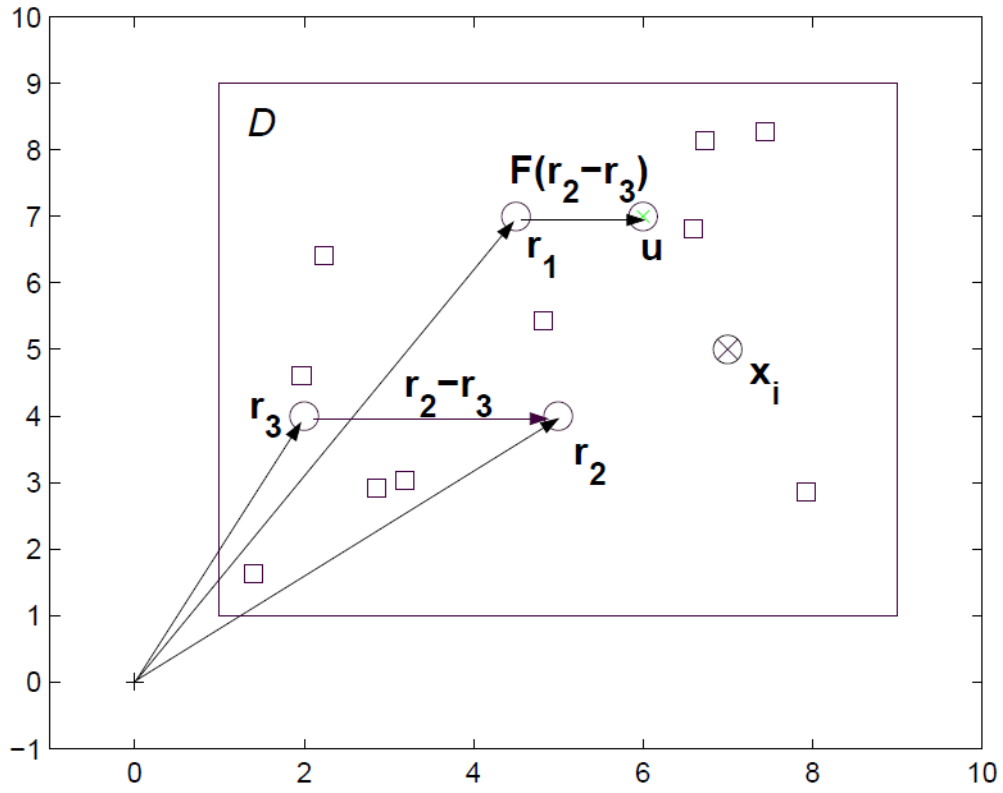
V této kapitole se seznámíte s algoritmem diferenciální evoluce. Tento algoritmus byl navržen nedávno a poprvé publikován v roce 1995. Během několika let se stal velmi populární a je často aplikován na problémy hledání globálního minima. Je to příklad jednoduchého heuristického hledání užívajícího evoluční operátory. Kapitola byla napsána podle [36]

Diferenciální evoluce (DE) je postup k heuristickému hledání minima multimodálních funkcí, který navrhli R. Storn a K. Price [33] koncem 90. let. Algoritmus DE dosáhl značné popularity, je často aplikován a dále rozvíjen. Experimentální výsledky i zkušenosti z četných aplikací ukazují, že často konverguje rychleji než jiné stochastické algoritmy pro globální optimalizaci.

Algoritmus diferenciální evoluce vytváří novou generaci *Q* tak, že postupně pro každý bod $\boldsymbol{x}_i$ ze staré generace *P* vytvoří jeho potenciálního konkurenta ***y*** a do nové populace z této dvojice zařadí bod s nižší funkční hodnotou. Algoritmus můžeme zapsat takto [36], obr. 12

generuj P (N bodů náhodně v D)
repeat
  for $i := 1$ **to** N **do**
    generuj vektor $\mathbf{u}$
    vytvoř vektor $\mathbf{y}$ křížením $\mathbf{u}$ a $\mathbf{x}_i$
    if $f(\mathbf{y}) < f(\mathbf{x}_i)$ **then** do Q zařaď $\mathbf{y}$
      else do Q zařaď $\mathbf{x}_i$
  endfor
  $P := Q$
until podmínka ukončení

Obrázek 12: Algoritmus diferenciální evoluce



Obrázek 13: Generování bodu **u**

Je několik způsobů, jak generovat nový bod **u,** což reprezentuje operátor mutace. Zde uvedeme dva nejčastěji užívané. Postup

označovaný RAND generuje bod u ze tří bodů ze staré populace podle vztahu:

$$\mathbf{u} = \mathbf{r}_1 + F(\mathbf{r}_2 - \mathbf{r}_3), \tag{23}$$

kde $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ jsou navzájem různé body náhodně vybrané z populace P různé od aktuálního bodu $\boldsymbol{x}_i$, $F > 0$ je vstupní parametr. Generování bodu **u** je graficky znázorněno na obrázku 13.

Postup označovaný BEST využívá nejlepší bod z populace P a čtyři další náhodně vybrané body podle vztahu

$$\mathbf{u} = \boldsymbol{x}_{best} + F(\mathbf{r}_1 + \mathbf{r}_2 - \mathbf{r}_3 - \mathbf{r}_4), \tag{24}$$

kde $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \mathbf{r}_4$ jsou navzájem různé body náhodně vybrané z populace P různé od aktuálního bodu $\boldsymbol{x}_i$ i od bodu $\boldsymbol{x}_{best}$ s nejnižší funkční hodnotou v P. $F > 0$ je opět vstupní parametr.

Nový vektor $\boldsymbol{y}$ vznikne křížením vektoru $\boldsymbol{u}$ a vektoru $\boldsymbol{x}_i$ tak, že kterýkoli jeho prvek x_{ij} je nahrazen hodnotou u_j s pravděpodobností C. Pokud žádné x_{ij} nebylo přepsáno hodnotou u_j nebo při volbě $C = 0$, nahrazuje se jeden náhodně vybraný prvek vektoru $\boldsymbol{x}_i$:

$$y_j = \begin{cases} u_j & \text{když } R_j \le C \quad \text{nebo } j = I \\ x_{ij} & \text{když } R_j > C \quad \text{a } j \ne I \end{cases} \tag{25}$$

kde I je náhodně vybrané celé číslo z $\{1, 2, \ldots, d\}$, $R_j \in (0, 1)$ jsou voleny náhodně a nezávisle pro každé j a $C \in [0, 1]$ je vstupní parametr. Toto křížení se označuje jako binomiální. Kromě toho existuje ještě křížení exponenciální, ve kterém se vyměňuje náhodně určený počet sousedních prvků vektorů. Je tedy podobné jednobodovému křížení u genetických algoritmů.

Selekci představuje v diferenciální evoluci výběr lepšího bodu z dvojice $\boldsymbol{y}$ a $\boldsymbol{x}_i$ do nové populace Q. V diferenciální evoluci jsou tedy přítomny všechny tři nejdůležitější evoluční operace: selekce, křížení a mutace.

Ali a Törn [36] navrhli určovat hodnotu parametru F v každém iteračním kroku podle adaptivního pravidla

$$F = \begin{cases} \max\left(F_{\min}, 1 - \left|\frac{f_{\max}}{f_{\min}}\right|\right) & \text{if} \quad \left|\frac{f_{\max}}{f_{\min}}\right| < 1 \\ \max\left(F_{\min}, 1 - \left|\frac{f_{\min}}{f_{\max}}\right|\right) & \text{jinak,} \end{cases} \tag{26}$$

kde f_{min}, f_{max} jsou minimální a maximální funkční hodnoty v populaci P a $F_{\min}$ je vstupní parametr, který zabezpečuje, aby bylo $F \in [F_{\min}, 1)$. Autoři doporučují volit hodnotu $F_{\min} \geq 0.45$.

Předpokládá se, že tento způsob výpočtu F udržuje prohledávání diverzifikované v počátečním stadiu a intenzivnější v pozdější fázi prohledávání, což má zvyšovat spolehlivost hledání i rychlost konvergence. Výhodou algoritmu diferenciální evoluce je jeho jednoduchost a výpočetně nenáročné generování nového bodu ***y***, které lze navíc velmi efektivně implementovat. Nevýhodou je poměrně velká citlivost algoritmu na nastavení hodnot vstupních parametrů F a C. Storn a Price doporučují volit hodnoty $N = 10\,d$, $F = 0.8$ a $C = 0.5$ a pak tyto hodnoty modifikovat podle empirické zkušenosti z pozorovaného průběhu hledání. To je ovšem doporučení dosti vágní. K tomu, aby bylo užitečné, je nutná zkušenost a dobrá intuice řešitele problému. V testovacích úlohách v článku [33] sami užívají pro různé úlohy velmi odlišné hodnoty těchto parametrů, a to $0.5 \leq F \leq 1$ a $0 \leq C \leq 1$. Také velikost populace N volí často menší než doporučovanou hodnotu $N = 10\,d$, kde d je počet argumentů v optimalizované funkci.

Zde jsme si ukázali původní a nejjednodušší verzi algoritmu diferenciální evoluce. Existuje však spousta dalších variant (strategií). Tyto různé strategie se označují formální zkratkou DE/*m*/*n*/*c*, kde *m* označuje užitý typ mutace, *n* počet přičítaných rozdílů náhodně vybraných vektorů v mutaci a *c* je užitý typ křížení. Strategie užívající (23) a (25) se označuje jako DE/rand/1/bin, strategie s (24) a (25) se označuje jako DE/best/2/bin.

Shrneme-li, tak v diferenciální evoluci se nenahrazuje nejhorší bod v populaci, ale pouze horší bod ve dvojici, což zabezpečuje větší

diverzitu bodů nové populace po delší období, takže při stejné velikosti populace většinou DE ve srovnání s náhodným prohledáváním má menší tendenci ukončit prohledávání v lokálním minimu, ale za to platíme pomalejší konvergencí při stejné podmínce ukončení.

Kontrolní otázky:

1. Proč musí být F ≠ 0? Co by se stalo, kdybychom zadali F < 0?
2. Co způsobí zadání C = 0?

Korespondenční úkol:

Vytvořte počítačový program pro realizaci algoritmu diferenciální evoluce.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili algoritmem diferenciální evoluce. Je to příklad jednoduchého heuristického hledání užívajícího evoluční operátory. V diferenciální evoluci se nenahrazuje nejhorší bod v populaci, ale pouze horší bod ve dvojici, což zabezpečuje větší diverzitu bodů nové populace po delší období, a proto má také menší tendenci ukončit prohledávání v lokálním minimu.

Pojmy k zapamatování

- operátor mutace v diferenciální evoluci,
- operátor křížení v diferenciální evoluci,
- selekce turnajem

6 Symbolická regrese

V této kapitole se dozvíte:

- Co je to symbolická regrese.
- Jak symbolická regrese souvisí s evolučními algoritmy;
- Jaké jsou metody symbolické regrese.
- Jaké jsou základní principy analytického programování.

Po jejím prostudování byste měli být schopni:

- Vysvětlit základní vlastnostmi genetického programování a gramatické evoluce.
- Vysvětlit, jaké jsou základní principy analytického programování.

Klíčová slova této kapitoly:

Symbolická regrese, genetické programování, gramatická evoluce, analytického programování.

Průvodce studiem

Symbolická regrese ke své správné funkci potřebuje evoluční algoritmy, které slouží k procesu optimalizace. Genetické programování bylo navrženo za účelem modifikace genetických algoritmů pro tvorbu tzv. programů v rámci symbolické regrese. Gramatická evoluce je dalším nástrojem pro symbolickou regresi řešenou pomocí evolučních algoritmů. Její výhodou je, v porovnání s genetickým programováním, že může vyvinout programy v jakémkoli programovacím jazyce. Analytické programování lze chápat jako alternativní přístup vzhledem ke genetickému programování a gramatické evoluci a není vázáno k jednomu evolučnímu algoritmu.

Symbolická regrese je proces, jehož lze přirovnat k činnosti, kdy se z malých stavebních kamenů staví složitější struktura, která má popisovat požadované chování systému. Například aproximovat sadu naměřených dat a určit funkční závislost mezi nimi, či nalézt vhodnou trajektorii robota, nebo navrhnout vhodný design logických obvodů a lze uvést spoustu dalších aplikací, pro které je tento princip návrhu vhodný. Jestliže hovoříme o symbolické regresi v souvislosti s evolučními algoritmy, existuje v současnosti několik nástrojů, jež využíváme. Nejznámější je určitě genetické programování [16, 17] a gramatická evoluce [26]. V poslední řadě se mezi metody pro symbolickou regresi řadí také analytické programování [44, 46, 47, 48].

6.1 Metody symbolické regrese

Genetické programování (GP) bylo představeno na konci 80. let minulého století Johnem Kozou [16, 17]. Navrhl modifikaci genetických algoritmů [3] pro tvorbu tvz. programů v rámci symbolické regrese a pojmenoval jej genetické programování.

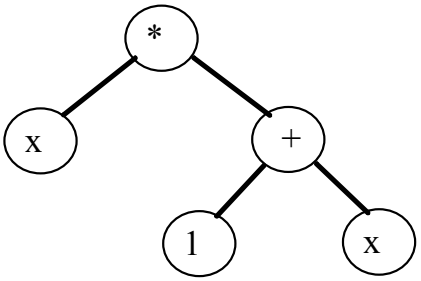
Gramatická evoluce je dalším ze známých nástrojů pro symbolickou regresi řešenou pomocí evolučních algoritmů. Jejími zakladateli jsou O'Neill a Conor Ryan [26, 32]. Její výhodou je, v porovnání s genetickým programováním, že může vyvinout programy v jakémkoli programovacím jazyce.

6.1.1 **Genetické programování**

V konceptu genetického programování není nová populace šlechtěna klasickým numerickým přístupem, ale analytickou cestou [5, 16, 17, 22, 47]. To znamená, že populace neobsahuje čísla, ale funkce samotné. Jinými slovy řečeno: ze základních kamenů typu elementárních funkcí, konstant, proměnných či operátorů je vyšlechtěn vhodný složitější tvar, který popisuje požadované chování zadaného problému. Například u aproximace funkcí jsou základními kameny matematické operátory a funkce typu plus, minus, cosinus, tangens

apod. a proměnné x, konstanty – číslo π či jiná čísla. Z takovýchto elementů pak může být vytvořeny libovolné funkce, např. $\frac{2x-\sin(y)}{\pi}$. Následnou optimalizací se pak hledá nejvhodnější komplexní výraz v analytickém tvaru.

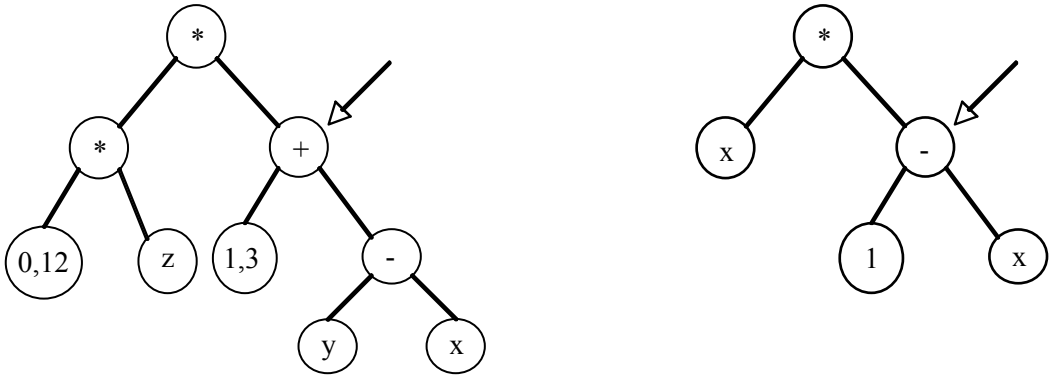
Jelikož je GP rozšířením genetických algoritmů, tak taxonomie je zde stejná. Každá hodnota parametru se jmenuje *gen*. Zde se ale na rozdíl od genetických algoritmů nevyskytují pouze hodnoty celočíselného či reálného typu, ale parametry v chromozómovém řetězci jsou přímo funkce. Pro snadnější zobrazení a reprezentaci se v základním tvaru GP používá *stromová struktura*. Vrchol stromové struktury se nazývá *kořenem*. Stromové struktury se čtou zleva zdola směrem ke kořeni. Obrázek 14 je možné přepsat do klasického zápisu jako funkce tvaru: fce(x) = x * (1 + x).



Obrázek 14: Stromová struktura

Operátor křížení

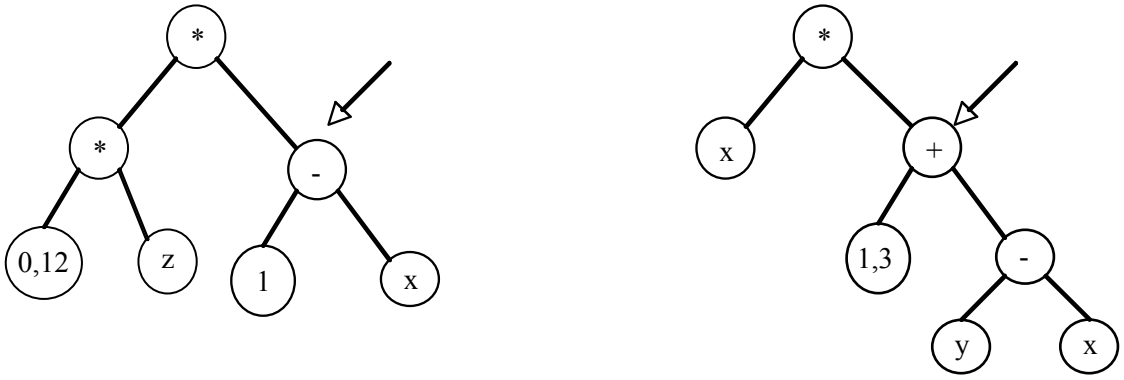
Po náhodném vytvoření populace se na jednotlivé jedince aplikují stejné operátory jako v genetických algoritmech. Operátor *křížení* pracuje následovně (Obr. 15 a Obr. 16). Náhodně se ve dvou vybraných stromech (*rodič 1*, *rodič 2*) zvolí místa (uzly), kde se řetězce funkcí od sebe oddělí (Obr. 15). Potomci (*potomek 1*, *potomek 2*) jsou vytvořeni tak, že se ve zvolených uzlech rodičů vymění jejich podřetězce (Obr. 16)



Obrázek 15: Operátor křížení v GP. Body křížení jsou naznačeny šipkou.

Rodič 1: $fce_1(x) = 0,12 * z * (1,3 + y - x)$.

Rodič 2: $fce_2(x) = x * (1 - x)$.



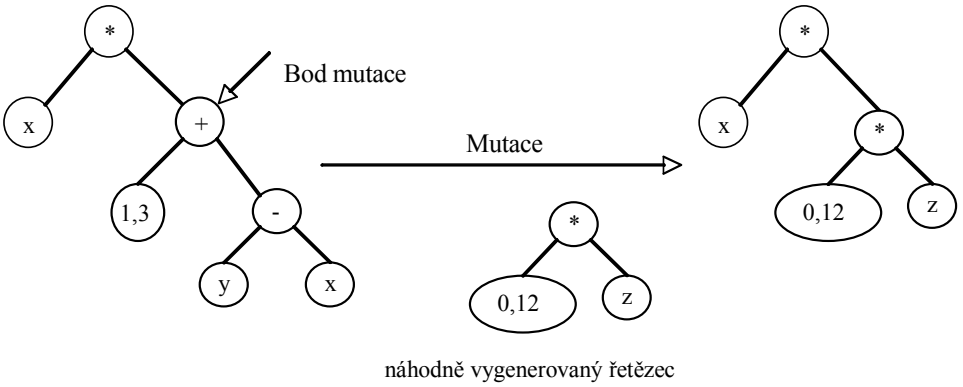
Obrázek 16: Operátor křížení v GP. Body křížení jsou naznačeny šipkou.

Potomek 1: $fce_1(x) = 0,12 * z * (1 - x)$.

Potomek 2: $fce_2(x) = x * (1,3 + y - x)$

Operátor mutace

Ve vybraném stromu náhodně zvolíme místo (uzel), kde se daný řetězec funkce nahradí náhodně vygenerovaným řetězcem (Obr. 17**Chyba! Nenalezen zdroj odkazů.**).



Obrázek 17: Operátor mutace v GP.

Tak jako v případě genetických algoritmů, tak i u genetického programování se objevuje parametr *pravděpodobnost mutace* a *pravděpodobnost křížení*. V případě genetického programování se ještě uvádí i *parametr pravděpodobnosti výběru* uzlu jako kořene pro začátek mutace.

Postup tvorby nových řešení lze zapsat následovně:

Nejprve je nastavena maximální hloubka stromu D_{max}.
Pak se používají dvě metody pro tvorbu nových řešení. První populace je tvořena běžně půl na půl oběma metodami.

1. *Plnící metoda* (každá větev má délku d = D_{max})
   - uzly v hloubce d < D_{max} jsou náhodně vybrány z množiny neterminálů (funkcí)
   - uzly v hloubce d = D_{max} jsou vybrány z množiny terminálů T

2. *Růstová metoda* (každá větev má délku d ≤ D_{max})
   - uzly v hloubce d < D_{max} jsou náhodně vybrány ze sjednocené množiny funkcí F a terminálů T
   - uzly v hloubce d = D_{max} náhodně vybrány z terminálů T

U genetického programování se objevuje tzv. problém *Bloat*, kdy potomci mohou mít delší řetězce než jejich rodiče. Obvykle průměrná

délka jedince narůstá s počtem iterací lineárně. Mezi možná řešení tohoto problému je zavést penalizaci dlouhých řešení, uzpůsobit operátory mutace či křížení, popř. použít více-kriteriální optimalizaci. Genetické programování obvykle v jednom jedinci kóduje pouze jedno řešení, pouze multivýrazové programování má v jednom chromozomu uloženou informaci o více řešeních.

Existuje několik přístupů ke genetickému programování [27]: lineární genetické programování, multivýrazové programování, kartézské genetické programování či infix forma genetického programování [27]. Všechny verze se liší v kódování jedince, striktně rozlišují mezi genotypem a fenotypem. Jedinci jsou zakódováni v lineární formě, ale vyjadřují se obvykle složitějšími nelineárními strukturami, například stromovými strukturami.

6.1.2 **Gramatická evoluce**

Gramatická evoluce je genetický algoritmus rozšířený o překladač gramatiky [12, 26, 32, 47]. Proces šlechtění nových potomků se pak provádí stejně jako v genetických algoritmech, tedy výběrem rodičů, použitím operátorů křížení a mutace. Nutné ale je, aby prohledávací algoritmus uměl pracovat s proměnlivým řetězcem v jedinci. Účelová funkce se pak obvykle tvoří jako rozdíl od požadovaného chování a její hodnota se převádí na vhodnou fitness, která je nutná k ohodnocení kvality jedince.

Gramatická evoluce se dále od genetického programování se liší následovně:

- používá lineární genomy;
- provádí ontogenetické mapování z genotypu na fenotyp (program);
- používá gramatiku k tvorbě legálních struktur v prostoru fenotypů.

Gramatická evoluce používá pro mapování programu Backus-Naurovu formu definice gramatiky, což je (normální) forma způsobu zápisu bezkontextových gramatik původně navržená pro popis syntaktických pravidel programovacího jazyka Algol 60.

Pravidla gramatické evoluce se zapisují ve tvaru:
<symbol>::=<možnosti>,

kde **<symbol>** patří do abecedy a **<možnosti>** jsou řetězce terminálů a neterminálů oddělené znakem '**|**', který odděluje jednotlivé možnosti generativní gramatiky.

Generativní gramatika slouží na rozdíl od analytické gramatiky k vytvoření syntakticky správně zapsaného programu z dané posloupnosti pravidel. Analytická gramatika pouze kontroluje syntaktickou správnost vytvořeného programu. Tedy můžeme psát:
<symbol>::=<možnost 1>|<možnost 2>|<možnost 3>|<možnost n>
kde **<možnosti>**substituují možné terminály či neterminály.

Ilustrační příklad:

Jako příklad lze uvést část definice pravidel pro zpracování kladných desetinných čísel.

1. <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
2. <unsigned integer> ::= <digit> | <unsigned integer><digit>
3. <decimal fraction> ::= .<unsigned integer>
4. <decimal number> ::= <unsigned integer> | <decimal fraction> | <unsigned integer><decimal fraction>

Pravidla lze pak interpretovat následovně:

- *pravidlo 1* – číslice je jeden z terminálů z intervalu < 0 , 9 >.
- *pravidlo 2* – celé číslo bez znaménka je jedna a více číslic.
- *pravidlo 3* – desetinná část je tečka následovaná celým číslem bez znaménka.

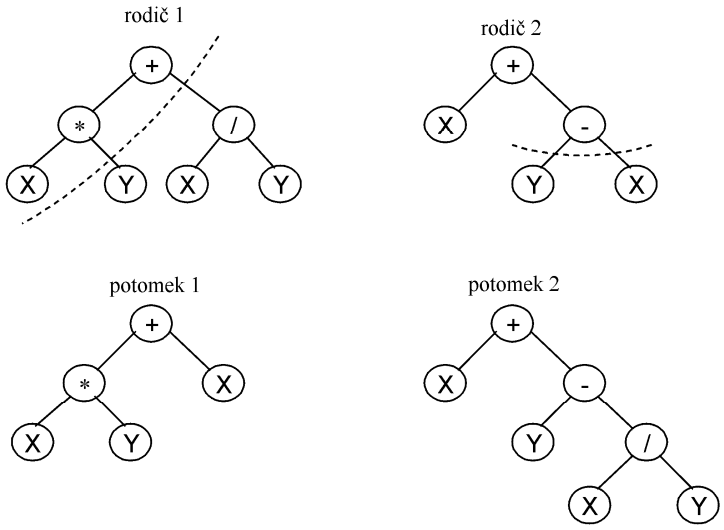
- *pravidlo 4* – desetinné číslo je buď celé číslo bez znaménka, nebo pouze desetinná část, nebo celé číslo bez znaménka následované desetinnou částí.
- *Terminály* jsou číslice ’0’ . . . ’9’ a znak tečka.
- *Počáteční symbol* je <decimal number>

V gramatické evoluci můžeme rozlišit, pokud používáme pouze terminály či jestli sada pravidel obsahuje i neterminál. Terminál je něco, co už nemá další možnost větvení. Neterminál obsahuje terminály, ale i další neterminály. *T - pravidlo* má pak na pravé straně (ve všech substitucích) pouze terminály. *N - pravidlo* je takové pravidlo, kde jsou použity i neterminály [26].

Standardně se v gramatické evoluci používá jednobodové křížení, které je aplikováno na chromozomy (obrázek 18). Tato jednoduchá operace má však poměrně velký dopad na nově vygenerované jedince.

48	16	120	38	51	230	79	17	84	63	49	122	165	213
56	80	71	168	214	147	31	3	91	112	67	135		

V místě křížení jsou chromozomy rozděleny na dvě části: první reprezentuje kostru nového jedince a druhá reprezentuje větve, které byly z rodičovského stromu odříznuty. Křížení se dokončí tak, že kostra nového jedince bude doplněna uzly a podstromy, které budou generovány použitím kodonů ze zbývající části druhého rodiče. Tyto geny si buď zachovají stejnou interpretaci, jakou měly v původním programu, nebo ji změní v závislosti na kontextu, v jakém budou v novém programu použity, tzn. mezi kolika pravidly a pro jaký neterminál mají rozhodnout. Ale v každém případě jsou použitelné a nemůže dojít k žádné syntaktické chybě, protože geny pouze zprostředkovávají volbu pravidla v daném kontextu.



Obrázek 18: Křížení v gramatické evoluci

Z obrázku 18 je zřejmé, že v praxi bude tímto řezem většinou oddělena a nahrazena celá skupina větví. To znamená, že byť se jedná o jednobodové křížení, je jeho efekt (pokud jde o vzájemné obměny genetického materiálu) výrazně vyšší než u jednoduchého prohazování podstromů obvyklého u genetického programování. V průměru si jedinci při křížení vymění polovinu genetického materiálu bez ohledu na jejich velikosti. To představuje velikou generativní schopnost této operace, což je klíčové pro funkci gramatické evoluce. Navíc se ukázalo, že křížení není nadměrně destruktivní, jak by se mohlo na první pohled zdát. Naopak, dochází k efektivnímu mixování bloků mezi jedinci. Jednoduché křížení je tak silným prostředkem pro globální prohledávání během celého výpočtu.

Gramatická evoluce byla testována na mnoha známých úlohách, jako jsou například symbolická regrese, symbolická integrace, hledání trigonometrických identit apod. Výsledky naznačují, že jde o velice úspěšnou metodu, která v mnoha případech předčí klasické genetické programování.

ŘEŠENÝ PŘÍKLAD

Uvažujme výrazy, ve kterých se mohou vyskytovat operace { +, –, *, / } a proměnné *X* a *Y*. Dohromady tvoří množinu terminálů *T*={+,–,*,/,*X*,*Y*}.

Množina neterminálů obsahuje symboly F = {**expr**, **op**, **var**}. Gramatika generující výrazy vypadá takto: **expr** je startovací symbol

expr	**::=**	**op expr expr**	**(0)**
	|	**var**	**(1)**
op	**::=**	**'+'**	**(0′)**
	|	**'–'**	**(1′)**
	|	**'*'**	**(2′)**
	|	**'/'**	**(3′)**
var	**::=**	**'X'**	**(0″)**
	|	**'Y'**	**(1″)**

Tato gramatika se nyní použije při dekódování lineárního chromozomu. Ten má v gramatické evoluci takový účel, že reprezentuje posloupnost pravidel tak, jak budou postupně aplikována během generování programu.

Uvažujme například, že máme rozvinout neterminál **op**, pro který si můžeme vybrat ze čtyř možných pravidel, a dále, že poslední přečtený kodon má hodnotu 17. V tomto případu by tedy byl neterminál **op** nahrazen podle pravidla (1′) znaménkem „–“, neboť 17 mod 4 = 1.

Čtení chromozomu se realizuje zleva doprava, dokud nedojde k jedné z následujících situací:

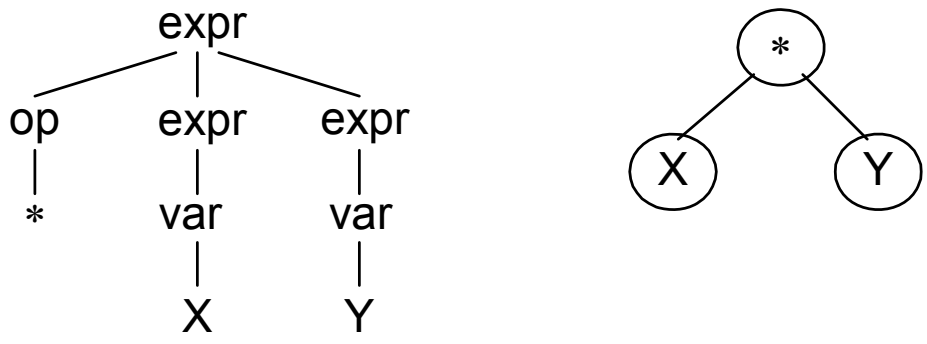
1. Program je ukončen, tj. všechny neterminály už byly přepsány na terminály. V tomto případě, kdy ještě zbývají v chromozomu nepoužité kodony, se jedná o tzv. *přespecifikovaný chromozom*. Tuto situaci nijak speciálně ošetřovat nemusíme, prostě zbytek chromozomu ignorujeme.
2. Program není ukončen, tj. ještě je třeba rozvinout jeden nebo více neterminálů, ale už byl přečten poslední kodon chromozomu. Ve tomto případě musíme dokončit odvození programu, jinak nebudeme schopni ohodnotit daného jedince. V takovém případě si můžeme pomoci například tak, že chybějící kodony budeme číst

znovu od začátku chromozomu. Aby nedošlo k nekonečnému nebo příliš dlouhému cyklení používá se omezení na počet průchodů chromozomem

Celý proces sestavení programu pomocí naší gramatiky si ukážeme na následujícím chromozomu

00101000	10100010	01000011	00000110	01111101	11100111	10010010	10001011
40	162	67	12	125	231	146	139

Generování programu začíná rozvinutím symbolu **expr**, pro který máme dvě pravidla (obrázek 19). Protože první kodon chromozomu má hodnotu 40, bude se **expr** přepisovat pomocí pravidla (0) na **op expr expr**. Následuje rozvinutí symbolu **op**. Ze čtyř možných pravidel vybereme pomocí kodonu 162 pravidlo (2') a **op** přepíšeme na „*“. Pokračujeme rozvinutím levého ze dvou zbývajících symbolů **expr**. Ze dvou možných pravidel vybereme pomocí kodonu 67 pravidlo (1) a **expr** se přepíše na **var**. Následující kodon 12 určuje, že symbol *var* bude přepsán podle pravidla (0") na **X**. Druhý symbol **expr** se opět přepíše na *var* podle pravidla (1), což je dáno kodonem 125. A konečně **var** bude přepsáno podle pravidla (1') na **Y**.



Obrázek 19: Příklad derivačního stromu jedince při použití gramatiky

Zajímavým rysem popsané reprezentace je to, že explicitně odděluje prohledávaný prostor (binární řetězce) od prostoru řešení (programy). Díky mapování genotypu na fenotyp je gramatická evoluce blíže

přirozenému konceptu genetického kódování než kterýkoliv jiný evoluční algoritmus. Toto mapování je navíc redundantní, jedno pravidlo je reprezentováno více hodnotami genu. To umožňuje tzv. *tiché mutace*, kdy drobné změny v chromozomu nezpůsobí změnu v programu.

6.2 Analytické programování

K pochopení AP je nutné se seznámit s principem ošetření diskrétního parametru jedince (Discrete Set Handling – DSH), na němž je celé AP založeno. Jak již z názvu vyplývá, jedná se o případ, kdy jednotlivé parametry jedince mohou nabývat pouze konečného počtu diskrétních hodnot. Jedinec evolučního algoritmu, který analytickému programování pomáhá nalézt nejlepší řešení, obsahuje celočíselné indexy do množiny diskrétních prvků. Evolučního procesu se tedy účastní indexy, ale hodnocení kvality jedince původní diskrétní hodnoty, které se dosadí na příslušná místa v účelové funkci [28, 44, 46, 47, 48].

Z pohledu principu evolučních algoritmů je tedy vhodnější konstatovat, že AP není samostatný evoluční algoritmus, ale spíše prostředek transformace či zobrazení z množiny základních symbolických objektů do množiny možných programů, které lze z těchto symbolických objektů zkonstruovat [28, 44, 46, 47, 48].

Za symbolické objekty považujeme podobně jako v genetickém programování či gramatické evoluci, terminály, funkce, operátory apod. Všechny tyto objekty je nutné před použitím AP rozdělit do skupin resp. množin podle počtu svých argumentů. Například operátor „+“ vyžaduje dva argumenty, funkce „*sin*“ má jeden argument, terminál „*x*“ nepotřebuje žádný argument apod. Dále je nutné vytvořit množinu, ve které se nacházejí všechny objekty dílčích skupin. Pro označení jednotlivých množin objektů se v AP používá označení GFS_{xarg} (*general function set* – obecná funkční množina), kde *x* v indexu *xarg* je číslo udávající počet argumentů u funkcí obsažených v této množině [28, 44, 46, 47, 48]. Pro množinu, která obsahuje všechny objekty je použito označení GFS_{all}. Toto rozdělení funkcí do množin má svůj hlavní

význam při zásahu proti generování patologických programů, tj. těch, které principiálně nemohou fungovat.

Pro řešení regrese, či symbolického integrování pomocí AP by definované množiny funkcí mohly vypadat například takto:

GFS_{0arg} = {*x, K*}

GFS_{1arg} = {*cos, sin, exp, log*}

GFS_{2arg} = {*+, - , * , /*}

GFS_{all} = $GFS_{2arg} \cup GFS_{1arg} \cup GFS_{0arg}$ = {*+, - , * , /, cos, sin, exp, log , x, K }*

Konstanta *K* je speciální konstanta v AP.

Na rozdíl od genetického programování či gramatické evoluce není zde třeba generovat různé číselné konstanty předem. Analytické programování sice potřebuje také číselné konstanty, ale ty se generují až v průběhu evoluce pomocí konstanty *K*, která se indexuje ještě před samotným ohodnocením účelové funkce jako K_1, K_2,… K_n. Tyto konstanty jsou poté numericky odhadnuty. Pro získání numerických hodnot konstant *K* používáme dva typy - AP_{nf} (*nf* – jako numerické či nelineární fitování (prokládání) pomocí balíčku nelineárního fitování v softwaru *Mathematica* od Wolfram Research, ve kterém je aktuálně AP dostupné – na FAI UTB ve Zlíně). Další možností, jak můžeme ohodnotit konstanty *K*, je AP_{meta}. Meta znamená metaevoluci. Pro ohodnocení konstant se používá další evoluční algoritmus. Struktura celého procesu je pak dána tímto zjednodušeným předpisem: EA_{master}>syntéza programu>indexování *K*>spuštění EA_{slave}>nastavení K_n, kde EA (podřízený - *slave*) „pracuje pod“ EA (nadřízený - *master*).

Úplně první verze analytického programování je AP_{basic}, která přistupuje k syntéze programů stejně jako kanonická GP, tzn. že společně s funkcemi a operátory byly vygenerovány náhodně i konstanty, které byly součástí syntézy analytických programů. Problém tohoto přístupu spočívá v tom, že velké množství vygenerovaných konstant přispívá ke zvýšení kardinality množiny všech možných kombinací a ke zpomalení nalezení optimálního řešení.

Další možnost spočívá v procesu, kde ohodnocování konstant není potřeba, např. při hledání vhodné trajektorie robota apod. [28, 44, 46, 47, 48].

ŘEŠENÝ PŘÍKLAD:

Postup vytváření programu je následující. Nejprve se podle celočíselných indexů, ze kterých je jedinec v procesu AP vytvořen, vybírají funkce z množiny GFS_{all}. V každém kroku se testuje, jak daleko je konec jedince. Podle této vzdálenosti se určuje, z jaké podmnožiny *GFS* budou vybírány příslušné funkce. Celý princip je dále vysvětlen na ukázkovém příkladu.

Vygenerovaný jedinec má tvar:

Jedinec = {4, 5, 7, 6, 10, 9}

Pro tvorbu programu bude využita množina GFS_{all}, tedy:

GFS_{all} = {+, - , * , /, Cos(), Sin(), Exp(), Log() , x, K }

Tabulka 3 – Ukázka setříděných terminálních a neterminálních symbolů

Funkce	+	-	*	/	*Cos*()	*Sin*()	*Exp*()	Log()	x	K
Počet argumentů	2	2	2	2	1	1	1	1	0	0

První parametr jedince je roven 4. Použijeme-li jej jako index do množiny GFS_{all}, získáme operátor „/“. Tento operátor má dva argumenty – čitatele a jmenovatele. Je tedy nutné vybrat, které další dva objekty, které dosadíme na tato dvě volná místa. Prvním argumentem bude funkce *cos,* protože se nachází na pozici 5, což je druhý parametr jedince. Tím druhým argumentem bude funkce *exp*, protože se nachází na pozici 7, což je třetí parametr jedince.

Prozatím tedy získáme výraz:

$$\frac{\cos(\ )}{\exp(\ )}$$

Je zřejmé, že výraz ještě není kompletní. Obě dvě funkce, které byly dosazeny v předchozím kroku, potřebují každá po jednom argumentu. Je tedy opět nutné zjistit, které dva argumenty to budou. Hodnoty dalších parametrů jedince se odkazují na funkci *sin* a na konstantu *K*. V tomto okamžiku má tedy výraz tento tvar:

$$\frac{\cos(\sin(\ ))}{\exp(K_1)}$$

Zbývá tedy zjistit, který argument má být dosazen do funkce *sin*. Jelikož má poslední parametr jedince hodnotu 9, bude jím *x*. Takto vypadá konečný tvar vygenerovaného výrazu:

$$\frac{\cos(\sin(x))}{\exp(K_1)}$$

Hodnota konstanty K_1 by před samotným ohodnocením účelovou funkcí musela být samozřejmě numericky nebo evolučně odhadnuta.

V rámci ukázkového příkladu zbývá vyřešit eventualitu, kdy by hrozilo, že vznikne *patologický program*. K této situaci by došlo v případě, že by měl jedinec například tento tvar:

Jedinec = {4, 5, 7, 6, 10, 4}

V závěrečném kroku by se tedy musel dosadit operátor „/“ do argumentu funkce *sin* a tím pádem by bylo potřeba určit ještě další dva argumenty. To by pochopitelně nebylo v daný okamžik možné, protože už byly použity všechny parametry jedince. Jak již bylo naznačeno, řeší se tato situace tak, že se v každém se kroku kontroluje vzdálenost od

konce jedince a podle toho se pak určuje, ze které podmnožiny se bude další objekt vybírat. V tomto případě, kdy do konce jedince zbývá jeden volný index, by poslední objekt byl vybírán z podmnožiny GFS_{0arg}. Jedinými kandidáty na volnou pozici jsou tedy terminály *x* a *K*. V ukázkovém příkladě by vybraným objektem byla konstanta *K* (sudé číslo odpovídá v tomto případě *K*, liché odpovídá *x*). Vznikl by tedy program ve tvaru:

$$\frac{\cos(\sin(K_2))}{\exp(K_1)}$$

Během chodu AP jsou samozřejmě vykonávány veškeré evoluční operace. Jedná se o selekci, křížení, mutace atp. Nutno poznamenat, že tyto jsou plně v moci použitého evolučního algoritmu, nad kterým AP pracuje. Jinými slovy, AP je pouze princip, jak zobrazit množinu vybraných objektů do množiny, kde tyto objekty tvoří program. To zda bude AP více či méně úspěšné, záleží ve hlavně na použitém evolučním algoritmu [28, 44, 46, 47, 48]

Posílené hledání je technika, která se může použít ke zlepšení výkonu analytického programování [47, 48]. Základním jejím principem je, že se částečně úspěšný program, který ještě nedosáhnul požadované vhodnosti, přidá do množiny *GFS* jako další objekt. Jestliže se v průběhu hledání nalezne program s lepší vhodností (ale stále ještě nedosáhnul uživatelsky stanovené hranice), než jakou má poslední vložený program, pak je předchozí program nahrazen tímto novým programem. Na takto přidaný program je pak nahlíženo jako na objekt s 0 argumenty – patří tedy do podmnožiny GFS_{0arg}.

Kontrolní otázky:

1. Jak jsou definovány operátory křížení a mutace v genetickém programování?
2. Jak jsou definovány operátory křížení a mutace v gramatické evoluci?
3. Jaký je základní princip evolučního programování?

Úkoly k textu:

1. Nalezněte na webu další aplikace genetického programování.
2. Nalezněte na webu další aplikace gramatické evoluce.

Úkoly k zamyšlení:

Navrhněte úlohu, kterou lze řešit za použití analytického programování. Zvolte algoritmus jejího řešení. Zapište jej formálně a pokuste se jej implementovat.

Korespondenční úkol:

1. Vytvořte počítačový program pro realizaci algoritmu genetického programování.
2. Vytvořte počítačový program pro realizaci algoritmu gramatické evoluce.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základními principy symbolická regrese, která k procesu optimalizace potřebuje evoluční algoritmy. Analytické programování lze chápat jako alternativní přístup vzhledem ke genetickému programování a gramatické evoluci, jež jsou nástrojem symbolické regrese

Pojmy k zapamatování

- genetické programování,
- stromová struktura,
- gramatická evoluce,
- analytické programování,
- DSH (Discrete Set Handling),
- GFS (General Function Set).

7 Úvod do neuronových sítí

V této kapitole se dozvíte:

- Jaká je geometrická interpretace funkce neuronu?
- Co specifikuje organizační, aktivní a adaptivní dynamika neuronové sítě?
- Jaký je princip Hebbova učení?

Po jejím prostudování byste měli být schopni:

- Popsat biologický neuron.
- Popsat formální neuron.
- Popsat topologii neuronové sítě.
- Vysvětlit princip Hebbova učení.

Klíčová slova této kapitoly:

Biologický neuron, formální neuron, neuronová síť, aktivační funkce, Hebbovo učení.

Průvodce studiem

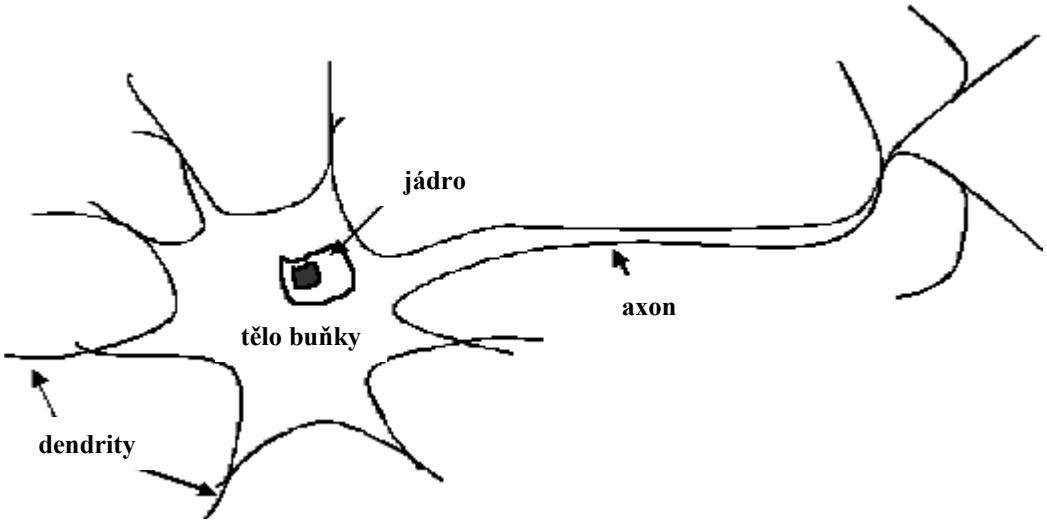
V této úvodní kapitole se stručně seznámíte se základním matematickým modelem biologického neuronu, tj. formálním neuronem. Z tohoto modelu budeme dále vycházet, a proto je nutné, abyste jeho pochopení věnovali zvýšenou pozornost. Dále si popíšeme topologii neuronové sítě a seznámíte se s Hebbovým adaptačním pravidlem, které budeme demonstrovat na řešeném příkladu.

7.1 Biologický neuron

Nervová soustava člověka je velmi složitý systém, který je stále předmětem zkoumání. Uvedené velmi zjednodušené neurofyziologické

principy nám však v dostatečné míře stačí k formulaci matematického modelu neuronové sítě. Základním stavebním funkčním prvkem nervové soustavy je nervová buňka, neuron. Neurony jsou samostatné specializované buňky, určené k přenosu, zpracování a uchování informací, které jsou nutné pro realizaci životních funkcí organismu. Struktura neuronu je schématicky znázorněna na obrázku 20.

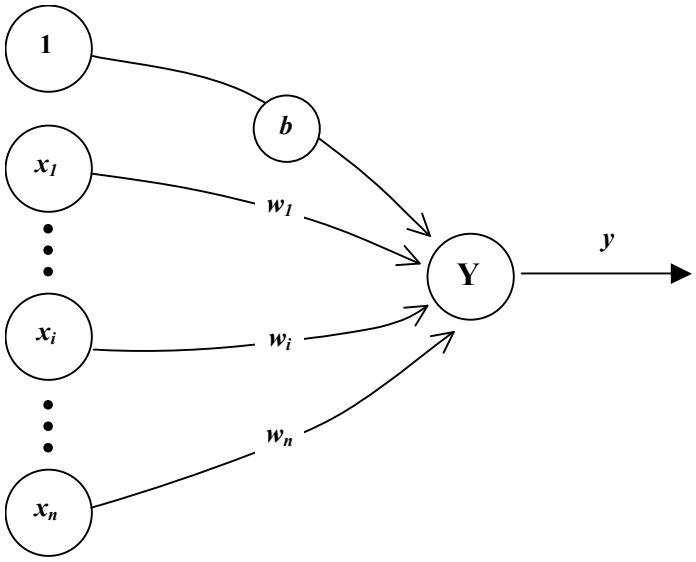
Neuron je přizpůsoben pro přenos signálů tak, že kromě vlastního těla (somatu), má i vstupní a výstupní přenosové kanály: dendrity a axon. Z axonu odbočuje řada větví (terminálů), zakončených blánou, která se převážně stýká s výběžky (trny), dendritů jiných neuronů. K přenosu informace pak slouží unikátní mezineuronové rozhraní, synapse. Míra synaptické propustnosti je nositelem všech význačných informací během celého života organismu. Z funkčního hlediska lze synapse rozdělit na excitační, které umožňují rozšíření vzruchu v nervové soustavě a na inhibiční, které způsobují jeho útlum. Paměťová stopa v nervové soustavě vzniká pravděpodobně zakódováním synaptických vazeb na cestě mezi receptorem (čidlem orgánu) a efektorem (výkonným orgánem). Šíření informace je umožněno tím, že soma i axon jsou obaleny membránou, která má schopnost za jistých okolností generovat elektrické impulsy. Tyto impulsy jsou z axonu přenášeny na dendrity jiných neuronů synaptickými branami, které svou propustností určují intenzitu podráždění dalších neuronů. Takto podrážděné neurony při dosažení určité hraniční meze, tzv. prahu, samy generují impuls a zajišťují tak šíření příslušné informace. Po každém průchodu signálu se synaptická propustnost mění, což je předpokladem paměťové schopnosti neuronů. Také propojení neuronů prodělává během života organismu svůj vývoj: v průběhu učení se vytváří nové paměťové stopy nebo při zapomínání se synaptické spoje přerušují.



Obrázek 20: Biologický neuron

7.2 Formální neuron

Základem matematického modelu neuronové sítě je *formální neuron*. Jeho struktura je schematicky zobrazena [4] na obrázku 21. Formální neuron *Y* (dále jen neuron) má obecně *n* reálných *vstupů,* které modelují dendrity a určují vstupní vektor $\boldsymbol{x} = (x_1, \ldots, x_n)$. Tyto vstupy jsou ohodnoceny reálnými *synaptickými váhami* tvořícími vektor $\boldsymbol{w} = (w_1, \ldots, w_n)$. Ve shodě s neurofyziologickou motivací mohou být synaptické váhy i záporné, čímž se vyjadřuje jejich inhibiční charakter.



Obrázek 21: Formální neuron s biasem

Vážená suma vstupních hodnot *y_in* představuje *vnitřní potenciál* neuronu *Y*:

$$y\_in = \sum_{i=1}^{n} w_i x_i \qquad (27)$$

Bias může být do vztahu včleněn přidáním komponent $x_0 = 1$ k vektoru $\boldsymbol{x}$, tj. $\boldsymbol{x} = (1, x_1, x_2, ..., x_n)$. Bias je dále zpracováván jako jakákoliv jiná váha, tj. $w_0 = b$. Vstup do neuronu Y je pak dán následujícím vztahem:

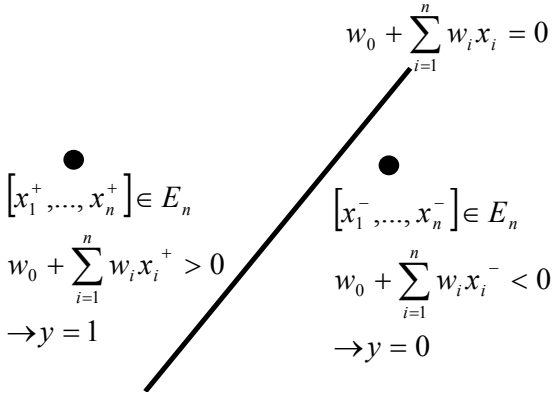
$$y\_in = \sum_{i=0}^{n} w_i x_i = w_0 + \sum_{i=1}^{n} w_i x_i = b + \sum_{i=1}^{n} w_i x_i \qquad (28)$$

Hodnota vnitřního potenciálu *y_in* po dosažení hodnoty *b* indukuje *výstup (stav) y* neuronu *Y*, který modeluje elektrický impuls axonu. Nelineární nárůst výstupní hodnoty $y = f(y\_in)$ při dosažení hodnoty potenciálu *b* je dán *aktivační* (*přenosovou*) *funkcí f*. Nejjednodušším typem přenosové funkce je *ostrá nelinearita*, která má pro neuron *Y* tvar:

$$f(y\_in) = \begin{cases} 1 \text{ pokud } y\_in \geq 0 \\ 0 \text{ pokud } y\_in < 0 \end{cases} \qquad (29)$$

Pokud místo váhového biasu, pracujeme s fixním prahem θ pro aktivační funkci, pak má přenosové funkce *ostrá nelinearita* pro neuron *Y* tvar:

$$f(y\_in) = \begin{cases} 1 \text{ pokud } y\_in \geq \theta \\ 0 \text{ pokud } y\_in < \theta \end{cases}, \qquad (30)$$



Obrázek 22: Geometrická interpretace funkce neuronu

K lepšímu pochopení funkce jednoho neuronu nám pomůže geometrická představa načrtnutá na obrázku 22. Vstupy neuronu chápeme jako souřadnice bodu v *n - rozměrném Euklidovském vstupním prostoru* E_n. V tomto prostoru má rovnice nadroviny (v E_2

přímka, v E_3 rovina) tvar: $w_0 + \sum_{i=1}^{n} w_i x_i = 0.$ Tato nadrovina dělí vstupní prostor na dva poloprostory. Souřadnice bodů [x_1^+,...,x_n^+], které leží v jednom poloprostoru, splňují následující nerovnost: $w_0 + \sum_{i=1}^{n} w_i x_i^+ > 0.$ Body [x_1^-,...,x_n^-] z druhého poloprostoru pak vyhovují relaci s opačným relačním znaménkem: $w_0 + \sum_{i=1}^{n} w_i x_i^- < 0.$ Synaptické váhy $\boldsymbol{w} = (w_0,..., w_n)$ chápeme jako koeficienty této nadroviny. Neuron *Y* klasifikuje, ve kterém z obou poloprostorů určených nadrovinou leží bod, jehož souřadnice jsou na vstupu, tj. realizuje *dichotomii* vstupního prostoru. Neuron *Y* je *aktivní*, je-li jeho stav *y = 1* a *pasivní*, pokud je jeho stav *y = 0*.

7.3 Neuronová síť

Každá neuronová síť je složena z formálních neuronů, které jsou vzájemně propojeny tak, že výstup jednoho neuronu je vstupem do (obecně i více) neuronů. Obdobně jsou terminály axonu biologického neuronu přes synaptické vazby spojeny s dendrity jiných neuronů. Počet neuronů a jejich vzájemné propojení v síti určuje *architekturu* (*topologii*) neuronové sítě. Z hlediska využití rozlišujeme v síti *vstupní*, *pracovní* (*skryté, mezilehlé, vnitřní*) a *výstupní neurony*. Šíření a zpracování informace v síti je umožněno změnou stavů neuronů ležících na cestě mezi vstupními a výstupními neurony. *Stavy* všech neuronů v síti určují *stav neuronové sítě* a *synaptické váhy* všech spojů představují *konfiguraci neuronové sítě*.

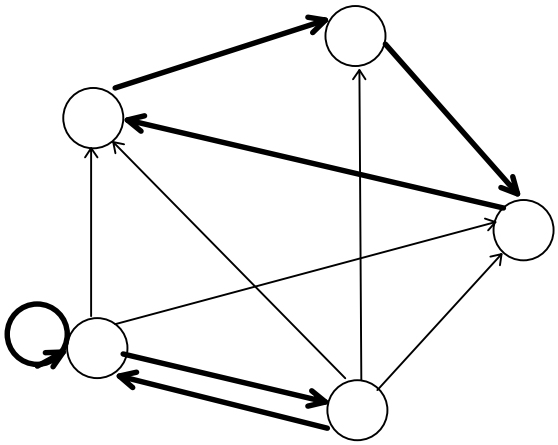
Neuronová síť se v čase vyvíjí, mění se stav neuronů, adaptují se váhy. V souvislosti se změnou těchto charakteristik v čase je účelné rozdělit celkovou dynamiku neuronové sítě do tří dynamik a uvažovat pak tři režimy práce sítě: *organizační* (změna topologie), *aktivní* (změna stavu) a *adaptivní* (změna konfigurace). Uvedené dynamiky neuronové sítě jsou obvykle zadány počátečním stavem a matematickou rovnicí, resp. pravidlem, které určuje vývoj příslušné charakteristiky sítě (topologie,

stav, konfigurace) v čase. Změny, které se řídí těmito zákonitostmi probíhají v odpovídajících režimech práce neuronové sítě. Konkretizací jednotlivých dynamik pak obdržíme různé modely neuronových sítí vhodné pro řešení různých tříd úloh.

7.3.1 **Organizační dynamika**

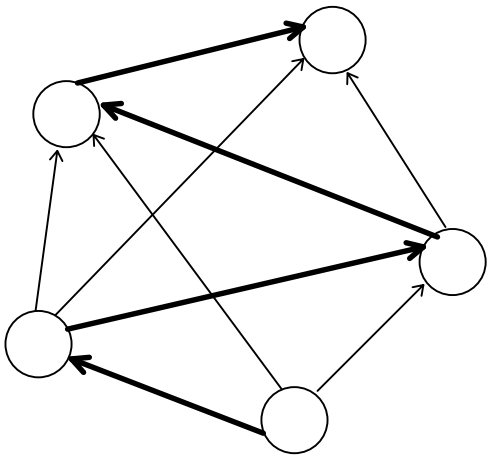
Organizační dynamika specifikuje architekturu neuronové sítě a její případnou změnu. Změna topologie se většinou uplatňuje v rámci adaptivního režimu tak, že síť je v případě potřeby rozšířena o další neurony a příslušné spoje. Avšak organizační dynamika převážně předpokládá pevnou architekturu neuronové sítě (tj. takovou architekturu, která se již v čase nemění). Rozlišujeme dva typy architektury: *cyklická* (*rekurentní*) a *acyklická* (*dopředná*) síť.

V případě *cyklické* topologie existuje v síti skupina neuronů, která je spojena v kruhu (tzv. *cyklus*). To znamená, že v této skupině neuronů je výstup prvního neuronu vstupem druhého neuronu, jehož výstup je opět vstupem třetího neuronu atd., až výstup posledního neuronu v této skupině je vstupem prvního neuronu. Nejjednodušším příkladem cyklu je *zpětná vazba* neuronu, jehož výstup je zároveň jeho vstupem. Nejvíce cyklů je v *úplné topologii* cyklické neuronové sítě, kde výstup libovolného neuronu je vstupem každého neuronu. Příklad obecné cyklické neuronové sítě je uveden na obrázku 23, kde jsou vyznačeny všechny možné cykly.



Obrázek 23: Příklad cyklické architektury

V ***acyklických*** sítích naopak cyklus neexistuje a všechny cesty vedou jedním směrem. Příklad acyklické sítě je na obrázku 24, kde je vyznačena nejdelší cesta.



Obrázek 24: Příklad acyklické architektury

U acyklické neuronové sítě lze neurony vždy (disjunktně) rozdělit do *vrstev*, které jsou uspořádány (např. nad sebou) tak, že spoje mezi neurony vedou jen z nižších vrstev do vrstev vyšších (obecně však mohou přeskočit jednu nebo i více vrstev). Speciálním případem takové architektury je ***vícevrstvá neuronová síť***. V této síti je první (dolní), tzv. *vstupní* vrstva tvořena vstupními neurony a poslední (horní), tzv. *výstupní* vrstva je složena z výstupních neuronů. Ostatní, tzv. *skryté* (*mezilehlé, vnitřní*) vrstvy jsou složeny ze skrytých (*vnitřních*) neuronů. V topologii vícevrstvé sítě jsou neurony jedné vrstvy spojeny se všemi neurony bezprostředně následující vrstvy. Proto architekturu takové sítě lze zadat jen počty neuronů v jednotlivých vrstvách (oddělených pomlčkou), v pořadí od vstupní k výstupní vrstvě. Také cesta v takové síti vede směrem od vstupní vrstvy k výstupní, přičemž obsahuje po jednom neuronu z každé vrstvy. Příklad architektury třívrstvé neuronové sítě je na obrázku 28.

7.3.2 **Aktivní dynamika**

Aktivní dynamika specifikuje *počáteční stav sítě* a způsob jeho změny v čase při pevné topologii a konfiguraci. V aktivním režimu se na začátku nastaví stavy vstupních neuronů na tzv. *vstup sítě* a zbylé neurony jsou v uvedeném počátečním stavu. Všechny možné vstupy, resp. stavy sítě, tvoří *vstupní prostor*, resp. *stavový prostor*, neuronové sítě. Po inicializaci stavu sítě probíhá vlastní výpočet. Obecně se předpokládá spojitý vývoj stavu neuronové sítě v čase a hovoří se o *spojitém modelu*, kdy stav sítě je spojitou funkcí času, která je obvykle v aktivní dynamice zadána diferenciální rovnicí. Většinou se však předpokládá diskrétní čas, tj. na počátku se síť nachází v čase 0 a stav sítě se mění jen v čase 1, 2, 3, ... V každém takové časové kroku je podle daného pravidla aktivní dynamiky vybrán jeden neuron (tzv. *sekvenční výpočet*) nebo více neuronů (tzv. *paralelní výpočet*), které *aktualizují* (mění) svůj stav na základě svých vstupů, tj. stavů sousedních neuronů, jejichž výstupy jsou vstupy aktualizovaných neuronů. Podle toho, zda neurony mění svůj stav nezávisle na sobě nebo je jejich aktualizace řízena centrálně, rozlišujeme *synchronní* a *asynchronní* modely neuronových sítí. Stav výstupních neuronů, který se obecně mění v čase, je *výstupem* neuronové sítě (tj. výsledkem výpočtu). Obvykle se však uvažuje taková aktivní dynamika, že výstup sítě je po nějakém čase konstantní a neuronová síť tak v aktivním režimu realizuje nějakou funkci na vstupním prostoru, tj. ke každému vstupu sítě vypočítá právě jeden výstup. Tato tzv. *funkce neuronové sítě* je dána aktivní dynamikou, jejíž rovnice parametricky závisí na topologii a konfiguraci, které se v aktivním režimu, jak již bylo uvedeno, nemění. Je zřejmé, že v aktivním režimu se neuronová síť využívá k vlastním výpočtům.

Aktivní dynamika neuronové sítě také určuje funkci jednoho neuronu, jejíž předpis (matematický vzorec) je většinou pro všechny (nevstupní) neurony v síti stejný (tzv. *homogenní* neuronová síť). Můžeme se setkat s následujícími *aktivačními funkcemi*:

$$f(x) = \begin{cases} 1 & \text{pokud } x \geq 1 \\ 0 & \text{pokud } x < 0 \end{cases} \quad \text{ostrá nelinearita}$$

$$f(x) = \begin{cases} 1 & x \geq 1 \\ x & 0 \leq x \leq 1 \\ 0 & x < 0 \end{cases} \quad \text{saturovaná lineární funkce}$$

$$f(x) = \frac{1}{1+e^{-x}} \quad \text{standardní (logistická) sigmoida}$$

$$f(x) = \frac{1-e^{-x}}{1+e^{-x}} \quad \text{hyperbolický tangens}$$

7.3.3 **Adaptivní dynamika**

Adaptivní dynamika neuronové sítě specifikuje *počáteční konfiguraci* sítě a způsob, jakým se mění váhové hodnoty na spojeních mezi jednotlivými neurony v čase. Všechny možné konfigurace sítě tvoří *váhový prostor* neuronové sítě. V adaptivním režimu se tedy na začátku nastaví váhy všech spojů v síti na počáteční konfiguraci (např. náhodně). Po inicializaci konfigurace sítě probíhá vlastní adaptace. Podobně jako v aktivní dynamice se obecně uvažuje spojitý model se spojitým vývojem konfigurace neuronové sítě v čase, kdy váhy sítě jsou (spojitou) funkcí času, která je obvykle v adaptivní dynamice zadána diferenciální rovnicí. Většinou se však předpokládá diskrétní čas adaptace. Víme, že funkce sítě v aktivním režimu závisí na konfiguraci. Cílem adaptace je nalézt takovou konfiguraci sítě ve váhovém prostoru, která by v aktivním režimu realizovala předepsanou funkci. Jestliže aktivní režim sítě se využívá k vlastnímu výpočtu funkce sítě pro daný vstup, pak adaptivní režim slouží k *učení* („programování“) této funkce.

Požadovaná funkce sítě je obvykle zadána tzv. *tréninkovou množinou* (*posloupností*) dvojic vstup/výstup sítě (tzv. *tréninkový vzor*). Způsobu adaptace, kdy požadované chování sítě modeluje učitel, který pro vzorové vstupy sítě informuje adaptivní mechanismus o správném

výstupu sítě, se nazývá *učení s učitelem* (supervised learning). Někdy učitel hodnotí kvalitu momentální skutečné odpovědi (výstupu) sítě pro daný vzorový vstup pomocí známky, která je zadána místo požadované hodnoty výstupu sítě (tzv. *klasifikované učení*). Jiným typem adaptace je tzv. *samoorganizace*. V tomto případě tréninková množina obsahuje jen vstupy sítě. To modeluje situaci, kdy není k dispozici učitel, proto se tomuto způsobu adaptace také říká *učení bez učitele*. Neuronová síť v adaptivním režimu sama organizuje tréninkové vzory (např. do shluků) a odhaluje jejich souborné vlastnosti.

7.4 Hebbovo učení

Hebbovo učení je založeno na myšlence, že váhové hodnoty na spojení mezi dvěma neurony, které jsou současně ve stavu „*on*“, budou narůstat a naopak, tj. váhové hodnoty na spojení mezi dvěma neurony, které jsou současně ve stavu „*off*“, se budou zmenšovat. Změna synaptické váhy spoje mezi dvěma neurony je úměrná jejich souhlasné aktivitě, tj. součinu jejich stavů. Donald Hebb tímto způsobem vysvětloval vznik podmíněných reflexů. Uvažujme jednovrstvou neuronovou síť, ve které jsou všechny vstupní neurony propojeny s jediným výstupní neuronem *Y* (viz. obr. 21), ale ne již navzájem mezi sebou. Pokud jsou složky vstupního vektoru $\boldsymbol{x} = (x_1, \ldots, x_n)$ reprezentovány v bipolární formě, lze složky příslušného váhového vektoru $\boldsymbol{w} = (w_1, \ldots, w_n)$ aktualizovat následovně:

$$w_i(new) = w_i(old) + x_i y, \qquad (31)$$

kde *y* je výstup z neuronu *Y*.

Hebbovo adaptační pravidlo [8]

Krok 0. Inicializace všech vah:

$w_i = 0$ *(i = 1* až *n)*

Krok 1. Pro každý vzor - tréninkový pár, tj. vstupní vektor (**s**) a příslušný výstup (**t**), opakovat následující kroky (2 až 4).

Krok 2. Aktivovat vstupní neurony:

$x_i = s_i$ *(i = 1 až n).*

Krok 3. Aktivovat výstupní neuron:

$y = t$.

Krok 4. Aktualizovat váhy podle

$w_i(new) = w_i(old) + x_i y$ *(i = 1 až n).*

Aktualizovat biasy podle

$b(new) = b(old) + y$.

Bias lze zapsat také jako váhovou hodnotu přiřazenou výstupu z neuronu, jehož aktivace má vždy hodnotu 1. Aktualizace váhových hodnot může být také vyjádřena ve vektorové formě jako

$$\boldsymbol{w}(new) = \boldsymbol{w}(old) + \boldsymbol{x}y. \tag{32}$$

Váhový přírůstek lze zapsat ve tvaru

$$\Delta\boldsymbol{w} = \boldsymbol{x}y \tag{33}$$

a potom

$$\boldsymbol{w}(new) = \boldsymbol{w}(old) + \Delta\boldsymbol{w}. \tag{34}$$

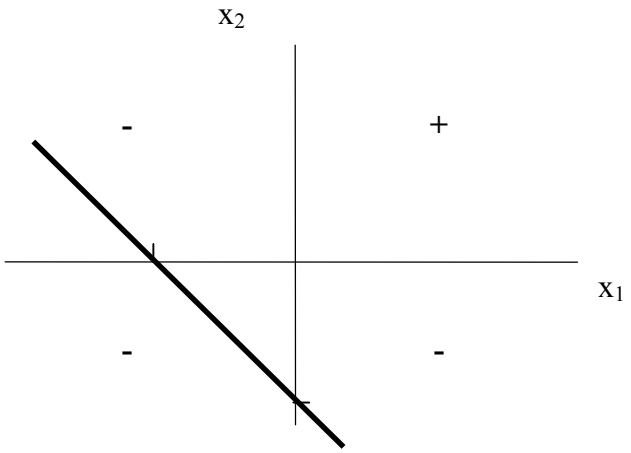
Výše uvedený algoritmus je pouze jedním z mnoha způsobu implementace Hebbova pravidla učení. Tento algoritmus vyžaduje jen jeden průchod tréninkovou množinou. Existují však i jiné ekvivalentní metody nalezení vhodných váhových hodnot, které jsou popsány dále.

Příklad:

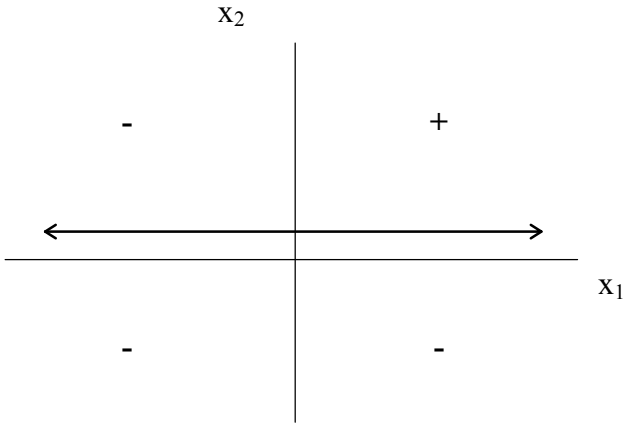
Hebbovovo pravidlo učení pro logickou funkci „AND“ v bipolární reprezentaci můžeme zapsat následovně:

čas	**VSTUP**		**POŽADOVANÝ VÝSTUP**	**PŘÍRUSTKY VAH**			**VÁHOVÉ HODNOTY**		
	x_1	x_2	t	Δw_1	Δw_2	Δb	w_1	w_2	b
0							0	0	0
1	1	1	1	1	1	1	1	1	1
2	1	-1	-1	-1	1	-1	0	2	0
3	-1	1	-1	1	-1	-1	1	1	-1
4	-1	-1	-1	1	1	-1	2	2	-2

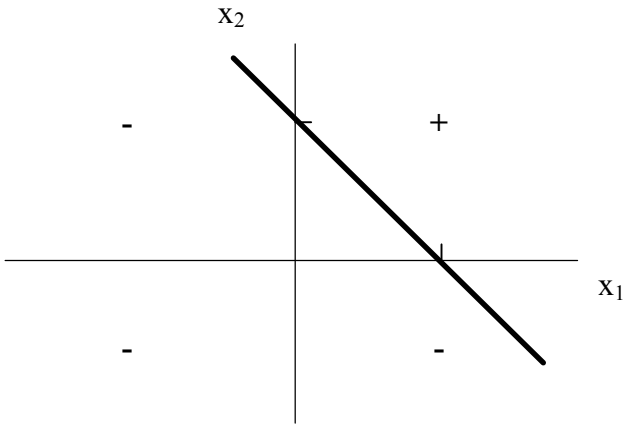
Grafický postup řešení je uveden na obrázku 25.



1. tréninkový vzor: rovnice přímky: $x_2 = -x_1 - 1$



2. tréninkový vzor: rovnice přímky: $x_2 = 0.$



3. a 4. tréninkový vzor: rovnice přímky: $x_2=-x_1+1$

Obrázek 25: Hebbovovo pravidlo učení pro logickou funkci „AND“ v bipolární reprezentaci.

Úkoly k zamyšlení:

Vytvořte geometrickou interpretaci funkce jednoho neuronu ve 2 - rozměrném Euklidovském prostoru. Vstupy neuronu jsou souřadnice bodu v E_2.

Korespondenční úkol:

Realizujte Hebbovo pravidlo učení pro logickou funkci „OR“ v bipolární reprezentaci.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základním matematickým modelem biologického neuronu, tj. formálním neuronem. Dále zde byl uveden popis topologie neuronové sítě a seznámili jste se i s Hebbovým adaptačním pravidlem, které bylo demonstrováno na řešeném příkladu.

Pojmy k zapamatování

- stav neuronu,
- vnitřní potenciál neuronu, s
- synaptické váhy,
- aktivační(přenosová) funkce,
- architektura (topologie) neuronové sítě,
- organizační, aktivní a adaptivní dynamika neuronové sítě,
- Hebbovo učení,

8 Základní modely neuronových sítí

V této kapitole se dozvíte:

- Jaký je princip adaptačního algoritmu perceptronu.
- Jaké jsou principy soutěžní strategie učení a procesu shlukování.

Po jejím prostudování byste měli být schopni:

- Charakterizovat perceptron, což je nejjednodušší neuronová síť s jedním pracovním neuronem.
- Charakterizovat modely neuronových sítí, které využívají soutěžní strategie učení.

Klíčová slova této kapitoly:

Perceptron, učení s učitelem, Kohonenovy samoorganizační mapy, soutěžní strategie učení.

Průvodce studiem

Perceptron je nejjednodušší neuronová síť s jedním pracovním neuronem a na jeho adaptačním algoritmu si vysvětlíme proces učení s učitelem. Dále se budeme věnovat modelům neuronových sítí, které využívají soutěžní strategie učení (competitive learning). Společným principem těchto modelů je, že výstupní neurony sítě spolu soutěží o to, který z nich bude aktivní. Na rozdíl od jiných učících principů (např. Hebbovo učení) je tedy v určitém čase aktivní vždy jen jeden neuron.

8.1 Perceptron

Autorem této nejjednodušší neuronové sítě je Frank Rosenblatt (r. 1957). Za typický perceptron je považována jednoduchá neuronová

síť s *n* vstupy ($x_1, x_2, ..., x_n$) a jedním pracovním neuronem spojeným se všemi svými vstupy. Každému takovému spojení je přiřazena váhová hodnota ($w_1, w_2, ..., w_n$). Signál přenášený vstupními neurony je buď binární (tj. má hodnotu 0 nebo 1), nebo bipolární (tj. má hodnotu -1, 0 nebo 1). Výstupem z perceptronu je pak $y = f(y\_in)$. Aktivační funkce *f* má tvar:

$$f(y\_in) = \begin{cases} 1 & pokud\ y\_in > \theta \\ 0 & pokud - \theta \le y\_in \le \theta \\ -1 & pokud\ y\_in < -\theta \end{cases}, \qquad (35)$$

kde θ je libovolný, ale pevný práh aktivační funkce *f.*

Váhové hodnoty jsou adaptovány podle adaptačního pravidla perceptronu tak, aby diference mezi skutečným a požadovaným výstupem byla co nejmenší. Adaptační pravidlo perceptronu je mnohem silnější než Hebbovo adaptační pravidlo.

Adaptační algoritmus perceptronu [7]

Krok 0. Inicializace vah w_i *(i = 1* až *n)* a biasu *b malými náhodnými čísly.*

Přiřazení inicializační hodnoty koeficientu učení α *(0<α≤1).*

Krok 1. Dokud není splněna podmínka ukončení výpočtu, opakovat kroky (2 až 6).

Krok 2. Pro každý tréninkový pár **s:**t (tj. vstupní vektor **s** a příslušný výstup t**)**, provádět kroky (3 až 5).

Krok 3. Aktivuj vstupní neurony:

$x_i = s_i$.

Krok 4 Vypočítej skutečnou hodnotu na výstupu:

$$y\_in = b + \sum_i x_i w_i ;.$$

$$y = \begin{cases} 1 & pokud\ y\_in > \theta \\ 0 & pokud - \theta \le y\_in \le \theta \\ -1 & pokud\ y\_in < -\theta \end{cases}$$

Krok 5 Aktualizuj váhové hodnoty a bias pro daný vzor

jestliže $y \neq t$,

$w_i(new)=w_i(old)+\alpha t x_i$ ($i=1$ až n).

$b(new) = b(old) + \alpha t$.

jinak

$w_i(new) = w_i(old)$

$b(new) = b(old)$

Krok 6. Podmínka ukončení:

jestliže ve *2. kroku* již nenastává žádná změna váhových hodnot, stop; jinak, pokračovat.

Aktualizaci podléhají pouze ty váhové hodnoty, které neprodukují požadovaný výstup *y*. To znamená, že čím více tréninkových vzorů má korektní výstupy, tím méně je potřeba času k jejich tréninku. Práh aktivační funkce je pevná nezáporná hodnota θ. Tvar aktivační funkce pracovního neuronu je takový, že umožňuje vznik pásu pevné šířky (určené hodnotou θ) oddělujícího oblast pozitivní odezvy od oblasti negativní odezvy na vstupní signál. Předcházející analýza o zaměnitelnosti prahu a biasu zde nemá uplatnění, protože změna θ mění šířku oblasti, ne však její umístění. Místo jedné separující přímky tedy máme pás určený dvěma rovnoběžnými přímkami:

1. Přímka separující oblast pozitivní odezvy od oblasti nulové odezvy na vstupní signál; tato hraniční přímka má tvar:

$$w_1 x_1 + w_2 x_2 + b > \theta. \qquad (36)$$

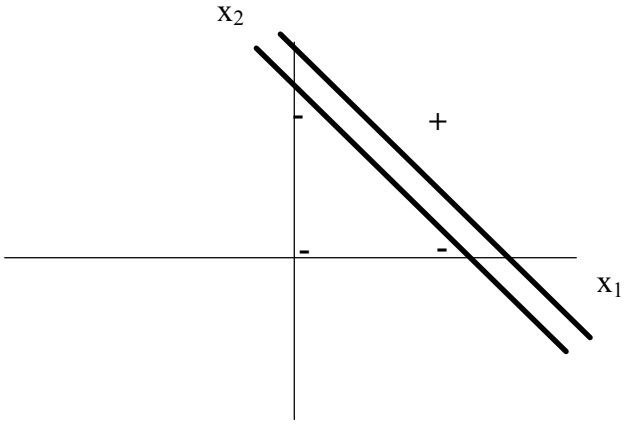
2. Přímka separující oblast nulové odezvy od oblasti negativní odezvy na vstupní signál;. tato hraniční přímka má tvar:

$$w_1 x_1 + w_2 x_2 + b < -\theta. \qquad (37)$$

Příklad:

Adaptační algoritmus perceptronu pro logickou funkci „AND“: binární vstupní hodnoty, bipolární výstupní hodnoty. Pro jednoduchost předpokládejme, že $\theta = 0,2$ a $\alpha = 1$.

čas	**VSTUP**		**VÝSTUP**			**PŘÍRUSTKY VAH**			**VÁHOVÉ HODNOTY**		
	x_1	x_2	y_in	y	t	Δw_1	Δw_2	Δb	w_1	w_2	b
0									0	0	0
1	1	1	0	0	1	1	1	1	1	1	1
2	1	0	2	1	-1	-1	0	-1	0	1	0
3	0	1	1	1	-1	0	-1	-1	0	0	-1
4	0	0	-1	-1	-1	1	0	0	0	0	-1
. . .											
37	1	1	1	1	1	0	0	0	2	3	-4
38	1	0	-2	-1	-1	0	0	0	2	3	-4
39	0	1	-1	-1	-1	0	0	0	2	3	-4
40	0	0	-4	-1	-1	0	0	0	2	3	-4



Obrázek 26: Hraniční pás pro logickou funkci „AND“ po adaptaci algoritmem perceptronu.

Oblast kladné odezvy je dána body, pro které platí: $2x_1 + 3x_2 - 4 > 0,2$

a hraniční přímka této oblasti má tvar: $x_2 = -\frac{2}{3}x_1 + \frac{7}{5}$.

Oblast záporné odezvy je dána body, pro které platí: $2x_1 + 3x_2 - 4 < -0,2$.

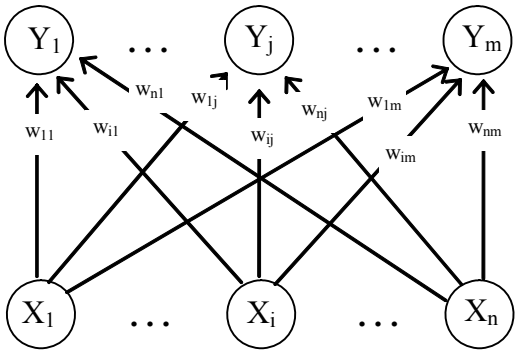
a hraniční přímka této oblasti má tvar: $x_2 = -\frac{2}{3}x_1 + \frac{19}{15}$.

8.2 Kohonenovy samoorganizační mapy

Tato neuronová síť (angl. *Self -Organizing Map*) byla poprvé popsána v roce 1982. Je nejdůležitější architekturou vycházející ze strategie soutěžního učení (tj. *učení bez učitele*). Základním principem učícího procesu je vytvoření množiny reprezentantů mající stejné pravděpodobnosti výběru. Přesněji, hledáme takové reprezentanty, pro které platí: vybereme-li náhodný vstupní vektor z rozdělení pravděpodobnosti odpovídající rozdělení tréninkové množiny, bude mít každý takový reprezentant přiřazenu pravděpodobnost, která je mu nejblíže. Algoritmus tedy nemá informace o požadovaných aktivitách výstupních neuronů v průběhu adaptace, ale adaptace vah odráží statistické vlastnosti trénovací množiny. Jsou-li si tedy dva libovolné vzory blízké ve vstupním prostoru způsobují v síti odezvu na neuronech, které jsou si fyzicky blízké ve výstupním prostoru. Hlavní ideou těchto neuronových sítí je nalézt prostorovou reprezentaci složitých datových struktur. Mnohodimenzionální data se tímto způsobem zobrazují v daleko jednodušším prostoru. Uvedená vlastnost je typická i pro skutečný mozek, kde například jeden konec sluchové části mozkové kůry reaguje na nízké frekvence, zatímco opačný konec reaguje na frekvence vysoké.

Jedná se o dvouvrstvou síť s úplným propojením neuronů mezi vrstvami. Výstupní neurony jsou navíc uspořádány do nějaké topologické struktury, nejčastěji to bývá dvojrozměrná mřížka nebo jednorozměrná řada jednotek. Tato topologická struktura určuje, které neurony spolu v síti sousedí (pro adaptační proces je to nezbytné). Pro adaptační proces je rovněž důležité zavést pojem *okolí J* výstupního neuronu (*j**) o *poloměru* (velikosti) *R*, což je množina všech neuronů ($j \in J$), jejichž vzdálenost v síti je od daného neuronu (*j**) menší nebo rovna *R*: $J = \{j; d(j,j^*) \leq R\}$.

To, jak měříme vzdálenost *d(j,j*)*, je závislé na topologické struktuře výstupních neuronů. Např. pro lineární oblast obsahující *m* neuronů ve výstupní vrstvě platí pro všechny $j \in J$: $\max(1, J - R) \leq j \leq \min(J+R, m)$. Obecná architektura Kohonenovy samoorganizační mapy obsahující *m* neuronů ve výstupní vrstvě (tj. $Y_1,..., Y_m$) a *n* neuronů ve vstupní vrstvě (tj. $X_1,..., X_n$) je zobrazena na obrázku 27.



Obrázek 27: Kohonenova samoorganizační mapa

Princip ***adaptivní dynamiky*** je jednoduchý: Procházíme celou tréninkovou množinu a po předložení jednoho tréninkového vzoru proběhne mezi neurony sítě kompetice. Její vítěz pak spolu s neurony, které jsou v jeho okolí, změní své váhové hodnoty. Reálný parametr učení $0 < \alpha \leq 1$ určuje míru změny vah. Na počátku učení je obvykle blízký jedné a postupně se zmenšuje až na nulovou hodnotu, což zabezpečuje ukončení procesu adaptace. Rovněž i velikost okolí *R* není konstantní: na začátku adaptace je okolí obvykle velké (např. polovina velikosti sítě) a na konci učení potom zahrnuje jen jeden samotný vítězný neuron (tj. *R = 0*).

Kohonenův algoritmus [7]:

Krok 0. Inicializace všech váhových hodnot w_{ij}:
Inicializace poloměru sousedství; tj okolí (*R*).
Inicializace parametru učení (α).

Krok 1. Není-li splněna podmínka ukončení, provádět kroky (2-8).

Krok 2. Pro každý vstupní vektor $\boldsymbol{x} = (x_1,...,x_n)$ opakovat kroky 3 až 5.

Krok 3. Pro každé j ($j = 1,..., m$) vypočítat:

$$D(j) = \sum_i \left(w_{ij} - x_i\right)^2 .$$

Krok 4. Najít index J takový, že $D(J)$ je minimum.

Krok 5. Aktualizace váhových hodnot všech neuronů ($j \in J$) tvořících topologické sousedství charakterizované indexem J, tj. pro všechna i ($i = 1,..., n$) platí:

$$w_{ij}(new) = w_{ij}(old) + \alpha\left[x_i - w_{ij}(old)\right].$$

Krok 6. Aktualizace parametru učení.

Krok 7. Zmenšení poloměru R topologického sousedství.

Krok 8. Test podmínky ukončení.

Geometrický význam popsaného algoritmu je takový, že vítězný neuron *i* a všichni jeho sousedé v síti, kteří by od něj neměli být příliš vzdáleni ani ve vstupním prostoru, posunou svůj váhový vektor o určitou poměrnou vzdálenost směrem k aktuálnímu vstupu. Motivací tohoto přístupu je snaha, aby vítězný neuron, který nejlépe reprezentuje předložený vstup (je mu nejblíže), ještě více zlepšil svou relativní pozici vůči němu. Problémem vzniklým při adaptaci může být nevhodná náhodná inicializace vah, která vede k blízkým počátečním neuronům ve výstupní vrstvě a tudíž pouze jeden z nich vyhrává kompetici zatímco ostatní zůstávají nevyužity.

V ***aktivním režimu*** se pak sousedství neuronů neprojevuje: předložíme-li síti vstupní vektor, soutěží výstupní neurony o to, kdo je mu nejblíže a tento neuron se pak excituje na hodnotu rovnu jedné, zatímco výstupy ostatních neuronů jsou rovny nule. Každý neuron tak reprezentuje nějaký objekt, či třídu objektů ze vstupního prostoru: tj.

pouze jeden neuron horní vrstvy, jehož potenciál ($\Sigma w.x$) je maximální odpovídá vstupnímu vektoru ***x***. Tento neuron je navíc schopen rozpoznat celou třídu takových, podobných si vektorů. Princip „vítěz bere vše“ se realizuje tzv. *laterální inhibicí*; všechny výstupní neurony jsou navzájem propojeny laterálními vazbami, které mezi nimi přenášejí inhibiční signály. Každý výstupní neuron se pak snaží v kompetici zeslabit ostatní neurony silou úměrnou jeho potenciálu, který je tím větší, čím je neuron blíže vstupu. Výsledkem tedy je, že výstupní neuron s největším potenciálem utlumí ostatní výstupní neurony a sám zůstane aktivním.

Proces shlukování ještě jednou vysvětlíme prostřednictvím funkce hustoty pravděpodobnosti. Tato funkce reprezentuje statistický nástroj popisující rozložení dat v prostoru. Pro daný bod prostoru lze tedy stanovit pravděpodobnost, že vektor bude v daném bodu nalezen. Je-li dán vstupní prostor a funkce hustoty pravděpodobnosti, pak je možné dosáhnout takové organizace mapy, která se této funkci přibližuje (za předpokladu, že je k dispozici reprezentativní vzorek dat). Jinými slovy řečeno, pokud jsou vzory ve vstupním prostoru rozloženy podle nějaké distribuční funkce, budou váhové vektory rozloženy analogicky.

Úkoly k zamyšlení:

1. Srovnejte Hebbovo adaptační pravidlo a adaptační pravidlo perceptronu.
2. Mějme pět vektorů: (1,1,0,0), (0,0,0,1), (0,0,1,1), (1,0,0,0), (0,1,1,0). Maximální počet shluků je: m=2. Řešte příklad algoritmem adaptace Kohonenovy samoorganizační mapy (vhodně si definujte vztah pro parametr učení).

Korespondenční úkol:

1. Vytvořte počítačový program pro realizaci adaptačního algoritmu perceptronu.
2. Vytvořte počítačový program pro realizaci adaptačního algoritmu pracujícího na principu soutěžní strategie učení.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základními modely neuronových sítí. Perceptron je nejjednodušší neuronová síť s jedním pracovním neuronem a na jeho adaptačním algoritmu reprezentuje proces učení s učitelem. Kohonenovy samoorganizační mapy zase reprezentují modely neuronových sítí, které využívají soutěžní strategie učení. Důraz v této kapitole byl kladen na pochopení rozdílů mezi procesem učení s učitelem a bez učitele.

Pojmy k zapamatování

- perceptron,
- adaptační pravidlo perceptronu,
- adaptace bez učitele,
- samoorganizace,
- soutěžní strategie učení (competitive learning),
- laterální inhibice,
- sousedství,
- proces shlukování,

9 Vícevrstvá neuronová síť

V této kapitole se dozvíte:

- Jaká je topologie vícevrstvé neuronové sítě.
- Jaký je princip adaptačního algoritmu backropagation.
- Co je to overfitting (přeučení).

Po jejím prostudování byste měli být schopni:

- Charakterizovat dopředné (feedforward) šíření signálu vícevrstvou neuronovou sítí.
- Uvést princip adaptačního algoritmu backropagation.

Klíčová slova této kapitoly:

Vícevrstvá neuronová síť, backpropagation, trénovací množina, generalizace.

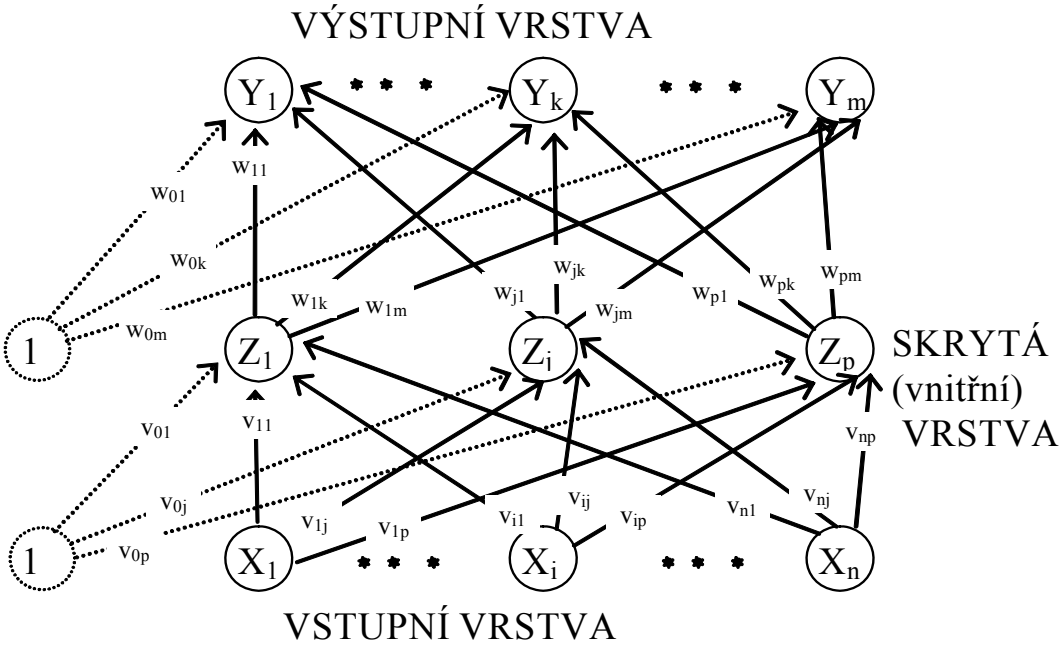
Průvodce studiem

V této kapitole rozebereme problematiku vhodné volby topologie vícevrstvé neuronové sítě, která by měla odpovídat složitosti řešeného problému. Dále se podrobně seznámíte s adaptačním algoritmem zpětného šíření chyby (backpropagation), jež je používán v přibližně 80% všech aplikací neuronových (tj. je nejrozšířenějším adaptačním algoritmem vícevrstvých neuronových sítí).

9.1 Topologie vícevrství sítě

Pravděpodobně nejrozšířenější způsob propojení neuronů se sigmoidní aktivační funkcí jsou vícevrstvé sítě. Vícevrstvá neuronová síť s jednou vnitřní vrstvou neuronů (neurony jsou označeny Z_j, $j = 1,..., p$) je

zobrazena na obrázku 28. Výstupní neurony (neurony jsou označeny Y_k, $k = 1,..., m$). Neurony ve výstupní a vnitřní vrstvě musí mít definovaný bias. Typické označení pro bias *k.* neuronu (Y_k) ve výstupní vrstvě je w_{0k}, a typické označení pro bias *j.* neuronu (Z_j) ve vnitřní vrstvě je v_{0j}. Bias (např. *j.* neuronu) odpovídá, jak již bylo dříve uvedeno, váhové hodnotě přiřazené spojení mezi daným neuronem a fiktivním neuronem, jehož aktivace je vždy *1*. Z uvedeného obrázku tedy vyplývá, že vícevrstvá neuronová síť je tvořena minimálně třemi vrstvami neuronů: vstupní, výstupní a alespoň jednou vnitřní vrstvou. Vždy mezi dvěma sousedními vrstvami se pak nachází tzv. *úplné propojení neuronů*, tedy každý neuron nižší vrstvy je spojen se všemi neurony vrstvy vyšší.



Obrázek 28: Neuronová síť s jednou vnitřní vrstvou neuronů

Velkým problémem modelu vícevrstvé neuronové sítě s adaptačním algoritmem backpropagation je (kromě minimalizace chybové funkce) volba vhodné topologie pro řešení konkrétního praktického problému. Zřídkakdy jsou podrobněji známy vztahy mezi vstupy a výstupy, které by se daly využít při návrhu speciální architektury. Většinou se používá vícevrstvá topologie s jednou nebo dvěma vnitřními vrstvami a očekává se, že učící algoritmus backpropagation zobecní příslušné vztahy z tréninkové množiny ve vahách jednotlivých spojů mezi neurony.

I v tomto případě je však potřeba vhodně volit počty neuronů ve vnitřních vrstvách. Je zřejmé, že tento problém organizační dynamiky úzce souvisí s adaptací a generalizací neuronové sítě. Architektura vícevrstvé neuronové sítě (tj. určení vhodného počtu vnitřních neuronů a jejich spojení), by měla odpovídat složitosti řešeného problému, tj. počtu tréninkových vzorů, jejich vstupů a výstupů a struktuře vztahů, které popisují. Je zřejmé, že malá síť nemůže řešit komplikovaný problém. Při učení pomocí algoritmu backpropagation se příliš malá síť obvykle zastaví v nějakém mělkém lokálním minimu a je potřeba topologii doplnit o další vnitřní neurony, aby adaptace měla větší stupeň volnosti. Na druhou stranu bohatá architektura sice při učení mnohdy umožní nalézt globální minimum chybové funkce, i když s větším počtem vah roste výpočetní náročnost adaptace. Avšak nalezená konfigurace sítě obvykle příliš zobecňuje tréninkové vzory včetně jejich nepřesností a chyb a pro nenaučené vzory dává chybné výsledky, tj. špatně generalizuje. Tomuto přesnému zapamatování tréninkové množiny bez zobecnění zákonitostí v ní obsažených se říká *přeučení* (*overfitting*) [34]. Zdá se tedy, že existuje optimální topologie, která je na jednu stranu dostatečně bohatá, aby byla schopna řešit daný problém, a na druhou stranu ne moc velká, aby správně zobecnila potřebné vztahy mezi vstupy a výstupy. Existují sice teoretické výsledky ohledně horního odhadu počtu vnitřních neuronů postačujících pro realizaci libovolné funkce z určité třídy, avšak pro praktické potřeby jsou příliš nadhodnocené, a tedy nepoužitelné. V praxi se obvykle topologie volí heuristicky, např. v první vnitřní vrstvě o něco více neuronů, než je vstupů a v druhé vrstvě aritmetický průmět mezi počtem výstupů a neuronů v první vnitřní vrstvě. Po adaptaci se v případě velké chyby sítě případně přidá, respektive při chudé generalizaci odebere několik neuronů a adaptivní režim se celý opakuje pro novou architekturu. Pro test kvality generalizace neuronové sítě se počítá chyba sítě vzhledem k tzv. *testovací množině*, což je část tréninkové množiny, která se záměrně nevyužila k adaptaci.

9.2 Standardní metoda backpropagation

Adaptační algoritmus zpětného šíření chyby (*backpropagation*) je používán v přibližně 80% všech aplikací neuronových sítí. Samotný algoritmus obsahuje tři etapy: dopředné (*feedforward*) šíření vstupního signálu tréninkového vzoru, zpětné šíření chyby a aktualizace váhových hodnot na spojeních. Během dopředného šíření signálu obdrží každý neuron ve vstupní vrstvě (X_i, $i = 1,..., n$) vstupní signál (x_i) a zprostředkuje jeho přenos ke všem neuronům vnitřní vrstvy ($Z_1, ..., Z_p$). Každý neuron ve vnitřní vrstvě vypočítá svou aktivaci (z_j) a pošle tento signál všem neuronům ve výstupní vrstvě. Každý neuron ve výstupní vrstvě vypočítá svou aktivaci (y_k), která odpovídá jeho skutečnému výstupu (*k.* neuronu) po předložení vstupního vzoru. V podstatě tímto způsobem získáme odezvu neuronové sítě na vstupní podnět daný excitací neuronů vstupní vrstvy. Takovým způsobem probíhá šíření signálů i v biologickém systému, kde vstupní vrstva může být tvořena např. zrakovými buňkami a ve výstupní vrstvě mozku jsou pak identifikovány jednotlivé objekty sledování. Otázkou pak zůstává to nejdůležitější, jakým způsobem jsou stanoveny synaptické váhy vedoucí ke korektní odezvě na vstupní signál. Proces stanovení synaptických vah je opět spjat s pojmem učení - adaptace - neuronové sítě.

Další otázkou je schopnost *generalizace* (zobecnění*)* nad naučeným materiálem, jinými slovy jak je neuronová síť schopna na základě naučeného usuzovat na jevy, které nebyly součástí učení, které však lze nějakým způsobem z naučeného odvodit.

Co je nutné k naučení neuronové sítě? Je to jednak tzv. *trénovací množina* obsahující prvky popisující řešenou problematiku a dále pak metoda, která dokáže tyto vzorky zafixovat v neuronové síti formou hodnot synaptických vah pokud možno včetně již uvedené schopnosti generalizovat. Zastavme se nejdříve u trénovací množiny. Každý vzor trénovací množiny popisuje jakým způsobem jsou excitovány neurony vstupní a výstupní vrstvy. Formálně můžeme za trénovací množinu ***T*** považovat množinu prvků (vzorů), které jsou definovány uspořádanými dvojicemi následujícím způsobem [34]:

$$T = \left\{ \left(\mathbf{x}_k, \mathbf{t}_k \right) \middle| \, \mathbf{x}_k \in \{0,1\}^n, \mathbf{t}_k \in \{0,1\}^m, k = 1, \cdots q \right\}$$

kde q počet vzorů trénovací množiny

$\boldsymbol{x}_k$ vektor excitací vstupní vrstvy tvořené n neurony

$\boldsymbol{t}_k$ vektor excitací výstupní vrstvy tvořené m neurony

Metoda, která umožňuje adaptaci neuronové sítě nad danou trénovací množinou se nazývá *backpropagation*, což v překladu znamená metodu zpětného šíření. Na rozdíl od už popsaného dopředného chodu při šíření signálu neuronové sítě spočívá tato metoda adaptace v opačném šíření informace směrem od vrstev vyšších k vrstvám nižším. Během adaptace neuronové sítě metodou backpropagation jsou srovnávány vypočítané aktivace y_k s definovanými výstupními hodnotami t_k pro každý neuron ve výstupní vrstvě a pro každý tréninkový vzor. Na základě tohoto srovnání je definována chyba neuronové sítě, pro kterou je vypočítán faktor δ_k ($k = 1, ..., m$). δ_k , jež odpovídá části chyby, která se šíří zpětně z neuronu Y_k ke všem neuronům předcházející vrstvy majícím s tímto neuronem definované spojení. Podobně lze definovat i faktor δ_j (j = *1, ..., p*), který je části chyby šířené zpětně z neuronu Z_j ke všem neuronům vstupní vrstvy, jež mají s tímto neuronem definované spojení. Úprava váhových hodnot w_{jk} na spojeních mezi neurony vnitřní a výstupní vrstvy závisí na faktoru δ_k a aktivacích z_j neuronů Z_j ve vnitřní vrstvě. Úprava váhových hodnot v_{ij} na spojeních mezi neurony vstupní a vnitřní vrstvy závisí na faktoru δ_j a aktivacích x_i neuronů X_i ve vstupní vrstvě. *Aktivační funkce* pro neuronové sítě s adaptační metodou backpropagation musí mít následující vlastnosti: musí být spojitá, diferencovatelná a monotónně neklesající. Nejčastěji používanou aktivační funkcí je proto standardní (logická) sigmoida a hyperbolický tangens.

Chyba sítě $E(\boldsymbol{w})$ je vzhledem k tréninkové množině definována jako součet parciálních chyb sítě $E_l(\boldsymbol{w})$ vzhledem k jednotlivým tréninkovým vzorům a závisí na konfiguraci sítě $\boldsymbol{w}$:

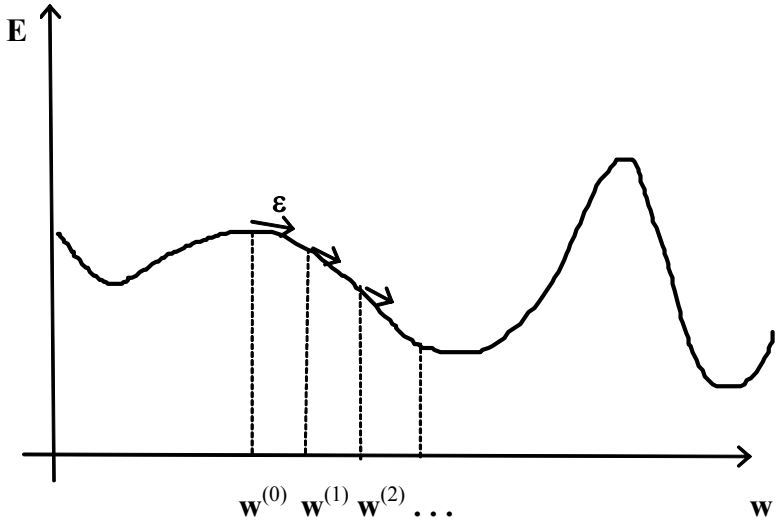
$$E(\mathbf{w}) = \sum_{l=1}^{q} E_l(\mathbf{w}). \qquad (38)$$

Parciální chyba $E_l(\boldsymbol{w})$ sítě pro *l.* tréninkový vzor (l = 1*, ...,q*) je úměrná součtu mocnin odchylek skutečných hodnot výstupu sítě pro vstup *l*-tréninkového vzoru od požadovaných hodnot výstupů u tohoto vzoru:

$$E_l(\mathbf{w}) = \frac{1}{2} \sum_{k \in Y} (y_k - t_k)^2 . \qquad (39)$$

Cílem adaptace je minimalizace chyby sítě ve váhovém prostoru. Vzhledem k tomu, že chyba sítě přímo závisí na komplikované nelineární složené funkci vícevrstvé sítě, představuje tento cíl netriviální optimalizační problém. Pro jeho řešení se v základním modelu používá nejjednodušší varianta gradientní metody, která vyžaduje diferencovatelnost chybové funkce. K lepšímu pochopení nám pomůže geometrická představa. Na obrázku 29 je schematicky znázorněna chybová funkce $E(\boldsymbol{w})$ tak, že konfigurace, která představuje mnohorozměrný vektor vah $\boldsymbol{w}$, se promítá na osu x. Chybová funkce určuje chybu sítě vzhledem k pevné tréninkové množině v závislosti na konfiguraci sítě. Při adaptaci sítě hledáme takovou konfiguraci, pro kterou je chybová funkce minimální. Začneme s náhodně zvolenou konfigurací $\mathbf{w}^{(0)}$, kdy odpovídající chyba sítě od požadované funkce bude pravděpodobně velká. V analogii s lidským učením to odpovídá počátečnímu nastavení synaptických vah u novorozence, který místo požadovaného chování jako chůze, řeč apod. provádí náhodné pohyby a vydává neurčité zvuky. Při adaptaci sestrojíme v tomto bodě $\mathbf{w}^{(0)}$ ke grafu chybové funkce tečný vektor (*gradient*) $\frac{\partial E}{\partial \mathbf{w}}\left(\mathbf{w}^{(0)}\right)$ a posuneme se ve směru tohoto vektoru dolů o ε. Pro dostatečně malé ε tak získáme novou konfiguraci $\mathbf{w}^{(1)} = \mathbf{w}^{(0)} + \Delta\mathbf{w}^{(1)}$, pro kterou je chybová funkce menší než pro původní konfiguraci $\mathbf{w}^{(0)}$, tj. $\boldsymbol{E}(\mathbf{w}^{(0)}) \geq \boldsymbol{E}(\mathbf{w}^{(1)})$. Celý proces konstrukce tečného vektoru opakujeme pro $\mathbf{w}^{(1)}$ a získáme tak $\mathbf{w}^{(2)}$ takové, že $\boldsymbol{E}(\mathbf{w}^{(1)}) \geq \boldsymbol{E}(\mathbf{w}^{(2)})$ atd., až se limitně dostaneme do lokálního minima chybové funkce. Ve vícerozměrném váhovém prostoru tento postup přesahuje naši představivost. I když při vhodné

volbě koeficientu učení (α) tato metoda vždy konverguje k nějakému lokálnímu minimu z libovolné počáteční konfigurace, není vůbec zaručeno, že se tak stane v reálném čase. Obvykle je tento proces časově velmi náročný i pro malé topologie vícevrstvé sítě (desítky neuronů).



Obrázek 29: Gradientní metoda

Hlavním problémem gradientní metody je, že pokud již nalezne lokální minimum, pak toto minimum nemusí být globální (viz obr. 29). Uvedený postup adaptace se v takovém minimu zastaví (nulový gradient) a chyba sítě se již dále nesnižuje. To lze v analogii s učením člověka interpretovat tak, že počáteční nastavení konfigurace v okolí nějakého minima chybové funkce určuje možnosti jedince učit se. Inteligentnější lidé začínají svou adaptaci v blízkosti hlubších minim. I zde je však chybová funkce definovaná relativně vzhledem k požadovanému „inteligentnímu“ chování (tréninková množina), které však nemusí být univerzálně platné. Hodnotu člověka nelze měřit žádnou chybovou funkcí. Elektrické šoky aplikované v psychiatrických léčebnách připomínají některé metody adaptace neuronových sítí, které v případě, že se učení zastavilo v mělkém lokálním minimu chybové funkce, náhodně vnáší šum do konfigurace sítě, aby se síť dostala z oblastí abstrakce tohoto lokálního minima a mohla popř. konvergovat k hlubšímu minimu.

Adaptační algoritmus backpropagation [7]:

Krok 0. Váhové hodnoty a bias jsou inicializovány malými náhodnými čísly.

Přiřazení inicializační hodnoty koeficientu učení α.

Krok 1. Dokud není splněna podmínka ukončení výpočtu, opakovat kroky (2 až 9).

Krok 2. Pro každý (bipolární) tréninkový pár **s:t** provádět kroky (3 až 8).

Feedforward:

Krok 3. Aktivovat vstupní neurony (X_i, *i=1, ...n)*

$x_i = s_i$..

Krok 4 Vypočítat vstupní hodnoty vnitřních neuronů:

(Z_j, *j=1,..., p)*:

$$z\_in_j = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}.$$

Stanovení výstupních hodnot vnitřních neuronů

$$z_j = f\left(z\_in_j\right).$$

Krok 5 Stanovení skutečných výstupních hodnoty signálu neuronové sítě (Y_k, *k=1, ..., m)*:

$$y\_in_k = w_{0k} + \sum_{j=1}^{p} z_j\, w_{jk},$$

$$y_k = f\left(y\_in_k\right).$$

Backpropagation:

Krok 6 Ke každému neuronu ve výstupní vrstvě (Y_k, *k=1, ..., m)* je přiřazena hodnota očekávaného výstupu pro vstupní tréninkový vzor. Dále je vypočteno $\delta_k = \left(t_k - y_k\right) f'\left(y\_in_k\right)$, které je součástí váhové korekce $\Delta w_{jk} = \alpha\, \delta_k z_j$ i korekce biasu $\Delta w_{0k} = \alpha\, \delta_k$.

Krok 7 Ke každému neuronu ve vnitřní vrstvě (*Zj, j=1, ..., p)* je přiřazena sumace jeho delta vstupů (tj. z neuronů, které se nacházejí v následující vrstvě), $\delta\_in_j = \sum_{k=1}^{m} \delta_k w_{j\,k}$.

Vynásobením získaných hodnot derivací jejich aktivační funkce obdržíme $\delta_j = \delta\_in_j f'\left(z\_in_j\right)$, které je součástí váhové korekce $\Delta v_{i\,j} = \alpha\, \delta_j x_i$ i korekce biasu $\Delta v_{0\,j} = \alpha\, \delta_j$.

Aktualizace vah a prahů:

Krok 8 Každý neuron ve výstupní vrstvě (Y_k, *k=1, ..., m)* aktualizuje na svých spojeních váhové hodnoty včetně svého biasu (*j=0, ..., p)*:

$w_{j\,k}\left(new\right) = w_{j\,k}\left(old\right) + \Delta w_{j\,k}$.

Každý neuron ve vnitřní vrstvě (Z_j, *j=1, ..., p)* aktualizuje na svých spojeních váhové hodnoty včetně svého biasu (*i=0, ..., n)*:

$v_{i\,j}\left(new\right) = v_{i\,j}\left(old\right) + \Delta v_{i\,j}$.

Krok 9. Podmínka ukončení:
pokud již nenastávají žádné změny váhových hodnot nebo pokud již bylo vykonáno maximálně definované množství váhových změn, stop; jinak, pokračovat.

Ačkoliv vlastní popis učícího algoritmu backpropagation je formulován pro klasický von Neumannovský model počítače, přesto je zřejmé, že jej lze implementovat distribuovaně. Pro každý tréninkový vzor probíhá nejprve aktivní režim pro jeho vstup tak, že informace se v neuronové síti šíří od vstupu k jejímu výstupu. Potom na základě externí informace učitele o požadovaném výstupu, tj. o chybě u jednotlivých vstupů, se počítají parciální derivace chybové funkce tak, že signál se šíří zpět od

výstupu ke vstupu. Výpočet sítě při zpětném chodu probíhá sekvenčně po vrstvách, přitom v rámci jedné vrstvy může probíhat paralelně.

Kontrolní otázky:

1. Co je to overfitting.
2. Co je to generalizace?
3. Jak probíhá adaptace metodou backpropagation.

Korespondenční úkol:

1. Vytvořte počítačový program pro realizaci adaptačního algoritmu backpropagation a použijte jej pro logickou funkci „XOR“.
2. Řešte vybranou logickou funkci standardním adaptačním algoritmem zpětného šíření chyby při stanovení různého počtu neuronů ve vnitřní vrstvě. Získané výsledky řešení srovnejte.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili s adaptačním algoritmem zpětného šíření chyby (backpropagation), jež je používán v přibližně 80% všech aplikací neuronových (tj. je nejrozšířenějším adaptačním algoritmem vícevrstvých neuronových sítí). Důraz v této kapitole byl kladen na problematiku vhodné volby topologie vícevrstvé neuronové sítě, která by měla odpovídat složitosti řešeného problému.

Pojmy k zapamatování

- backpropagation (adaptační algoritmus zpětného šíření chyby),
- generalizace,
- trénovací množina,
- dopředné (feedforward) šíření signálu,
- overfitting (přeučení).

10 Použití evolučních technik při adaptaci neuronové sítě

V této kapitole se dozvíte:

- Co je to typický cyklus evoluce adaptace váhových hodnot na spojeních mezi neurony?
- Jak lze reprezentovat váhové hodnoty?
- Co je to hybridní trénink?

Po jejím prostudování byste měli být schopni:

- Uvést výhody a nevýhody binární reprezentace váhových hodnot na spojeních mezi neurony.
- Uvést výhody a nevýhody reálné reprezentace váhových hodnot na spojeních mezi neurony.

Klíčová slova této kapitoly:

Binární reprezentace váhových hodnot, reálná reprezentace váhových hodnot, evoluce váhových hodnot, hybridní trénink.

Průvodce studiem

Evoluce váhových hodnot na spojeních mezi neurony provádí adaptivní a globální řešení úlohy týkající se adaptace neuronových sítí. Evoluční metody adaptace mají snahu nahradit metodu zpětného šíření při optimalizaci váhových a prahových koeficientů. Hybridní přístup používá genetické algoritmy a metodu backpropagation pro adaptaci neuronových sítí je v současné době předmětem výzkumu.

Učení umělých neuronových sítí je formulováno jako proces adaptace vah, jehož cílem je nalezení optimální množiny váhových hodnot na spojeních mezi neurony sítě podle optimalizačních kritérií. Jedním z nejpopulárnějších adaptačních algoritmů pro vícevrstvé sítě s dopředným šířením signálu je metoda backpropagation. Nevýhoda tohoto adaptačního algoritmu zpětného šíření chyby plyne z jeho gradientní povahy, neboť často uvízne v lokálním minimu chybové funkce a prohledávání prostoru řešení je pak neefektivní. Jedním ze způsobů, jak překonat nedostatky metody backpropagation (stejně jako i jiných gradientních prohledávacích technik), je považovat adaptační proces za evoluci váhových hodnot na spojení mezi neurony v prostředí determinovaném architekturou sítě a úlohou. Definice fitness funkce jedince pro každou takovou úlohu pak vychází z hodnoty celkové „chyby“ sítě (tj. rozdílu mezi aktuálním a očekávaným výstupem sítě) a to tak, že maximum fitness funkce odpovídá minimální hodnotě chybové funkce. Z tohoto pohledu použijeme globální prohledávací proceduru evolučních algoritmů pro adaptaci neuronových sítí efektivněji, protože fitness (chybová) funkce nemusí být diferencovatelná a dokonce ani spojitá, jelikož nezávisí na gradientních informacích.

Evoluce adaptace váhových hodnot na spojeních mezi neurony v neuronové síti je rozdělena do dvou hlavních fází: (1) musíme zvolit reprezentaci váhových hodnot, tj. zda bude ve formě binárního řetězce nebo reálná a (2) popsat samotnou evoluci evolučními algoritmy (tj. definovat operátory křížení, mutace apod.). Různá reprezentace nebo různě definované evoluční operátory mohou vést i při řešení stejné úlohy ke zcela odlišným výsledkům, co se týče tréninkového času i přesnosti výpočtu. Typický cyklus evoluce adaptace váhových hodnot na spojeních mezi neurony je následující [41]:

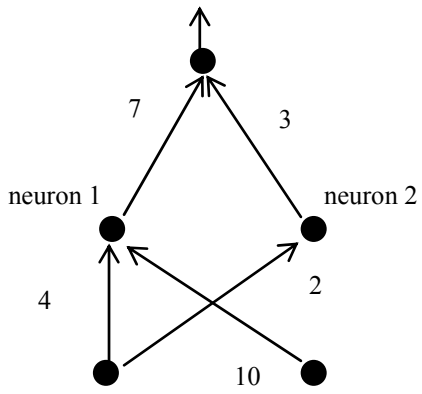
1. Zakódovat každého jedince (neuronovou síť) v dané generaci do předem definovaného schématu propojení neuronů sítě (architektura je definována předem a je fixní pro všechny sítě) a vytvořit chromozóm, tj. dosadit odpovídající váhové hodnoty těchto spojení do vytvořeného schématu.
2. Pro každého jedince (neuronovou síť) a pro předložený tréninkový vzor vypočítat střední kvadratické chybu sítě. Fitness jedince pro každou neuronovou síť obvykle definujeme jako funkci celkové „chyby“ sítě (tj. čím je větší hodnota chyby sítě, je její fitness hodnota menší), ale můžeme pro ni použít i jiný definiční vztah v závislosti na typu neuronové sítě.
3. Vybrat rodiče pro reprodukci na základě výpočtu pravděpodobností, která odpovídá jejich fitness hodnotě, např. použitím rulety.
4. Aplikovat genetické operátory jako křížení, mutace aj. s danou pravděpodobností a vytvořit novou generaci.

10.1 Binární reprezentace

Nejběžnější reprezentací váhových hodnot na spojeních mezi neurony je z pohledu evolučních algoritmů reprezentace binární V tomto reprezentačním schématu je každá váhová hodnota na spojeních mezi neurony reprezentována binárním číslem určité délky. Množina všech váhových hodnot sítě (jedince v evolučním procesu) je pak reprezentována svým umístěním do binárního řetězce, tj. neuronová síť je zakódována zřetězením všech váhových hodnot sítě do chromozómu. Uspořádání váhových hodnot v řetězci je náhodně stanoveno před zahájením výpočtu a je pro danou úlohu stálé, ačkoliv může ovlivnit celkový výsledek adaptace, tj. tréninkový čas i její přesnost. Obrázek 30 uvádí případ binární reprezentace neuronové sítě s předem definovanou architekturou. Každá hodnota na váhovém spojení mezi neurony je reprezentována čtyřmi bity, celá neuronová síť

je pak reprezentována řetězcem s 24 - bity, kde hodnota „0000“ představuje neexistenci spojení mezi neurony.

Výhoda binární reprezentace je v její jednoduchosti a obecnosti. Lze na ni přímo použít klasicky definovaný operátor křížení (jednobodové křížení) a mutaci a usnadňuje i hardwarovou implementaci neuronové sítě, neboť váhové hodnoty musí být reprezentovány z hlediska bitů s limitovanou přesností. Pro binární reprezentaci existuje několik kódovacích technik [21], např. kódování uniformní, Grayovo, exponenciální apod. Jistým omezením pro binární reprezentaci je předem definovaná přesnost zobrazení váhových hodnot. Pokud použijeme příliš málo bitů, adaptace může trvat extrémně dlouho nebo dokonce selže, protože reálná váhová hodnota nemůže být aproximována pouze diskrétními hodnotami z jistého definovaného okolí. Na druhé straně použitím příliš mnoha bitů se stávají binární řetězce příliš dlouhé, což výrazně prodlouží i čas evoluce a evoluční adaptace se tak stává nepraktickou. Používáme-li pro váhové hodnoty binární reprezentaci, musíme nejprve vyřešit následující problémy: (1) jak optimalizovat počet bitů pro každé váhové spojení; (2) jak určit rozsah okolí a (3) jakou použít kódovací metodu.



0100 1010 0010 0000 0111 0011

(a) (b)

Obrázek 30: (a) architektura neuronové sítě s váhovými hodnotami na spojeních mezi neurony; (b) binární reprezentace váhových hodnot za předpokladu, že každá z nich je reprezentována čtyřmi bity.

10.2 Reprezentace reálnými čísly

Pro reprezentaci váhových hodnot lze používat i reálná čísla, např. jedno reálné číslo pro každé váhové spojení. Reálná reprezentace neuronové sítě uvedená na obrázku 30 (a) může být (4.0, 10.0, 2.0, 0.0, 7.0, 3.0). Samotná reálná čísla tak pomáhají překonávat nedostatek binárních reprezentačních schémat, neboť mohou reprezentovat přímo konkrétní váhové spojení tak, že každému váhovému spojení přísluší právě jedno reálné číslo. Chromozóm je zde reprezentován zřetězením těchto reálných čísel (také zde záleží na pořadí čísel v chromozómu) a každý jedinec v populaci tak je reprezentován reálným vektorem. V reálném reprezentačním schématu nemohou být přímo použity standardní binární genetické operátory pracující s binárními řetězci (křížení a mutace), a proto byly vytvořeny speciální operátory, které jsou vhodné pro reálnou reprezentaci a které navíc zlepšují její rychlost a přesnost, např. [21] křížení průměrem, náhodná mutace, diferenciální křížení apod. Hlavním účelem těchto operátorů je udržet užitečné funkční bloky (užitečné rysy v neuronové síti) vcelku v průběhu celé evoluce. Evoluční adaptace je pak mnohem rychlejší než konvenčními algoritmy (např. backpropagation), ale na rozdíl od nich výslední jedinci (naadaptované neuronové sítě) nejsou schopni stejně dobře generalizovat.

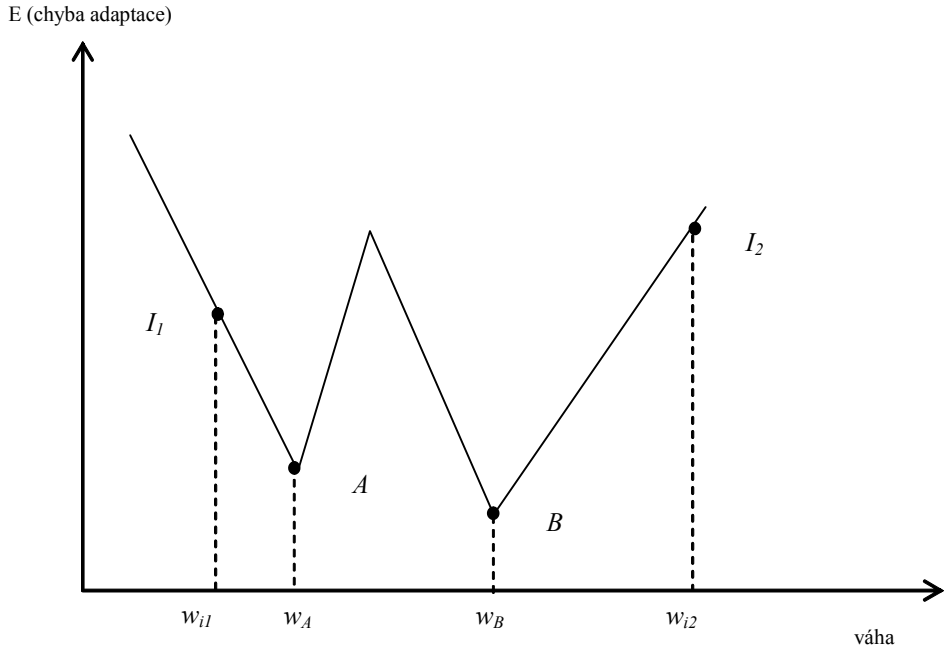
Evoluční metody adaptace mají sice snahu nahradit metodu zpětného šíření při optimalizaci váhových a prahových koeficientů, však pro tento typ úloh nejsou nejvhodnější, protože metoda zpětného šíření je výpočtově méně náročná a její výsledky jsou většinou přijatelné. Pokud však máme vytvořit efektivní síť s minimem vnitřních neuronů a spojů, jsou evoluční algoritmy nenahraditelné a používají se zejména při řešení problémů, kde nejsou k dispozici korektní odpovědi jednotlivých výstupních neuronů sítě pro daný vstup - to se především vyskytuje u problémů řízení nebo při navigaci robotů v neznámé prostředí. Evoluční adaptace však může být při řešení některých problémů i výrazně pomalejší ve srovnání s metodou backpropagation nebo jinými

gradientními metodami, protože evoluční algoritmy jsou mnohem méně citlivé na nastavení počátečních podmínek adaptace a vždy hledají globální optimální řešení, zatímco gradientní algoritmy mohou nacházet jen lokální optimum v okolí počátečního řešení. Evoluční algoritmy nenastavují žádná omezení na typ neuronové sítě a její adaptace může trvat neomezeně dlouho, může tedy pracovat s velkým okruhem umělých neuronových sítí, můžeme je použít pro adaptaci rekurentních neuronových sítí, neuronových sítí vyššího řádu, fuzzy neuronových sítí apod.

10.3 Hybridní trénink

Optimální hybridizace používající genetické algoritmy a metodu backpropagation pro adaptaci neuronových sítí je v současné době předmětem výzkumu. Evoluční metody dokáží relativně rychle nalézat váhové a prahové koeficienty, které jsou blízké svým optimálním hodnotám, ale na přechod od těchto téměř optimálních hodnot k optimálním hodnotám potřebují již velmi mnoho času. To zejména platí pro genetické algoritmy. Účinnost evoluční adaptace může být zlepšena začleněním lokální prohledávací procedury do evoluce, tj. kombinací globálního prohledávání evolučními algoritmy s lokálním laděním, kde evoluční algoritmus nalezne nejbližší hodnoty optima a zpětné šíření optimalizaci dokončí do globálního optima. Evoluční algoritmy tedy nejprve rychle lokalizují vhodnou oblast reprezentující počáteční bod ve váhovém prostoru a potom lokální prohledávání nalezne řešení blízké optimu v dané oblasti. Lokální prohledávací procedurou může být např. gradientní algoritmus backpropagation. Hybridní trénink je úspěšný v mnoha aplikačních oblastech. Obvykle nejprve použijeme genetické algoritmy pro nalezení množiny počátečních hodnot na váhových spojeních mezi neurony a pak aplikujeme algoritmus backpropagation pro jejich následné ladění. Získané výsledky ukázaly, že hybridní GA - BP přístup je mnohem efektivnější než samotné genetické algoritmy nebo algoritmus backpropagation. Ačkoliv hybridní evoluční přístup vyžaduje více

výpočetního času než gradientní algoritmus, není tomu tak ve skutečnosti, protože např. algoritmus backpropagation vynakládá mnoho času na nalezení dobré počáteční váhové inicializace. Použití genetických algoritmů pro lokalizaci dobrých počátečních váhových hodnot je proto mnohem výhodnější než jejich náhodné nastavení v metodě backpropagation. Obrázek 31 ilustruje jednoduchý případ nastavení inicializační váhové hodnoty. Pokud evoluční algoritmus nalezne počáteční váhovou hodnotu w_{I2}, bude pro lokální prohledávací algoritmus jednoduché nalézt globální optimální váhu w_B a to dokonce i v případě, že w_{i2} není tak dobrá jako w_{i1}.



Obrázek 31: Ilustrační použití evolučního algoritmu pro nalezení dobré inicializace vah, aby lokální prohledávací algoritmus mohl nalézt globální řešení. Váha w_{I2} je podle obrázku optimální inicializační váhová hodnota, protože vede k nalezení globálního optima w_B.

Úkoly k zamyšlení:

1. Zamyslete se nad výhodami a nevýhodami binární a reálné reprezentace váhových hodnot.
2. Jak byste vymezili hybridní trénink?

Korespondenční úkol:

1. Navrhněte genetický algoritmus pro adaptaci neuronové sítě otestujte jej na úloze XOR (topologii neuronové sítě zvolte 2-2-1). Dále zadejte následující: Velikost populace = 30 jedinců, P_{cross} = 0.5 a P_{mut} = 0,005. Znázorněte průběhy křivek minimální i průměrné chyby adaptace.
2. Porovnejte výsledky adaptace předchozího příkladu s řešením stejné úlohy metodou backpropagation, popř. navrhněte a implementujte hybridní algoritmus (pak v korespondenčním úkolu uveďte i výsledky této adaptace).

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili s aplikací evolučních algoritmů na adaptaci váhových hodnot na spojeních mezi neurony. Evoluční metody adaptace mají snahu nahradit metodu zpětného šíření při optimalizaci váhových a prahových koeficientů. Hybridní přístup používá pro adaptaci neuronových sítí genetické algoritmy a backpropagation. Důraz v této kapitole byl kladen reprezentaci váhových hodnot.

Pojmy k zapamatování

- binární reprezentace
- reálná reprezentace
- hybridní trénink

11 Použití evolučních technik při optimalizaci architektury neuronové sítě

V této kapitole se dozvíte:

- Jaký je princip konstruktivních a destruktivních algoritmů při stanovení topologie neuronové sítě.
- Jaký je princip přímého a nepřímého kódování topologie neuronové sítě.

Po jejím prostudování byste měli být schopni:

- Charakterizovat proces evoluce architektury umělé neuronové sítě.
- Vybrat způsob pro zakódování topologie neuronové sítě.

Klíčová slova této kapitoly:

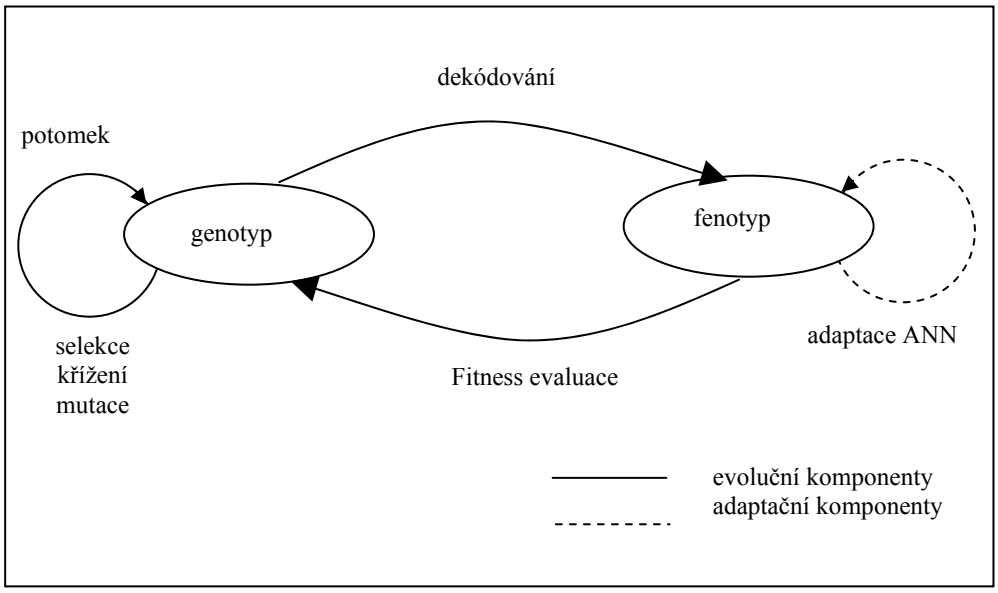
Konstruktivní algoritmus, destruktivní algoritmus, přímé a nepřímé kódování topologie neuronové sítě, evoluce architektury umělé neuronové sítě.

Průvodce studiem

Během evoluční adaptace neuronové sítě jsme doposud předpokládali, že její architektura je předem daná a je fixní v průběhu celého výpočtu. Nyní budeme diskutovat problém, jak navrhnout architekturu neuronové sítě pro řešenou aplikaci.

Evoluce architektury umožňuje neuronovým sítím adaptovat jejich topologii bez zásahu člověka tak, že provádějí automatickou optimalizaci rozmístění neuronů i jejich propojení a navíc umožňuje adaptovat i parametry aktivačních funkcí neuronů.

Navrhnout architekturu neuronové sítě znamená navrhnout následující [41]: A) *topologii sítě* (tj. rozložení neuronů v síti); B) *propojení mezi neurony* a C) *tvar přenosové funkce jednotlivých neuronů*. Je zřejmé, že použitá architektura neuronové sítě má rozhodující vliv na úspěšnost řešení dané aplikace, protože má i podstatný dopad na zpracování informací neuronovou sítí. Řešíme-li např. úlohu, kdy architektura neuronové sítě obsahuje jen několik spojení a lineární neurony, pak taková síť má pouze limitované schopnosti a obvykle není schopna úlohu vyřešit. Naopak úloha řešená neuronovou sítí, jejíž architektura obsahuje příliš velký počet nelineárních neuronů a jejich spojení, může přenášet poruchy obsažené v tréninkových datech a selhat i při generalizaci. Bohužel, architektura neuronové sítě bývá obvykle stanovena na základě zkušeností řešitele nebo experimentálně formou pokus-omyl. Nejčastější další metodou, která se používá k návrhu architektury neuronové sítě, jsou konstruktivní/destruktivní algoritmy *Konstruktivní algoritmus* zpočátku pracuje s minimální topologií sítě (tj. se sítí s minimálním počtem vnitřních neuronů a spojení) a pokud je to nutné, přidává během adaptace další neurony a spojení, zatímco *destruktivní algoritmus* dělá právě opačné (tj. zpočátku pracuje s maximální topologií sítě a během adaptace ruší nepotřebné neurony nebo spojení mezi nimi).



Obrázek 32: Proces evoluce architektury umělé neuronové sítě.

Podobně jako v průběhu evoluce váhových hodnot na spojeních mezi neurony, můžeme i v průběhu evoluce architektury neuronové sítě zaznamenat dvě významné fáze, jimiž jsou [41]: (1) stanovení vhodného genotypu a (2) výběr vhodného evolučního algoritmu pro řešení. Problémem nyní není, zda použít reprezentaci binární nebo reálnou. Klíčovou otázkou v kódování architektury neuronové sítě je rozhodnutí, jaké množství informace o architektuře by mělo být zakódováno do chromozómu. Jelikož pracujeme s diskrétními hodnotami, používáme obvykle binární reprezentaci, tj. matice, grafy nebo generování pravidel. Pokud jsou v chromozómu specifikovány všechny detaily topologie sítě, tj. všechny neurony a jejich vzájemné propojení, pak se tento způsob reprezentace nazývá *přímé kódování*. U větších sítí je možné nekódovat přímo všechna vzájemná propojení neuronů, ale jen propojitelnost jednotlivých vrstev sítě, což reprezentuje *parametrické kódování* [6]. Pro optimalizaci rozsáhlých síťových architektur je možné zvolit ještě složitější typ kódování, kdy jsou zakódovány pouze ty nejdůležitější parametry nebo rysy architektury sítě a reprezentace dalších detailů již závisí na adaptačním procesu. Takový způsob reprezentace se nazývá *nepřímé kódování*. Síťová architektura pak může být specifikována pravidly růstu, větami formálního jazyka apod. Typickým případem takovéhoto kódování je tzv. *gramatické kódování*, kdy evoluční algoritmy nevyvíjí přímo architekturu sítě, ale pravidla formálních gramatik, která se následně používají na generování vlastní topologie sítě. Proces evoluce architektury neuronové sítě, bez ohledu na vybraný typ její reprezentace (způsobu zakódování) je uveden na obrázku 32 a probíhá následovně:

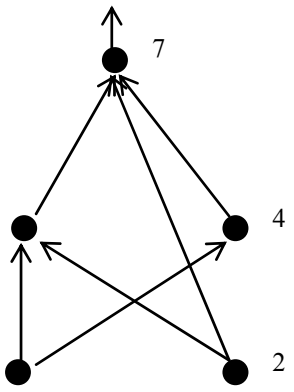
1. Zakódovat každého jedince v každé generaci do architektury s nezbytnými detaily.
2. Adaptovat každou neuronovou síť (tj. nejprve dekódovat její architekturu); náhodně inicializovat hodnoty vah na spojeních mezi neurony a pokud je to potřeba pak náhodně inicializovat i parametry adaptačních pravidel.

3. Výpočet fitness každého jedince (neuronové sítě) založené na adaptačních výsledcích; tj. založené na nejmenší celkové střední kvadratické chybě tréninku, nebo při testování (pokud je větší důraz kladen na generalizaci), na čase výpočtu, komplexnosti architektury (méně neuronů a spojení apod.)
4. Výběr rodičů pro další reprodukci, založený na vyčíslené hodnotě jejich fitness.
5. Aplikovat genetické operátory, jako je křížení, mutace, inverze apod. a vytvořit potomky pro novou generaci.

11.1 Přímé kódování topologie neuronové sítě

V přímém kódovacím schématu topologie neuronové sítě můžeme rozlišit dva přístupy. První z nich (chromozóm je tvořen řetězcem specifikujícím pouze spojení mezi jednotlivými neurony) odděluje evoluci topologie neuronové sítě od evoluce váhových hodnot. Druhý přístup (chromozóm je tvořen řetězcem specifikujícím váhové hodnoty na spojeních mezi jednotlivými neurony) pak řeší obě úlohy společně. Oba přístupy vyžadují stanovit fixní maximální topologii sítě již před začátkem výpočtu. V přímém kódování je každé spojení libovolných dvou neuronů v síti specifikováno přímo svou binární reprezentací, např. $N \times N$ matice $C = (c_{ij})_{NxN}$ může reprezentovat topologii vzájemného propojení N neuronů, kde c_{ij} označuje existenci nebo neexistenci spojení z neuronu i do neuronu j. Pokud je $c_{ij} = 1$, tak dané spojení existuje a pokud je $c_{ij} = 0$, spojení neexistuje. Ve skutečnosti může c_{ij} reprezentovat reálnou hodnotu váhy na spojení mezi neuronem i a j, čímž pak reprezentuje jak topologii sítě tak i váhové hodnoty na spojeních mezi neurony. Každá matice C tak přímo mapuje odpovídající spojení v topologii neuronové sítě. Binární řetězec reprezentující vzájemné propojení neuronů v síti je tvořen zřetězením řádků (nebo sloupců) matice C. Obrázky 33 a 34 znázorňují dva příklady přímého kódování topologie neuronové sítě. Obrázek 33 (a) zobrazuje síť s dopředným šířením signálu, jejíž topologie obsahuje dva

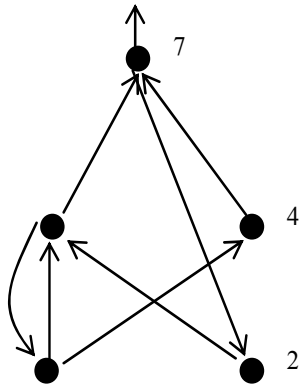
vstupy a jeden výstup. Její matice spojení je uvedena na obr. 33 (b), kde vstupy c_{ij} označují existenci, či neexistenci spojení vedoucího z neuronu *i* do neuronu *j*. První řádek tak znázorňuje spojení vedoucí z neuronu *1* ke všem ostatním neuronům. První dva sloupce jsou rovny 0, protože neexistuje spojení vedoucí z neuronu *1* k sobě samému a spojení vedoucí z neuronu *1* k neuronu *2*. Nicméně, neuron *1* je spojen s neuronem *3* a *4*, stejně tak lze vysvětlit „1“ ve 3. a 4. a 5. sloupci. Konverze této matice do chromozómu je jednoznačná. Můžeme zřetězit všechny řádky (nebo sloupce) a obdržíme: „00110 00101 00001 00001 00000“. Jestliže matice reprezentuje síť s dopředným šířením signálu, pak může být reprezentována pouze horní trojúhelníkovou maticí a tím lze redukovat délku chromozómu, viz obrázek 33 (c). Je zřejmé, že kódovací schéma z obrázku 34 může být použito jak u sítě s dopředným šířením signálu, tak i u rekurentní sítě. Obrázek 34 zobrazuje rekurentní síť se dvěma vstupy a jedním výstupem. Její matice spojení je uvedena na obr. 34 (b), kde vstupy c_{ij} označují existenci či neexistenci spojení vedoucího z neuronu *i* do neuronu *j*. Na rozdíl od obr. 33 (c) zde nemůžeme redukovat délku chromozómu, protože reprezentace topologie rekurentní neuronové sítě pracuje s celou maticí spojení, viz obr. 34 (b).



$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad (0110\ \ 101\ \ 01\ \ 1)$$

(a) (b) (c)

Obrázek 33: Příklad přímého kódování topologie neuronové sítě s dopředným šířením signálu. (a), (b) a (c) zobrazují topologii, matici spojení a její binární reprezentaci. Jelikož uvažujeme pouze neuronovou síť s dopředným šířením signálu, binární řetězec této reprezentace zahrnuje pouze horní trojúhelníkovou matici.



$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \qquad (00110\ \ 00100\ \ 10001\ \ 00001\ \ 01000)$$

(a) (b) (c)

Obrázek 34: Příklad přímého kódování rekurentní neuronové sítě. (a), (b) a (c) zobrazují architekturu, matici spojení a její binární reprezentaci.

Přímé kódování je pro implementaci evolučními algoritmy velmi jednoduché a je vhodné pro přesné deterministické výpočty v malých neuronových sítích, tj. sítích s malým množstvím neuronů. Velmi jednoduše zde můžeme včlenit do sítě další neuron nebo spojení, anebo z ní neurony a spojení odstranit. Délka genotypu je zde úměrná složitosti odpovídajícího fenotypu, a tak se prostor řešení zvětšuje exponenciálně s rostoucí velikostí sítě. Jedním ze způsobů, jak redukovat velikost matice, je používat doménových znalostí k redukci stavového prostoru řešení. Pokud např. používáme kompletní propojení dvou sousedních vrstev v neuronové síti s dopředným šířením signálu, její topologii můžeme zakódovat jen počtem skrytých vrstev a počtem neuronů v každé z nich, přičemž je délka chromozómu redukována. Využívat tyto doménové znalosti síťové topologie v praxi je velmi obtížné, a proto se zde dává přednost manuální restrikci prohledávaného prostoru. Dalším problémem přímého kódování je nemožnost zakódovat opakující se struktury, např. pokud jsou neuronové sítě vytvořené z několika subsítí s podobným lokálním propojením. V přímém kódování musí být struktury, které se opakují na úrovni fenotypu opakovaně zakódovány i do genotypu, a to není pro délku chromozómu efektivní. Plná specifikace fenotypu s opakujícími se strukturami ve skutečnosti předpokládá, že jsou adaptivní změny

opakujících se struktur během aplikace genetických operátorů nezávisle nalézány stále znovu.

11.2 Nepřímé kódování topologie neuronové sítě

Jedním ze způsobu redukce délky genotypu reprezentačního schématu zakódování topologie umělé neuronové sítě je nepřímé kódování, jež zahrnuje pouze nejdůležitější rysy topologie sítě. Detaily o každém propojení neuronů v síti jsou buď předdefinovány podle předchozí znalosti, nebo specifikovány množinou deterministických vývojových pravidel. Nepřímé kódování topologie neuronové sítě může sice vytvářet mnohem kompaktnější genotypovou reprezentaci, ale na druhé straně vůbec nemusí mít nalezená neuronová síť schopnost dobré generalizace. Pro kódování topologie neuronové sítě existuje mnoho nepřímých kódovacích strategií při aplikaci vývojových pravidel, jež jsou inspirována např. *Lindenmayerovými systémy*. Posun od přímé optimalizace topologie k optimalizaci vývojových pravidel přináší výhody kompaktnější genotypová reprezentace a zároveň chromozóm nezvětšuje svou délku s rostoucí velikostí topologie sítě, protože se velikost pravidla nemění. Vývojové pravidla bývají obvykle popsána rekurzivní rovnicí nebo se vytvářejí podobně jako v produkčních systémech s levou a pravou stranou. Na obr. 35 je uveden příklad množiny vývojových pravidel, kde každé z nich je tvořeno levou stranou, která je neterminálem a pravou stranou, jež je maticí typu 2x2 obsahující buď terminály nebo neterminály. Typickým krokem konstrukce matice spojení je nalezení pravidel, kdy levá strana obsažena v matici nahradí všechny své výskyty příslušnými pravými stranami. Jestliže získaná matice obsahuje pouze *1* a *0* (tj. pouze terminály), nemůžeme již na ni uplatnit žádné další vývojové pravidlo a matice tak reprezentuje matici spojení pro výslednou neuronovou síť. Obrázek 36 představuje souhrn předchozích tří kroků přepisovacích pravidel a výsledná architektura neuronové sítě je pak vytvořena podle obr. 36 (d), kde neurony *2,4*,a *6* nejsou v topologii sítě zobrazeny, protože nemají žádná spojení s ostatními neurony. Příklady popsané na

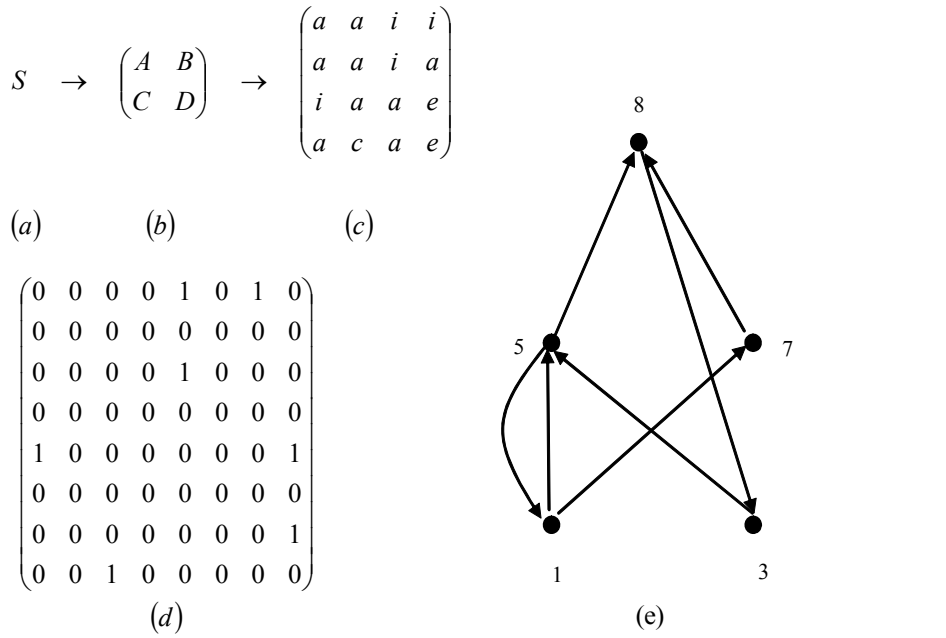
obrázcích 35 a 36 ilustrují způsob, jak může být topologie neuronové sítě definována pomocí množiny pravidel. Otázkou nyní je: *Jak můžeme získat takovou množinu pravidel vytvářejících topologii umělé neuronové sítě?* Jednou z možných odpovědí je nechat je vyvíjet. Množinu pravidel můžeme zakódovat jako jedince v evolučním procesu a optimalizovat jej genetickými algoritmy. Každé pravidlo pak může být reprezentováno čtyřmi pozicemi alel (tj. projev genu), korespondujícími se čtyřmi elementy na pravé straně v pravidlech, přičemž levá strana pravidel může být reprezentována implicitně. Každá pozice v chromozómu může nabývat množství různých hodnot závisejících na množství neterminálních symbolů, jichž je v množině pravidel použito, např. neterminální symboly mohou být v rozsahu „*A*“ až „*Z*“a „*a*“ až „*p*“. Dále je definováno šestnáct pravidel „*a*“ až „*p*“ na levé straně a matice typu 2x2 s hodnotami „*1*“ a „*0*“ na pravé straně. Levá strana je implicitně determinována svou pozicí v chromozómu, např. množina pravidel z obr. 36 může být reprezentována následujícím chromozómem: **ABCDaaaaiiiaiaacaeae,** kde první čtyři pozice reprezentují pravou stranu pravidla „*S*“, další čtyři pozice reprezentují pravou pravidla „*A*“ apod. Metoda založená na reprezentaci topologie neuronové sítě pomocí vývojových pravidel obvykle odděluje evoluci architektury od evoluce váhových hodnot na spojeních mezi neurony. Výsledné grafy jsou učené pomocí backpropagation a ohodnocené sumou chyb pro vzory z testovací množiny. Aplikace operátoru křížení se děje rozsekáním řetězců chromozomů. na několik podřetězců a následnou výměnou těchto podřetězců. Aplikace operátoru mutace se děje náhradou náhodně zvoleného symbolu symboly A-Z a a-p abeced.

$$S \to \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

$$A \to \begin{pmatrix} a & a \\ a & a \end{pmatrix} \quad B \to \begin{pmatrix} i & i \\ i & a \end{pmatrix} \quad C \to \begin{pmatrix} i & a \\ a & c \end{pmatrix} \quad D \to \begin{pmatrix} a & e \\ a & e \end{pmatrix} \quad \cdots$$

$$a \to \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \quad c \to \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \quad e \to \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \quad i \to \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \quad \cdots$$

Obrázek 35: Příklady vývojových pravidel používaných pro konstrukci matice spojení, kde S je inicializační stav.



S	A	B	C	D	A	a	a	a	a	B	i	i	i	a	C	i	a	a	c	D	a	e	a	e

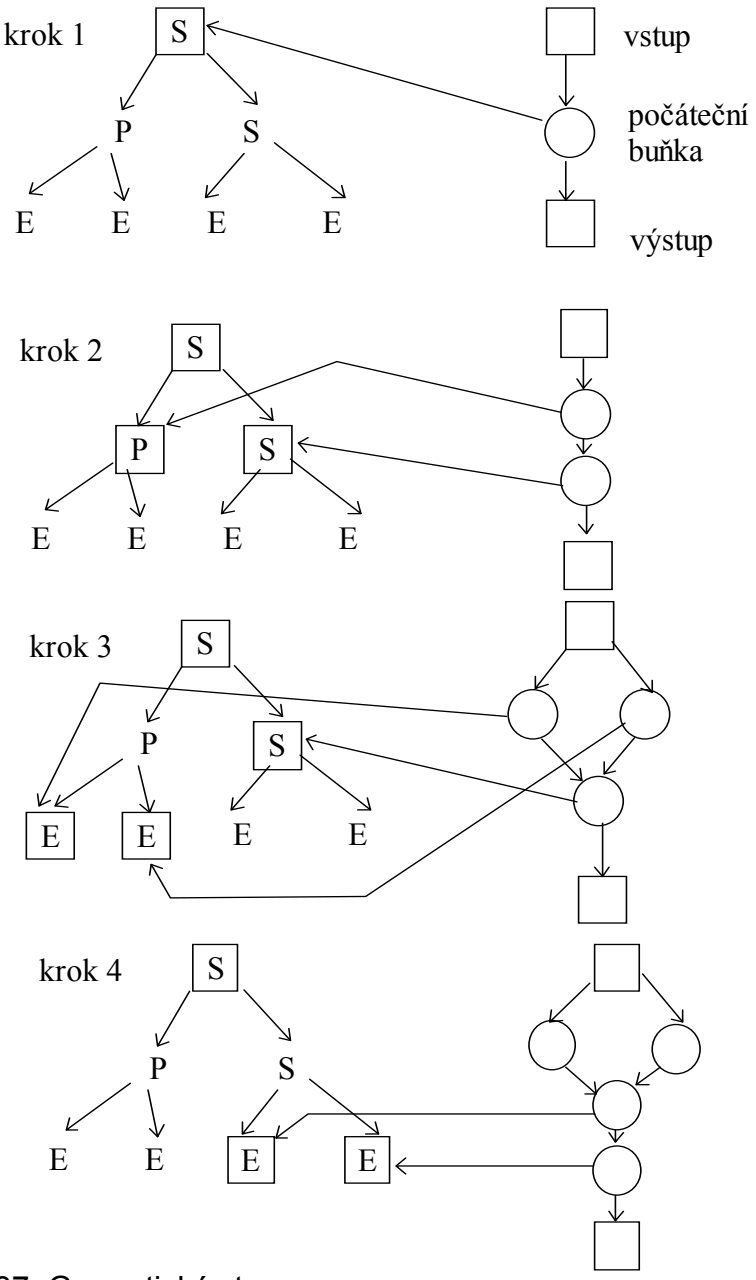
(*f*)

Obrázek 36: Vývoj topologie umělé neuronové sítě použitím pravidel z obr. 35. (a) počáteční stav; (b) krok 1; (c) krok 2; (d) krok 3, kdy jsou všechny prvky matice terminály, tj. buď 1 nebo 0; (e) získaná topologie neuronové sítě; (f) chromozóm odpovídající těmto pravidlům. Neurony v topologii jsou číslovány od 1 do 8. Neurony bez spojení nejsou zobrazeny.

11.3 Stromová reprezentace topologie umělé neuronové sítě

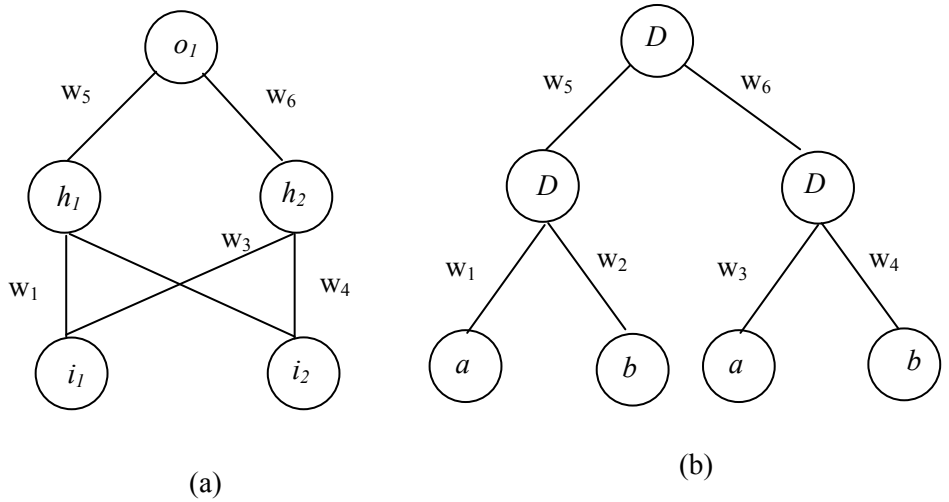
Základem dalšího kódování je „*gramatický strom*“ (viz obrázek 37). Je vždy umístěn nalevo a napravo je síť s neurony (buňkami) označenými kroužkem. Každá buňka ukazuje na určité místo v tomto stromu a podle toho, co se na této pozici nachází, se dělí na dvě buňky, nebo změní své vnitřní parametry (např. práh neuronu), nebo přemístí svůj ukazatel na jinou pozici ve stromě (čímž je možné vytvořit rekurzi). Vývoj začíná od jedné počáteční buňky ukazující na vrchol stromu. Tato buňka je spojená se vstupem a výstupem. V sekvenčním (vertikálním) rozdělení, označeném **S**, se rodičovská buňka rozdělí na dvě dceřiné buňky tak,

že první z nich zdědí všechny vstupy rodičovské buňky a druhá dceřiná buňka zdědí všechny jejich výstupy. Přitom je definované spojení obou dceřiných buněk mezi sebou. Při paralelním (horizontálním) rozdělení, označeném **P**, se rodičovská buňka rozdělí na dvě nespojené dceřiné buňky a každá z nich zdědí vstupy i výstupy rodičovské buňky. Symbol **E** je symbolem ukončení práce. Je možné použít i označení **R**, kdy se ukazatel přesune na některý předcházející uzel, čímž je možné zavést rekurzi.



Obrázek 37: Gramatický strom.

Každý bod genotypového stromu kóduje operace, které se mohou provádět na odpovídající buňce a dva podstromy bodů specifikujících operace, jež by měly být aplikovány na obě dceřiné buňky. Genotyp obsahuje množinu pravidel růstu, proces dělení buněk (jednotlivá buňka se rozdělí na dvě „dceřiné“ buňky) a transformace (nová spojení mohou být přidána a váhy na spojeních vycházejících z buňky mohou být modifikovány). V tomto modelu jsou proto spojení vytvářena během procesu dělení buněk. Gramatické stromy se optimalizují pomocí genetických algoritmů a ohodnocení chromozomu (gramatického stromu) je odvozené z chybové funkce neuronové sítě. Tato kódovací metoda má dvě výhody: (1) kompaktní genotyp může vytvářet komplexní fenotypové sítě; (2) evoluce může využívat fenotyp, v němž jsou opakující se podstromy kódovány pouze v jedné části genotypu. To podstatně šetří místo v genomu.



(a) (b)

Obrázek 38: (a) konfigurace neuronové sítě s dopředným šířením signálu a její reprezentace (b) založená na genetickém programování.

Na obr. 38 (a) je zobrazena topologie neuronové sítě s dopředným šířením signálu obsahující dva vstupní neurony i_1 a i_2, dva skryté neurony h_1 a h_2 a jeden výstupní neuron o_1 a spojením mezi neurony jsou přiřazeny váhové hodnoty. Na obr. 38 (b) reprezentuje „*D*“ funkci s dvěma spojeními, „*a*“ a „*b*“ jsou hodnoty vstupního signálu. Celá topologie sítě včetně váhových hodnot na spojeních mezi neurony je

pak zakódována do lineárního chromozómu následovně [8]: ***DDDababw$_6$w$_5$w$_4$w$_3$w$_2$w$_1$.***

11.4 Evoluce parametrů přenosové funkce neuronů

Doposud probíhala diskuse o architektuře neuronové sítě pouze z hlediska její topologie a tvar přenosové funkce každého neuronu v topologické struktuře sítě byl předem dán experty. Přenosová funkce je však důležitou součástí architektury a má významný vliv na výkon celé neuronové sítě. V principu mohou mít přenosové funkce neuronů v evolučním procesu různé tvary (např. skoková funkce s prahem, Gaussova funkce, sigmoidální funkce apod.), jež mohou být buď dány expertem, nebo vyvíjeny v průběhu evoluce evolučními algoritmy. Rozhodnutí, jak kódovat přenosové funkce v chromozómu, závisí na předchozí znalosti a na době výpočtu. Neurony uvnitř skupiny (ve vrstvě) mívají tendenci zachovat stejné přenosové funkce pouze s malými rozdíly v jejích parametrech, zatímco jiné skupiny neuronů mohou mít různé typy přenosových funkcí. To umožňuje používat nepřímé kódovací metody, které podporují vývoj pravidel pro specifikaci parametrů funkce, čímž můžeme očekávat i kompaktnější kódování chromozómu a rychlejší průběh evoluce.

11.5 Evoluce architektury současně s váhovými hodnotami

Doposud diskutovaný evoluční přístup řeší evoluci architektury odděleně od adaptace sítě, která probíhá až po nalezení její optimální (suboptimální) architektury. Toto je obvyklý postup hlavně v případě používání nepřímých metod kódování architektury sítě. Velkým problémem spojeným s evolucí architektury neuronové sítě bez současné její adaptace je tzv. problém „*noisy fitness evaluation*“, v důsledku čehož je ohodnocení fitness funkce velmi nepřesné. Na rozdíl od kódování váhových hodnot na spojení mezi neurony obsahuje kódování architektury mnoho nepřesností, neboť to, co je opravdu

během evoluce vyvíjeno je *fitness fenotypu* (tj. fitness jedinců – neuronové sítě s danou architekturou a úplnou sadou váhových hodnot na spojeních mezi neurony), která je pouze hrubou aproximací *fitness genotypu* (tj. kódování architektury neuronové sítě bez informace o váhových hodnotách na spojeních mezi neurony). Máme zde tedy optimalizovat genotyp jedince tak, aby mohl pracovat bez ohledu na počáteční nastavení vah, avšak aproximovat můžeme pouze jeho fenotyp s úplnou sadou nastavení váhových hodnot na spojeních mezi neurony.

Rozeznáváme zde dva hlavní zdroje nepřesností [41]:

1. Prvním zdrojem jsou náhodně inicializovány hodnoty vah na spojeních mezi neurony, neboť různé počáteční inicializace váhových hodnot mohou vést k různým tréninkovým výsledkům. Z tohoto důvodu mohou mít stejné genotypy právě kvůli náhodnému inicializačnímu nastavení váhových hodnot zcela různé hodnoty fitness.
2. Druhým zdrojem je adaptační algoritmus, neboť různé adaptační algoritmy mohou vést k různým tréninkovým výsledkům, dokonce i v případě stejného inicializačního nastavení váhových hodnot: metoda backpropagation může např. redukovat chybu během adaptace neuronové sítě na hodnotu 0.05, ale evoluční algoritmus může díky svému globálnímu charakteru redukovat chybu adaptace až na 0.001.

Takováto nepřesnost ovlivňuje i samotný průběh evoluce. Je-li fitness genotypu G_1 větší než fitness genotypu G_2, pak předpokládáme, že i genotyp G_1 kvalitnější než genotyp G_2, což může být pravda, ale nemusí. Tento problém obvykle řešíme tak, že je architektura neuronové sítě adaptována vícekrát s různým inicializačním váhovým nastavením. Průměrný výsledek adaptace je pak použit k výpočtu fitness hodnoty jedince. Tato metoda vyžaduje pro vyhodnocení každé fitness značný výpočetní čas. To je největší důvod proč se používá hlavně pro sítě malého rozsahu. Výsledek takovéto evoluce je ovšem

nevěrohodný. Jedním ze způsobů, jak řešit tento problém je vyvíjet architekturu neuronové sítě zároveň s její adaptací. Pokud je každý jedinec v populaci plně specifikován neuronovou sítí s danou architekturou a s úplnou sadou váhových informací, pak zde probíhá jednoznačné mapování mezi genotypem a fenotypem a evaluace fitness jedince je přesná.

Kontrolní otázky:

1. Čím se liší přímé a nepřímé kódování topologie neuronové sítě?.
2. Čím se liší konstruktivní a destruktivní algoritmy?
3. S jakými nepřesnostmi se potýkáme, pokud neprobíhá evoluce architektury současně s váhovými hodnotami.

Úkoly k zamyšlení:

Zamyslete se nad dalšími možnostmi zakódování topologie neuronové sítě.

Korespondenční úkol:

1. Navrhněte genetický algoritmus pro optimalizaci topologie vícevrstvé neuronové sítě s dopředným šířením signálů. Algoritmus implementujte a ověřte např. při řešení problému XOR.
2. Navrhněte genetický algoritmus pro optimalizaci spojení mezi neurony vícevrstvé neuronové sítě s dopředným šířením signálů. Algoritmus implementujte a ověřte např. při řešení problému XOR.

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základními principy evolučního návrhu architektury neuronových sítí včetně parametrů aktivačních

funkcí neuronů. Důraz v této kapitole byl kladen na možnost adaptovat topologii sítě bez zásahu člověka.

Pojmy k zapamatování

- konstruktivní/destruktivní algoritmy,
- gramatický strom,
- přímé, nepřímé a parametrické kódování topologie neuronové sítě.

12 Modularita neuronových sítí

V této kapitole se dozvíte:

- Jaké jsou typy modulární architektury neuronových sítí.
- Jaké jsou vlastnosti modulární architektury neuronových sítí.
- Jak se vytváří modulární architektura neuronových sítí.

Po jejím prostudování byste měli být schopni:

- Charakterizovat vlastnosti modulární architektury neuronových sítí.
- Charakterizovat vývoj modulární architektury neuronových sítí.
- Charakterizovat typy modulární architektury neuronových sítí.
- Charakterizovat vztahy mezi moduly.

Klíčová slova této kapitoly:

Modulární architektura neuronových sítí, ARTMAP, bránové sítě, autoasociativních moduly

Průvodce studiem

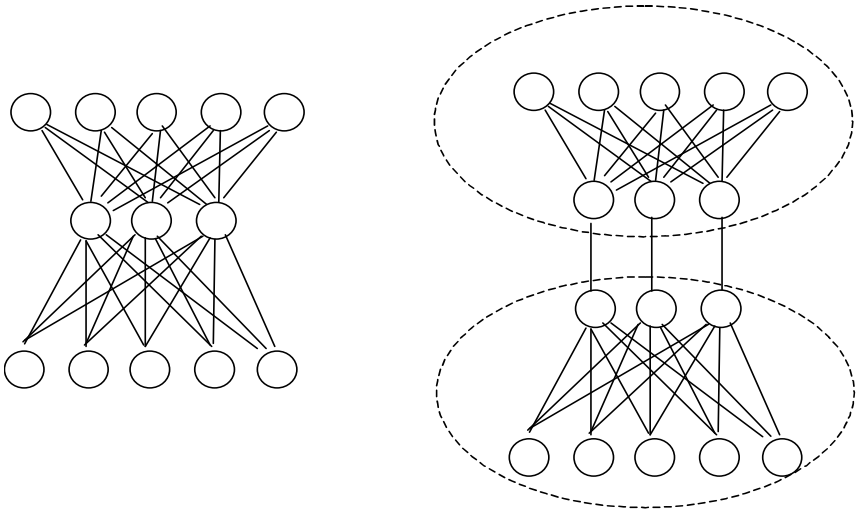
V této kapitole se budeme věnovat otázkám modularity, jež má velmi významnou úlohu nejen v neuronových sítích, ale např. i v kognitivních vědách, kde patří mezi základní atributy. Celá kapitola je zpracována podle [13].

12.1 Modulární architektura neuronových sítí

Moduly lze v neuronových sítích definovat různými způsoby. V závislosti na způsobu definice jim lze přiřadit i různé vlastnosti. Moduly lze např. považovat za množinu bodů, které formují přesnou

strukturu celé sítě podle jistých strukturálních kritérií, tj. počet spojení je mezi body v modulu je mnohem větší než počet spojení mezi moduly. Taková definice není omezována ani způsobem externích spojení mezi moduly a ani množstvím spojení ve vnitřním uspořádání modulu, která pak mohou být i rekurentní. Mezi dnes nejpoužívanější neuronové sítě patří: vícevrstvé sítě adaptované metodou backpropagation, Hopfieldovy sítě, Kohonenovy mapy, sítě typu couterpropagation a ART. Mezi těmito modely jsou i takové, které modulární architekturu vůbec nevytvářejí, např. Hopfieldovy sítě a Kohonenovy mapy. Zbývající modely určité aspekty modularity vykazují.

Vícevrstvá neuronová síť adaptovaná metodou backpropagation je prototypem sítě s dopředným šířením signálu a lze ji považovat za modulární po vrstvách, tj. vždy dvě sousední vrstvy sítě formují „přesné“ moduly, jež přijímají vstupní signál současně a současně také vytvářejí svůj výstup. Tento modulární charakter vícevrstvé neuronové sítě je zobrazen na obrázku 39 a je zřejmé, že tato síť má pevná pouze mezimodulární spojení, spojení uvnitř modulů jsou proměnlivá. Modularita po vrstvách je velmi slabá, protože umožňuje pouze sériové propojení modulů, tj. každá následující vrstva provádí jen další transformace výstupu předešlé vrstvy.

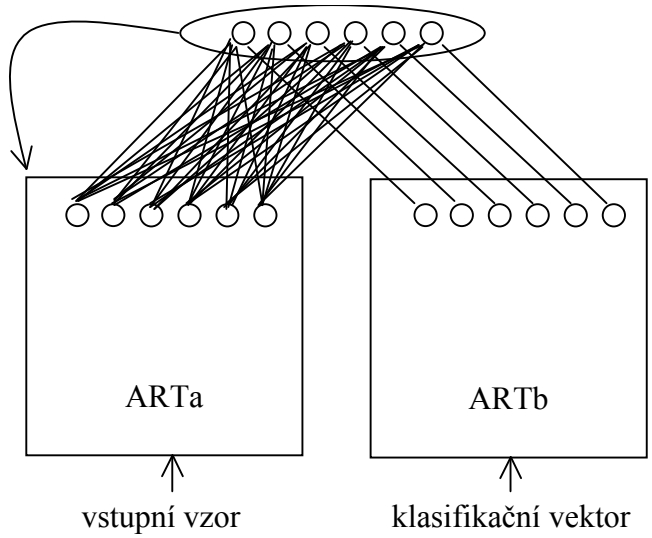


(*a*) (*b*)

Obrázek 39: Architektura vícervstvé sítě adaptované metodou backpropagation: (a) standardní, (b) modulární.

Síť typu ***counterpropagation*** představuje zajímavý hybridní model, který sdružuje dva sériově spojené moduly: (1) Kohonenovu mapu, jež vede do (2) sítě „instar“ ve tvaru hvězdy. Adaptace této sítě probíhá ve dvou krocích. V první fázi se Kohonenova vrstva adaptuje na vstup a rozdělí vstupní prostor do tříd. Ve druhé fázi „instar“ vrstva přiřadí každé třídě její vektorovou reprezentaci. Během těchto dvou procesů, pracuje síť typu counterpropagation jako statisticky optimální samoprogramující se vyhledávací tabulka. Na neuronové sítě lze tedy pohlížet jako na stavební blokové komponenty, které mohou být sestaveny do nových konfigurací, jež nabízejí nové a odlišné informace.

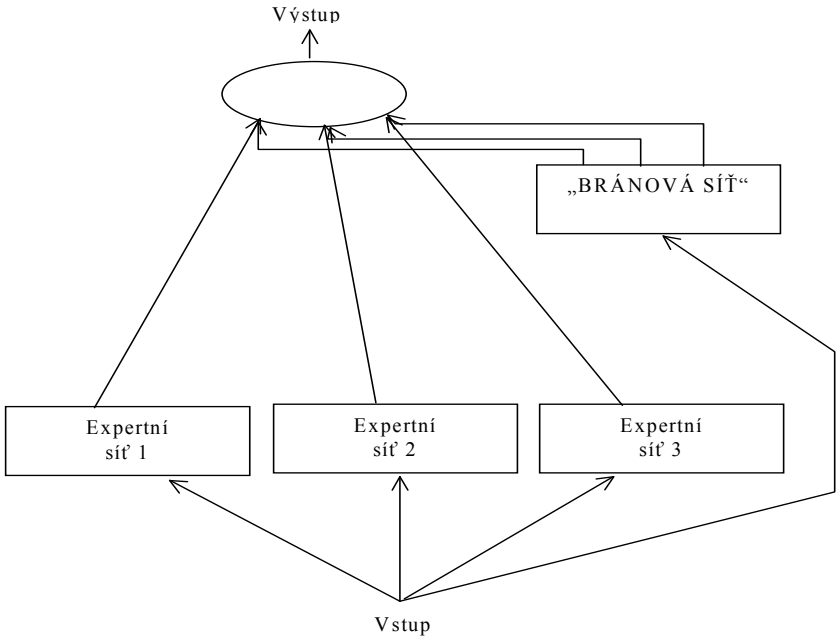
ART-sítě jsou považovány za samoorganizující klasifikátory vzorů. Jejich modulární rozšíření představuje síť ARTMAT, která integruje dva moduly sítě ART za pomoci třetí síťové struktury (viz obr. 40).



Obrázek 40: Struktura sítě ARTMAP.

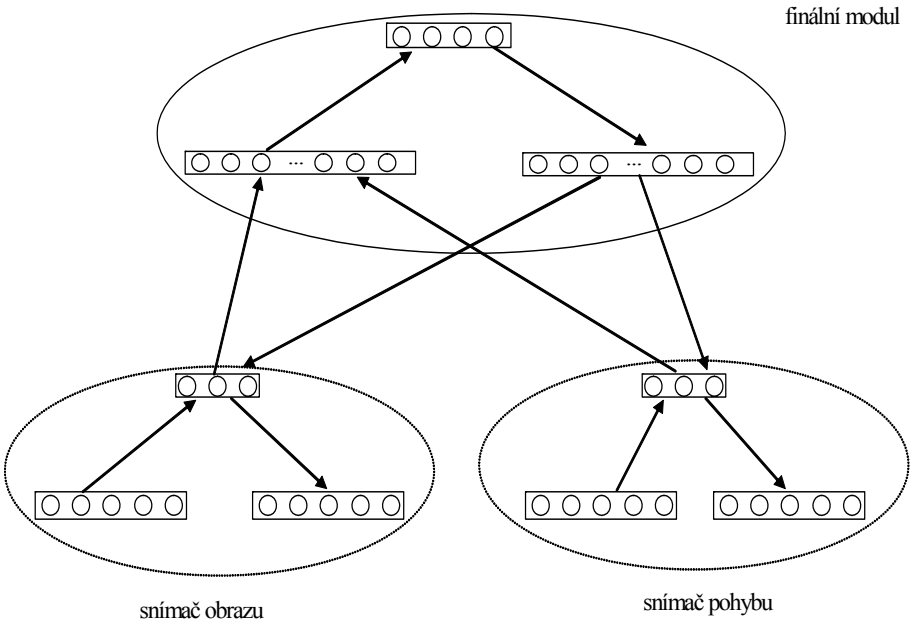
Dalším zajímavým modelem modulárních sítí je ***bránová síť*** tzv. ***„gating network“***, která obsahuje moduly jako základní stavební bloky. Její architektura obsahuje tři části (viz. obr. 41): A) expertní síťové moduly, jež zpracovávají zcela nezávisle stejný vstupní vzor; B) modul „bránová síť“, který má na vstupu tentýž vstupní vektor jako expertní moduly a na výstupu po jedné hodnotě každého z expertních modulů (tato hodnota odráží pravděpodobnost reprezentující správnost výstupu

z jednotlivých modulů); C) integrátor, jehož výstupem je vážená sumace výstupů expertních sítí s pravděpodobnostmi danými „bránovým“ modulem. Expertní moduly i „bránový“ modul jsou vícevrstvé sítě adaptované metodou backpropagation. Během adaptace jsou vstupní data analyzována s výstupními odezvami všech expertních systémů (tj. pouze jedna výstupní odezva je nejbližší očekávanému výstupu). Pokud systém vykoná daný tréninkový vzor lépe než v minulosti, váhové hodnoty bránové sítě jsou změněny tak, aby výstup ze systému korespondoval s výstupem vítězné expertní sítě. Pokud však systém nevykazuje zlepšení, váhy bránové sítě jsou přizpůsobeny tak, aby posouvala své výstupy směrem k neutrálním hodnotám. Během adaptace všechny expertní sítě modifikují své váhy na základě chybové hodnoty, která je vážena korespondující pravděpodobnostní hodnotou v bránové síti. Tato pravděpodobnostní hodnota určuje, jak se daná expertní síť podílí na výstupní konfiguraci systému. Bránová síť tedy determinuje velikost chybových vektorů expertních sítí v závislosti na schopnosti každé expertní sítě se učit daný tréninkový vzor. Bránové sítě jsou vhodné především pro řešení problémů, kde se dá množina možných zadání dekomponovat na jednotlivé typy úloh řešitelné samostatně jako dílčí podúlohy pomocí modulů. Je proto nezbytné zabývat se i problémem přiřazení modulů k dílčím úlohám tak, aby zároveň neřešily úlohy jiné, a dále je nutné se zabývat i problémem formátu vstupních informací, protože různé úlohy vyžadují i různé formy vstupů. Tento základní model může být jednoduše rozšiřitelný na systém obsahující moduly či v jednotlivých expertních sítích, které mají opět architekturu bránové sítě.



Obrázek 41: Jednoduchý model bránové sítě.

Na obrázku 42 je prezentována modulární síť, která používá jako stavební bloky ***autoasociativní moduly***. Jednotlivé moduly představují vícevrstvé sítě adaptované metodou backpropagation a to tak, že soubory dat na vstupu i na výstupu jsou stejné. Skryté neurony se tak učí kompaktnější formou reprezentaci vstupního vzoru. Tento model je složený z asociativních modulů, kdy jeden modul formuje interní reprezentaci ze snímače obrazů, druhý modul ji formuje ze snímače pohybu a třetí modul akceptuje obě interní reprezentace jako vstup a učí se svou interní reprezentaci. Tento model má několik výhod: (1) jeho interní struktura je pro experimenty výhodnější než standardní vícevrstvá síť adaptovaná metodou backpropagation; (2) je flexibilnější, neboť komponenty modulů mohou být použity pro přípravu vstupů i k jiným modulům; (3) je komplexnější, protože používá kompaktní reprezentaci. V modulární autoasociativní síti tak může každý vstup prezentovat různé moduly a formovat z nich svou interní reprezentaci, jež je integrována do finální sítě, tj. model je tak rezistentní na změny.



Obrázek 42: Síť autoasociativních modulů.

Další modely modulární architektury neuronové sítě jsou popsány v [13].

12.2 Vytváření modulární architektury

Modulární architekturu lze u modelů buď definovat fixně a to tak, že existujícím modelům modularitu přiřadíme (tato modularita pak z vlastností modelu nevychází a rovněž nemění ani jeho architekturu) nebo vytvořit. Metody vytváření modulárních sítí mohou obecně spadat do čtyř různých kategorií [13]:

1. neuronové sítě jsou trénovány nezávisle a pak integrovány do systému.
2. neuronové sítě jsou vytvářeny systémem z několika síťových modulů s fixním vnitřním propojením, jež obecně nemusí být stejného typu. Adaptují se postupně formou strojového učení.
3. Do formy komplexních sítí jsou začleňovány moduly fixního typu, které lze kombinovat specifickým způsobem. Mají speciální mezimodulární učící pravidla.

4. Do konfigurace sítí jsou začleněny dispozice, jež formují moduly během učení. Obecně však nevytváří striktní moduly, protože stupeň modularity sítě závisí na řešeném problému.

Dekompozice úloh do jednotlivých modulů může být prováděna (a) explicitně nebo (b) automaticky. Explicitní dekompozice úlohy závisí na pochopení řešené úlohy a na možnostech modulárních komponent. To je většinou prováděno na základě využití dřívějších znalostí a zkušeností s řešením podobných úloh, např. částečná dekompozice může vycházet ze struktury úlohy, kdy jsou vstupní data z různých zdrojů. Dekompozice úloh je tedy často prováděna za pomocí teoretických znalostí o řešeném problému. Na druhé straně automatické dekompozice je dosaženo "naslepo" během spuštění samotného výpočtu. Používá se hlavně v případech, kdy nemáme potřebné znalosti o řešené úloze. Dekompozici monolitické neuronové sítě do modulů lze řešit i s použitím evolučních algoritmů, kde emergence modulů je velmi důležitou součásti procesu učení [13]. Pojmem „*modul*“ označujeme zcela nezávislou funkční jednotku. Mezi problémem řešeným jako celkem a stejným problémem řešeným pomocí modulů pak vznikají odlišnosti. Modulární architektura neuronové sítě vykazuje následující vlastnosti: (1) provádí dekompozici řešené úlohy v tom smyslu, že úlohu rozdělí na dvě nebo více funkčně nezávislé úlohy; (2) má tendenci přidělit každé úloze takovou síťovou topologii, která je pro její řešení nejvhodnější; (3) má tendenci přidělovat každému síťovému modulu k řešení úlohy podobného typu, výsledkem čehož je urychlení adaptace a jiné úlohy pak bude přidělovat jiným modulům, tj. lze se tak vyhnout zpomalení adaptace; (4) Každý modul je trénován pouze na vykonání jednotlivé dílčí úlohy a celkové řešení úlohy vzniká spojením řešení jednotlivých modulů. Nevýhodou modulárního systému někdy je, že brání sdílení vědomostí mezi moduly. Tento problém vzniká v případě, když je komplexní úloha lépe řešitelná při použití sdílené strategie v celém kontextu, jejíž modifikace je na kontextu také závislá.

12.3 Vlastnosti modulární architektury neuronových sítí

Hlavním důvodem pro volbu modulární neuronové sítě namísto monolitické, je zlepšení funkčnosti a výkonu systému. Při použití modulů v neuronových sítích máme možnost si volit nejrůznější způsoby nastavení. Modulární systémy použijeme hlavně pro řešení problémů, které jsou jen těžko řešitelné monolitickou neuronovou sítí, tj. modulární systém může využít speciálních schopností svých modulů a v důsledku toho obdrží výsledky, jež jsou monolitickou sítí hůře dosažitelné při stejném čase učení i při stejném počtu kroků učení modulární síť obsahuje méně optimalizovaných vah. Modulární systém je možné nastavit tak, že jeden modul řídí chování ostatních modulů, čehož by se v monolitickém systému pomocí klasických adaptačních metod nedalo dosáhnout. Dalším rozšířením modulární architektury neuronových sítí jsou hybridní systémy, kde uvažujeme i speciální moduly, které nemusí být nutně neuronovými sítěmi, ale mohou být konstruovány různými technikami.

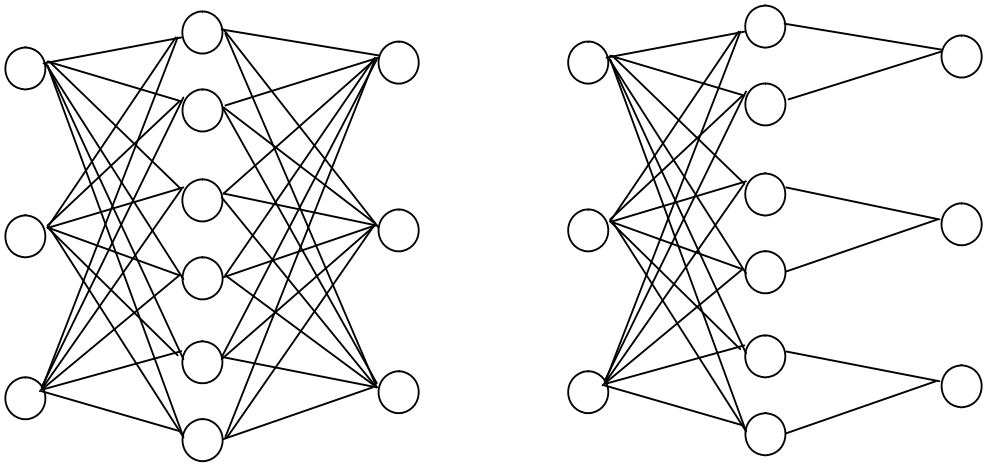
Rozeberme si nyní přednosti řešení úloh při použití modulární architektury podrobněji:

1. *Modulární architektura neuronové sítě má výhody v případě rychlosti učení.* Lze v podstatě říci, že se modulární sítě učí rychleji určitou třídu problému než monolitické sítě. Modulární architektura má výhodu v provádění funkční dekompozice problému. Přirozeným způsobem tak rozkládá komplexní funkci na množinu jednoduchých funkcí a je proto schopna naučit se tuto množinu jednoduchých funkcí rychleji než síť monolitická, která se učí nerozloženou komplexní funkci. Modulární architektura snižuje výskyt konfliktní tréninkové situace, které mají tendenci zpožďovat učení. Konflikty v tréninkových informacích označujeme jako interference, které mohou být dvojího typu: prostorové (*spatial crosstalk*) a časové (*temporal crosstalk*). *Prostorové* interference

se vyskytují např. při adaptaci metodou backpropagation, kdy je skrytý neuron spojen se dvěma výstupními neurony pozitivními váhami, dochází tedy ke konfliktní situaci. Obrázek 43 zobrazuje síťovou architekturu, kde část (a) zobrazuje jednoduchou síť a část (b) zobrazuje modulární architekturu obsahující tři různé sítě. Ačkoliv obě síťové architektury mohou být použity pro řešení týchž úloh, modulární architektura je vůči prostorové interferenci odolná. *Časová* interference může zase vzniknout, pokud neuron obdrží nekonzistentní tréninkovou informaci v jiném čase. Nastává v případě, že síť adaptujeme pro vykonání dvou různých funkcí postupně, tj. je síť nejprve adaptována na vykonávání jedné funkce a teprve potom na vykonávání funkce druhé. Druhou funkci by se síť měla naučit bez zbytečného znehodnocení vykonávání funkce první. Zde může nastat konfliktní situace, protože po tréninku druhé funkce převažuje tendence, že vykonávání první funkce je významně potlačeno. Vhodně vytvořená modulární architektura sítě by oproti monolitické síti, měla umožnit její rychlejší adaptaci, protože se rozdílné funkce učí sítěmi různých modulů, což vede k pozitivnímu ovlivnění tréninku.

2. *Modulární architektura neuronové sítě může mít výhody v případě generalizace.* Architektura sítě je obvykle svázaná s funkcí, na kterou je natrénována. Jestliže modulární architektura bude schopna rozložit komplexní funkci na množinu jednoduchých funkcí a náležitě přidělit každé jednoduché lineární síti jednoduchou funkci, pak lze očekávat i dobrou generalizaci systému. Proto by měl být takový mechanismus součástí daného procesu. Další příčina, proč modulární architektura neuronové sítě v určitých případech lépe generalizuje než monolitická síť, je způsobena rozdílem mezi lokální a globální generalizací. Globální generalizace se vyskytuje u monolitické architektury sítě, kdy tréninkové vzory vycházejí z rozsáhlé oblasti vstupního prostoru a naopak, modulární architektura vykonává lokální generalizaci v tom smyslu, že každá dílčí síť architektury se pouze učí vzory z omezené oblasti vstupního prostoru, proto i adaptace modulární architektury na

tréninkové vzory z jedné oblastí nemá vliv na trénink vzorů z jiných oblastí.



(a) (b)

Obrázek 43: Architektura vícervstvé sítě (a) monolitické; (b) modulární.

3. *Modulární architektura neuronové sítě má výhody v případě reprezentace vzorů trénovaní množiny.* Modulární architektura má tendenci vytvářet reprezentace, které jsou jednodušeji interpretovány než reprezentace vytvářené monolitickou sítí. Tím je míněno, že modulární architektura implementuje danou funkci pochopitelněji než monolitická síť. Tyto vlastnosti jsou demonstrovány např. v [30], kde skryté neurony pracují v oddělených sítích pro rozpoznávací úlohy a lokalizační úlohy a přispívají tak pochopitelnějším způsobem k řešení obou úloh, na rozdíl od monolitické sítě, kde skryté neurony řeší obě úlohy současně. Díky této vlastnosti modulárních sítí mohou i experimentátoři lépe pochopit, jak jejich systémy interpretují různé reprezentace vstupních dat. Pokud navíc jeden modul vykonává úlohu, která navazuje na úlohu vykonávanou jiným modulem, potom lze požadovat, aby byla tato úloha vykonávána až ve vhodné situaci. Máme-li k dispozici viditelné zobrazení procesu adaptace, má modulární architektura i další výhodu spojenou s použitím ošetřujícího mechanismu, kterým lze docílit toho, aby

adaptací vznikla úplná modulární architektura sítě bez zbytečných interferencí.

12.4 Vtahy mezi moduly

Dalším důležitým ukazatelem při používání modulární architektury neuronových sítí jsou ***vztahy mezi moduly navzájem***, které můžeme označit jako: (a) následné - *succesive*, (b) spolupracující - *cooperative*, (c) kontrolní - *supervisory*. "Následné" vztahy mezi moduly spatřujeme při dekompozici úlohy do postupně po sobě jdoucích jednotlivých dílčích úloh, kde každá z nich je řešena ve speciálním modulu. Tyto vztahy mezi moduly především existují v hybridních systémech, např. u systémů obsahující moduly pro rozpoznávání řeči, v systémech obsahující moduly pro předzpracování vstupních údajů, apod. Vstupní data u "spolupracujících" modulů jsou zpracována ve více speciálních modulech a do hlavního výpočtového modulu jsou pak buď vybrána předzpracována data pouze z jednoho modulu, nebo jsou výsledky z několika modulů vzájemně zkombinovány. Neuronová síť může být použita i jako hlavní výpočtový modul a všechny vstupní moduly (experti) jsou pak s ní spojeny. Na těchto spojích jsou adaptivně nastaveny váhové hodnoty, přičemž je vstupní signál komponován jako vážený signál jednotlivých modulů. "Kontrolní" vztahy mezi moduly mohou být realizovány v případě, kdy je jedna neuronová síť trénována tak, aby byla schopna nastavit parametry druhé neuronové sítě, nebo pokud je doplňující neuronová síť trénována pro predikci chyby hlavní neuronové sítě z daných vstupních a výstupních hodnot.

12.5 Vývoj modulární architektury neuronové sítě

Presentovaná metoda vychází z článku [16, 39]. Nejdůležitějším pojmem hill-climbing optimalizačního algoritmu je pravděpodobnostní vektor, který je zde používán při náhodném generování *n*-bitového vektoru formujícího jedince v příští generaci. Hlavním rysem uvedené metody je taková modifikace pravděpodobnostního vektoru, aby byl v

každé další generaci (tj. s každou jeho další aktualizací) posouván směrem k nejlepšímu řešení. Celý proces je opakován tak dlouho, dokud jeho složky neobsahují pouze hodnoty *1* (*true*) nebo *0* (*false*). Výsledný pravděpodobnostní vektor pak jednoznačně determinuje optimální řešení daného problému. Předložená metoda reprezentuje hodnotnou abstrakci genetických algoritmů a představuje jednoduchý model evoluční emergence modularity topologie neuronové sítě. Tato metoda vychází z principů evolučních algoritmů a vytváří modulární architekturu neuronové sítě pro řešení problému obsahujícího soubor různých problémů. Každý vzniklý modul pak reprezentuje jednu nezávislou část neuronové sítě - tj. jedno dílčí řešení zadaného problému. Postupně je tak prohledáván prostor řešení daného problému - charakterizovaného řetězci (chromozómy), přičemž každý z nich představuje jedno jeho potenciální řešení. Jednotlivé řetězce pak kódují architekturu neuronové sítě tak, že reprezentace každé síťové architektury je zakódovaná do odpovídajícího genotypu vycházejícího z matice spojení. Váhové hodnoty na spojeních mezi jednotlivými neurony jsou následně nastaveny metodou backpropagation. Vítězná modulární síťová architektura řešení je výsledkem procesu emergence při použití evolučních algoritmů. Moduly v neuronové síti emergují, tj. nevytváříme je přímo za použití penalizace, ale jsou důsledkem redukce interference mezi jednotlivými moduly neuronové sítě. Tento jednoduchý model emergence modularity topologie neuronové sítě je produktem jak procesu učení, tak i evoluce. Proces učení pak představuje algoritmus metody backpropagation.

Model se vyvíjí v procesu se zpětnou vazbou: (a) Fitness se počítá pomocí přesnosti klasifikace tréninkových objektů; (b) přesnost klasifikace je výsledkem chyby sítě po adaptaci metodou backpropagation. Po mnoha jeho etapách začnou v populaci emergovat neuronové sítě, které jsou optimalizované z hlediska modulární architektury spojení, ale i z hlediska k váhovým koeficientům po provedení procesu učení. Nezanedbatelným výstupem z uvedeného modelu je vedle vzniku modulární architektury spojení (nežádoucí

interference mezi moduly byly potlačeny) i velká redukce spojení vedoucí ke vzniku modulů neuronové sítě. Mechanismus tohoto „prunningu“ můžeme vysvětlit např. analogií v prudké redukci počtu spojení neuronů v mozku v dětském věku.

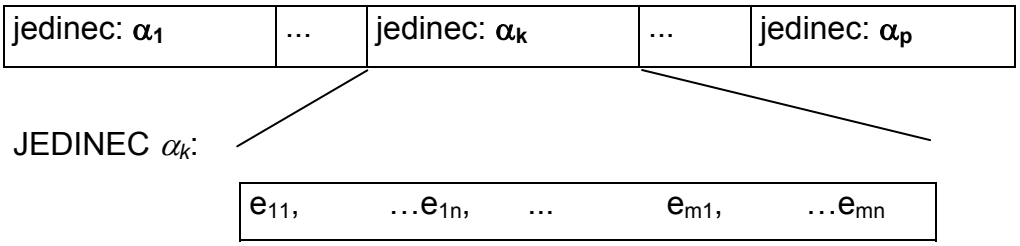
V následující části se podrobně seznámíme s navrhovanou metodou, která je založená na evolučních algoritmech a je vhodná pro optimalizaci modulární architektury neuronové sítě. Vycházíme z populace jedinců, jejíž velikost je konstantní (tj. počet jedinců v dané populaci se nemění). Každý jedinec tak představuje jednu vícevrstvou neuronovou síť s jednou skrytou vrstvou neuronů. Dříve než zahájíme výpočet, musíme v závislosti na řešené úloze definovat maximální možnou architekturu jedinců, tj. maximální počet vnitřních neuronů (počet vstupních a výstupních neuronů je určen řešeným problémem). Architektura neuronové sítě by měla odpovídat složitosti řešeného problému [34]: počtu tréninkových vzorů, jejich vstupů a výstupů a struktuře vztahů, které popisují. Je zřejmé, že *malá* síť nemůže řešit komplikovaný problém. Při učení pomocí algoritmu backpropagation se příliš malá síť obvykle zastaví v nějakém mělkém lokálním minimu, a proto je potřeba topologii doplnit o další skryté neurony tak, aby adaptace měla větší stupeň volnosti. Na druhou stranu *bohatá* architektura sice při učení mnohdy umožní nalézt globální minimum chybové funkce, ale s větším počtem vah roste i výpočetní náročnost adaptace. Nalezená konfigurace sítě však obvykle příliš zohledňuje tréninkové vzory včetně jejich nepřesností a chyb a pro nenaučené vzory dává chybné výsledky, tj. špatně generalizuje. Je tedy zřejmé, že existuje *optimální* topologie, která je na jednu stranu dostatečně bohatá, aby byla schopna řešit daný problém, a na druhou stranu ne moc velká, aby správně zobecnila potřebné vztahy mezi vstupy a výstupy. Základní heuristiky pro stanovení maximálního počtu neuronů ve vnitřních vrstvách neuronové sítě můžeme nalézt např. v knize [23], která je považována za jednu z nejlepších prakticky zaměřených knih.

Populace *P* je tvořena jedinci, $P = \{\alpha_1, \alpha_2, ..., \alpha_p\}$, kde *p* je kardinalita populace a vyjadřuje počet chromozómů v *P*, $p = |P|$. Každý jedinec v populaci je popsán chromozómem, což je binární vektor pevné délky

$\alpha_i = (e_{11}, ..., e_{1n}, ..., e_{m1}, ..., e_{mn})_i \in \{0,1\}^{mn}$, $(i=1,...p)$.

V počáteční populaci jsou binární číslice vygenerované náhodně s pravděpodobností 0.5. Chromozóm, který obsahuje *m* skrytých neuronů a *n* výstupních neuronů je znázorněn na obrázku 44, kde $e_{ij} = 0$, pokud spojení mezi *i-tým* neuronem ve skryté vrstvě a *j-tým* neuronem ve výstupní vrstvě jedince α neexistuje a $e_{ij} = 1$, pokud toto spojení existuje ($i = 1,\ldots,m$; $j = 1,\ldots n$). Spojení mezi neurony ve vstupní a skryté vrstvě nejsou součástí chromozómu, protože pro vytvoření modulární síťové architektury nejsou nezbytná.

Populace *P*:



Obrázek 44: Populace jedinců.

Každý jedinec (tj. síťová architektura jedince) je potom částečně adaptován metodou backpropagation a ohodnocen na základně výsledků tohoto učení. Chyba adaptační metody backpropagation je definována následovně [4, 7, 14]:

$$E = \frac{1}{2}\sum_{k=1}^{vz}\sum_{i=1}^{n}\left(y_i^{(k)} - o_i^{(k)}\right)^2, \qquad (40)$$

kde

vz je počet vzorů trénovací množiny;

n je počet neuronů ve výstupní vrstvě;

$y_i^{(k)}$ je skutečný výstup *i-tého* neuronu pro *k-tý* vzor trénovací množiny;

$o_i^{(k)}$ je očekávaný výstup *i-tého* neuronu pro *k-tý* vzor trénovací množiny.

Zvolit vhodný počet cyklů metody backpropagation pro částečnou adaptaci sítě je velice důležité kritérium, které odlišuje adaptaci modulární sítě od adaptace sítě monolitické. Adaptace modulární sítě probíhá zpočátku rychleji než adaptace odpovídající monolitické sítě [15, 30]. Je zřejmé, že našim cílem je získat takovou topologii sítě, která je schopna se naučit daný typ problému co nejrychleji a v odpovídající kvalitě (tj. pokud možno při zachování schopnosti predikce). Vycházíme zde z heuristiky, která závisí na složitosti řešeného problému a na velikosti topologie sítě. Pro každého jedince je po jeho adaptaci vyčíslen rozdíl mezi požadovaným a skutečným výstupem, tj. vypočtena chyba sítě podle vztahu (40) a všichni jedinci jsou pak na základě této vypočtené chyby sítě ohodnoceni podle vztahu (41).

Naší úlohou je nalézt takový *mn*-bitový vektor $\boldsymbol{\alpha_{opt}} \in \{0,1\}^{mn}$, který bude korespondovat s globálním minimem funkce *f* ($f : \{0,1\}^{mn} \to R$), tj. funkce vyjadřující průměrnou chybu po adaptaci sítě (= jedince) metodou backpropagation (viz vztah (38)). Vektor řešení $\boldsymbol{\alpha_{opt}}$ by měl splňovat vztah $\boldsymbol{\alpha}_{opt} = \arg \min_{\boldsymbol{\alpha} \in \{0,1\}^{mn}} f(\boldsymbol{\alpha})$, na prohledávaném prostoru $\boldsymbol{S} = \{0,1\}^{mn}$. Funkce *f* „reprezentuje“ prostředí, ve kterém existují chromozómy (= jedinci) populace. Mírou úspěšnosti jedince je jeho funkční hodnota. Protože hledáme minimum účelové funkce, chromozóm je tím úspěšnější, čím je jeho funkční hodnota menší. S tímto ohledem zavedeme ohodnocení fitness (*F*) populace $P \subseteq \{0,1\}^{mn}$ obsahující *p* chomozómů jako zobrazení chromozómů z populace *P* na kladná reálná čísla:

$$F : P \to R_+ .$$

Toto ohodnocení pak splňuje následující podmínku:

$$\forall \boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2 \in P : f(\boldsymbol{\alpha}_1) \leq f(\boldsymbol{\alpha}_2) \Rightarrow F(\boldsymbol{\alpha}_1) \geq F(\boldsymbol{\alpha}_2) \geq 0.$$

Hodnota chyby sítě (38) po částečné adaptaci metodou backpropagation je parametrem fitness funkce. Hodnota fitness funkce *i*-tého jedince je pak vypočtena podle následného vztahu:

$$F_i = \frac{\sum_{k=1}^{con} f_{ik}}{con} \tag{41}$$

kde

i = 1, …, *p* (*p* je počet jedinců v populaci);

k = 1, …, *con* (*con* je definovaná celočíselná konstanta větší než jedna);

E_{ik} je chyba adaptační metody backpropagation pro *i*-tého jedince v *k*-té adaptaci vyčíslená podle vztahu (38).

$f_{ik} = \frac{1}{E_{ik}}$ je hodnota fitness funkce pro *i*-tého jedince v *k*-té adaptaci (na začátku každé adaptace se prahové a váhové faktory sítě náhodně vygenerují vždy znovu a učení probíhá pomocí backpropagation pro každou adaptaci každého jedince konstantní počet kroků)

Konstanta *con* nám zajišťuje, že fitness hodnota F_i (tj. hodnota fitness funkce *i*-tého jedince) bude vypočtena ze statisticky významného množství dílčích adaptací sítě, proto by její hodnota měla být dostatečně velká.

Prezentovaná metoda vychází z paralelní verze hill-climbing algoritmu s učením [17, 18] a z důvodu lepší konvergence nepoužívá žádné

genetické operátory. Použití genetických operátorů (např. mutace nebo křížení) by bylo celkem bezvýznamné, protože v pravděpodobnostním vektoru je již náhodnost obsažena v dostatečné míře. Algoritmus je založen na emergenci pravděpodobnostního vektoru, který je zkvalitňován na základě nejlépe ohodnocených jedinců v populaci. Pravděpodobnostní vektor ***w*** zapíšeme:

$\boldsymbol{w} = (w_{11}, \ldots, w_{1n}, \ldots, w_{m1}, \ldots, w_{mn}) \in [0,1]^{mn}$,

kde *mn* je počet složek pravděpodobnostního vektoru (*m* je počet neuronů ve skryté vrstvě a *n* je počet neuronů výstupní vrstvě). Jeho jednotlivé složky $0 \leq w_{ij} \leq 1$ určují pravděpodobnost výskytu proměnné '*1*' v dané pozici.

Řešení *R* optimalizačního problému, můžeme s ohledem na evoluci pravděpodobnostního vektoru $\boldsymbol{w} \in [0,1]^{mn}$ zapsat následovně:

$$R : [0,1]^{mn} \rightarrow \{0,1\}^{mn},$$

potom v čase $t \rightarrow \infty$ nalezne algoritmus řešení:

$$\boldsymbol{w}_{opt} = \boldsymbol{\alpha}_{opt} = \lim_{t \rightarrow \infty} \left(\arg \min_{\boldsymbol{\alpha} \in P_t} f(\boldsymbol{\alpha}) \right),$$

kde P_t je populace jedinců α_i ($i = 1, \ldots, p$) v čase *t*.

Pokud všechny složky pravděpodobnostního vektoru budou blízko nuly nebo jedničky, je binární vektor $\boldsymbol{\alpha} \in \{0,1\}^{mn}$ jednoznačně určen pravděpodobnostním vektorem $\boldsymbol{w} \in [0,1]^{mn}$ (tj. zaokrouhlením jeho složek na 0 nebo 1). Můžeme tedy zapsat:

$\boldsymbol{w}' \leftarrow \boldsymbol{w} + \lambda (\boldsymbol{\alpha} - \boldsymbol{w})$,

tj. nový pravděpodobnostní vektor $\boldsymbol{w}'$ leží blíže nejlepšímu řešení $\boldsymbol{\alpha}$.

V každé další generaci určíme složky pravděpodobnostního vektoru následujícím způsobem:

(1) Vypočítáme hodnotu F_{prum} , tj. průměrnou hodnotu fitness funkce jedinců pro populaci v daném časovém kroku podle vztahu:

$$F_{prum} = \frac{\sum_{i=1}^{p} F_i}{p}, \tag{42}$$

kde

p je počet jedinců v populaci,

F_i je hodnota fitness funkce *i-tého* jedince vypočítaná podle vztahu (4.8).

(2) Vybereme množinu q jedinců s hodnotou fitness funkce (F_i) větší, než je průměrná hodnota fitness funkce v populaci (F_{prum}),
tj. $\alpha_1, \alpha_2, \ldots, \alpha_q$ ($1 \leq q \leq p$, kde p je počet jedinců v populaci).
Pro hodnotu fitness funkce každého vybraného jedince musí být tedy splněno

$F_i \geq F_{prum}$,

kde

i = 1, …, q (q je počet vybraných jedinců v populaci splňující danou nerovnost);

(3) Jednotlivé složky pravděpodobnostního vektoru pro populaci v daném časovém kroku $w'_k \in [0,1]$ určíme podle vztahu

$$w'_k = (1-\lambda) w_k + \lambda w''_k \tag{43}$$

kde

$k = 1, \ldots, mn$ (mn je počet složek pravděpodobnostního vektoru $\boldsymbol{w}$);

w_k je hodnota k-té složky pravděpodobnostního vektoru v předchozí generaci.

λ je konstanta ($0 < \lambda < 1$);

w_k'' je hodnota *k-tého* bitu pravděpodobnostního vektoru ***w*** vypočítaného jako průměrná hodnota *k-tých* složek chromozómů u jedinců α_i (*i = 1, ...,q*) s hodnotou fitness funkce $F_i \geq F_{prum}$;

počítáme jej podle vztahu:

$$w_k'' = \frac{\sum_{i=1}^{q} (e_k)_i}{q} . \qquad (44)$$

$(e_k)_i$ je hodnota *k-tého* bitu chromozómu jedince α_i (*i = 1, ..., q*) v populaci;

a zároveň platí, že hodnota jeho fitness funkce (F_i) je $F_i \geq F_{prum;}$

q je počet jedinců s hodnotou fitness funkce $F_i \geq F_{prum}$ (*i* = 1, …, *q);*

Je – li v čase $t \to \infty$ $\boldsymbol{\alpha}$ je nejlepší řešení, tj. je – li $\lim_{t\to\infty} \boldsymbol{w} = \boldsymbol{\alpha}$, pak platí:

$$\boldsymbol{w'} \leftarrow (1 - \lambda)\, \boldsymbol{w} + \lambda\, \boldsymbol{\alpha} , \qquad (45)$$

což po úpravě odpovídá vztahu:

$$\boldsymbol{w'} \leftarrow \boldsymbol{w} + \lambda\, (\boldsymbol{\alpha} - \boldsymbol{w}) . \qquad (46)$$

Do populace je pro další generaci automaticky zařazen jedinec s nejvyšší hodnotou fitness funkce. Hodnoty chromozómů ostatních jedinců v populaci α_i (*i = 2, …,p*) jsou stanoveny následovně: například je-li w_k *= 0(1),* potom *k-tý* gen *i-tého* jedince α_i má v chromozómu na dané pozici hodnotu $(e_k)_i$ *= 0(1);* pro *0 < w_k < 1* je proměnná $(e_k)_i$ náhodně určena vztahem

$$(e_k)_i = \begin{cases} 1 & \text{pokud } random < w_k \\ 0 & \text{jinak} \end{cases} . \qquad (47)$$

kde *random* je náhodné číslo z intervalu [0, 1) s rovnoměrnou distribucí.

Samotný proces evolučních algoritmů je ukončen, pokud je parametr nasycení (kap.2) větší než definovaná hodnota.

Prezentovanou metodu lze vnímat jako určité propojení mezi genetickým algoritmem a hill-climbing algoritmem s učením. Metoda sice pracuje s populací chromozómů (viz *genetické algoritmy*), ale dochází zde jak k evoluci populace (viz *genetické algoritmy*), tak i pravděpodobnostního vektoru (viz *hill-climbing s učením*), který je zkvalitňován na základě nejlépe ohodnocených jedinců v populaci (viz *genetické algoritmy*). Z takto aktualizovaného pravděpodobnostního vektoru jsou vygenerovaní jedinci pro novou populaci s tím, že nejlépe ohodnocený jedinec přechází do nové populace automaticky (viz elitismus v *genetických algoritmech*). Podmínka ukončení výpočtu odpovídá *hill-climbing algoritmu*, tj. parametr nasycení je větší než definovaná hodnota.

Kontrolní otázky:

1. Jaké jsou znáte typy modulární architektury neuronových sítí.
2. Jaké jsou vlastnosti modulární architektury neuronových sítí
2. Jak se vytváří modulární architektura neuronových sítí

Úkoly k zamyšlení:

Zamyslete se nad výhodami a nevýhodami modulární architektury neuronových sítí.

Korespondenční úkol:

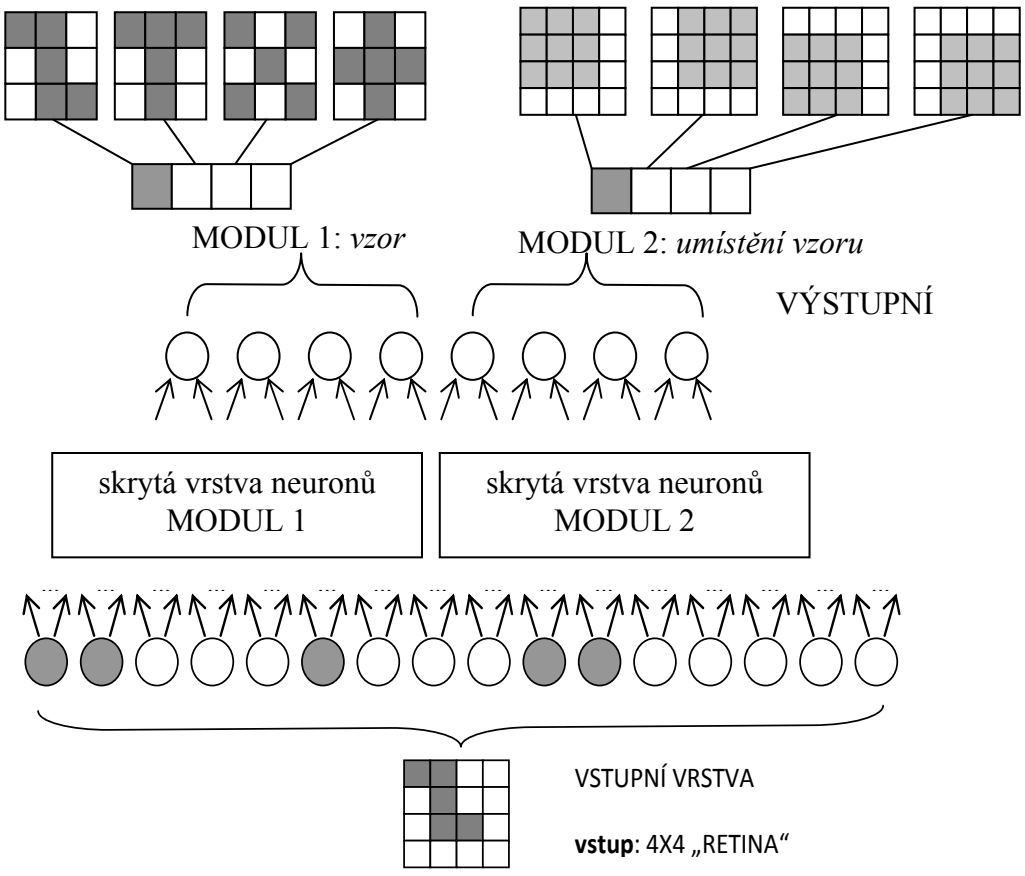
Neuronová síť z obrázku 44 musí: (a) rozpoznat vzor (b) rozpoznat jeho umístění na sítnici („retina“). Základní vzory trénovací množiny jsou organizovány do matice (mřížky) typu 3x3, která je reprezentována

devítisložkovým binárním vektorem a jeho umístění je charakterizováno čtyřmi možnostmi. Trénovací množina obsahuje čtyři vzory, z nichž je každý definován ve čtyřech různých pozicích. Celkový počet vstupních vektorů trénovací množiny je pak 16. Neurony ve vnitřní a výstupní vrstvě patří do dvou subsystémů detekujících: (1) tvar objektu a (2) jeho umístění na sítnici.

(a) Úlohu řešte *monolitickou* neuronovou síti s topologií 16 – 16 –8. (tj. 16 neuronů ve vstupní vrstvě, 16 neuronů ve vnitřní vrstvě a 8 neuronů ve výstupní vrstvě).

(b) Úlohu řešte *modulární* neuronovou síti s topologií 16 – 16 –8. Každý modul má topologii 16 – 8 - 4.

Srovnejte průběh obou adaptací.



Obrázek 44: Grafické znázornění úlohy pro korespondenční úkol

Shrnutí obsahu kapitoly

V této kapitole jste se seznámili se základními principy modularity, jež má významnou úlohu nejen v neuronových sítích, ale např. i v kognitivních vědách, kde patří mezi základní atributy. Důraz v této kapitole byl kladen na vývoj modulární architektury vícevrstvé neuronové sítě založené na restrikci vzájemného propojení neuronů.

Pojmy k zapamatování

- modulární architektura neuronové sítě,
- Artmap,
- bránová síť,
- asociativní moduly,
- dekompozice úlohy.

Literatura

[1] Bäck, T., Ho_meister F., Schwefel, H.-P. (1993). Applications of Evolutionary Algorithms, TR SYS-2/92, University of Dortmund

[2] Bäck, T. (1996). Evolutionary Algorithms in Theory and Practice. Oxford University Press, New York.

[3] Back T., Fogel D. B., Michalewicz Z. (1997), Handbook of evolutionary algorithms, Oxford University Press, , ISBN 0750303921

[4] Beale, R. - Jackson, T. (1992) Neural Computing: An Introduction. J W Arrowsmith Ltd, Bristol, Greit Britain.

[5] Banzhaf W. (ed.): Genetic Programming and Evolvable Machines, Vol. 7, Nr. 1, March, 2006, Springer, ISSN: 1389-2576.

[6] Davis, L. (1991) Handbook of genetic algorithms. Van Nostrand Reinhold, New York.

[7] Fausett, L. V. (1994) Fundamentals of Neural Networks. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

[8] Ferreira C. (2006) Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence, Springer, , ISBN: 3540327967

[9] F. Glover:Tabu Search - Part I. ORSA Journal of Operations Research, 1 (1989),190.

[10] F. Glover:Tabu Search - Part II. ORSA Journal of Operations Research, 2 (1990),4.

[11] Goldberg, D. E. (1989) Genetic algorithm in search optimalization and machine learning. Addison-Wesley, Reading., Massachusets.

[12] Hugosson J., Hemberg E., Brabayon A., O'neill M. Genotype Representations in Grammatical Evolution. Applied Soft Computing, pp.36-43 Vol.10, 2010

[13] Hussain, T. (1995) Modularity within neural networks. Queen´s University Kingston, Ontario, Canada.

[14] Herz, J. - Krogh, A. - Palmer, R. G. (1991) Introduction to the Theory of Neural Computation. Addison Wesley Publishing Company, Redwood City.

[15] Jacobs, R. (1990) The task decomposition through competition in a modular connectionist architecture. Technical Report 90-44, University of Massachusetts.

[16] Koza J. R. (1998) Genetic Programming, MIT Press, , ISBN 0-262-11189-6

[17] Koza J. R. et al. (1999) Genetic Programming III; Darwinian Invention and problem Solving, Morgan Kaufmann Publisher, , ISBN 1-55860-543-6

[18] V. Kvasnička, J. Pospíchal, and M. Pelikán: Hill Climbing with Learning (An Abstration of Genetic Algorithm). Proceedings of Mendel'95, First International Conference on Genetic Algorithms, Brno, September 26-28, 1995, pp.65-70.

[19] Kvasnička, V., Pelikán, M., Pospíchal, J. Hill climbing with learning (an abstraction of genetic algorithm). Neural network world (1996) **5** 773-796.

[20] Kvasnička, V., Beňušková, L., Pospíchal, J., Farkaš, I., Tiňo, P., Kráľ, A. (1997) Úvod do teórie neurónových sietí. IRIS, Bratislava.

[21] Kvasnička, V., Pospíchal, J., Tiňo, P. (2000) Evolučné algoritmy. STU Bratislava.

[22] Langdon, W. B. (2000) Genetic Programming and Data Structures, Spriner, 1998, ISBN 0-7923-8135-1..

[23] Masters, T. (1993) Practical neural network in C++. Academic Press.

[24] Míka, S. (1997). Matematická optimalizace. vydavatelství ZČU, Plzeň.

[25] Michalewicz, Z. (1992). Genetic Algorithms + Data Structures = Evolution Programs. Berlin, Springer Verlag.

[26] O'Neill M., Ryan C. (2003) Grammatical Evolution. Evolutionary Automatic Programming in an Arbitrary Language, Kluwer Academic Publishers, , ISBN 1402074441

[27] Oltean M., Grosan C. (2003) A Comparison of Several Linear Genetic Programming Techniques, Complex Systems,

[28] Oplatková, Z. (2009) Metaevolution - Synthesis of Optimization Algorithms by means of Symbolic Regression and Evolutionary Algorithms. Saarbrücken : Lambert-Publishing,. ISBN: 978-8383-1808-0.

[29] Pospíchal, J. (1996) Evolučné optimalizačné algoritmy. Habilitační práce. Bratislava,.

[30] Rueckl, J. G.: Why are “What” and “Where” processed by separate cortical visual systems? A computational investigation. *Journal of Cognitive Neuroscience* **2** (1989) 171-186.

[31] Spall J. C. (2003). Introduction to Stochastic Search and Optimization, Wiley-Intersience.

[32] O'Sullivan J., Ryan C. (2002) An Investigation into the Use of Different Search Strategies with Grammatical Evolution, Proceedings of the 5th European Conference on Genetic Programming, p.268 - 277, Springer-Verlag London, UK, ISBN:3-540-43378-3

[33] Storn, R., Price, K. (1997) Diferential Evolution a Simple and Efficient Heuristic for Global Optimization. J. Global Optimization, 11, 341-359.

[34] Šíma, J., Neruda, J. (1996) Teoretické otázky neuronových sítí. Matfyzpress, Praha.

[35] Törn, A., Žilinskas A. (1989). Global Optimization, Lecture Notes in Computer Science, No. 350, Springer.

[36] Tvrdík, J. (2010) Evoluční algoritmy. Distanční studijní opora. Ostravská univerzita v Ostravě,.

[37] del VALLE, Y. a kol. Particle swarm optimization: Basic concepts, variants and applications in power systems. In *IEEE Transactions on Evolutionary Computation*. sv.12, č. 2, 2008. s. 171-195

[38] VESELÝ, F. (2010) Aplikace optimalizační metody PSO v podnikatelství. Brno: Vysoké učení technické v Brně, Fakulta podnikatelská,. 66 s. Vedoucí diplomové práce doc. Ing. Petr Dostál, CSc.

[39] Volná, E. (2003) Modularita neuronových sítí. Disertační práce STU Bratislava.Vedoucí Doc. RNDr. Jiří Pospíchal, CSc.

[40] Volná, E. Evoluční algoritmy. *Automatizace*. ročník 51, číslo 2 (2008) pp.91-94. ISSN 0005-125X.

[41] Yao, X. (1999) Evolving artificial neural networks. In Proceedings of the IEEE 89 (9) 1423-1447,.

[42] Zelinka, I. (1998) Umělá intelligence I. VUT v Brně, Brno.

[43] Zelinka I. (2001) Prediction and analysis of behavior of dynamical systems by means of artificial intelligence and synergetic, Ph.D. thesis,

[44] Zelinka I. (2002) Umělá inteligence v problémech globální optimalizace, BEN, Praha, , ISBN 80-7300-069-5

[45] Zelinka I. (2004) SOMA – Self Organizing Migrating Algorithm“, In: Babu B.V., Onwubolu G. (eds), New Optimization Techniques in Engineering, Springer-Verlag, , ISBN 3-540-20167X

[46] Zelinka I., Oplatkova Z, Nolle L., Boolean Symmetry Function Synthesis by Means of Arbitrary Evolutionary Algorithms-Comparative Study, International Journal of Simulation Systems, Science and Technology, Volume 6, Number 9, August 2005, pages 44 - 56, ISSN: 1473-8031,

[47] Zelinka, I., Oplatková, Z., Ošmera, P., Šeda, M., Včelař, F. (2008) Evoluční výpočetní techniky - principy a aplikace. BEN - technická literatura, Praha, , ISBN 80-7300-218-3.

[48] Zelinka, I., Davendra, D., Šenkeřík, R., Jašek, R., Oplatková, Z.: Analytical Programming - a Novel Approach for Evolutionary Synthesis of Symbolic Structures. In Evolutionary Algorithms. Rijeka: InTech, 2011, s. 149-176. ISBN 978-953-307-171-8