VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

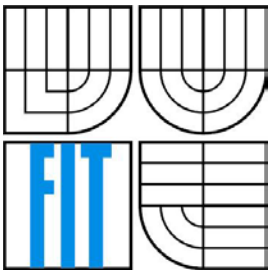
STROJOVÉ UČENÍ - APLIKACE PRO DEMONSTRACI ZÁKLADNÍCH PŘÍSTUPŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Pavel Kefurt
AUTHOR

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

STROJOVÉ UČENÍ - APLIKACE PRO DEMONSTRACI ZÁKLADNÍCH PŘÍSTUPŮ

MACHINE LEARNING - THE APPLICATION FOR DEMONSTRATION OF MAIN APPROACHES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Pavel Kefurt
AUTHOR

VEDOUCÍ PRÁCE — Doc. Ing. František Vítězslav Zbořil, CSc.
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce se zabývá především základními algoritmy strojového učení. V první části práce jsou vybrané algoritmy popsány. Zbývající část se následně věnuje implementaci těchto algoritmů a vytvoření demonstračních úloh pro každý z nich.

Abstract

This work mainly deals with the basic machine learning algorithms. In the first part, the selected algorithms are described. The remaining part is then devoted to the implementation of these algorithms and a demonstration of tasks for each of them.

Klíčová slova

Umělá inteligence, strojové učení, učení s učitelem, učení bez učitele, posilované učení, ID3 algoritmus, Back Propagation, K-Means clustering, Self-Organizing Maps, SOM, Q-Learning, State-Action-Reward-State-Action, SARSA

Keywords

Artificial learning, machine learning, supervised learning, unsupervised learning, reinforcement learning, ID3 algorithm, Back Propagation, K-Means clustering, Self-Organizing Maps, SOM, Q-Learning, State-Action-Reward-State-Action, SARSA

Citace

Kefurt Pavel: Strojové učení - aplikace pro demonstraci základních přístupů, bakalářská práce, Brno, FIT VUT v Brně, 2013

Strojové učení - aplikace pro demonstraci základních přístupů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Doc. Ing. Františka Vítězslava Zbořila, CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Pavel Kefurt
15. 5. 2013

Poděkování

Tímto bych chtěl poděkovat především mému vedoucímu bakalářské práce Doc. Ing. Františku Vítězslavu Zbořilovi, CSc. za jeho cenné rady a čas, který mi věnoval při řešení této práce.
V neposlední řadě také rodině, která mě celou dobu podporovala.

Obsah

1 Úvod

Strojové učení je jeden z velkých segmentů oblasti nazývané Umělá inteligence. Jak již název napovídá, celé odvětví umělé inteligence se snaží všemi možnými prostředky docílit toho, aby si stroje osvojily jednu ze schopností, kterou v současné době disponují pouze živé organismy včele s člověkem, který je doteď považován za jedinou živou bytost se schopností uvědomění si svojí existence.

Nabízí se jednoduchá otázka. Proč lidé chtějí, aby stroje uměly samy nezávisle myslet?

Odpověď, ač na první pohled nemusí být jasná, je zcela jednoduchá. Je to lenost. Lenost druhu homo sapiens. To je přesně ten důvod, proč vznikly všechny pomůcky, které v současné době používáme, včetně počítačů. Lidé jsou od přírody pohodlní, a proto vytvářejí nástroje, které jim mohou usnadnit práci. A to je i tento případ. Počítač je super věc, ale na spoustu věcí nestačí, protože není schopen sám nezávisle učinit rozhodnutí ve zcela nové situaci. A to je pro líného člověka problém, protože by musel u toho stroje sedět a říkat mu co má dělat. Proto se hledají nástroje, jak docílit toho, aby stroj dokázal v dnešním informačním věku, kdy objem informací každým dnem obrovsky narůstá, všechno zpracovat za co možná nejmenší asistence člověka.

Pokud se podíváme na dnešní svět, všude uvidíme výsledky tohoto snažení. Stačí se jen podívat například na svůj mobilní telefon. V dnešní době má již většina obyvatel moderního světa vlastní chytrý mobilní telefon, který kromě samotného telefonování a posílání SMS zpráv dokáže přijímat povely pomocí hlasu. Zvládne dokonce odesílat zprávy, které mu jeho majitel nadiktuje. A právě toto je jedno konkrétní využití umělé inteligence (pomocí strojového učení bylo dosaženo schopnosti rozpoznat jednotlivé hlásky/slova a následně je převést do psaného textu), což usnadňuje línému člověku život, protože místo mačkání kláves (namáhavé a zdlouhavé) stačí pouze slovně zadat povel - třeba nadiktovat SMS zprávu, která bude odeslána příjemci.

Kromě výše zmíněného příkladu se strojové učení používá napříč všemi možnými odvětvími. Velké uplatnění má v průmyslu, kde je možné nechat řízení některých částí výrobního procesu pod kontrolou počítače místo člověka. Například, pokud necháme algoritmus dostatečně dlouho sledovat práci daného zaměstnance, po určitém čase bude program schopen vykonávat stejnou práci 24 hodin denně bez nároku na mzdu. A to je už pro zaměstnavatele velice zajímavá nabídka.

Velmi zajímavých výsledků v oblasti strojového učení dosahuje společnost IBM a její projekt Watson[1]. Tento počítač je schopen na položenou otázku v přirozeném jazyce správně odpovědět, a to velmi rychle. Dokonce dokázal ve vědomostní soutěži Jeopardy (v česku známé pod názvem Riskuj) porazit dva nejlepší hráče planety. Jeho praktické využití je směřováno především do oblasti medicíny, kde je schopen nalézt ideální léčbu pro konkrétního pacienta, a bankovnictví, kde může

[1]Stránka projektu: http://www-03.ibm.com/innovation/us/watson

provádět analýzu rizikovosti úvěrů, apod. Samozřejmě existuje mnohem více možností jeho využití a všechny mají jedno společné – usnadnit člověku práci.

Strojové učení nepřišlo jen tak z nebe. Ve skutečnosti se vyvíjí již několik desítek let a neustále vychází ze stejných základních principů, které byly vymyšleny již před více než 50-ti i více lety. A to je důvod, proč se tato práce zabývá především vysvětlováním a ukazováním základních přístupů ke strojovému učení, neboť obdoby níže popsaných algoritmů můžete nalézt v méně či více pozměněné formě i v těch nejnovějších aplikacích.

2 Rozdělení

Tato část práce popisuje základní rozdělení strojového učení do tří skupin a jejich následný popis se zaměřuje na to, jak jednotlivé metody fungují, jaké jsou mezi nimi rozdíly a jaké algoritmy jsou typickými zástupci daných principů.
Strojové učení lze rozdělit do tří hlavních kategorií. A to:

- Učení s učitelem (supervised learning)
- Učení bez učitele (unsupervised learning)
- Posilované učení (reinforcement learning)

První dva způsoby se vyznačují tím, že algoritmus dostane dostatečně velkou a různorodou trénovací množinu příkladů a na jejím základě se naučí požadované chování. Rozdíl mezi nimi je pouze v tom, zda mají k dispozici ještě nějaké další doplňující informace anebo si musí vystačit bez nich. Zatímco třetí skupina funguje tak, že zadáme manévrovací prostor s různými omezeními/výhodami a algoritmus si následně sám projde celý prostor a na základě průběžného zaznamenávání nových poznatků se naučí daný prostor chápat.

2.1 Učení s učitelem

je první z možných přístupů. Zároveň se jedná o nejčastěji používaný segment strojového učení. Pokud budete hledat nějaký algoritmus s tímto typem přístupu, čeká vás výběr z nepřeberného množství různých variant.

Základní princip strojového učení s učitelem spočívá, jak již samotný název napovídá, v učení (často označovaném jako trénování) daného algoritmu s pomocí učitele, tedy za pomoci někoho či něčeho, co bude algoritmu pomáhat s rozpoznáním, jak danou informaci interpretovat. Je to stejné, jako když my máme svého učitele, který nám pomáhá rychleji a přesněji pochopit nějakou problematiku a v případě, že uděláme chybu, nás opraví a ukáže nám, jaký měl mít například daný výpočet výsledek. Stejně tomu je i v případě učení (trénování) našeho algoritmu. Kromě samotného vstupu, který je zapotřebí zpracovat, poskytneme algoritmu i informaci o správném/očekávaném výstupu. A ten ji může nějakým způsobem využít ke svému učení. Jednou z možností je například zpracování vstupu a následné porovnání vypočteného výsledku s očekávaným s tím, že se případný rozdíl mezi těmito hodnotami promítne do následně prováděných operací. Tedy poučí se ze své chyby. Celý trénovací proces pokračuje do doby, dokud rozdíl mezi očekávaným a vypočteným výstupem neklesne pod určitou hodnotu. Data používaná k trénování těchto algoritmů označujeme jako trénovací množinu. Ta se skládá z jednotlivých prvků označovaných jako příklady. Ty jsou tvořeny dvojicemi vektorů (či jinak definovaných dat), kde jeden vektor je množina vstupních hodnot

a druhý je množina požadovaných/správných výstupních hodnot. Tedy trénovací množina T s počtem příkladů m může vypadat následovně:

$$\mathrm{T} = \{(\overrightarrow{x_1}, \overrightarrow{y_1}), (\overrightarrow{x_2}, \overrightarrow{y_2}), \dots, (\overrightarrow{x_m}, \overrightarrow{y_m})\}$$

kde

$\overrightarrow{x_i}$ je i-tý vstupní vektor a

$\overrightarrow{y_i}$ je i-tý výstupní vektor.

Samotný algoritmus spočívá v tom, že se snaží hledat takovou funkci $g: X \to Y$, kde X je vstupní a Y výstupní prostor. Funkce g patří do prostoru G, obvykle označovaného jako prostor hypotéz.

Ovšem samotné nalezení této funkce nezaručuje efektivní zpracování. Velmi často totiž existuje více možností g, ze kterých lze vybírat. Proto základní způsob, jak zvýšit výkon algoritmu spočívá ve správném výběru funkce g. Toho lze docílit například pomocí hodnotící funkce $f: X \times Y \to \mathbb{R}$. Potom $g(x) = \arg\max_y f(x, y)$ vybere takové y, které zajistí maximální hodnotu.

Dva základní algoritmy s různými přístupy k tomuto typu učení, kterými se dále budu zabývat, jsou:

- ID3 algoritmus (založen na rozhodovacích stromech)
- Back Propagation (založen na neuronových sítích)

2.2 Učení bez učitele

Na rozdíl od předchozího přístupu, kde jsme měli k dispozici doplňující informace, které algoritmu pomáhaly s učením, v tomto případě žádné další informace k dispozici nejsou. Jedinou možnou výjimku tvoří počet skupin, do kterých se má vstup rozdělit.

Celkově se dá tento přístup znovu přirovnat k tomu, jak probíhá učení lidí, ovšem tentokrát k té části učení, kterou absolvuje každý člověk sám bez pomoci druhého. Jedná se o takové situace, kdy nám nikdo neřekne, jaké řešení je správné, a záleží pouze na nás, jakým způsobem danou situaci vyřešíme, zda zvolíme cestu nejmenšího odporu nebo nějakou obtížnější, která nám přinese šťavnatější ovoce.

Z výše uvedeného vyplývá, že tentokrát bude trénovací množina chudší o jednu složku, a to o tu, ve které byla v předchozím typu učení uložena očekávaná hodnota výstupu. Tedy zbývá nám pouze samotný vstup. Ten je opět tvořen vstupními hodnotami. Většinou se u těchto dat bavíme o číselných vektorech v n-rozměrném prostoru, což v případě trénovací množiny T s počtem příkladů m jde zapsat následujícím způsobem:

$$\mathrm{T} = \{\overrightarrow{x_1}, \overrightarrow{x_2}, \dots, \overrightarrow{x_m}\}$$

kde $\overrightarrow{x_i}$ je i-tý n-dimenzionální vstupní vektor.

Jak jsem již naznačil výše, algoritmy založené na tomto stylu učení velmi často rozdělují vstupní data do skupin. Ty symbolizují nějaké společné nebo velmi podobné vlastnosti pro všechny prvky dané skupiny, jako je například barva nebo poloha v prostoru. Většinou takovéto skupiny označujeme jako shluky. Této problematice se podrobně věnuje technika nazývaná shluková analýza. Ta se snaží o identifikaci jednotlivých shluků v n-rozměrném prostoru a jejich následnou klasifikaci/rozdělení do jednotlivých podobnostních tříd.

Obecně lze říci, že pro rozpoznávání shluků se využívá vzdálenost mezi jednotlivými objekty tak, že na základě tohoto údaje se určí, ke které skupině má daný objekt nejblíže. Celkově by mělo platit, že jednotlivé prvky třídy si jsou mezi sebou mnohem podobnější než prvky z rozdílných tříd.

Základní vlastnosti vzdálenosti d mezi dvěma objekty O_1 a O_2 jsou:

1. nezápornost: $d(O_1, O_2) \geq 0$
2. symetrie: $d(O_1, O_2) = d(O_2, O_1)$
3. shodné objekty mají vzdálenost rovnu nule: $d(O_1, O_1) = 0$
4. mezi třemi různými objekty platí trojúhelníková nerovnost: $d(O_1, O_2) \leq d(O_1, O_3) + d(O_3, O_2)$

Dále v této práci najdete následující algoritmy založené na učení bez učitele:

- K-Means clustering
- Self-Organizing Maps (SOM)

2.3 Posilované učení

Tento typ učení funguje na úplně jiném principu než obě předchozí skupiny. Zatímco u učení s učitelem a učení bez učitele předáváme algoritmu trénovací množinu s příklady a následně očekáváme výsledek v podobě nějaké konkrétní hodnoty (s učitelem) nebo třídu klasifikace (bez učitele), v tomto případě od nás algoritmus dostane stavový prostor, ve kterém pobíhá kompletní činnost algoritmu. Výstupem je pak cesta. Jako cestu lze uvažovat posloupnost akcí, které je zapotřebí učinit k tomu, abychom se dostali ze startovacího stavu do stavu cílového.

Stejně jako v ostatních případech i tento typ učení lze připodobnit k učení lidí. Tentokrát si paralelu můžeme představit jako případ, kdy se přestěhujeme do nového města, které vůbec neznáme. To se ovšem postupem času mění, protože se v něm začneme pohybovat a postupně ho poznáváme. Čím déle v daném městě žijeme, tím více uliček máme prozkoumaných a tím lépe umíme najít například nejkratší cestu do našeho oblíbeného obchodu. Jde hlavně o to, že postupně chodíme i do míst, kde jsme ještě nebyli, a rozšiřujeme si tak své obzory o nové možnosti, které se nám mohou někdy v budoucnu hodit. Výhodou je, že jakmile jednou danou uličku poznáme, při jiné příležitosti ji můžeme použít, protože již víme, kam vede.

Chování algoritmů založených na tomto principu je stejné. Jen místo nás se pohybuje v novém městě (stavovém prostoru) postava, kterou nazýváme agent. Tento agent postupně prochází celý

stavový prostor pomocí nějaké konkrétní zvolené strategie tak, aby se nakonec dostal k cílové pozici, kterou může představovat například supermarket.

Agent buď využívá stále stejnou strategii, nebo ji v průběhu výpočtu mění. To záleží na tom, do které skupiny algoritmus patří. Možnosti máme dvě, a to:

- aktivní politika (On-policy)
- pasivní politika (Off-policy)

V případě aktivní politiky si algoritmus může průběžně svoji strategii měnit, a tím se přizpůsobovat nově získaným poznatkům (využívat nově získané poznatky). V případě pasivní politiky je na začátku zvolena jedna konkrétní strategie a ta se již nemění.

Vybraná strategie agentovi slouží především k rozhodnutí, jakou další akci má vybrat k získání nového (lepšího) stavu. Mezi hlavní možné strategie patří především:

- ε-greedy (hladová strategie)
    - Vybírá akci vedoucí k největší možné odměně.
    - Pouze s malou pravděpodobností ε vybírá náhodnou akci.
    - Vhodný pro získání optimální cesty.
- softmax [1]
    - Je vhodná v případech, kdy nechceme při náhodném výběru vybírat ty nejhorší možné akce.

Náhodný výběr akcí je velice důležitý, protože díky němu je agent schopný dále objevovat nové možnosti stavového prostoru.

Již několikrát jsem zde zmínil pojem stavový prostor. Jedná se o prostor skládající se ze stavů, které mohou obsahovat různé druhy překážek, bonusů či penalizací. V každého stavu může existovat nula až n akcí, které vedou do dalšího stavu. V celém prostoru se ještě navíc musí nacházet minimálně jeden druh speciálního stavu, a to cílový stav. Ten může být buď jeden, anebo jich může existovat pro daného agenta hned několik.

Celkově posilované učení většinou potřebuje znát následující informace:

- množinu stavů S
- množinu možných akcí A
- pravidla pro přechodové funkce
- pravidla pro určení odměny po přechodu do nového stavu
- definované cíle

Pokud bychom tento seznam porovnali s Markovskými rozhodovacími procesy, které jsou definovány takto:

Markovský rozhodovací proces je uspořádaná čtveřice $(S, A, P.(\cdot,\cdot), R.(\cdot,\cdot))$, *kde*

- S *je konečná množina stavů,*

- *A je konečná množina akcí (alternativa: A_s je konečná množina akcí dostupných ve stavu s),*
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ *je pravděpodobnost, že akce a ve stavu s v čase t povede do stavu s' v čase $t + 1$,*
- $R_a(s, s')$ *je okamžitý užitek (ne očekávaný okamžitý užitek) dosažený po přechodu stavu na s' ze stavu s s pravděpodobností přechodu $P_a(s, s')$.*

(Teorie Markovských rozhodovacích procesů nevyžaduje, aby S nebo A byly konečné množiny) [2]

můžeme si povšimnout, že posilované učení z těchto procesů vychází.

Většina agentů je stavěna pro práci v diskrétním prostoru. Uvažujme tedy diskrétní čas t jako jeden krok, ve kterém agent získá informaci o prostředí/situaci o_t. Součástí této informace většinou bývá i užitná hodnota daného kroku r_t. Následně se provede volba akce a_t z množiny přípustných akcí. Po provedení této akce se prostředí dostává do nového stavu s_{t+1} s užitnou hodnotou r_{t+1}. Tuto akci můžeme nazvat přechod se zápisem (s_t, a_t, s_{t+1}). Cílem agenta je získat svými přechody co možná největší užitek.

Dále se budu zabývat následujícími algoritmy:

- Q-Learning
- State-Action-Reward-State-Action (SARSA)

3 Algoritmy

V této kapitole se budu věnovat již konkrétním algoritmům. Všechny zde zmíněné algoritmy spadají do jedné z výše popsaných kategorií. Při jejich výběru jsem dbal především na to, aby co možná nejlépe odrážely základní principy své skupiny.

3.1 ID3

Tento algoritmus spadá do skupiny strojového učení s učitelem.

Základem jsou rozhodovací stromy, díky kterým je hlavní využití tohoto algoritmu především v oblasti dolování dat (data mining). V současné době se využívá hlavně k výukovým účelům, protože pro nasazení v praxi nyní slouží jeho mladší bratříčci, kteří zachovávají základní princip a ten zdokonalují. Konkrétně se jedná o algoritmy C4.5[2] a C5.0[3]. Všechny tyto algoritmy jsou z dílny jediného člověka, a to Rossa Quinlana[4].

3.1.1 Rozhodovací stromy

Jak jsem již zmínil, základem ID3 algoritmu jsou rozhodovací stromy. Ty jsou tvořeny, jak samotný název napovídá, datovou strukturou strom. Tato struktura, anebo její modifikace, je velice často používaná především pro její rychlost při vyhledávání položek. Ta je vykoupena ne zcela rychlým sestavením stromu a pomalou editací. Ovšem v tomto případě editaci nepotřebujeme, samotné vytvoření stromu provedeme pouze jednou a následně z něj už jenom čteme. Jednou ze slabin této struktury je právě samotné vytváření stromu. Ne všechny stromy jsou dostatečně efektivní. Rychlost vyhledávání je ovlivněna především tím hlubokou a šířkou stromu. Pokud bychom měli náš strom hodně hluboký, budeme potřebovat hodně kroků k získání výsledku, a tedy ztrácíme efektivitu. Stejné je to v případě příliš širokého stromu s malou hloubkou, která je natolik malá, že musíme procházet sekvenčně velké množství položek, a tím se nám znovu ztrácí efektivita spojená s touto strukturou. Je dobré si tuto skutečnost uvědomit, protože pokud podceníme samotné vytvoření stromu, následně můžeme přijít o hlavní výhodu této struktury.

Nyní se již vraťme k samotným rozhodovacím stromům. Hlavní myšlenka spočívá v rozdělení sledovaných atributů vstupních objektů do jednotlivých uzlů stromu a hodnoty těchto atributů do hran vedoucích z daného uzlu. Počet hran musí být konečný. Proto v případě nekonečné množiny hodnot

[2]Další informace na http://en.wikipedia.org/wiki/C4.5_algorithm

[3]Další informace na Zdroj: http://www.rulequest.com/see5-info.html

[4]Další informace na Zdroj: http://en.wikipedia.org/wiki/Ross_Quinlan

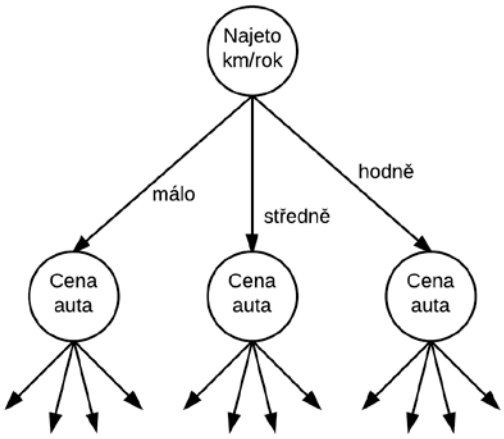
je zapotřebí tuto množinu diskretizovat například jejím rozdělením do intervalů. Příkladem může být množina přirozených čísel. Ta je nekonečná, a pokud bychom ji chtěli využít, musíme vytvořit konečný počet intervalů. Například tak, že všechna čísla menší než 10 budou samostatně a číslice spadající nad tuto hranici budou patřit do jedné velké množiny.

Nejlepší bude si tento způsob tvorby stromu ukázat na příkladu. V tabulce 1 je několik řádků z možné trénovací množiny (povšimněte si u prvních dvou atributů rozdělení do intervalů). Po vytvoření výsledného stromu bychom měli být schopni jednoznačně odpovídat na otázku: „Jak velký risk je pojištění auta našemu zákazníkovi?“. Vstupní vektor je zde reprezentován sloupci 2-4, kterým se také říká podmínkové atributy. Výsledek je reprezentován pátým sloupcem, kterému se říká rozhodovací atribut. Tuto úlohu můžeme označit jako klasifikační úlohu. Tedy my klasifikujeme našeho zákazníka do jedné ze skupin rizikovosti pojištění. Klasifikace je právě jedna z úloh, kterým se zabývá dolování dat[5].

	Cena auta v Kč	nehod za loňský rok	najetých km/rok	risk
1	do 100 000	0	málo	malý
2	nad 100 000 do 250 000	1	málo	střední
3	nad 250 000 do 500 000	0	hodně	malý
4	nad 500 000	5 a více	málo	velký
5	nad 500 000	2	hodně	střední
6	nad 100 000 do 250 000	1	hodně	malý
7	do 100 000	4	středně	velký
8	nad 100 000 do 250 000	0	středně	malý

Tabulka 1

Aby byl výsledek opravdu použitelný, tabulka by musela být mnohem větší. Ovšem i tak by byla menší, než skutečný počet všech kombinací, kterých je $3^{(4\cdot6\cdot3)} = 2{,}25 \cdot 10^{34}$.



obr. 1: Náhled na jednu větev možného rozhodovacího stromu

Samotné sestavení stromu by se dalo popsat zhruba následovně. Odeber kterýkoliv podmínkový atribut z trénovací množiny a použij jej jako uzel. Z tohoto uzlu veď tolik větví, kolik

[5]Další informace na http://en.wikipedia.org/wiki/Data_mining#Data_mining

má daný atribut možných hodnot. Pro každou větev vyhodnoť, zda všechny odpovědi spadající do této větve mají stejnou hodnotu pro rozhodovací atribut. Pokud ano, uzel na konci této větve označ jako losový a nastav mu jeho hodnotu na hodnotu tohoto rozhodovacího atributu. V opačném případě pro tento uzel opakuj celý postup.
Pro atribut *najeto kilometrů za rok* by mohl výsledek vypadat jako na obr. 1.

3.1.2 Popis algoritmu

Vraťme se zpět k algoritmu ID3. Z předchozího textu jsme se dozvěděli, co jsou to rozhodovací stromy. Pozorný čtenář si určitě všiml, který se právě u rozhodovacích stromů objevil. Jedná se o již zmíněné neefektivní vytváření stromu. Algoritmus pro sestavení základního rozhodovacího stromu nám totiž říká, že si můžeme vybrat kterýkoliv podmínkový atribut. Tedy nemusíme zkoumat, který atribut by byl nejvhodnější, prostě si nějaký vybereme. Což může mít za následek zbytečně hluboký strom. Algoritmus ID3 se tedy snaží tento problém vyřešit. Základní princip zůstává stejný, jen při samotném výběru podmínkového atributu začneme více přemýšlet. Místo výběru libovolného atributu zvolíme ten, který nám pomůže jednotlivé objekty od sebe co možná nejvíce odlišit, čehož můžeme docílit pomocí maximalizace informačního zisku. Ten získáme prostřednictvím informační entropie [3],tedy míry neurčitosti náhodného procesu. Ta pro n možných hodnot rozhodovacího atributu $S \in \{s_1, s_2, \dots, s_n\}$ a jejich pravděpodobnostmi $P(s_i)$ vypadá následovně:

$$E(S) = -\sum_{i=1}^{n} P(s_i) \cdot \log_2 P(s_i)$$

Pro výpočet informačního zisku dále potřebujeme zjistit, jaké množství informace bude zapotřebí pro dokončení stromu. K tomu nám poslouží následující vztah:

$$E(A) = \sum_{i=1}^{m} \frac{|S_{A_i}|}{|S|} \cdot E(S_{A_i})$$

kde

A je atribut, který má m různých hodnot

S je množina příkladů

S_{A_i} je podmnožina S obsahující příkladyA_i

Nyní se dostáváme k samotnému výpočtu informačního zisku $G(S, A)$:

$$G(S, A) = E(S) - E(A)$$

Tento výpočet se provede v každém cyklu pro všechny atributy a vybere se ten, který má největší informační zisk. Atributy s pravděpodobností výskytu 1 nebo 0 mají intuitivně přírůstek informačního zisku roven nule.

Pokud budeme uvažovat informace z tabulky 1, pak by výpočet pro atribut *najetých kilometrů za rok* vypadal následovně:

$$S \in tabulka\ 1, \qquad A \in najetých\ kilometrů\ za\ rok$$

$$E(S) = -\frac{4}{8}\cdot \log_2 \frac{4}{8} - \frac{2}{8}\cdot \log_2 \frac{2}{8} - \frac{2}{8}\cdot \log_2 \frac{2}{8} = 1.5$$

$$S_{A_1} = \{1,2,4\}, \qquad S_{A_2} = \{3,5,6\}, \qquad S_{A_3} = \{7,8\}$$

$$E(A) = \frac{3}{8}\cdot\left(-\frac{1}{3}\cdot \log_2 \frac{1}{3} - \frac{1}{3}\cdot \log_2 \frac{1}{3} - \frac{1}{3}\cdot \log_2 \frac{1}{3}\right) + \frac{3}{8}\cdot\left(-\frac{2}{3}\cdot \log_2 \frac{2}{3} - \frac{1}{3}\cdot \log_2 \frac{1}{3}\right) +$$

$$+\frac{2}{8}\cdot\left(-\frac{1}{2}\cdot \log_2 \frac{1}{2} - \frac{1}{2}\cdot \log_2 \frac{1}{2}\right) = 1.188$$

$$F(S,A) = 1.5 - 1.188 = \mathbf{0.312}$$

V prvé řadě vypočteme $E(S)$, a to tak, že použijeme pravděpodobnost výskytu jednotlivých hodnot ve sloupci *risk* (v našem případě se v tabulce vyskytují tři hodnoty s těmito četnostmi: 4x *malý*, 2x *střední*, 2x *velký*. Celkový počet hodnot v tabulce je 8). Dále rozdělíme množinu příkladů vybraného atributu do podmnožin podle jejich hodnot (zde řádky 1,2,4 mají hodnotu *málo*, řádky 3,5,6 *hodně* a 7,8 *středně*). V rámci těchto podmnožin zjistíme, kolik mají možností ve sloupci *risk* a to promítneme do výpočtu, kde je to znázorněno různými barvami.

Pro ostatní atributy vycházejí výpočty následovně:

$$A \in \text{cena auta} = 0.844, \qquad F(S,A) = \mathbf{0.656}$$

$$A \in \text{nehod za rok} = 0.25, \qquad F(S,A) = \mathbf{1,25}$$

Pokud porovnáme vypočítané hodnoty, zjistíme, že největší informační zisk má atribut *nehod za rok*.

A na závěr to nejdůležitější. Už máme dostatek znalostí na to, abychom se podívali na samotný algoritmus ID3, který vypadá následovně:

1. Vytvoř kořen stromu
2. Pokud mají všechny příklady množiny S stejné hodnoty rozhodovacího atributu, označ kořen touto hodnotou a vrať jej jako listový uzel. Jinak pokračuj.
3. Pokud již nejsou další podmínkové atributy, označ kořen nejčastěji se vyskytující hodnotou rozhodovacího atributu a vrať ho jako listový uzel. Jinak pokračuj
4. Vyjmi atribut s největším informačním ziskem z množiny atributů a tímto atributem označ kořen.
5. Pro každou hodnotu A_i vybraného atributu:
    a. přidej kořenu jednu větev označenou touto hodnotou
    b. vytvoř podmnožinu S_{A_i} množiny S obsahující pouze prvky mající hodnotu A_i
    c. rekurzivně zavolej algoritmus ID3 s parametry S_{A_i} a množiny atributů (tato množina již neobsahuje atribut vybraný v bodě 4)
    d. připoj vrácený strom/list k této větvi
6. Vrať vytvořený strom (kořen)

3.2 Back Propagation

Za chvíli si ukážeme druhý algoritmus založený na strojovém učení s učitelem. Ovšem ještě před tím si povíme něco o základním kameni tohoto algoritmu, a to neuronech a jejich sítích.

3.2.1 Neuron a neuronové sítě

Na rozdíl od předchozího řešení, kde hlavní roli hrály rozhodovací stromy, tentokrát využijeme jiné stavební kameny, které se v současné době využívají velice často. Těmito stavebními kameny jsou neurony, tedy něco, co je mnohem blíže člověku už jen proto, že se jedná o jednu ze součástí našeho těla - mozku.

Jak jsem zmiňoval již v úvodu tohoto textu, lidé jsou pohodlní a dělají vše pro to, aby si ulehčili práci - a co je lepší řešení, než aby za nás dělal práci stroj, který je kopií člověka. Konkrétně se v tomto případě jedná o mozek, díky kterému jsme schopni zpracovávat informace z okolí a reagovat nejen na rutinní situace, ale i na nově vzniklé problémy.

3.2.1.1 Biologický neuron

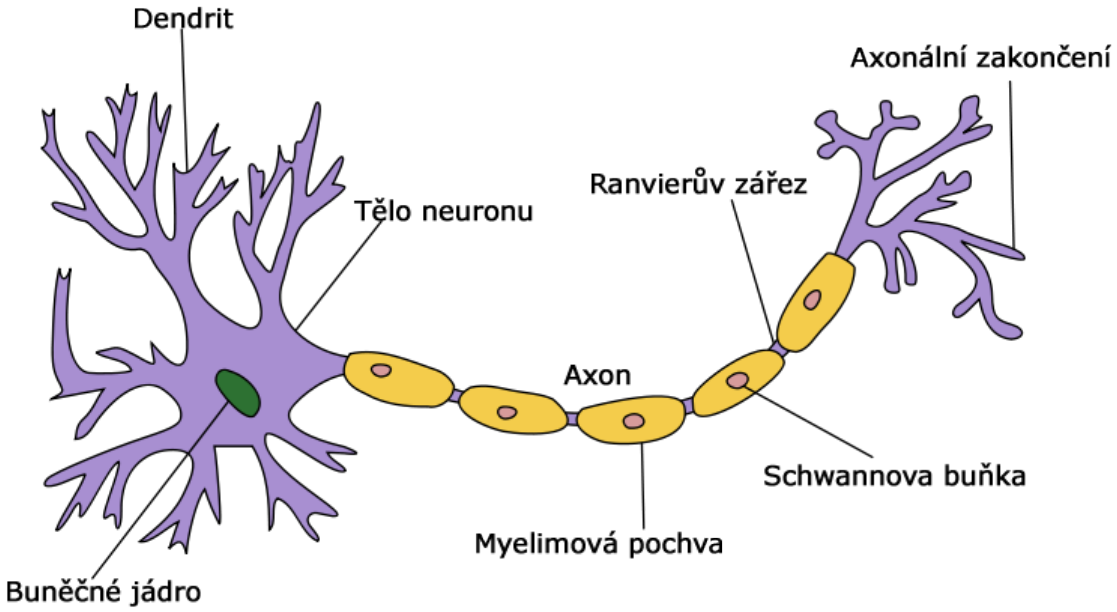
Biologický neuron, který můžeme vidět na obr. 2, má několik pro nás zajímavých částí, a to především dendrity, které slouží jako vstupy, dále tělo neuronu, často označované jako soma, a axon, který slouží jako výstup. Jednotlivé neurony jsou pak spolu propojeny pomocí synapsí, které se nacházejí na axonálních zakončeních. Detail tohoto propojení je vyobrazen na obr. 3, kde si můžeme povšimnout, že mezi jednotlivými spoji je mezera. V této mezeře probíhá komunikace pomocí chemických procesů. To je zapříčiněno tím, že dendrit a tělo neuronu je pokryto nevodivou membránou, na rozdíl od axonu, který je pokryt membránou vodivou.

Samotné chování biologického neuronu můžeme zjednodušeně pospat takto. Na dendrity je přivedena vstupní informace v podobě elektrických impulzů. Ty se sejdou v těle neuronu, kde se spojí v jeden signál, na který neuron může reagovat. Každý neuron má vlastní práh, od kterého je citlivý. V případě, že síla signálu tento práh nepřekročí, neuron nereaguje. V opačném případě sám vyšle signál přes axon do dalších neuronů. Poté, co je signál vyslán na výstup, se neuron stává na určitý čas necitlivý na další vstupní podměty. Pokud je i po uplynutí této doby stále na vstupu hodnota přesahující práh, vyslání impulsu se opakuje.

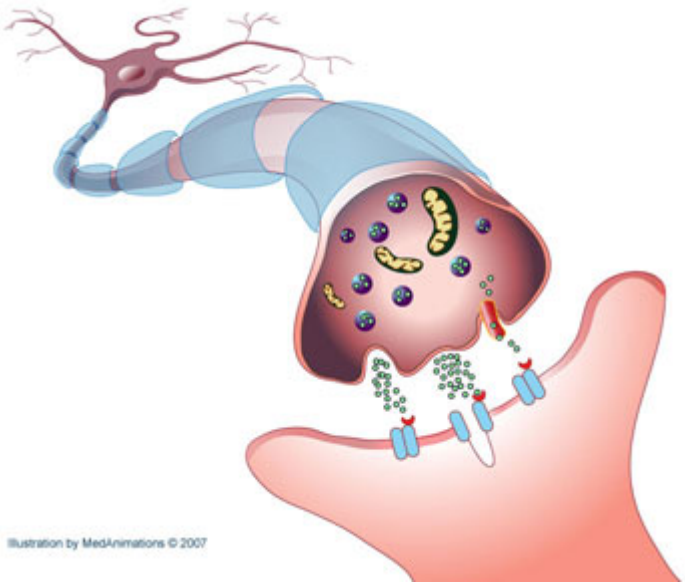
3.2.1.2 Umělý neuron

Umělý neuron vychází ze svého biologického kolegy, ale ve značně zjednodušené podobě. Na druhou stranu, základní funkce zůstává stejná. Tedy neuron přijme ze svých vstupů (stimulů) signály ve formě reálných čísel, které reprezentují buďto podnět z vnějšího okolí, nebo výstupy z jiných neuronů. Ty zpracuje, a pokud je výsledný potenciál dostatečně velký, vyšle signál vlastní.

Vyobrazení umělého neuronu můžeme vidět na obr. 4, kde je jeden neuron s n vstupy a jedním výstupem.



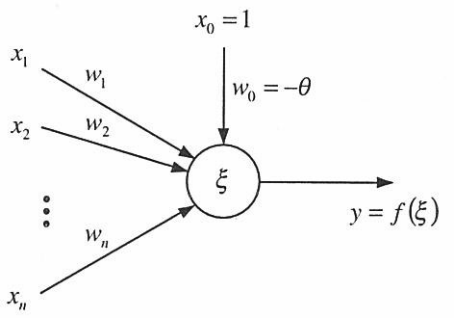
obr. 2: Biologický neuron[6]



obr. 3: Detail propojení neuronů (Synapse). [7]

[6] Zdroj: http://cs.wikipedia.org/wiki/Neuron

[7] Zdroj: http://www.medanimations.com/illustrations



obr. 4: Znázornění umělého neuronu, kde $x_0, \dots, x_n$ jsou vstupy, $w_0, \dots, w_n$ váhy, ξ vnitřní potenciál a y výstup neuronu získaný použitím aktivační funkce f. [4]

Každý vstup má vlastní synaptickou váhu reprezentovanou reálným číslem, díky které je neuron schopen adaptace (učení). Tuto váhu si lze představit jako reprezentaci důležitosti daného stimulu. Speciálním vstupem je x_0, který je vždy roven jedné. Váha k němu připadající zastupuje prahovou hodnotu neuronu. Dále zde máme vnitřní potenciál ξ, který se vypočítá jako vážený součet stimulů. Tedy tak, že se všechny vstupy vynásobí svými vahami a následně se sečtou. Výstup poté získáme aplikací přenosové funkce f na tento potenciál. Zmíněnou přenosovou funkcí může být jakákoliv omezená rostoucí diferencovatelná funkce. Velice často se využívá rovnice sigmoidy v tomto tvaru:

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

kde λ je parametr strmosti, který určuje, jak rychle proběhne přechod mezi nulou a jedničkou. Nejstrmější možná funkce je funkce signum (obr. 5 a obr. 6), která je definována takto:

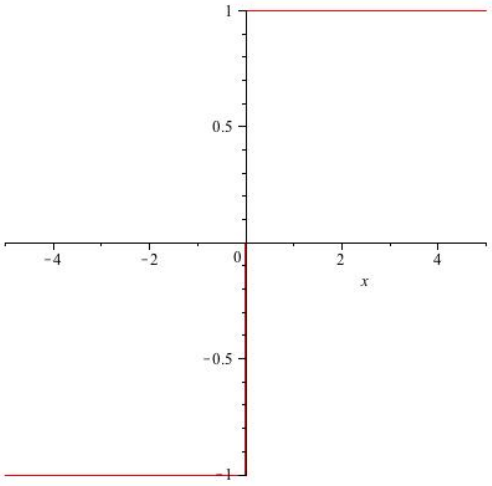
$$sign\, x = \begin{cases} 1 : x > 0 \\ 0 : x = 0 \\ -1 : x < 0 \end{cases}$$

Na rozdíl od této nelineární funkce je vztah sigmoidy diferencovatelný a jeho největší derivace je v bodě nula. Tato funkce je vykreslena na obr. 7 a její derivace pak na obr. 8. Mezi další často používané přenosové funkce se řadí například hyperbolický tangens, který můžeme vidět na obr. 9 a jeho derivaci pak na obr. 10.

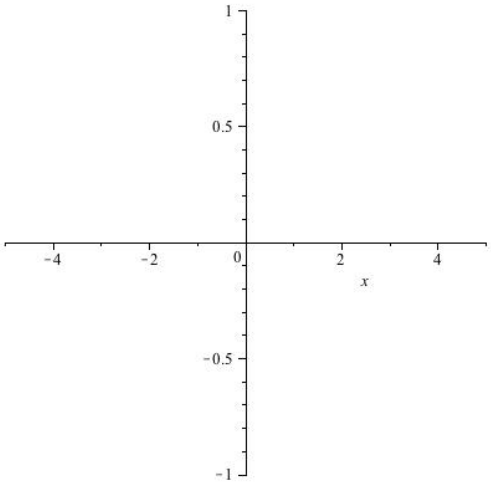
Matematicky tedy můžeme odezvu neuronu zapsat takto:

$$y = f\left(\sum_{i=0}^{n} x_i \cdot w_i\right)$$

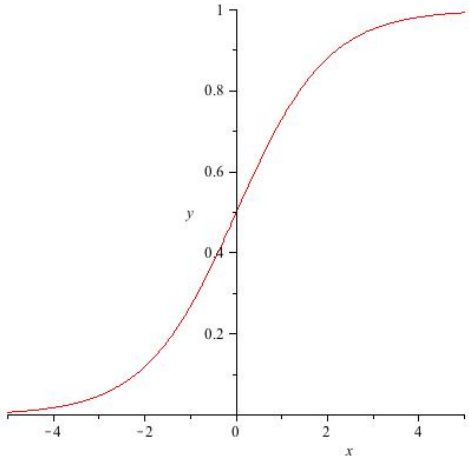
Pokud bychom se podívali na výsledek geometricky, zjistíme, že nám neuron rozdělil prostor na dva podprostory. Když budeme chápat vstupní vektor jako bod v prostoru a neuron jako nadrovinu separující prostor na dva podprostory, výstup reagující na vstupní vektor určí, zda tento bod patří do kladného nebo záporného podprostoru, a to tak, že pokud vnitřní potenciál překročí stanovený práh, jedná se o bod v kladném poloprostoru, v opačném případě v poloprostoru záporném. Tato vlastnost je základem schopnosti neuronů klasifikovat vstup do různých tříd. V průběhu učení se separující nadrovina umisťuje do prostoru pomocí normálového vektoru tvořeného váhami ze vstupů 1 až n a posunutím určeným váhou w_0.



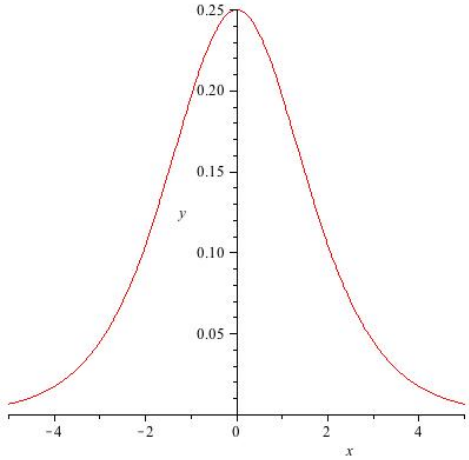
obr. 5: Průběh funkce signum



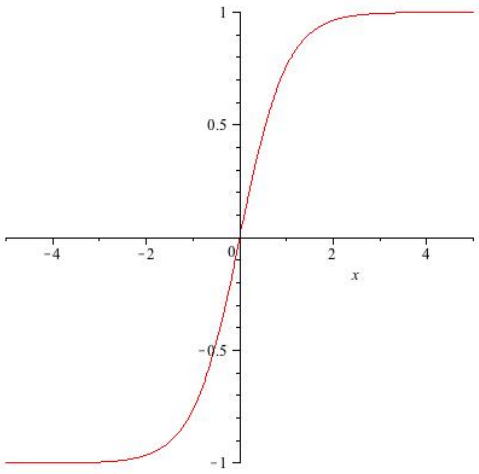
obr. 6: Průběh derivace pro funkci signum neexistuje



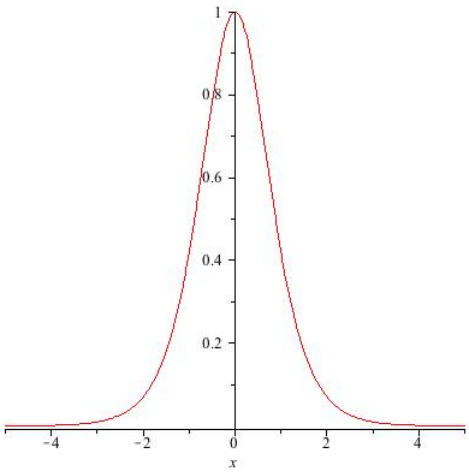
obr. 7: Průběh funkce sigmoid s parametrem λ=1



obr. 8: Průběh derivace funkce sigmoid s parametrem λ=1



obr. 9: Průběh funkce hyperbolického tangentu



obr. 10: Průběh derivace hyperbolického tangentu

3.2.1.3 Jedno a více vrstvé sítě

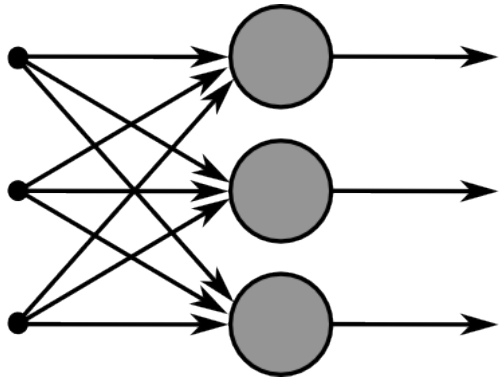
Doteď jsme se bavili o jednom neuronu. Ovšem samotný neuron je dobrá věc, ale sama o sobě nám moc neposlouží. K tomu, abychom mohli začít řešit složitější úlohy, jich budeme potřebovat více. A tím nastává důležitá otázka. Jak takových více neuronů spojit, aby pracovaly společně na nějakém složitějším problému? Odpovědí jsou právě neuronové sítě. Ty sdružují více neuronů do různých seskupení tak, aby je bylo možné použít jako jeden celek. Máme dvě základní možnosti seskupení.

1. jednovrstvá síť – např. Kohenova síť
2. vícevrstvá síť – např. vícevrstvá perceptronová síť

Každá z těchto variant se hodí na jiný typ úloh. Jednovrstvé sítě mají své použití například při optimalizaci a adaptaci, zatímco vícevrstvé sítě se dají využít třeba pro klasifikaci nebo predikci.

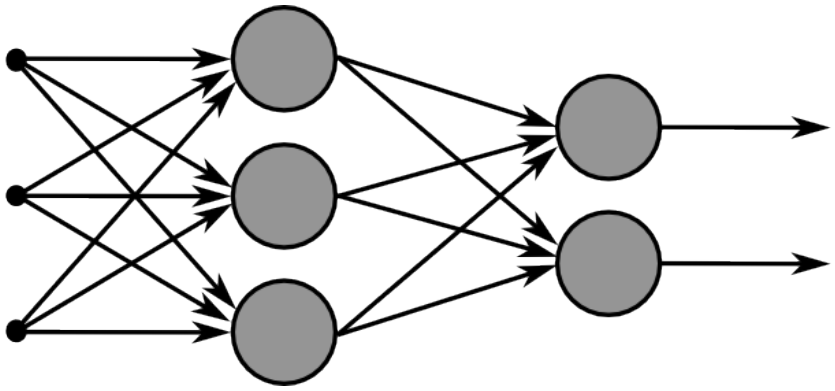
Pro každý typ sítě je zapotřebí nějaký učící algoritmus, díky kterému bude síť schopna natrénovat trénovací množinu a následně produkovat správné výsledky. Každý typ sítě má většinou více či méně specifický algoritmus určený právě pro její potřeby. V každém případě, základem je nějaké propojení jednotlivých neuronů, od kterého se následně vše odvíjí. V případě první vrstvy se bavíme především o vstupním bodě do naší sítě, kde jsou očekávána data ke zpracování. Tento vektor dat je přiveden na každý vstup neuronu ve vrstvě. Tedy pokud máme vektor o pěti rozměrech a v první vrstvě se nachází pět neuronů, celkový počet spojení pro tuto vrstvu bude 25. V dalších vrstvách se většinou vyskytuje typ spojení každý s každým, kde každý neuron předchozí vrstvy má svůj výstup přiveden na vstup ke každému neuronu další vrstvy. Takto to pokračuje pro všechny vrstvy až k té výstupní (v případě jednovrstvé sítě je vstupní vrstva zároveň i výstupní). Z té již dostaneme výsledný vektor. Všechny vrstvy pod touto vrstvou jsou takzvané skryté vrstvy, protože z pohledu výpočtu nás zajímá pouze ta vrstva, ze které získáme výsledek, a nikoliv ty ostatní. Počet neuronů pro jednotlivé vrstvy není nijak omezen. Pouze jich musí být konečný počet.

Oba typy sítě můžeme vidět na obr. 11 respektive obr. 12.



obr. 11: Ukázka jednovrstvé neuronové sítě[8]

[8] Zdroj: http://commons.wikimedia.org/wiki/File:SingleLayerNeuralNetwork.png



obr. 12: Ukázka dvouvrstvé neuronové sítě[9]

3.2.2 Popis algoritmu

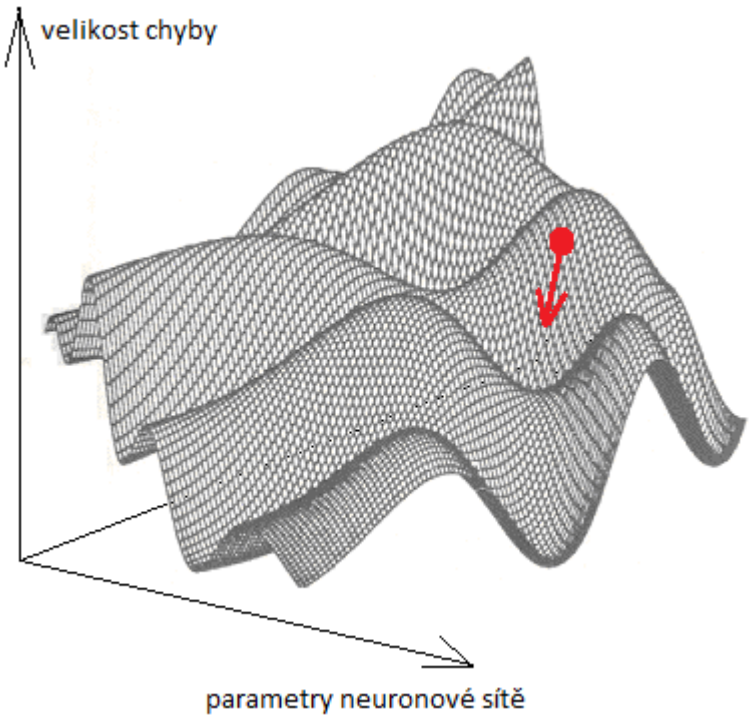
Back Propagation je velice důležitý algoritmus, kterému vděčíme za to, že se v současné době neuronové sítě používají v tak hojném počtu. Vznik neuronových sítí je možné datovat až do roku 1943, kdy W. McCulloch a W.Pitts matematicky popsali jednoduchý model neuronu. Až do roku 1969 se tato oblast zkoumala a rozvíjela, ale v tomto roce byla vydána práce M. Minského a S.Peperta s názvem Perceptrons, kde zpochybnili možnosti neuronových sítí, protože neexistovalo řešení, jak separovat logickou funkci XOR. Díky této práci se vývoj téměř zastavil, znovu pokračoval až v 80. letech. Velká změna nastala v roce 1986, kdy D. Rumelhart, G. Hinton a R. Williams zveřejnili přelomový algoritmus s názvem Back Propagation, který umožňoval učení vícevrstvých neuronových sítí a tím se vyřešil do té doby neřešitelný problém se separací funkce XOR. Tento algoritmus se pro svoji jednoduchost hojně využívá dodnes. [4]

Hlavní myšlenkou algoritmu Back Propagation je informovat nižší vrstvy sítě o výsledku klasifikace vstupních dat. Toho je docíleno tak, že se první přiloží trénovací vektor na vstup sítě a síť se nechá, aby provedla svůj výpočet. Následně se porovná vypočtený výsledek s očekávaným a vypočte se velikost chyby, o kterou se síť odchýlila. Tato chyba se následně zpětně distribuuje do nižších vrstev sítě a neurony si na jejím základě upraví svoje vstupní váhy tak, aby tuto chybu snížily. Tedy po provedení této zpětné propagace a přiložení stejného vzorku na vstup sítě by měla být nově spočtená chyba menší než v předchozím případě.

Před započetím samotného trénování je zapotřebí stanovit koeficient učení. Ten představuje rychlost, jakou se síť bude učit nové vzory. Většinou se tato hodnota volí z intervalu $\langle 0.1, 0.9 \rangle$. Dále musíme nastavit počáteční hodnoty vah pro všechny vstupy u každého neuronu. Většinou se využívá generování náhodných hodnot v intervalu $\langle -0.5, 0.5 \rangle$, nebo $\langle -\frac{3}{\sqrt{n}}, \frac{3}{\sqrt{n}} \rangle$, kde n je počet vstupů daného neuronu. Po inicializaci máme síť v určitém stavu chybovosti vůči trénovací množině. To si lze představit například jako graf zobrazený na obr. 13, kde svislá osa odpovídá velikosti chyby, které se

[9] Zdroj: http://commons.wikimedia.org/wiki/File:MultiLayerNeuralNetwork.png

dopustíme pro konkrétní nastavení parametrů sítě vyznačených na ostatních osách. Cílem algoritmu je najít v tomto prostoru globální minimum.



obr. 13: Prostor znázorňující velikost chyby sítě v závislosti na jejích parametrech s vyznačeným aktuálním stavem sítě a vypočteným gradientem z tohoto bodu [5]

Jeden z možných stavů sítě máme zvýrazněn na obr. 13 červeným bodem. Z něj hledáme směr, kterým nejrychleji klesá chyba sítě. Tomuto směrovému vektoru se říká gradient (na obrázku znázorněn červenou šipkou). Tento vektor následujeme až do doby, než nalezneme dané minimum. V některých případech se může stát, že lokální minimum je v tak velké „prohlubni“, že se z něj již algoritmus nedostane. V takovém případě je zapotřebí proces učení ukončit a spustit znovu.
V popisu algoritmu budeme požívat tyto značky:

- l - aktuálně zpracovávaná vrstva
- j- aktuálně zpracovávaný neuron vrstvy
- i - aktuálně zpracovávaný vstup neuronu
- m - počet neuronů výstupní vrstvy
- n_{l+1} - počet neuronů v následující (vyšší) vrstvě
- x_i - hodnota vstupu i
- d_j - očekávaný výstup neuronu j
- y_j - skutečný výstup neuronu j
- λ - parametr strmosti
- μ - koeficient učení – obvykle hodnota z intervalu $\langle 0.1, 0.9 \rangle$

Kompletní algoritmus Back Propagation:

1. Inicializace sítě a nastavení parametrů λ a μ.
2. Vynuluj velikost globální chyby.
3. Opakuj pro celou trénovací množinu.
    a. Načti vektor z trénovací množiny.
    b. Opakuj pro všechny vrstvy sítě.
        i. Opakuj pro všechny neurony vrstvy.
            1. Spočti vnitřní potenciál neuronu pomocí vztahu $\sum_{i=0}^{n} x_i \cdot w_i$.
            2. Vypočti výstup neuronu pomocí přenosové funkce (například pro sigmoidu vztahem: $\frac{1}{1+e^{-\lambda\xi}}$).
    c. Spočti velikost chyby, které se síť dopustila pomocí vztahu $\sum_{j=1}^{m}\left(d_j - y_j\right)^2$.
    d. Vypočtenou chybu vynásob 0.5 a přičti ke globální chybě.
    e. Vypočti δ_j pro výstupní vrstvu pomocí vztahu $\left(d_j - y_j\right) \cdot \lambda \cdot y_j \cdot \left(1 - y_j\right)$.
    f. Vypočti δ_j pro všechny neurony všech skrytých vrstev pomocí vztahu$\left(\sum_{k=1}^{n_{l+1}} {}^{l+1}\delta_k \cdot {}^{l+1}w_{kj}\right) \cdot \lambda \cdot {}^{l}y_j \cdot \left(1 - {}^{l}y_j\right)$, kde ${}^{l+1}\delta_k$je δ k-tého neuronu v následující vrstvě, ${}^{l+1}w_{kj}$ je váha vstupu spojujícího aktuální neuron s neuronem následující vrstvy.
    g. Vypočti nové váhy u všech neuronů pomocí vztahu ${}^{l}w_{ji} = {}^{l}w_{ji} + \mu \cdot {}^{l}\delta_j \cdot {}^{l}x_i$.
4. Pokud je globální chyba větší než maximální povolená odchylka, přejdi na bod 2.

3.3 K-Means clustering

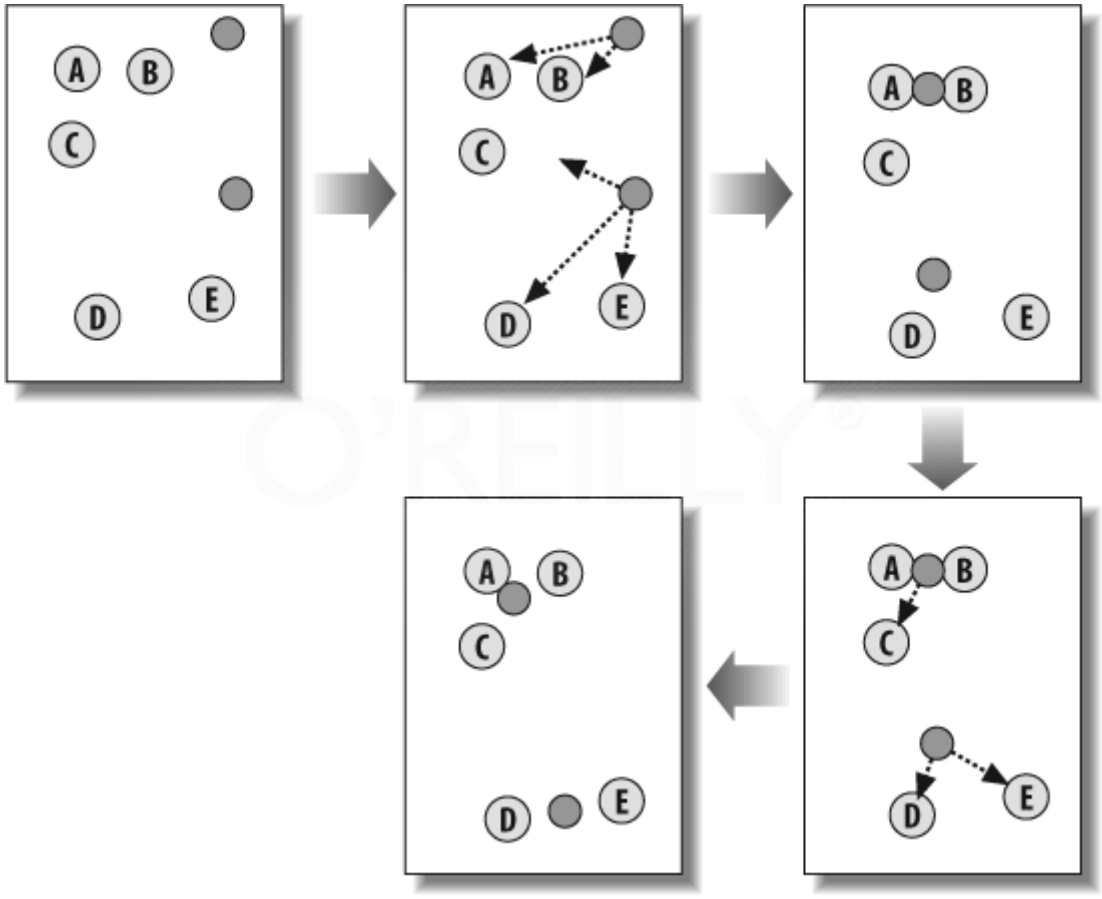
První algoritmus patřící do kategorie strojového učení bez učitele, kterým se zde budeme zabývat, je K-Means clustering. Ten klasifikuje vstupní body v n-dimenzionálním prostoru do k shluků. Jedná se o jeden ze základních algoritmů tohoto typu a jeho princip, jak bude uvedeno níže, je velice intuitivní. Silnou stránkou tohoto algoritmu je především vícedimenzionální shlukování, ve kterém je velice rychlý.

V kapitole 2.2 jsem již uvedl, že některé algoritmy učení bez učitele vyžadují informaci o počtu shluků, do kterých má být vstup klasifikován. K-Means mezi ně patří. Před započetím učení mu tedy tuto informaci musíme sdělit. Intuitivně počet shluků, do kterých se má klasifikovat, musí být menší nebo roven počtu prvků v trénovací množině. V opačném případu to nedává velký smysl.

K samotnému výpočtu se využívají těžiště/středy shluků. Ty se na začátku inicializují na hodnoty vybraných prvků trénovací množiny a následně dochází k jejich přepočtu. Počáteční výběr těžiště je velice důležitým faktorem zásadně ovlivňujícím výsledné rozložení shluků. Pokud totiž

vybereme prvních x prvků z trénovací množiny, a ty budou ležet blízko sebe, nedojde k tak dobrému rozpoznání středů, jako v případě náhodného výběru, kde je mnohem větší pravděpodobnost, že vybrané prvky budou ležet dále od sebe.

Kompletní princip algoritmu probíhá velice intuitivně, a to tak, že se všem příkladům trénovací množiny přidělí některý ze shluků, a to ten, který je nejblíže. Tím se změní velikost jednotlivých skupin, a proto dojde k přepočtu jejich těžišť. Následně se opět projdou všechny prvky trénovací množiny a umístí se k nejbližšímu středu. Tento postup pokračuje tak dlouho, dokud dochází k přesouvání bodů mezi shluky, anebo se zmenšuje velikost chyby. Ukázku tohoto postupu máme vykreslenou na obr. 14. Když se nad tímto principem trochu zamyslíme, spatříme jeden možný problém, který tento způsob ukrývá. Jedná se o vliv zašumění dat na výsledné shluky. Pokud budeme mít vysoce zašuměná data, výsledné shluky mohou být naprosto odlišné od případu, kdy data zašuměná nebudou. Je to způsobeno právě tím, že každý bod se přiřazuje do nějaké skupiny a tím se mění velikost a těžiště tohoto shluku.



obr. 14: Znázornění fungování algoritmu K-Means clustering v několika krocích[10]

Algoritmus K-Means clustering v bodech:

1. Inicializuj k shluků/klastrů (náhodně vybraných k různých bodů z trénovací množiny).
2. Pro všechny body trénovací množiny:

[10] Zdroj: http://answers.oreilly.com/uploads/monthly_02_2010/post-9-126592805763_thumb.png

a. Nastav hodnotu nejmenší vzdálenosti mezi aktuálním bodem a těžištěm na maximální hodnotu.
b. Pro všechny shluky:
    i. Vypočti vzdálenost mezi těžištěm tohoto shluku ta aktuálním bodem z trénovací množiny$x$, kde n je počet dimenzí pomocí vztahu $\sum_{i=1}^{n}(x_i - t_i)^2$.
    ii. Pokud je vypočtená hodnota menší než dosud nejmenší vzdálenost, nově vypočtenou hodnotu nastav jako tu nejmenší.
c. Přiřaď zpracovávaný bod do shluku, který je nejblíže.

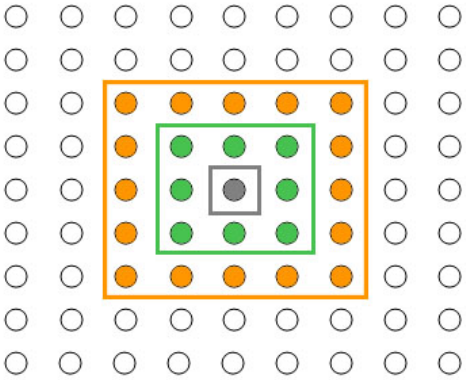
3. Pro všechny klastry:
    a. Přepočítej těžiště podle vztahu $t = \frac{\sum_{x_o \in C} x_o}{m}$, kde t je těžiště aktuálního shluku, C je množina bodů shluku, m mohutnost množiny$C$ a x_o je prvek množiny C.
4. Vypočti aktuální chybu shlukování pomocí vztahu $\sum_{j=1}^{k}\sum_{x_o \in C_j}\left(x_o - t_j\right)^2$, kde k je počet shluků, C je množina bodů jednoho shluku, x_o je prvek množiny C a t je těžiště daného shluku.
5. Pokud došlo k přesunu nějakého bodu, nebo se snížila velikost chyby, jdi na bod 2.

3.4 Self-Organizing Maps (SOM)

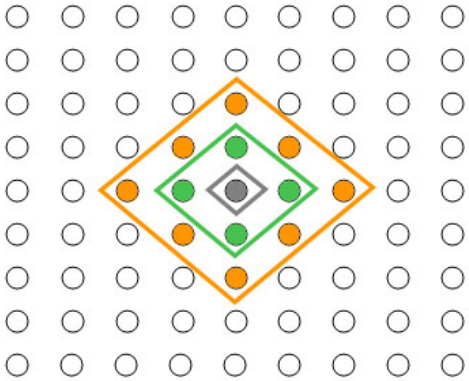
Tento algoritmus je další, který využívá ke své činnosti neurony, kterými jsme se již zabývali v kapitole 3.2.1. Na rozdíl od minulého použití, se tentokrát jedná o jednovrstvou síť. Neurony v ní jsou v tomto případě umístěny nezávisle. Tedy nejsou navzájem nijak spojeny. Ovšem to neznamená, že by mezi nimi nebyla nějaká struktura. Naopak, většinou jsou rozmístěny ve 2D prostoru do tvaru čtverce nebo obdélníku. Dále se můžeme setkat s 1D variantou, kde jsou neurony umístěny na přímce, anebo s hexagonální mřížkou. Důvod, proč jsou neurony takto uspořádány, spočívá v tom, že SOM, také označovaný jako Kohenova mapa, využívá svoje okolí, a proto je zapotřebí vědět, jaké mají rozvržení. Ukázky možného rozvržení sítě a různá okolí výsledného neuronu můžete vidět na obr. 15a obr. 16, kde máme neurony umístěny do čtverce sokolím, které je taktéž čtvercové respektive kosočtvercové. Dále pak na obr. 17jsou neurony rozmístěny hexagonálně s okolím do šestiúhelníku.

Kohenova mapa patří mezi soutěživé neuronové sítě. Ty spočívají v tom, že jednotlivé neurony mezi sebou soupeří o vítězství. To trvá tak dlouho, než zůstane jediný neuron, kterému se říká vítěz, a představuje shluk, do kterého byl vstupní vektor přiřazen. V průběhu učení, tzv. adaptaci, probíhá v síti samoorganizační proces, což znamená, že se jednotlivé neurony v rámci prostoru pohybují. Jinak tomu není ani zde, kde se vítěz posouvá směrem k danému vstupnímu bodu a zároveň s ním se posouvají i neurony z jeho okolí. To může mít různé tvary a velikosti v závislosti na tom, jak chceme, aby se daná síť učila (viz. obr. 15, obr. 16 a obr. 17). Okolí je totiž velice významný faktor, který

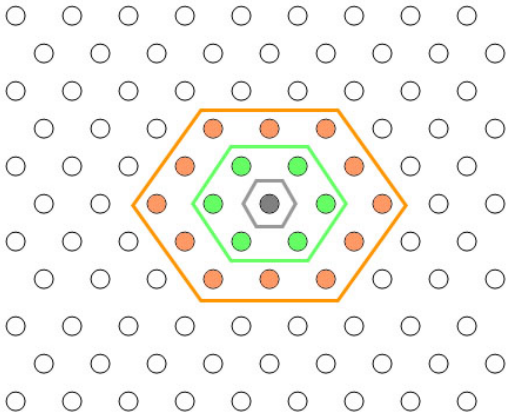
ovlivňuje chování celé sítě a její finální stav. Tedy pokud budeme za okolí vítěze považovat neurony umístěné například v kruhu, dostaneme jinou výslednou síť než v případě použití čtvercového okolí. Důležitou roli zde také hraje postupné zmenšování aktivované oblasti. Bez použití tohoto faktoru se naše síť nebude učit dostatečně kvalitně, protože nebude docházet k jemnému doladění samotného vítěze nezávisle na ostatních. Většinou se volí varianta, kdy dochází k postupnému zmenšování okolí v závislosti na počtu kroků. Dobře to je patrné na obr. 18, kde máme čtvercový prostor s různými velikostmi okolí, které se v průběhu času zmenšuje až na velikost nula. Tím je docíleno toho, že síť po určité době posouvá pouze vítězem a ostatní neurony zůstávají na svém místě.



obr. 15: Čtvercová síť se čtvercovým okolím.
Velikosti okolí: oranžová = 2, zelená = 1 a šedá = 0



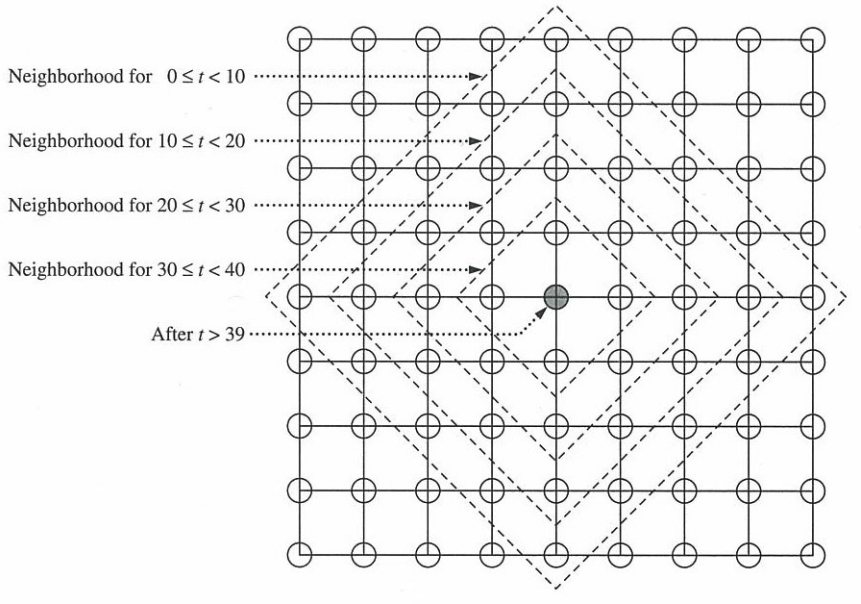
obr. 16: Čtvercová síť s kosočtvercovým okolím.
Velikosti okolí: oranžová = 2, zelená = 1 a šedá = 0



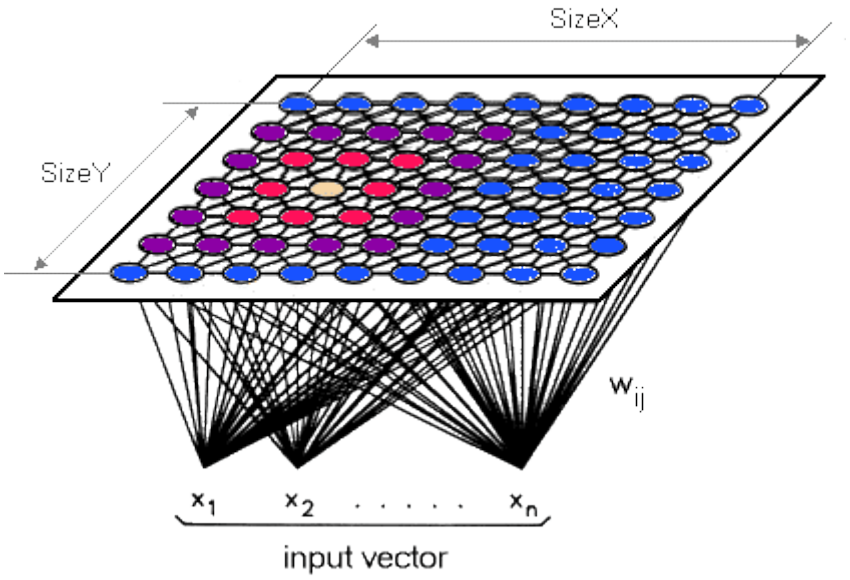
obr. 17: Hexagonální síť s šestiúhelníkovým okolím.
Velikosti okolí: oranžová = 2, zelená = 1 a šedá = 0

Jelikož tento algoritmus spadá do kategorie učení bez učitele, trénovací množina obsahuje pouze vstupní vektor bez žádné další informace. Na rozdíl od předchozího algoritmu dokonce i bez informace o počtu shluků, a to proto, že v SOM vyhrává pouze jeden neuron označující výsledný shluk, jak jsem již zmínil. Tedy maximální počet shluků, které můžeme rozpoznat, záleží na počtu neuronů v síti. Ovšem to neznamená, že nejsme schopni rozpoznat menší počet shluků, anebo se jim úplně vyhnout. Velmi často se totiž tento algoritmus využívá k rozpoznávání písma a tam nás shluky

přímo nezajímají. Kompletně znázorněnou Kohenovu mapu, včetně vstupního vektoru a čtvercového okolí, můžeme vidět na obr. 19.



obr. 18: Změna velikosti okolí v čase.[11]



obr. 19: Kohenova mapa se vstupy $x_1, \ldots, x_n$ a čtvercovým okolím o velikosti dva.[12]

Princip algoritmu, jak je uvedeno výše, je založen na soutěžení. To spočívá v tom, že se hledá neuron, s nastavením vah, které je co možná nejblíže k hodnotám přiloženým na vstup. Poté co dokončíme učení, nastává čas pro klasifikaci. Tu můžeme rozdělit do dvou režimů na:

- adaptační
- neadaptační

[11] Zdroj: Mehrotra, K., Mohan, C., K., Ranka, S.: Elements of Artificial Neural Networks, The MIT Press, 1997, ISBN 0-262-13328-8

[12] Zdroj: http://www.lohninger.com/helpcsuite/kohonen_network_-_background_information.htm

V případě adaptační varianty jde o možnost neustálého dolaďování vah vítěze, což může být výhodné, protože se síť neustále doučuje a je tedy schopná reagovat i na dlouhodobé změny vstupních hodnot. To ovšem neplatí pro variantu druhou. Zde dochází pouze k vyhodnocení vítěze a další změny se už neprovádějí.

Nyní se již podíváme na popis celého algoritmu. V něm jsou využity následující značky:

- n – počet vstupů neuronu
- i - aktuálně zpracovávaný vstup neuronu
- x_i - hodnota vstupu i
- w_i – váha vstupu i
- μ - koeficient učení – obvykle hodnota z intervalu $\langle 0.1, 0.9 \rangle$

Samotný algoritmus pak vypadá takto:

1. Inicializuj síť na náhodné hodnoty vah.
2. Vyber příklad z trénovací množiny.
3. Nastav hodnotu nejmenší vzdálenosti mezi vstupem a neuronem na maximální hodnotu
4. Pro všechny neurony sítě:
    a. Vypočítej vzdálenost mezi vstupním vektorem a vahami neuronu pomocí vztahu $\sum_{i=1}^{n}(x_i - w_i)^2$.
    b. Pokud je vypočtená hodnota menší než dosud nejmenší vzdálenost, tak nově vypočtenou vzdálenost nastav jako tu nejmenší.
5. Uprav váhy vítězného neuronu a jeho okolí podle vztahu $w_i = w_i + \mu \cdot (x_i - w_i)$.
6. Pokud počet iterací nepřekročil stanovenou hranici, pokračuj na bod 2.

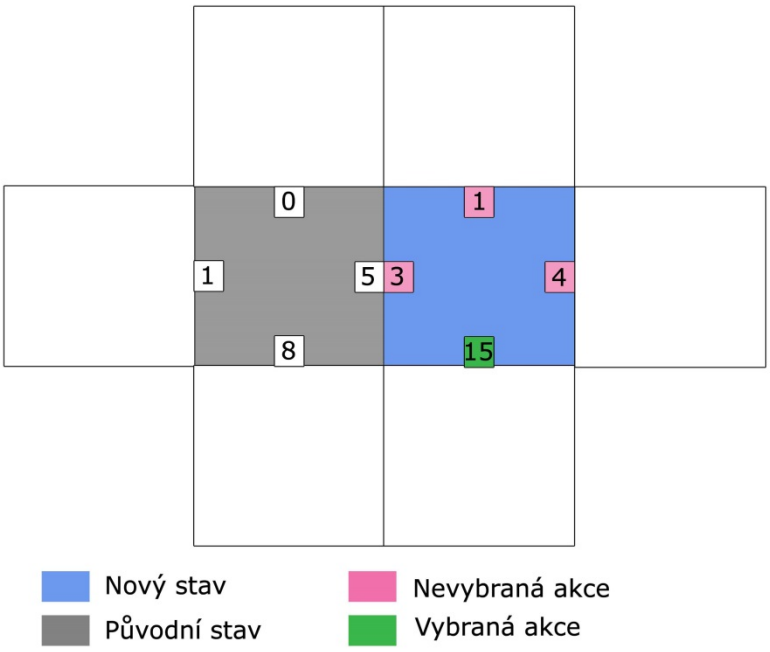
3.5 Q-Learning

Nyní se dostáváme k poslední skupině učících algoritmů, a to k posilovanému učení. Typickým zástupcem, patřícím do této skupiny, je právě Q-Learning, který patří mezi velice často zmiňovaný algoritmus v této souvislosti. Celkově se řadí mezi tři nejčastěji zmiňované algoritmy při výuce základů strojového učení. Těmi dalšími jsou již výše popsané ID3 a K-Means clustering.

Základní chování tohoto algoritmu spočívá v procházení stavového prostoru. Pohyb po celém prostoru probíhá zcela náhodně a není do něj nijak zasahováno. Během této činnosti Q-Learning hodnotí jednotlivé provedené akce v daném stavu a toto hodnocení si zaznamenává. Vzhledem k tomu, že procházení probíhá náhodně, je zapotřebí počítat s určitým časem, než dojde k projití všech stavů. Jednotlivým průchodům začínajícím na startovní pozici a končícím v cíli říkáme procházky. Tedy při startu algoritmu agent začne na startovacím stavu a prochází celým prostorem tak dlouho, dokud nenarazí na cíl. Poté se vrátí na začátek a celý proces se opakuje. To trvá tak dlouho, dokud

agent nenasbírá dostatek informací k tomu, aby byl schopen dorazit přímou cestou až do cíle. Pozice startu nemusí být pevně dána, může se po každé procházce změnit. To pomůže agentovi prozkoumat celý stavový prostor v rychlejším čase.

Při hodnocení jednotlivých akcí se algoritmus snaží maximalizovat zisk z daného stavu tak, že po vybrání náhodného přechodu se vyhledá nejlépe ohodnocená akce ve stavu, který je umístěn v cíli tohoto přechodu. Toto ohodnocení se následně využije pro výpočet hodnoty vybraného přechodu. Díky tomuto postupu Q-Learning najde vždy optimální cestu od startu k cíli. Přechod z jednoho stavu do druhého můžeme vidět na obr. 20, kde si agent náhodně vybral akci, která provádí přechod o jedno pole vpravo. V tomto novém stavu našel nejlépe ohodnocený přechod s hodnotou 15. Tu následně využije pro výpočet nové hodnoty akce s aktuálním ohodnocením 5.



obr. 20: Přechod Q-Learningu do nového stavu. Bílé akce nebyly hodnoceny (algoritmus náhodně vybral).

Výpočet nového ohodnocení přechodu v aktuálním stavu využívá, kromě informace o maximální hodnotě akce v následujícím stavu, také odměnu, kterou agent za vstup do tohoto stavu získá. Ta může být buď kladná, nebo záporná. Pro správné fungování celého algoritmu je třeba, aby taková kladná hodnota existovala minimálně jedna, a to v cílovém stavu. V opačném případě by nebylo možné provádět hodnocení přechodů správně. Kladné hodnoty jsou brány jako odměny za správné chování, zatímco záporné mohou být použity pro penalizaci za vstup do nesprávných míst. Další hodnoty, které se k výpočtu využívají, jsou dvě konstanty. První z nich se jmenuje discount factor, který budeme označovat znakem γ. Volně lze tento název přeložit jako srážkový faktor. Ten zajišťuje srážku/úbytek z nejlepší akce v novém stavu. Díky této hodnotě můžeme regulovat vliv ohodnocení následujícího stavu na stav aktuální. Velikost tohoto faktoru se většinou pohybuje v rozsahu $0 \leq \gamma < 1$. Druhou hodnotu budeme označovat znakem α a její název je koeficient učení. Ten nám dává možnost nastavit, jak rychle se náš agent bude učit. Většinou se využívá v rozsahu

$0 < \alpha \leq 1$. Navíc se velice často v průběhu výpočtu snižuje s počtem vybrání konkrétní akce v daném stavu. Tedy čím vícekrát agent využije konkrétní akci v aktuálním stavu, tím méně nové informace získá. Celý vzorec pro výpočet nové hodnoty akce vypadá takto:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot \left(r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right)$$

kde

$Q(s_t, a_t)$ značí ohodnocení akce a_t provedené ve stavu s_t,

α koeficient učení,

r_{t+1} odměnu získanou v novém stavu,

γ srážkový faktor a

$\max_a Q(s_{t+1}, a)$ maximální hodnotu akce v novém stavu.

Samotný algoritmus vypadá následovně:

1. Inicializuj všechny stavy.
2. Vyber startovací pozici a nastav ji jako aktuální stav.
3. Pokud aktuální stav není cíl, opakuj.
    a. Vyber náhodnou akci pro přechod do nového stavu.
    b. V novém stavu najdi nejlépe ohodnocenou akci.
    c. Vypočti nové ohodnocení náhodně vybrané akce.
    d. Nastav nový stav jako stav aktuální.
4. Pokud jsi nepřekročil počet procházek, pokračuj bodem 2.

3.6 State-Action-Reward-State-Action (SARSA)

Dostáváme se k poslednímu algoritmu, který si zde ukážeme. Ve všech předchozích případech jsme si ukazovali zcela odlišné algoritmy, které fungovaly každý jiným způsobem. Tentokrát tomu bude jinak. Algoritmus, který si ukážeme teď, je modifikací již popisovaného Q-Learningu. Takže nyní máme možnost porovnat dva algoritmy fungující na stejném základu lišícím se pouze v některých detailech.

Základní rozdíl mezi těmito dvěma algoritmy spočívá v tom, že Q-Learning, který byl popsán v předchozí kapitole, patří mezi off-policy algoritmy, zatímco SARSA je on-policy, což se projevuje ve způsobu, jakým daný algoritmus přistupuje k procházení stavového prostoru. V kapitole 3.5 bylo uvedeno, že Q-Learning prochází mezi jednotlivými stavy vždy pomocí náhodného výběru přechodu a tento způsob nemění. SARSA k tomuto problému přistupuje poněkud odlišným způsobem. Místo využívání neustále té samé strategie se snaží svoje chování v průběhu učení měnit. Například v případě použití ε-greedy policy algoritmus vybírá přechody s největší váhou a pouze v určitém

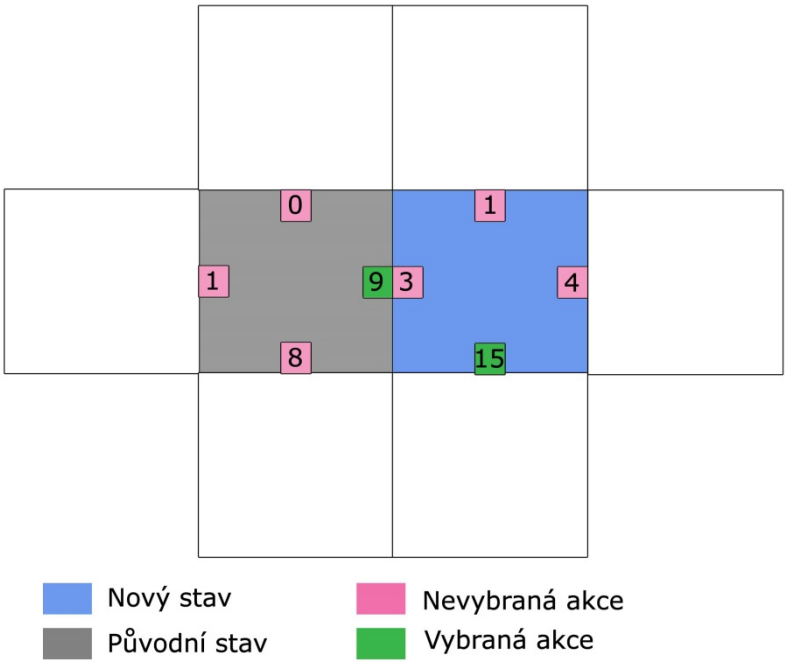
procentu případů zvolí náhodnou akci. Tato náhodnost je pro správné fungování velice důležitá, protože bez ní by se algoritmus nesnažil hledat lepší cesty a využíval by neustále tu, kterou již našel.

Pokud se podíváme na vzorec, který se využívá pro aktualizaci hodnoty akce:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot \left(r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right)$$

zjistíme, že rozdíl se skrývá pouze ve výběru hodnoty přechodu, kde místo hledání maxima bereme hodnotu odpovídající akci, která se skutečně provede. Což v případě ε-greedy policy bude znamenat právě tu největší (pokud se zrovna nevybere náhodně).

Podíváme-li se na obr. 21, uvidíme znázornění aktuálního stavu a nového stavu, kde algoritmus provedl přechod na základě nejvyššího ohodnocení akce (například právě pomocí ε-greedy policy), nikoliv jako v předchozím případě pouze pomocí náhody. V novém stavu následně znovu vybral akci s největším ohodnocením. Tyto dvě hodnoty následně využije pro svůj výpočet. Druhá varianta, znázorněná již na obr. 20, zobrazuje případ, kdy se využil náhodný výběr akce nezávisle na tom, jaké hodnoty přechodů v aktuálním stavu jsou, a až následně v novém stavu vybral maximum. To se shoduje právě s Q-Learningem.



obr. 21: Přechod SASRA do nového stavu. Při výběrech akcí byla aktivní ε-greedy politika.

Každá modifikace se snaží svůj vzor v něčem zdokonalit. V tomto případě jedna z velkých výhod algoritmu SARSA oproti Q-Learningu spočívá ve schopnosti reagovat na další agenty. Ti můžou mít zcela jiný způsob chování a jejich cíl může být odlišný. Představme si například jednoduchou situaci s kočkou, myší a sýrem. Kočka má za cíl sníst myš, zatímco ta se chce dostat k sýru, který je někde v prostoru, ale nechce být potravou kočky. Máme tedy dva agenty, kde jedním

je kočka, která má jediný cíl, a to sníst myš, a přitom na ni nečíhá žádné nebezpečí. To je téměř ideální stav, takže kočka se nepotřebuje na nic ohlížet a jednoduše jí stačí najít nejkratší cestu k myši. Tento úkol může bez problémů obstarat například Q-Learning. Nyní se dostáváme k druhému agentovi, a to myši. Ta má situaci o poznání těžší. Jí nestačí jít nejkratší cestou k sýru, protože by ji po cestě mohla potkat kočka a učinit z ní svoji večeři. Místo toho musí reagovat na aktuální pozici kočky a podle toho se v dostatečném odstupu od ní dál pohybovat směrem k sýru. Tato úloha je jak stvořená právě pro náš algoritmus. SARSA, jak jsem si již uvedl, je schopná v průběhu učení modifikovat svůj přístup k procházení prostoru a tím lépe reagovat na nepříznivé situace v podobě, například, dalšího agenta.

Na závěr ještě samotný algoritmus:

1. Inicializuj stavový prostor.
2. Vyber startovací pozici a nastav jí jako aktuální stav.
3. Vyber akci z tohoto stavu podle nějaké politiky pro přechod do nového stavu.
4. Pokud aktuální stav není cíl, opakuj.
    a. Vyber akci z nového stavu podle nějaké politiky.
    b. Vypočti nové ohodnocení akce, která má být provedena z aktuálního stavu.
    c. Proveď přechod z aktuálního stavu do nového stavu.
    d. Nastav akci z nového stavu jako akci z aktuálního stavu.
5. Pokud jsi nepřekročil počet procházek, pokračuj bodem 2.

4 Analýza

Do této chvíle jsem se zabýval především tím, jak jednotlivé algoritmy fungují, jaké mají výhody a nevýhody. Teď se posuneme o trochu dál. Už víme, s čím pracujeme, a proto je na čase si rozvrhnout samotnou práci.

Dle zadání je zapotřebí udělat dvě věci:

1. Implementovat vybrané algoritmy.
2. Vytvořit ukázkové úlohy pro každý z nich.

Bavíme se tedy o dvou rozdílných věcech. První bod se zabývá tvorbou šesti zcela odlišných algoritmů, které by měly být schopny zpracovávat různé úlohy v plném rozsahu funkčnosti zvolené metody, zatímco druhý bod vyžaduje implementaci jednoduchých základních ukázek toho, co daný algoritmus dělá.

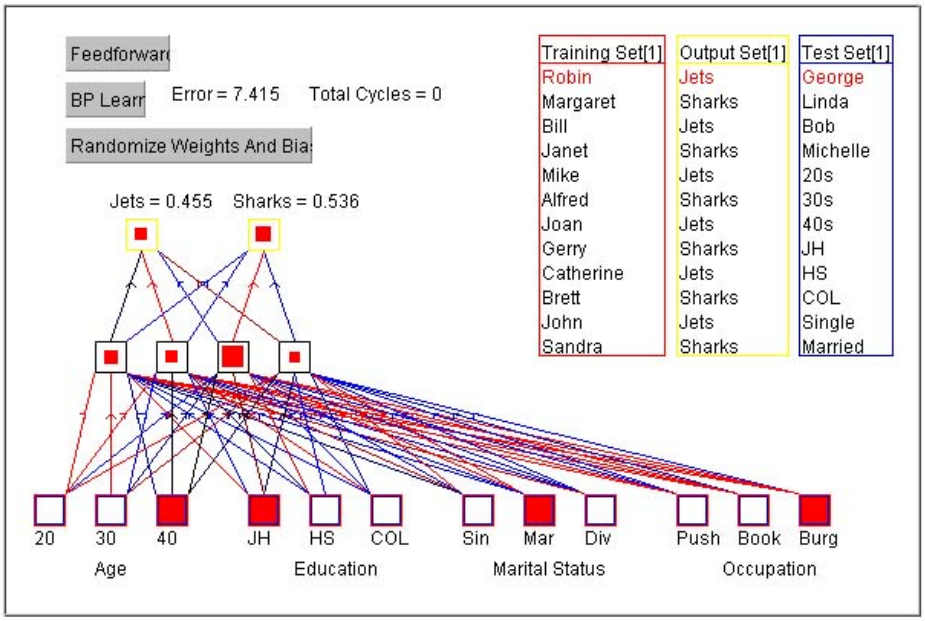
Existuje více cest, jak tyto požadavky splnit. Nastíním zde dvě základní.

1. Implementovat jednu velkou aplikaci obsahující všechny možnosti všech vybraných algoritmů.
2. Vytvořit jednotlivé malé aplikace, které budou řešit pouze svůj algoritmus.

První varianta zní celkem honosně. Vytvoříme jednu „super aplikaci“, která bude schopna dělat naprosto vše, na co by uživatel pomyslel, a třeba ještě něco navíc. To je velice lákavá myšlenka, protože odpadne zdlouhavá a nudná práce s tvorbou několika uživatelských rozhraní. Zároveň uživatel bude mít vše po ruce v jednom balíčku bez nutnosti přemýšlet, co vlastně chce. Existoval by totiž pouze jeden exe soubor, který by stačilo spustit. Ovšem toto řešení má také své nevýhody. Tou asi nejzásadnější je přehlednost. Pokud budeme mít velkou aplikaci, která obsahuje všechny funkce ze všech algoritmů, pro nového uživatele, který bude chtít takovou aplikaci využít, bude velice obtížné se zorientovat v takovém množství možností. Zároveň tím chtě nechtě částečně ořízneme funkčnost jednotlivých algoritmů, protože většina z nich zvládá mnohem více, než jen to, co jsme schopni zobrazovat na obrazovce.

Pokud se podíváme na druhou variantu, na první pohled si můžeme říct, že to není moc dobré řešení. Vytvářet hromadu malých aplikací, mezi kterými se bude muset uživatel prodírat a hledat tu správnou přece jen není nijak uživatelsky přívětivé. Když se nad tím ale zamyslíme, tak ve skutečnosti i v té jedné aplikaci by uživatel musel stejně vědět, kterou metodu chce využít a najít ji mezi těmi ostatními. Pokud tedy ví, co požaduje, je zbytečné rozptylovat ho těmi ostatními a raději mu poskytnout maximální uživatelské pohodlí a možnosti k řešení daného problému. Tedy hlavní výhodou je to, že každá aplikace může být trochu jiná, a tím je možné umožnit uživateli soustředit se pouze na daný problém, aniž by byl rozptylován dalšími možnostmi. Co se týče nevýhod, jedná se hlavně o nutnost hledat správnou aplikaci mezi ostatními a samotný vývoj bude zdlouhavější, protože bude potřeba dělat hodně věcí vícekrát.

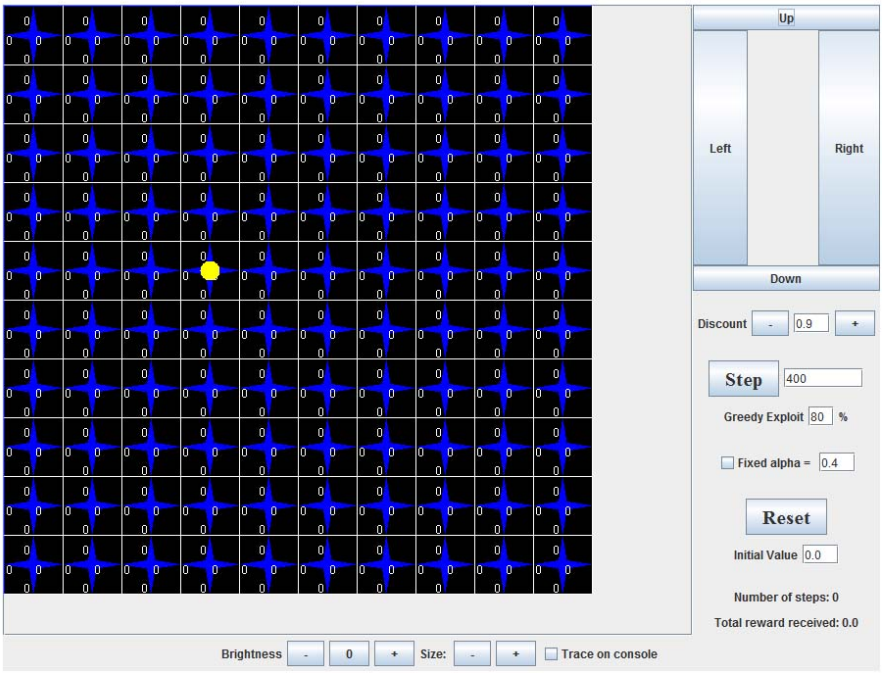
Podívejme se, jak řeší toto dilema ostatní. Většina aplikací, které jsou na internetu k dispozici, se věnuje vždy pouze jedné konkrétní problematice, kde se snaží dát uživateli maximální možnosti v nastavení všech parametrů. Uveďme dva konkrétní případy. Na obr. 22 a obr. 23 můžeme vidět dvě aplikace, které řeší některý z algoritmů. V obou případech se jedná o aplikaci zaměřenou pouze na jednu konkrétní úlohu s nastavením všech dostupných parametrů. Výsledkem je pro lajka zcela nesrozumitelná a nepoužitelná aplikace, neboť nemá tušení, které číslo co znamená a jaké hodnoty je třeba zadat, aby aplikace začala fungovat. Přitom se bavíme pořád o aplikaci, která zpracovává pouze jeden jediný problém. Jak by tedy potom vypadala aplikace, která má na starosti více problémů? Pravděpodobně ještě mnohem hůře. Takže větší potenciál na vytvoření srozumitelné aplikace se skrývá spíše v druhé variantě, kde budou k dispozici ovládací prvky pouze k té konkrétní úloze.



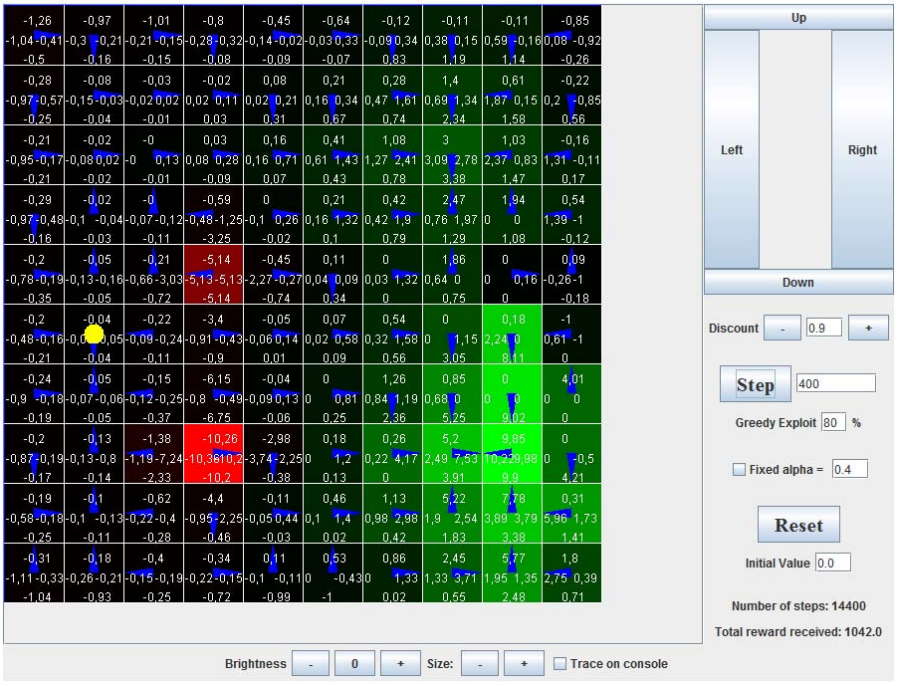
obr. 22: Applet ze stránky http://itee.uq.edu.au/~cogs2010/cmc/chapters/BackProp/ na algoritmus Back Propagation.

Zvolili jsme tedy směřování celého zadání. Teď je ještě zapotřebí trochu se zamyslet nad tím, jak zpřístupnit kompletní funkcionalitu implementovaných algoritmů a tím dát možnost případnému zájemci využít veškerý potenciál, který není možné zpřístupnit v dané úloze, například práce s více dimenzemi. Oba popisované algoritmy patřící do strojového učení bez učitele jsou totiž dělané pro více dimenzí a teprve až tam se skutečně projevuje jejich síla. Tam už ale chybí názornost, která je pro ukázku základní funkce zapotřebí. Tento požadavek může vyřešit například samostatná knihovna s implementacemi všech algoritmů. Pak již nikdo nebude svazován samotnou aplikací a navíc se tím otevírá možnost opakovaného využití. Protože jedna věc je aplikace demonstrující základní funkce a druhá věc je možnost využít algoritmus v jiném projektu bez nutnosti studia celé problematiky a následné implementace něčeho, co již bylo vytvořeno mnohokrát. Z tohoto důvodu jsem se rozhodl kompletně oddělit aplikaci s konkrétní úlohou a samotný algoritmus. Tento krok bude mít kromě již zmíněné výhody ještě jedno plus, a to možnost vyměnit nebo upravit algoritmus nezávisle na celé

aplikaci, a tím využít již existující úlohu pro předvedení například nějaké modifikace původního algoritmu. Poté bude mnohem snazší obě varianty porovnávat a vyhodnocovat rozdíly, protože samotná úloha zůstane nezměněna.



obr. 23: Applet ze stránky http://www.cs.ubc.ca/~poole/demos/rl/q.html na algoritmus Q-Learning.



obr. 24: Applet ze stránky http://www.cs.ubc.ca/~poole/demos/rl/q.html na algoritmus Q-Learning. Během učení.

5 Návrh

Shrňme si informace z předchozí kapitoly. Výsledkem bude více specializovaných aplikací a každá z nich bude řešit vlastní úlohu připravenou pro konkrétní algoritmus. Samotná aplikace pak bude od implementace konkrétního algoritmu oddělena pro účely opakovaného použití a její rozhraní by mělo být pro uživatele co možná nejjednodušší.

A proto si návrh celého řešení rozdělíme do těchto částí:

1. Algoritmy
2. Úlohy
    a. Samotná úloha.
    b. Uživatelské rozhraní aplikace.

5.1 Algoritmy

Každý algoritmus je specifickou záležitostí, která vyžaduje trochu něco jiného. Proto si teď rozebereme každý z nich vzhledem k jeho návrhu.

Ovšem ještě před tím se podíváme na společné rysy. V kapitole 2, jsem zmiňoval, že strojové učení s učitelem a bez učitele vyžaduje trénovací množinu a v případě posilovaného učení je třeba stavový prostor. Všechny tři varianty se shodují v tom, že se jedná o nějaký soubor dat, který je zapotřebí načíst a zpracovat. Aby všechny algoritmy správně fungovaly, je zapotřebí poměrně velké množství dat. Tedy nejedná se jen o pár informací, které by mohl uživatel vždy znovu zadat, ale o něco, co jednou vytvoří buď on, anebo nějaký skript, a pak už půjde jen o opakované použití. Nejvhodnější způsob, jak tento požadavek zajistit, je použití externího souboru, který se vždy při startu načte.

Jak by měl takový soubor vlastně vypadat? Trénovací množiny jsou definovány jako množiny vektorů. Dalo by se tedy jednotlivé prvky množiny uložit na samostatný řádek. Vzhledem k tomu, že jsou všechny prvky vektory, tak s uložením do řádku nebude problém. Jediné, co je v tomto ohledu potřeba ještě učinit, je nalezení oddělovače, který bude oddělovat jednotlivé složky vektoru. Jako vhodný symbol se jeví středník a to ze dvou důvodů. V textu se vyskytuje velmi zřídka a používá se jako oddělovač v CSV formátu. Ten je pro svoji jednoduchost velice rozšířený a existuje spousta aplikací, které jej umí vytvářet. Za všechny uveďme například Microsoft Office a jeho Excel, který umí uložit vytvořenou tabulku do tohoto formátu. Tím máme elegantně vyřešený způsob, jak může uživatel komfortně vytvářet velké trénovací množiny.

5.1.1 ID3

Jak jsem uvedl v kapitole 3.1, tvorba rozhodovacího stromu probíhá tak, že každá větev je rekurzivní zanoření. Budeme tedy potřebovat jeden objekt tvořící kořen stromu. Ten se bude následně vytvářet pro každou větev celého stromu.

Rozhraní tohoto objektu bude velice jednoduché. Veškerá práce totiž probíhá uvnitř a nás tedy bude zajímat pouze předání hodnot konstruktoru a pak až výsledek pro náš dotaz. Parametry konstruktoru jsem koncipoval tak, aby docházelo k co možná nejmenšímu množství přenášených dat. Vznikla tak potřeba na tři parametry a to:

- trénovací množinu s daty,
- seznam atributů, které se mají využít a
- čísla řádků z trénovací množiny, kterých se tento výpočet bude týkat.

Pokud potřebujeme získat výsledek klasifikace pro konkrétní data, poslouží nám metoda s názvem `getResult` a jejím jediným parametrem budou data, která mají být klasifikována a výstupem je třída klasifikace.

Další částí je samotné načítání trénovací množiny. V případě ID3 trénovací množina nemusí obsahovat pouze čísla, ale například i text. Navíc nás zde zajímají i dodatečné informace v podobě počtu sloupců a řádků celé množiny. Proto jsem se rozhodl vytvořit objekt s těmito daty. Jeho rozhraní bude obsahovat především metody pro přidávání řádků, zpřístupnění samotných dat a zpřístupnění jednotlivých informací o množině.

Máme tedy dva objekty. Jeden sloužící jako samotný strom a ve druhém ukládáme trénovací množinu. Už nám tedy chybí pouze něco, co zařídí odstartování celého procesu. To může být například funkce `learn`, která dostane cestu k trénovacím datům a vrátí hotový strom.

5.1.2 Back Propagation

Back Propagation je založen na práci s vícevrstvými neuronovými sítěmi.

Jak už víme, neuronové sítě mají nějakou topologii, která obsahuje jednotlivé vrstvy sítě. Každá vrstva může mít jiný počet neuronů, a to musíme při našem návrhu zohlednit. Navíc do sítě ještě vede vektor s nějakým počtem vstupních hodnot. Na tento vektor jsem se rozhodl pohlížet jako na speciální nultou vrstvu především proto, že algoritmus s těmito hodnotami pracuje úplně stejně jako s výstupy z neuronů a tento krok zajistí sjednocení i na úrovni kódu.

Tento algoritmus se týká neuronů. Otázkou ale zůstává, jak na takový neuron pohlížet v kódu. Prakticky máme dvě možnosti.

1. Vlastní objekt pro neuron.
2. Uchování zajímavých hodnot pouze v jedné struktuře.

První varianta spočívá ve vytvoření vlastního objektu, který bude zpřístupňovat chování celého neuronu, což je sice z hlediska abstrakce zajímavý způsob, ale je v podstatě zbytečný. Tento postup

by se dal přirovnat ke známému přísloví „s kanónem na vrabce“. My totiž nepotřebujeme uchovávat nějaká velká komplexní data, která by potřebovala složité operace. Jde nám pouze o jednoduché číselné hodnoty. V takovém případě je režie spojená s obsluhou objektu mnohem větší, než užitek získaný z abstrakce. Proto zde máme variantu druhou, která odstraňuje právě zmíněný problém se zbytečnou režií. Uvědomme si, jaké hodnoty potřebujeme z daného neuronu znát. Jediný údaj, který nás opravdu zajímá, je velikost jednotlivých vah na vstupech. Ostatní je totiž pro všechny neurony společné, a proto není třeba uvádět je u každého z nich. Stačí nám uložit pouze seznam vah daného neuronu a to už není žádný větší problém. Výsledná struktura uchovávající celou topologii by tedy měla vypadat zhruba následovně:

- První úroveň: seznam vrstev
- Druhá úroveň: seznam neuronů v dané vrstvě
- Třetí úroveň: seznam vah vstupů daného neuronu

Pro další výpočet jsou potřebné pouze jednotlivé číselné hodnoty, jako je například maximální velikost povolené chyby.

Celý algoritmus by měl ve výsledku být jediným objektem, který si v konstruktoru vyžádá cestu k trénovací množině, seznam vrstev s údajem o počtu neuronů v každé z nich, velikost koeficientu učení a maximální velikost povolené chyby. Velikost parametru strmosti jsem se rozhodl stanovit pevně na hodnotu jedna - a to proto, že jeho vliv na výslednou hodnotu není nijak výrazný. Pouze ovlivňuje rychlost konvergence celého algoritmu. Co se týče zbytku rozhraní tohoto objektu, bude nás zajímat především metoda na odstartování učení, kterou jsem se rozhodl nazvat `learn`, a následně pak metoda `getResult`, která bude mít na starosti vrátit výsledek klasifikace na zadaný vstup.

5.1.3 K-Means clustering

A dostáváme se k prvnímu algoritmu ze skupiny učení bez učitele. V tomto případě nás čeká velice nenáročný algoritmus, kterému stačí pouze trénovací množina a počet shluků, do kterých má provést klasifikaci. Žádná další složitost v podobě ukládání, či načítání dat zde není, protože jde pouze o seznam bodů, které se přeskupují mezi skupinami.

Jedná se tedy znovu o jeden objekt, kterému do konstruktoru předáme trénovací množinu a počet klasifikačních shluků. Následně nám stejně jako v předchozích případech poslouží metoda `learn`, která započne samotné učení. Změna nastane pouze ve způsobu získání výsledku. Jak jsem již uvedl, tentokrát pracujeme s body a shluky, takže výsledné rozložení bodů do shluků nám zařídí dvě metody, kde jedna, `getPoints`, nám vrátí seznam bodů, se kterými algoritmus pracuje, a druhá, `getLabels`, vrátí seznam synchronizovaný se seznamem bodů a v každé položce bude číslo skupiny, do které byl bod na této pozici přiřazen. Získáme tedy dvojici bod-shluk.

5.1.4 SOM

U tohoto algoritmu stojí za zmínku především topologie neuronové sítě. Ta je totiž jednovrstvá, ale nemusí být, a většinou také není, pouze jednořádková. V kapitole 3.4, kde jsem se tímto algoritmem zabýval, jsem uvedl, že sítě jsou většinou maticemi vektorů. Tato forma je využívána velice často a my ji pro naše účely využijeme taktéž. Z maticové sítě totiž vytvoříme jednořádkovou (jedna ze stran matice bude mít velikost jedna), ale opačně nikoliv.

Od uživatele budeme potřebovat, kromě samotné trénovací množiny, také počet neuronů pro horizontální a vertikální rozměr matice a velikost okolí, které má být ovlivněno při samoorganizaci. V souvislosti s touto hodnotou jsem uvažoval, jestli dát uživateli k dispozici možnost ovlivnit po jaké době má dojít ke snížení velikosti, ale po provedení několika testů jsem usoudil, že tato funkce může zůstat pevně součástí algoritmu, protože při změně této hodnoty se mění pouze počet cyklů, které jsou potřeba provést k úspěšnému naučení trénovací množiny. Poslední hodnota, která je zapotřebí k započetí učení, je hodnota koeficientu učení. Ten by měl být, stejně jako všechny ostatní výše zmíněné hodnoty, součástí konstruktoru objektu, který bude tento algoritmus vykonávat.

Dále bude zapotřebí, stejně jako u všech předchozích algoritmů, metoda `learn`, která odstartuje učení. Dále by zde měla být metoda `getResult`, která provede přiřazení zadaného bodu k nejbližšímu neuronu, který vrátí.

5.1.5 Q-Learning a SARSA

V této kapitole budu popisovat oba algoritmy současně, protože jsou si velmi podobné. Hlavní prvek, který oba algoritmy používají, je stav. Většina operací je navázána právě na něj. Bohužel, tento prvek je zcela závislý na konkrétní úloze, která se zpracovává. Takže je zapotřebí jej odstínit. K tomu využijeme jeden objekt, který se bude muset vytvořit vždy pro konkrétní úlohu, kterou bude algoritmus řešit. Základní rozhraní zůstane stejné. To bude obsahovat především dva konstruktory, kde jeden bude vytvářet stav, který není agentovi přístupný, a druhý pak normální přístupný stav. Ten by měl získat informaci o tom, zda je cílovým stavem, jaké je jeho ohodnocení, jaké akce je z něho možné učinit a jejich rating. Objekt by měl také obsahovat metody, které každou z těchto vlastností testují/zpřístupňují, a metody, které vracejí některé z možných akcí v tomto stavu, a to na základě daných strategií.

Samotný algoritmus je také částečně závislý, protože na základě úlohy se teprve rozpozná jak definovat možné akce a podobně. Ovšem rozhraní se nemění. Opět jde o jeden objekt reprezentující celý algoritmus. Jeho konstruktor by měl získat soubor s trénovací množinou a koeficient učení. Metoda, která zůstává stejná i pro tento objekt, je metoda `learn`. Ta provede naučení daného stavového prostoru. Chybí nám ještě způsob dosažení výsledku, tedy cesty od startu k cíli. To by mohla obstarávat například funkce `getWay`, která dostane stav, na kterém má agent začít, a výsledkem by měla být cesta, kterou agent učinil k dosažení cílového stavu. Otázkou ale zůstává, jak

takovou cestu reprezentovat. Buď můžeme vrátit seznam všech stavů, které agent navštívil, nebo seznam akcí, které musel cestou provést. Ani jedna varianta není špatná, záleží čistě na dané úloze, kterou variantu vybereme. Někdy pro nás může být výhodnější mít k dispozici rovnou dané stavy, ale jindy je nepotřebujeme a stačí nám samostatné akce. Osobně preferuji způsob druhý, tedy seznam provedených akcí, a to proto, že pomocí tohoto seznamu a známého počátečního stavu jsem schopen jednotlivé stavy jednoduše dohledat, pokud by to bylo zapotřebí. To je důvod, proč jsem pro tento případ zvolil seznam akcí. Užitečnou metodou může být také nastavení startu na konkrétní stav. To se může hodit v případě, že náš algoritmus nevybírá počáteční stavy náhodně.

5.2 Úlohy

Nyní se podíváme na to, jaké úlohy bude vhodné implementovat pro co možná nejlepší demonstraci konkrétního algoritmu.

5.2.1 ID3

Jak jsem již uvedl, ID3 algoritmus se často využívá k dolování dat. Výsledná úloha by tedy měla být schopna této funkce. Způsobů, jak toho dosáhnout, je hned několik. Já jsem se rozhodoval především mezi těmito dvěma možnostmi:

- Zpracování trénovací množiny a zobrazení výsledného stromu.
- Zobrazení trénovací množiny se samotným zpracováním na pozadí.

V prvním případě se jedná o to, že aplikace vytvoří celý stromu na pozadí a zobrazí výsledný strom, se kterým se následně pracuje. Odnoží této varianty je možnost zobrazení trénovací množiny na popředí a následné vykreslení výsledného stromu. Jako velká výhoda se jevila možnost zobrazení celé struktury, na jejímž základě probíhá rozhodování. Ovšem po detailnějším zkoumání jsem usoudil, že se jedná spíše o nevýhodu, protože výsledné stromy, které nejsou stvořeny pouze z několika řádků trénovací množiny, jsou poněkud rozsáhlé a jejich rozumné zobrazení by vyžadovalo velký monitor, a i v tom případě by se jednalo o značně nepřehlednou strukturu. Uživatel by viděl pouze jakousi strukturu, která by ho spíš zastrašovala, než učila. Možným řešením tohoto problému by bylo vytvoření celého simulátoru, který by rozumným způsobem tuto strukturu zpřístupňoval. Bohužel, tato práce se věnuje ještě dalším pěti algoritmům a tvorba takového simulačního prostředku by svým rozsahem byla spíše na samostatné téma. Proto jsem se rozhodl pro variantu druhou, která zobrazuje trénovací množinu, se kterou se právě pracuje, a samotný strom zůstává skryt. Toto řešení je rozšířením aktuálně využívaného způsobu ukázek různých algoritmů v předmětu Základy umělé inteligence (IZU), kde vyučující nejprve otevře speciální soubor, ve kterém je uložená trénovací množina, a následně spustí konzolovou aplikaci, která tento soubor načte a vrátí výsledek. Já bych rád vytvořil novou verzi, která odstraní první krok, a to otevírání zvláštního

souboru. Místo toho si uživatel zvolí soubor, se kterým bude chtít pracovat, a ten se mu zobrazí v aplikaci do přehledné tabulky, ve které bude moci zvýrazňovat jednotlivé buňky. Díky této možnosti bude mít student neustále na očích trénovací množinu a neztratí tak kontext, ve kterém klasifikace probíhá. Bude tedy možné provést několik klasifikací hned po sobě bez toho, aby se student ztratil, neboť bude mít trénovací data stále na očích.

Úlohu, kterou by aplikace pro tento algoritmus měla provádět, můžeme shrnout zhruba do následujících bodů:

- Zobrazování trénovací množiny.
- Možnost zvýraznění některých buněk trénovací množiny.
- Klasifikace nově zadaných dat a zobrazení tohoto výsledku.
- Možnost provádět více klasifikací po sobě.

5.2.2 Back Propagation

Vytvoření úlohy pro tento algoritmus bude značně podobné předchozímu případu. Oba algoritmy totiž mají nějakou síť, kterou by bylo vhodné zobrazit (v tomto případě síť neuronovou). Ovšem i zde přetrvávají problémy popsané v předchozí kapitole. Chování na venek je navíc u obou algoritmů velice podobné. Oba provádí klasifikaci nad trénovacími daty. Jen v tomto případě jsou všechny zpracovávané hodnoty číselné.

To, že si jsou oba algoritmy na venek podobné, může posloužit jako výhoda pro uživatele. Když by se totiž vytvořila stejná, nebo velice podobná úloha, mohl by lépe porovnat rozdíly ve výstupech a možnostech nastavení. Proto jsem se rozhodl nechat základní parametry úlohy stejné. Bude se tedy znovu jednat o klasifikaci, kde aplikace bude zobrazovat trénovací množinu a dovolí v ní označovat zajímavé hodnoty. Stejně tak by měla zůstat možnost provádění několika klasifikací po sobě.

5.2.3 K-Means clustering

Hlavním zaměřením tohoto algoritmu je shlukování. Navržená úloha by tedy měla provádět shlukování nějakých dat s rozumnou vizualizací výsledku. V praxi se shlukování využívá například pro hledání objektů v obraze, anebo k hledání souvislostí mezi nějakými daty, ve kterých na první pohled ani žádná souvislost být nemusí (například výstup z nějakého zařízení).

Vzhledem k tomu, že účelem navrhované úlohy je demonstrovat tento algoritmus v co možná nejsrozumitelnější podobě, rozhodl jsem se druhou variantu rovnou vyloučit, a to proto, že v datech, vhodných pro účel klasifikace, by si uživatel nebyl schopen jednoduše ověřit, zda tomu tak opravdu je, takže by celá úloha fungovala jako jeden velký black box. To je přesný opak toho, čeho bych rád dosáhl. Proto jsem se začal soustředit na první možnost, a to vyhledávání shluků v obraze. To se

úspěšně využívá například k rozpoznání silnice z jedoucího auta, nebo nalezení a označení osoby v prostoru pomocí termovizní kamery.

Po konzultaci této možnosti s vedoucím práce, jsem se ale rozhodl tuto úlohu zjednodušit a tím poskytnout větší prostor k pochopení celé funkce. Nechal jsem se tedy inspirovat existujícími řešeními, která většinou fungují na bázi plátna, na němž se vygenerují body a ty se následně rozčlení do skupin. Tento způsob jsem následně rozšířil o další věci, jako je například možnost zadání vlastních bodů, anebo provádění krokování.

Celá úloha tedy bude spočívat v rozdělení vykreslených bodů do jednotlivých skupin. Body, které budou k tomuto účelu sloužit, vytvoří uživatel pomocí myši. Následně pak bude mít možnost provést okamžité rozdělení do skupin, anebo si celý proces shlukování postupně krokovat a tím vidět, jak dochází k přerozdělování bodů mezi skupinami.

5.2.4 SOM

Máme zde znovu podobnou situaci jako v případě ID3 a Back Propagation. Tentokrát v podání algoritmů SOM a K-Means. Oba algoritmy totiž provádějí shlukování a bylo by tedy vhodné vytvořit možnost alespoň částečně je porovnat. Rozhodl jsem se tedy přestat hledat speciální úlohu, která by demonstrovala možnosti tohoto algoritmu, a zaměřil jsem se na porovnání obou dvou variant. Přesto si ovšem neodpustím zmínit jednu aplikaci tohoto algoritmu. Jak již bylo uvedeno v kapitole 3.4, tento algoritmus funguje na základě jednovrstvé neuronové sítě, která se přizpůsobuje zadanému prostoru, a ten je následně schopna rozpoznat. Velice zajímavý příklad na toto téma je zaznamenán ve videu uloženém na serveru YouTube.com[13], kde se tento algoritmus adaptuje na fotografii ruky.

Nyní již zpět k definování úlohy pro Self-Organizing maps. Bude se jednat o úlohu, která využije body zadané uživatelem ve 2D prostoru. Do něj se také inicializují jednotlivé neurony. Uživatel bude mít možnost buďto provést celé učení najednou, nebo postupovat po jednotlivých krocích. Výsledné zobrazení by mělo obsahovat zadané body s naučenými neurony.

5.2.5 Q-Learning a SARSA

Poslední úloha, kterou je zapotřebí definovat, se týká algoritmů ze skupiny posilovaného učení. Vzhledem k tomu, že si jsou oba zde popisované algoritmy tak blízké, rozhodl jsem se je spojit a vytvořit pro ně společnou úlohu.

Jako inspiraci jsem využil úlohu, která je prezentována v předmětu Základy umělé inteligence (IZU). Tu můžeme vidět na obr. 25. Jedná se o bludiště, kterým agent prochází. Myslím si, že to je ideální příklad, jak tyto algoritmy prezentovat, protože velice názorně ukazují pohyb agenta ve stavovém prostoru. Navíc se v tomto řešení dá najít i paralela s tím, co jsem popisoval v kapitole 2.3 ohledně přirovnání k učení lidí.

[13] Zmiňované video: http://youtu.be/aQkIg69ZAXs

V této již existující aplikaci jsem z hlediska studenta našel dva hlavní nedostatky, které bych rád ve své verzi eliminoval. Více se jimi budu zabývat v kapitole5.3.5, protože se jedná především o způsob řešení vzhledu a chování aplikace.

Celá úloha pro tyto dva algoritmy bude mít následující parametry:

- Bludiště, ve kterém se agent bude moci volně pohybovat.
- Jednotlivé zdi bludiště bude možné vytvářet/mazat/upravovat.
- Ke každému poli v bludišti bude možné nastavit odměnu (jak kladnou, tak zápornou), kterou agent dostane, pokud na něj vstoupí.
- Bude možné definovat více cílových polí.
- Bude možné zobrazit/skrýt ohodnocení jednotlivých akcí v polích.
- Bude možné zobrazovat/skrývat cestu od startu k cíli.
- Se startovacím bodem bude možné hýbat.
- Úlohu bode možné krokovat.

5.3 Rozhraní

Jako poslední část této kapitoly nás čeká návrh rozhraní jednotlivých aplikací, které bude vycházet z navržených úloh.

Před samotným návrhem je dobré se podívat, jak už byly podobné aplikace vyřešeny a vzít si z nich ponaučení. Ponaučení, která jsem si z těchto aplikací odnesl, spočívají především v tom, čeho se chci při vývoji rozhraní vyvarovat. Existuje totiž velké množství appletů, které řeší všemožné úkoly, ale všechny mají jednu společnou vlastnost. Jsou uživatelsky naprosto nevyhovující. Pokud, jakožto student, se snažím pochopit nějaký problém, jsem vždy rád, když na internetu najdu nějaký applet, který daný problém řeší. Bohužel jsem hned zase zklamán, protože vyžaduje nastavení velkého množství možností, aby bylo možné vůbec applet spustit. Ovšem já ten applet nepotřebuji proto, že vím, co který parametr znamená, ale proto, že právě to nevím! Na toto se velice často v těchto aplikacích zapomíná, a tak jsou velice nepřehledné. Většinou se mi je ani nepovede rozběhnout, a pokud ano, stejně netuším, co to vlastně dělá. Uvedu dva příklady toho, jak by to dle mého názoru rozhodně nemělo vypadat.

První applet se týká algoritmu Back Propagation. Ten můžeme vidět na obr. 22. Hned na první pohled člověk většinou netuší, co tím chtěl vlastně autor říci. Text v tlačítkách není zobrazen celý, hodnoty jsou rozházené po celé obrazovce a bez předchozího nastudování celé funkčnosti je uživatel naprosto ztracen, protože absolutně nemá ponětí kam, kdy a proč kliknout.

Podobně tomu je i v druhé ukázce, kde se jedná o applet na Q-Learning. Ten můžeme vidět na obr. 23. Nejprve uživatel vidí cosi barevného. Vpravo a dole jsou nějaká tlačítka s hodnotami, která zabírají spoustu místa a jsou naprosto bez jakéhokoliv oddělovače poházeny po celém vyhrazeném

prostoru. V každém případě je to oproti variantě první velký posun dopředu. Uživatel se v této aplikaci zorientuje a pochopí základní ovládání. Po provedení několika kroků se celá část vyhrazená pro vizualizaci začne zaplňovat dalšími číselnými a barevnými údaji, jak můžeme vidět na obr. 24. V tuto chvíli utrpěla přehlednost další přímý zásah. Ve změti všech možných informací totálně zanikl hlavní účel celého algoritmu, protože bez důkladného zkoumání není vůbec jasné, co to tam vlastně je zobrazeno.

Oba dva výše zmíněné applety znázorňují přesně to, čemu se v následujícím textu budu snažit vyhnout a upřednostnit jednoduchost a přehlednost ovládání.

Všechny zde vytvořené aplikace by měly mít co možná nejvíce sjednocené ovládání. Proto jsem navrhl základní šablonu, která se bude dodržovat ve všech aplikacích. Ta obsahuje pouze nástrojovou lištu, umístěnou nahoře, a zbývající volný prostor využít pro samotnou vizualizaci. Bude tedy pouze jedno místo pro nástroje s tím, že pokud nebude dostačovat, bude k němu možné přidat ještě horní pruh s nabídkami. Všechna tlačítka by měla být v textové podobě, protože text v tomto případě napoví mnohem více než obrázek. U všech by navíc měl být tooltip, který upřesní chování daného tlačítka.

5.3.1 ID3

Úloha pro tento algoritmus je zcela jednoduchá a tedy i výsledná aplikace by měla být jednoduchá. Shrňme si úkony, které bude nutné v této aplikaci provádět z hlediska rozhraní:

- Otevření trénovací množiny
- Klasifikace zadaných dat

Máme pouze dvě základní funkce, které musíme zajistit. Budou tedy stačit jen dvě tlačítka v nástrojové liště. Ta seřadíme podle posloupnosti úkonů. Takže první bude otevření trénovací množiny a druhé pak klasifikace.

Když uživatel klikne na tlačítko klasifikace, zobrazí se nové okno, do kterého zadá hodnoty pro klasifikaci. Po jeho potvrzení aplikace provede vyhodnocení a vyskočí nové informační okno, ve kterém bude zopakován vstup, který uživatel zadal, a výsledek pro tyto hodnoty.

Trénovací množina bude po celou dobu zobrazena v tabulce s názvy jednotlivých sloupců.

5.3.2 Back Propagation

V tomto případě potřebujeme znát více údajů, ale celkově máme podobný počet akcí jako u ID3 algoritmu. Podívejme se, jaké funkce by mělo rozhraní nabízet:

- Vytvoření nové sítě.
  - Nastavení počtu vrstev a počtu neuronů v daných vrstvách.
  - Nastavení všech dalších potřebných hodnot pro funkci sítě.
  - Specifikace trénovací množiny.

- Klasifikace vstupních hodnot.

Mělo by se tedy jednat pouze o dvě základní tlačítka, která následně budou zpřístupňovat ostatní možnosti.

V prvém případě se jedná o tlačítko pro vytvoření nové sítě. To by mělo zpřístupnit nové okno, ve kterém uživatel zadá potřebné údaje. Těch je ale větší množství, zvláště při rozsáhlejších sítích. Zadávání všech těchto hodnot najednou by mohlo způsobit nepřehlednost, protože uživatel by musel vyplnit velké množství polí. Proto bude rozumnější rozdělit vyžadované pole do dvou oken, která se zobrazí po sobě, takže uživatel bude vždy vyplňovat pouze několik polí, ve kterých se lépe zorientuje. V prvním okně by mělo dojít k zadání základních údajů, jako je počet vrstev sítě, velikost chyby a soubor s trénovací množinou, v následujícím okně počty neuronů pro každou z vrstev. Vzniknou tak dvě jednoduchá okna místo jednoho velkého a složitého.

Důležitou součástí učícího procesu algoritmu Back Propagation je aktuální velikost chyby, které se síť dopouští. Tento údaj je velice důležitý, aby byl uživatel schopen rozpoznat, zda nedošlo k zaseknutí algoritmu v lokálním extrému. Proto by od začátku učení, které se spustí po vytvoření sítě, měla být tato hodnota vidět. To zajistíme pomocí informačního okna, které bude tuto informaci zobrazovat, a v případě potřeby nám dovolí proces učení zastavit.

Po dokončení učícího procesu se uživateli zpřístupní tlačítko pro klasifikaci. To by mělo vyžadovat zadání vstupního vektoru, který má být klasifikován. Po jeho potvrzení by mělo dojít k vyhodnocení a zobrazení výsledku v informačním okně, jehož součástí by měl být i zopakovaný vstupní vektor, aby si uživatel mohl překontrolovat, jaké hodnoty zadal.

Stejně jako v předchozím příkladu budou všechna tlačítka umístěna v nástrojové liště a ve zbytku okna bude zobrazena trénovací množina, se kterou se pracuje.

5.3.3 K-Means clustering

Třetím v pořadí je K-Means clustering. Zde je situace poněkud složitější, protože přibude potřebných tlačítek. Podívejme se na požadované akce:

- Vytvoření nového plátna
- Uložení vytvořeného plátna
- Načtení uloženého plátna
- Vytváření nových bodů
- Provádění algoritmu
    - Kompletní naučení
    - Postupné krokování

První čtyři akce budou mít každá po jednom tlačítku, které danou operaci vykoná. Ta pátá bude vyžadovat tlačítka pro krokování. To je velice užitečná věc pro pochopení celé funkce, a proto zde nesmí chybět, i když přináší drobnou komplikaci. Místo jednoho tlačítka, které by provedlo

inicializaci celého algoritmu i jeho chod, musíme tyto dvě věci od sebe oddělit. Vznikne nám tedy tlačítko pro inicializaci, které bude vyžadovat informaci o počtu shluků, do kterých má proběhnout klasifikace, a dále pak dvě tlačítka, kde jedno provede všechny cykly učení, zatímco druhé vždy jen jeden. Máme tedy šest tlačítek. Ale to ještě není úplně všechno. Neustálé klikání na tlačítko provádějící jeden krok je přece jen trochu nepohodlné, a proto jsem se rozhodl ještě přidat automatické krokování s volbou rychlosti. To bude vyžadovat další dvě tlačítka a jeden posuvník, kde první tlačítko tuto funkci zapne, druhé vypne a posuvník bude regulovat rychlost.

Celkově by tedy rozhraní mělo obsahovat osm tlačítek a jeden posuvník. Ten je možné umístit, stejně jako tlačítka, na nástrojovou lištu nebo do nabídky k tlačítku start krokování. Rozhodl jsem se pro druhou variantu, a to především z důvodu úspory místa na liště a umístění posuvníku blíže k funkci, kterou ovládá.

Máme tedy znovu všechny ovládací prvky na jednou místě v nástrojové liště a zbytek místa může využít samotné plátno na kreslení bodů, které bude probíhat pomocí tlačítka myši.

5.3.4 SOM

Vzhledem k tomu, že úloha pro SOM se shoduje s úlohou pro algoritmus K-Means clustering, bude jejich rozhraní velice podobné. Rozdíly najdeme především v inicializaci, kde SOM vyžaduje více informací v podobě horizontální a vertikální velikosti neuronové sítě a velikost okolí, které bude ovlivňováno. Tyto informace uživatel zadá v novém okně, které se zobrazí při kliknutí na tlačítko inicializace celého algoritmu.

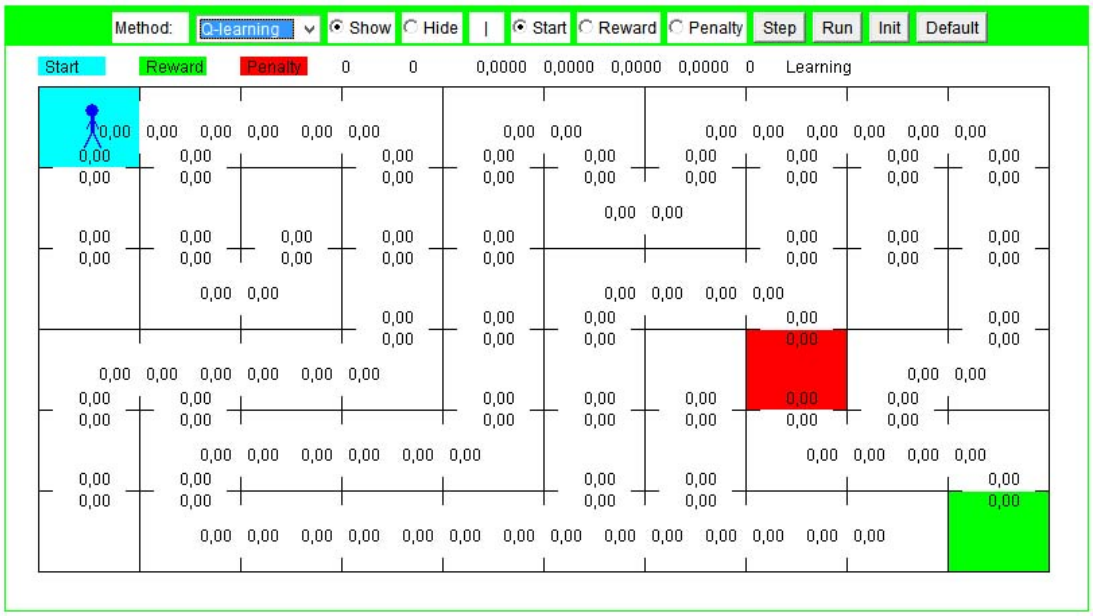
Jediná další změna je v tom, že tento algoritmus skončí, až dokončí stanovený počet cyklů. To znamená, že i v případě plného učení musíme stanovit počet cyklů, které budou vykonány, a v případě, že to nebude stačit, provést operaci znovu. Správný počet cyklů bude zapotřebí následně zjistit experimentálně. Z hlediska rozhraní se jedná o možnost dalšího použití tlačítka s více cykly.

5.3.5 Q-Learning a SARSA

Dostáváme se k největší aplikaci, která v rámci této práce vznikne. Úloha bludiště je totiž nejkomplexnější navrženou úlohou a zároveň tato aplikace bude obsluhovat rovnou dva algoritmy, a to Q-Learning a SARSA.

Nejprve se vrátím ke zmínce z kapitoly 5.2.5, kde jsem uvedl, že aplikace, ze které jsem vycházel, má dva nedostatky. Pro lepší názornost můžeme rozhraní této aplikace vidět na obr. 25. Jako první nedostatek vidím špatnou viditelnost hran bludiště. Ty jsou dělané pomocí slabých čar mezi jednotlivými poli a v případě zbarveného pole není zcela jasně viditelné, kde je zeď a kde ne. Tento problém vyniká zvláště na projektoru, kde se tyto čáry ztrácí. Proto jsem se rozhodl, že v nové verzi bude zeď jasně viditelná. Toho lze docílit buď použitím širších čar, nebo obětováním některých stavů a vytvoření zdí z nich. Obě varianty mají své pro a proti. Rozšíření čar zní jako logická úvaha,

která nebude zbytečně zabírat stavy, které mohou sloužit agentovi. Ovšem druhá varianta umožní velice důležitou věc, a to snadné kreslení vlastního bludiště. Protože zeď lze vytvořit tažením myši přes stavy, zatímco použitím čar by nebylo možné dosáhnout stejného efektu a uživatel by je musel tvořit jednotlivě klikáním myší. Navíc výsledné bludiště stejně nebude tak přehledné jako v případě začerněného celého stavu.



obr. 25: Applet bludiště využívaný v předmětu IZU.[14]

Druhý nedostatek pak vidím v nemožnosti automatického krokování. S touto funkcí jsme se už setkali u aplikací pro K-Means clustering a SOM. V obou případech má uživatel možnost provádět ruční krokování, ale to je uživatelsky nepřívětivé, pokud pro něj zajímavý stav nastane až za velké množství cyklů. Proto jsem k této funkci umístil i možnost automatického krokování s volbou rychlosti a stejně tomu bude i zde.

Kromě již zmíněného krokování musí aplikace zvládat vytváření bludiště. K tomu poslouží tři tlačítka, kde jedno zapne vytváření cílů, druhé kreslení zdí a třetí přidávání odměn. Všechny tyto věci se musí nejen vytvořit, ale také mazat. To lze provádět buď pomocí dalšího tlačítka v liště, nebo jiným tlačítkem myši. Budeme-li používat levé tlačítko myši k vytváření, pravé může sloužit k mazání. Tento přístup se mi osvědčil již v mnoha aplikacích, a proto jej aplikuji i zde. Výhoda tohoto řešení spočívá především v rychlosti ovládání, protože člověk nemusí překlikávat mezi jednotlivými nástroji. Celé kreslení by pak mělo probíhat na plátno s mřížkou označující jednotlivé stavy a speciální typy těchto stavů, jako je například zeď nebo cíl, budou zvýrazněny speciální barvou.

Stejně jako v předchozích aplikacích, které využívají krokování, bude zapotřebí oddělit inicializaci algoritmu od samotného provádění, jak již bylo vysvětleno výše. Zároveň zde přichází další faktor v podobě druhého algoritmu, který by mělo být možné využít bez nutnosti vypnutí

[14]https://www.fit.vutbr.cz/study/courses/IZU/public/rfcl.html

aplikace. Proto kromě inicializace algoritmu bude zapotřebí i destrukce, která přivede aplikaci do stavu před inicializací a tím dá uživateli možnost vybrat druhý algoritmus. Tyto dvě funkce by tedy měly být řešeny dalšími dvěma tlačítky, která jsou společná pro oba algoritmy, a proto by stejně jako vytváření mapy a krokování měla být umístěna v nástrojové liště.

Poslední dvě tlačítka, která jsou společná pro oba algoritmy, jsou zobrazení/skrytí cesty od startu k cíli a zobrazení/skrytí ohodnocení jednotlivých akcí ve stavech.

Zbývají nám ještě funkce týkající se ukládání a otevírání bludiště, vytváření čistého plátna a volby algoritmu. Tyto funkce už nebudou v nástrojové liště, protože by byla přeplněná a ztratila by přehlednost. Proto jsem se rozhodl umístit je do horního pruhu nabídek, kam zároveň přidám všechny akce, které jsou již umístěny v nástrojové liště. Je jich totiž tolik, že pokud budou ještě v nabídce, která bude pouze pro danou skupinu nástrojů, tak se uživateli lépe asociuje, které nástroje k čemu slouží. Vzniknou tedy tři nabídky. Jedna bude obsahovat funkce sloužící k manipulaci s celým prostorem, jako je například načtení bludiště. Druhá nabídka bude obsahovat nástroje pro editaci bludiště. A třetí bude obsahovat nástroje k ovládání algoritmů, včetně jejich výběru.

Samotná dialogová okna pro nastavení jednotlivých hodnot budou řešena stejně jako v předchozích aplikacích, tedy formou co možná nejmenšího počtu vyplňovaných polí.

6 Implementace

Dostáváme se k samotné implementaci. Tu jsem provedl v jazyce C++ za pomoci Qt toolkitu[15] ve verzi 5.0.1. Tato kombinace je velice často využívaná k vývoji různých aplikací. Za všechny uvedu například VLC media player[16], který je uživateli velice oblíbený. Jeho výhody spočívají především v usnadnění práce s uživatelským rozhraním a s tím související multiplatformností, díky které stačí zdrojové kódy pouze přenést na jinou platformu, zkompilovat a vše bude funkční, protože daná platforma pracuje s okny naprosto odlišným způsobem. Zdrojové soubory vytvořené v rámci této práce budou zkompilovány na platformě Windows, která je ve škole standardně k dispozici a počítače v přednáškových místnostech mají nainstalován právě tento systém.

Aby případný zájemce, který by chtěl využít některý z implementovaných algoritmů, hned nepotřeboval celý toolkit, rozhodl jsem se, že samotné algoritmy budou napsány v čistém C++ bez použití tohoto nástroje a teprve až jejich úlohy budou Qt využívat. Tím zajistím plnou nezávislost algoritmů na vytvořené aplikaci.

Nejlepší způsob, jak získat bližší informace o implementaci, je nahlédnout přímo do zdrojového kódu. Proto v této kapitole uvedu pouze zajímavé části z procesu vývoje.

6.1 ID3 a Back Propagation

Jak jste si jistě všimli, základní princip u aplikací pro ID3 algoritmus a Back Propagation je velice podobný, a tedy i zajímavé pasáže jsou podobné. Proto se v této kapitole budu zabývat oběma případy najednou.

Velmi důležitou věcí týkající se obou aplikací, která se ukázala jako nezbytná až v průběhu vývoje, je využití vláken. Tato potřeba vznikla při načítání trénovací množiny a následně pak v průběhu činnosti celého algoritmu a to především proto, že trénovací množina může být celkem rozsáhlá. V takovém případě je zapotřebí udržet celou aplikaci v režimu, ve kterém je schopná dále reagovat na uživatele a "nezamrznout". Proto jsem se rozhodl využít jedno pracovní vlákno, které provede načtení trénovací množiny do tabulky a následně zapne učící proces algoritmu. V případě Back Propagation je tento přístup ještě částečně rozšířen, protože v průběhu učení je zapotřebí dávat uživateli informaci o aktuálním stavu. Takže jsem navíc provedl zapouzdření C++ objektu, který provádí tento algoritmus do QObjektu, který umožňuje využití nadstavbových funkcí Qt toolkitu. V tomto případě jsem využil mutex, který umožňuje udělit výlučný přístup do určité části kódu. Touto speciální částí jsou dvě virtuální metody, ve kterých dochází k manipulaci s proměnnou obsahující informaci o aktuální velikosti globální chyby. Tyto dvě metody jsem předefinoval tak, že

[15] Stránky projektu: http://qt-project.org/

[16] Stránky projektu: http://www.videolan.org/

před zápisem nebo čtením dané hodnoty si vlákno vyžádá výlučný přístup, čímž je docíleno možnosti v jednom vlákně provádět zápis a v druhém čtení bez toho, aby docházelo k nekonzistentním stavům. Zároveň jsem neporušil samotný algoritmus, který je tak nadále napsán v čistém C++ bez nutnosti použití Qt.

6.2 K-Means clustering a SOM

Stejně jako v minulém případě jsem se rozhodl spojit tyto dvě aplikace do jedné kapitoly, protože obě mají také velice podobnou úlohu.

Zde stojí za zmínku především ovládání. Konkrétně vytváření bodů na plátně. Původní myšlenka spočívala v tom, že vytváření bodů se bude provádět levým tlačítkem myši. Ovšem během testování se ukázalo, že uživatelé velice často levým tlačítkem jen tak klikaly na plátno a tím vytvářely body, které nechtěly. Proto jsem se rozhodl přesunout tuto funkci na tlačítko pravé, protože u něj tento problém nenastává. Navíc se nejedná o nějak málo využívané tlačítko, takže se k této funkci člověk velice rychle dostane. Většinou se totiž intuitivně jako první zkouší levé a následně pravé tlačítko. Toto řešení se, ale časem ukázalo jako špatné, protože jsem opomněl vytvořit funkci mazání bodů. Proto jsem nakonec zvolil řešení, které spočívá v dalším tlačítku na nástrojové liště, které zapíná a vypíná možnost manipulace s body. Následně pak levé tlačítko slouží k vytváření a pravé k mazání bodů.

6.3 Q-Learning a SARSA

Dostáváme se k posledním dvěma algoritmům, které mají na rozdíl od ostatních, společnou úlohu i aplikaci. Stejně jako u aplikací pro K-Means a SOM se i zde provádí vykreslení do plátna. Ovšem v tomto případě uživatel pracuje s více nástroji, takže bude pro každý nástroj vlastní tlačítko, které jej bude zpřístupňovat. Uživatel tedy vytváří požadovaný objekt pomocí levého tlačítka myši a pravým tlačítkem jej může zase smazat. V případě, že nechce nic vytvářet ani mazat, pouze zruší výběr daného nástroje. Výjimku tvoří případ, kdy již máme inicializovaný algoritmus. V tu chvíli se nám na levém tlačítku zapne funkce přesouvání startovací pozice agenta. Tato funkce se hodí především po dokončení celého učení, kdy je již celý stavový prostor prozkoumán a my chceme pouze zobrazit cestu k cíli z různých míst.

Pro implementaci algoritmu SARSA jsem se rozhodl využít ε-greedy policy, protože je velice často zmiňovaná, a v případě, že by se uživatel chtěl dozvědět více informací, nebude mít problém s nalezením rozumného zdroje. Zároveň se během implementace ukázalo, že je velice důležité dbát na to, aby po každé procházce došlo k vygenerování nové startovací pozice, a pokud není možné s ohledem na aktuální hodnoty vybrat jednoznačného vítěze (existují například dvě stejně kvalitně hodnocené akce), je nutné vybrat náhodně jednu z nich, a nevolit například vždy první nalezenou.

Obě dvě zjištění jsou důležitá hlavně pro procházení ve stavovém prostoru a hledání nových cest. Pokud by totiž zůstal stále stejný počáteční stav, při použití ε-greedy policy bude algoritmus nacházet nové cesty velice pomalu.

Poslední, o čem se v této kapitole zmíním, je zobrazení cesty od zvoleného startu k cíli. Oba algoritmy jsou schopny zobrazit aktuálně nejlepší cestu od chvíle, kdy existuje posloupnost ohodnocených akcí od startu k cíli. Ovšem do té doby není možné tuto cestu nalézt a počet potřebných kroků k získání této posloupnosti je díky velké nahodilosti procházení různý. Proto vyžádání zobrazení cesty ještě v době, kdy to není možné, vyvolá chybové okno s odůvodněním a návrhem řešení. Nejlepší způsob, jak zjistit, zda je algoritmus již schopný nalézt cestu, spočívá v zobrazení si ohodnocení akcí. Což se provede pomocí vedlejšího tlačítka v nástrojové liště. Ovšem ani to nemusí být zcela dostačující. Existuje totiž možnost, že algoritmus uvedenou cestu vůbec nenajde a zacyklí se. Této situace lze dosáhnout naprosto jednoduchým způsobem. Stačí umístit vedle sebe dva stavy s odměnou, která je zhruba stejně velká jako odměna v cíli. Agent následně bude přecházet pořád mezi těmito dvěma stavy a cíl nebude považovat za dostatečně zajímavý. Proto je zapotřebí vždy nastavit dostatečně velkou odměnu za vstup na cílové pole. Důvod, proč jsem takovou odměnu nenastavil všem cílovým stavům implicitně, spočívá v tom, že jsem chtěl nechat uživateli možnost zvolit si rozsah hodnot, které budou v bludišti použity, a především k tomu, aby bylo možné vytvořit i cíl se záporným ohodnocením, tedy pole, kde agent například „zemře“ nebo nastane jiný nepříznivý stav, který nedovoluje pokračovat. Tuto situaci jsem vyřešil tak, že při vytváření cesty je kontrolován počet polí, která byla navštívena, a pokud tato hodnota překročí celkový počet polí, je jasné, že došlo k zacyklení.

7 Testování

Dostáváme se k finální části. Všechny aplikace už máme vytvořené a nyní je zapotřebí ověřit, jak naše dosavadní snažení obstojí v konfrontaci s realitou.

7.1 Rozhraní

Důležitou věcí při vývoji aplikace je uživatelská přívětivost. Pod tímto termínem si každý z nás může představit něco jiného. Já jsem své pojetí popsal v předchozím textu. Jde mi především o jednoduchost a intuitivnost ovládání. Testování těchto dvou věcí jsem prováděl za pomoci pěti dobrovolníků, kteří v několika etapách zkoušeli s aplikacemi pracovat. Většina z nich už prošla kurzem, ve kterém se zmíněné algoritmy (nebo alespoň jejich část) probírala a zbývajícím jsem před začátkem vysvětlil, oč se jedná. Hlavním účelem těchto testů bylo prověřit a upravit ovládání tak, aby bylo co možná nejkomfortnější a přehledné.

V tomto ohledu zaznamenala největší úspěch aplikace bludiště. Tu dokázali ovládat bez problému všichni zúčastnění. Naopak nejhůře dopadly první dvě aplikace (ID3 a Back Propagation), Obě se ukázaly jako nevhodné pro ty uživatele, kteří ještě dané algoritmy dostatečně neznali. Podobný problém, ale v menším měřítku, se projevil i u aplikace s algoritmem SOM, která vyžaduje nastavení neuronové sítě, což je pro neznalého uživatele problém.

Jako velice důležitý podmět k zlepšení se také ukázala jedna z připomínek, která při testování zazněla. Šlo o možnost informovat uživatele o tom, že pokud provede změny na plátně, je nutné tyto změny před použitím uložit. Proto jsem tuto informaci přidal před inicializaci. Pokud tedy nebudou změny uloženy, není možné začít s inicializací a uživatel je o tom informován formou dialogového okna.

7.2 Algoritmy

Nyní se podíváme podrobněji na dva testy, které jsem ještě v tomto textu neuváděl.

Jako první zmíním porovnání dvou způsobů vybírání počátečních umístění těžišť v algoritmu K-Means clustering. V kapitole 3.3 jsem zmínil, že je vhodné vybírat tyto body náhodně, protože následně dochází k lepšímu rozpoznání shluků. To, zda tomu tak opravdu je, nebo stačí vybrat prvních x bodů z trénovací množiny, nyní uvidíme v číslech.

Vytvořil jsem 8 trénovacích množin, kde každá má jiné parametry. Liší se nejen v počtu shluků, ale i pořadím, v jakém byly body vytvářeny. Všechny množiny jsou součástí přiloženého CD. Jsou uloženy ve složce testy pod názvem test1. Každý test jsem zopakoval patnáctkrát a zapisoval výsledek rozpoznání shluků. Všechny výsledky můžeme vidět v tabulce 2. Z výsledků je patrné, že

pokud se jedná o rozpoznání dvou shluků, jsou oba způsoby bezchybné (1.csv a 7.csv). Změna nastává u ostatních příkladů. Zatímco v případě náhodného výběru se vždy povedlo alespoň několikrát shluky správně rozpoznat, v druhém případě hodně záleží na tom, jak jsou body uspořádány. Pokud se nepovede rozpoznání napoprvé, nepovede se vůbec. Je to dáno tím, že jsme odstranili jediný náhodný faktor v celém algoritmu, a tím jsme zcela závislí na rozložení prvních bodů. Pokud například víme, že každý z prvních x bodů bude patřit vždy do jiného shluku, můžeme s velkou výhodou využít druhé možnosti. Stejně tak je možné umístit středy shluků rovnou na předpokládané místo a získat tak výhodu vždy správného rozpoznání. V opačném případě více vyhovuje využití varianty náhodného výběru.

Název trénovací množiny	Náhodný výběr		Prvních x bodů	
	Správně	Špatně	Správně	Špatně
1.csv	15	0	15	0
2.csv	12	3	0	15
3.csv	5	10	0	15
4.csv	7	8	15	0
5.csv	11	4	0	15
6.csv	3	12	0	15
7.csv	15	0	15	0
8.csv	5	10	0	15

Tabulka 2

U porovnání náhodné a přímé varianty ještě zůstaneme. Jen se přesuneme k jinému algoritmu. V případě Q-Learningu máme na výběr, zda startovací bod zvolíme náhodně nebo nikoliv. Pro otestování, která varianta je lepší, jsem vytvořil speciální bludiště, které je dostatečně velké na to, aby vynikl rozdíl v rychlostech. Prohlédnout si ho můžete na přiloženém CD ve složce testy pod názvem testu test2. Název souboru je maze.csv. Současně ve složce najdete zdrojový kód, na kterém jsem testování provedl. Zadání úlohy je prosté. Nalézt v co možná nejkratším čase cestu od bodu v levém horním rohu (první bod bludiště) do cílového stavu umístěnému v pravém dolním rohu bludiště. První varianta si vždy po ukončené procházce vygeneruje novou náhodnou startovní pozici, zatímco v druhém případě agent začíná vždy ze stejného bodu na prvním stavu bludiště (tedy stejný stav, od kterého má být nalezena cesta). Výsledný rozdíl mezi oběma variantami je až překvapivě velký. Zatímco v případě náhodného výběru startovací pozice algoritmus skončil s časem 55,243 sekund, při neměnící se pozici to trvalo 6 minut a 54,285 sekundy, což je 7,5 krát více než v prvním případě. Na základě tohoto výsledku jsem přehodnotil využití statického startovacího stavu u Q-Learningu a stejně jako v případě SARSA již aplikace generuje tyto body náhodně.

8 Závěr

V této práci bylo vysvětleno šest základních algoritmů strojového učení s ohledem na možné přístupy k nim. A to konkrétně:

- Učení s učitelem:
  - ID3 algoritmus.
  - Back Propagation.
- Učení bez učitele:
  - K-Means clustering.
  - Self-Organizing Maps (SOM).
- Posilované učení:
  - Q-Learning.
  - State-Action-Reward-State-Action (SARSA)

Ke všem těmto algoritmům byly navrženy a následně zpracovány úlohy formou samostatných aplikací. Ty uživateli umožňují zaměřit se na konkrétní algoritmus, aniž by byl rozptylován ostatními. Všechny aplikace byly navrženy tak, aby byly co možná nejjednodušší a s intuitivním ovládáním. To se následně v rámci testování potvrdilo jako správné. Zvláště pak u aplikace pro algoritmy Q-Learning a SARSA, kterou uživatelé hodnotili jako velice povedenou. Naopak aplikace s algoritmy ID3 a Back Propagation u uživatelů propadly. Byly hodnoceny jako velice špatně použitelné. Testování také potvrdilo, že využití náhodných akcí je v algoritmech velice důležité a bez tohoto faktoru dochází k nepříznivému chování, například v podobě prodloužení času výpočtu.

Kromě samotných aplikací s úlohami byla zároveň vytvořena knihovna se všemi algoritmy, takže jakýkoliv zájemce má možnost kterýkoliv z nich využít ve svém vlastním projektu.

Všechny zdrojové kódy jsou napsané v jazyce C++ s využitím Qt toolkitu. Výjimku tvoří samotné algoritmy, které jsou napsány v čistém C++ bez využití jakýchkoliv dalších nástrojů. To umožňuje jednoduší použití třetím osobám, které tak nemusí dodatečně nic instalovat.

Tato práce by mohla být v budoucnu rozšířena směrem k vytvoření simulací každého z těchto algoritmů, především ID3 a Back Propagation, které by tím získaly mnohem lepší informační potenciál. Bylo by také možné rozšířit stávající sadu šesti algoritmů o další, tentokrát již třeba o ne zcela základní algoritmy.

Existují částečně podobné práce, které vznikly v současné době. Jedná se především o dvě bakalářské práce, a to „Rozpoznávání objektů v obrazech“ a „Rozpoznávání ručně psaného písma“. Obě práce se totiž věnují problematice, kterou by bylo možné alespoň z části řešit některým ze zde popsaných algoritmů.

9 Použitá literatura

1. SUTTON, R. S. a G. BARTO. *Reinforcement Learning: An Introduction*. Cambridge: The MIT Press, 2005 [cit. 2013-04-20]. Dostupné z: http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node17.html
2. Markovův rozhodovací proces. *Wikiperie* [online]. 2013, verze 27. 3. 2013 [cit. 2013-04-20]. Dostupné z: http://cs.wikipedia.org/wiki/Markov%C5%AFv_rozhodovac%C3%AD_proces#Definice
3. Entropie. *Wikiperie* [online]. 2013, verze 12.4.2013 [cit. 2013-04-21]. Dostupné z: http://cs.wikipedia.org/wiki/Entropie#Informa.C4.8Dn.C3.AD_entropie
4. MAŘÍK, V. O. ŠTĚPÁNKOVÁ a J. LAŽANSKÝ. *Umělá inteligence 4*.. Praha: Academia, 2003. ISBN 8020010440.
5. CHALUPNÍK, V. Biologické algoritmy (5) - Neuronové sítě. *root.cz* [online]. 2012, verze 14. 6. 2012 [cit. 2013-04-25]. Dostupné z: http://www.root.cz/clanky/biologicke-algoritmy-5-neuronove-site/
6. ZELINKA, I. *Umělá inteligence - hrozba nebo naděje?*. Praha: Nakladatelství BEN, 2003. ISBN 80-7300-068-7.
7. NILSSON, N. J. *Introduction to Machine Learning*. Stanford University, 1998. Dostupné také z: http://robotics.stanford.edu/~nilsson/mlbook.html
8. MAŘÍK, V. O. ŠTĚPÁNKOVÁ a J. LAŽANSKÝ. *Umělá inteligence*.. Praha: Academia, 1993. ISBN 8020004963.
9. MEHROTRA, K. K. MOHAN a S. RANKA. *Artificial neural networks*.. Cambridge: MIT Press, 1997. ISBN 0262133288.

10 Seznam příloh

Příloha 1. DVD obsahující:

- Elektronickou verzi této práce.
- Zdrojové kódy všech aplikací.
- Spustitelné aplikace s ukázkovými příklady.
- Uživatelskou příručku pro vytvořené aplikace.
- DLL knihovnu se všemi algoritmy.
- Testované trénovací množiny.