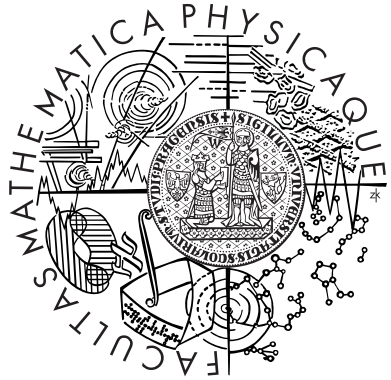
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Vojtěch Havránek

Umělé neuronové sítě a zpětnovazebné učení

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Doc. RNDr. Iveta Mrázová, CSc.
Studijní program: Informatika

Rád bych zde poděkoval všem, kteří mi byli nápomocní při psaní této práce. Především děkuji Doc. RNDr. Ivetě Mrázové, CSc. za cenné rady a připomínky. Děkuji také své rodině za její podporu a trpělivost v době vzniku této práce.

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

V Praze dne 7.3.2008 Vojtěch Havránek

Obsah

Název práce: *Umělé neuronové sítě a zpětnovazebné učení*
Autor: *Vojtěch Havránek*
Katedra: *Katedra teoretické informatiky a matematické logiky*
Vedoucí diplomové práce: *Doc. RNDr. Iveta Mrázová, CSc.*
e-mail vedoucího: *iveta.mrazova@mff.cuni.cz*
Abstrakt: *Při řešení složitých úloh strojového učení bývá často obtížné specifikovat přesný postup vedoucí k jejich správnému řešení. V praxi proto může být výhodnější využít např. zpětnovazebného učení, které vyžaduje jedinou informaci o řešené úloze, a to odměnu úměrnou vhodnosti akcí agenta. Ukazuje se, že algoritmy zpětnovazebného učení pracující s neuronovými sítěmi mohou v řadě takových úloh dosáhnout dobrých výsledků. Dosažené výsledky mohou být lepší, má-li neuronová síť schopnost modelovat prostředí nebo je-li rozšířena o rekurentní vazby. V práci ukazujeme, že pro danou síť, která správně predikuje odměnu, je nalezení optimální akce v obecném případě NP-úplná úloha.*
Popisujeme tři modely neuronových sítí. Jedním z nich je námi přizpůsobená varianta Suttonova algoritmu TD(λ) pro nemarkovské prostředí. Všechny tři modely jsme důkladně otestovali v námi vytvořeném simulátoru dravce a kořisti. Nejúspěšnější z testovaných modelů - modifikovaný TD(λ) jsme následně aplikovali při řízení reálného mobilního robota. V práci se zároveň zabýváme vhodným způsobem odměňování, biologickou opodstatněností zvažovaných modelů neuronové sítě, důležitostí explorativních schopností algoritmu a mezemi použitelnosti zpětnovazebného učení. Součástí práce je knihovna neuronových sítí napsaná v jazyku C++, ve které jsou popsané modely implementovány.
Klíčová slova: *Dynamické neuronové sítě, zpětnovazebné učení, robotika, Temporal Difference learning, prohledávání*

Title: *Artificial neural networks and reinforcement learning*
Author: *Vojtěch Havránek*
Department: *Department of Theoretical Computer Science and Mathematical Logic*
Supervisor: *Doc. RNDr. Iveta Mrázová, CSc.*
Supervisor's e-mail address: *iveta.mrazova@mff.cuni.cz*
Abstract: *When solving complex machine learning tasks, it is often more practical to let the agent find an adequate solution by itself using e.g. reinforcement learning rather than trying to specify a solution in detail. The only information required for reinforcement learning is a reward that gives the agent reinforcement about the desirability of his actions. Our experiments suggest that good results can be achieved by reinforcement learning with online learning neural networks. The functionality of such neural network may be further extended by allowing it to model the environment and/or by providing it with recurrent connections. In this thesis, we show that for a given network predicting the reward, it is NP-complete to find the agent action that maximizes this reward.*
We describe three neural network models, one of them being an original modification of Sutton's TD(λ) algorithm that extends its domain to non-Markovian environments. All three models were thoroughly tested with our predator-prey simulator. The most powerful of them, the modified TD(λ) was then applied to control of a real mobile robot. Simultaneously, we have discussed the principles of rewarding the agents, the biological plausibility of the algorithms, the importance of the exploration capabilities and general bounds of reinforcement learning. As a significant part of the thesis, a C++ neural networks library was developed and the described models were implemented.
Keywords: *Dynamic neural networks, reinforcement learning, robotics, Temporal Difference learning, exploration*

Kapitola 1

Úvod

Umělá inteligence často řeší úlohy, ve kterých se snažíme naučit agenta dosáhnout požadovaného cíle určitou sekvencí akcí. Správná posloupnost akcí přitom může být tak složitá, že by bylo velmi komplikované ji přesně formulovat a agentovi naprogramovat. Často ani správnou posloupnost akcí vedoucích k cíli neznáme. Dokážeme ale posoudit, nakolik je dosažený výsledek kvalitní, nebo můžeme v průběhu aktivity agenta posoudit, do jaké míry se jeho činnost ubírá správným směrem.

Příkladem takového agenta a úlohy může být mobilní robot - automatický vysavač, kterého chceme naučit vyluxovat celou místnost. Naprogramovat robotovi konkrétní trajektorii, po které se má místností pohybovat, by bylo pracné, a robot by navíc byl přizpůsoben jedné konkrétní místnosti a v ostatních místnostech by mohl zabloudit.

Vhodnou technikou v takové situaci může být tzv. zpětnovazebné učení[1] [2]. Zpětnovazebné učení představuje jednu z metod strojového učení, při které agentovi neposkytujeme informaci o tom, jak se má chovat, ale pouze mu prostřednictvím speciálního signálu, tzv. odměny nebo též zpětné vazby[2], dáváme informaci o tom, nakolik je jeho momentální aktivita žádoucí. V případě robotického vysavače bychom mohli agentovi v každém okamžiku udělit odměnu úměrnou například objemu smetí, které v daném okamžiku nasbíral. Problematika zpětnovazebného učení se pak zabývá hledáním vhodných algoritmů, které agentovi umožní najít správné řešení zadané úlohy pouze s využitím informace o jeho vstupech, výstupech a výši odměny v každém okamžiku. Agent se tedy v jistém smyslu učí z vlastních chyb.

Živé organismy, zejména vyšší živočichové, jsou vybaveny vysokou schopností adaptace na vnější podmínky. Lidé a dokonce i lidoopové jsou schopni naučit se velmi složitému chování, aby dosáhli svých cílů. Je možné, že podstatná část tohoto učení probíhá způsobem podobným zpětnovazebnému učení. Proto je na místě otázka, zda lze s využitím zpětnovazebného učení dosáhnout srovnatelných výsledků i u umělých agentů. Nejen z tohoto důvodu jsou zajímavým nástrojem zpětnovazebného učení umělé neuronové sítě, které byly navrženy podle vzoru nervové soustavy biologických organismů. V této práci se budeme zabývat právě použitím umělých neuronových sítí pro účely zpětnovazebného učení. Při jejich aplikaci budeme kromě dosažené úspěšnosti zvažovat též biologickou opodstatněnost použitých modelů.

[1] anglicky reinforcement learning

[2] anglicky reward respektive reinforcement

Úloha zpětnovazebného učení bude záměrně vymezena velmi obecně. Lze ale očekávat, že čím obecnější úlohy dokáže konkrétní algoritmus zpětnovazebného učení řešit, tím bude méně efektivní. Budeme tedy hledat určitý kompromis mezi obecností a efektivitou algoritmu. Naším cílem tedy nebude najít univerzální algoritmus, který by dokázal optimálně vyřešit libovolnou úlohu zpětnovazebného učení. Naším cílem bude najít algoritmus a model neuronové sítě, který dává dostatečně dobré výsledky pro co nejširší podmnožinu úloh, se kterými se v praxi můžeme setkat. Mimo hledání vhodného algoritmu se budeme též zabývat otázkou, jakým způsobem je vhodné odměňovat, abychom agentovi co nejvíce usnadnili hledání řešení. Metodou, která je často alternativou k použití zpětnovazebného učení, jsou tzv. genetické algoritmy[6]. Těmi se v této práci zabývat nebudeme.

V následující kapitole jsou formálně zavedeny pojmy z oblasti zpětnovazebného učení a představeny základní algoritmy zpětnovazebného učení. V kapitole 3 jsou pak zavedeny pojmy z oblasti neuronových sítí a popsán algoritmus zpětného šíření. Kapitola 4 se nejprve zabývá možnostmi použití neuronových sítí pro zpětnovazebné učení obecně, poté popisuje vybrané konkrétní modely neuronových sítí a algoritmy, které s nimy pracují.

V kapitole 5 jsou pak popsány experimenty se simulátorem dravce a kořisti a experiment z oblasti robotiky, ve kterých byly tyto algoritmy otestovány. V kapitole 6 pak srovnáváme zpětnovazebné učení s úlohou prohledávání stavového prostoru a z tohoto srovnání vyvodíme určité doporučení, jak vhodně stanovit odměnu. Dále v této kapitole uvažujeme o mezích použitelnosti metody zpětnovazebného učení. V závěru pak shrneme přínos této práce a naznačíme možnosti jejího dalšího pokračování. Součástí práce je též autorem vyvinutá knihovna funkcí pro práci s neuronovými sítěmi a simulátor dravce a kořisti, které jsou přiloženy k práci na samostatném CD. Popis tohoto software a popis obsahu přiloženého CD lze nalézt v přílohách v závěru této práce.

Kapitola 2

Zpětnovazebné učení

2.1 Princip zpětnovazebného učení

Definujme nyní potřebné pojmy precizněji. Máme dáno prostředí a v něm agenta, který s prostředím interaguje. Agentem může být například počítačový program, robot nebo živý organismus. Čas považujeme za diskrétní a začínající od okamžiku 0. V každém časovém okamžiku t dostává agent na vstup potenciálně neúplnou informaci o stavu prostředí s_t (dále jen "stav") z množiny možných stavů S a zasahuje do prostředí vykonáním akce a_t z množiny proveditelných akcí $A(s_t)$, čímž se dostává do stavu s_{t+1}. Stav s_0 nazýváme počáteční. Mohou existovat tzv. koncové stavy $s \in S$, pro něž je množina $A(s)$ prázdná a v nichž činnost agenta končí. Výsledek agentovy akce je obecně nedeterministický, protože může záviset na informacích o prostředí, které nejsou součástí agentova vstupu s, a navíc prostředí samo se může chovat stochasticky. Ve stavu s dostává agent odměnu[1] $r(s) \in \mathbb{R}$, což je číslo ohodnocující úspěšnost agenta[2]. Odměna $r(s)$ může být i záporná, což značí, že stav s je nežádoucí. Záporná odměna se nazývá trest. Odměnu získanou v čase t značíme r_t a platí $r_t = r(s_t)$.

Nechť se agent nachází ve stavu s_t, do kterého se dostal přes stavy $s_0, s_1, \ldots, s_{t-1}$ vykonáním akcí $a_0, a_1, \ldots, a_{t-1}$. Potom pravděpodobnost, že po vykonání akce a_t se agent dostane do stavu s, značíme:

$$Pr\{s_{t+1} = s \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0\} \tag{2.1}$$

Agent může, ale nemusí mít možnost si uchovávat svůj vnitřní stav (paměť). Řídí se nějakým algoritmem. K formalizaci algoritmu, který řídí jeho chování, používáme pojem strategie[3]. Strategie π je funkce, která každému stavu $s \in S$ přiřadí nějakou proveditelnou agentovu akci $a \in A(s)$.

[1] anglicky reward

[2] Odměnu můžeme alternativně definovat jako funkci $r(s, a)$ označující odměnu, kterou agent dostane při vykonání akce a ve stavu s. Formalismy jsou vzájemně ekvivalentní. Lze totiž položit $r(s, a) = r(s')$, ale převedení úlohy používající $r(s, a)$ na námi používaný formalismus může vyžadovat rozdělení jednoho stavu s na více různých stavů v tom případě, že při různých způsobech, jak se dostat do stavu s, dostává agent různou odměnu.

[3] anglicky policy

Naším cílem je najít pro agenta takovou strategii, kterou získá co možná největší kumulativní odměnu[4]

$$R = r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots = r_0 + \sum_{k=1}^{\infty} \gamma^k r_k, \tag{2.2}$$

kde $0 \leq \gamma \leq 1$ je pevně zvolená konstanta, tzv. discount factor. V extrémním případu, kdy $\gamma = 0$, bude agent motivován vykonávat takové akce, které přinesou co nejvyšší okamžitou odměnu. Naopak pro vyšší γ blízké jedničce bude agent přisuzovat větší význam i odměnám, které získá později. Pokud není zaručeno, že agent časem dospěje do koncového stavu, mělo by platit, že $\gamma < 1$ a výše odměny r by měla být omezená, aby byl problém dobře definován.

Optimální strategií je taková strategie, která přináší maximální možnou kumulativní odměnu. Úlohu jsme ale definovali velmi obecně, a proto by bylo v řadě případů nalezení optimální strategie příliš těžké. Často nám však bude postačovat strategie, která přináší "dostatečně vysokou" kumulativní odměnu.

V obecném případě zpočátku agent nemá znalosti o řešené úloze a nezná pravděpodobnost výsledků svých akcí. Musí se naučit maximalizovat odměnu jen z vlastní zkušenosti. Agent je v podobné situaci jako pokusná myš v labyrintu, která je v cíli odměněna potravou, zatímco dostane elektrický šok, zvolí-li špatnou cestu.

Poznamenejme, že hranice mezi agentem a prostředím se nemusí shodovat s fyzickou hranicí agentova těla. Za součást agenta považujeme pouze to, co spadá pod agentovu výhradní kontrolu. Speciálně, systém poskytující agentovi odměnu není jeho součástí. Například je-li součástí robota i obvod, který ze senzorů počítá výši odměny, nepovažujeme jej za součást agenta, ale spíše součást prostředí.

Příklad: Agentem je robot, jehož cílem je uklidit odpadky v místnosti. Stav s je informace, kterou robot zaznamená svými senzory. Možné akce jsou změna směru pohybu robota, změna rychlosti, pohyb končetin. Pokaždé, když robot vyhodí nějaký odpad do koše, dostane odměnu. Výsledek robotových akcí je nejistý, předměty mu mohou z končetiny vyklouznout, kolemjdoucí člověk může odhodit další odpadky, apod. Robot není zpočátku vybaven programem na uklízení, pouze nějakým algoritmem, podle kterého se uklízet učí.

2.2 Konečný Markovský rozhodovací proces

K výše uvedenému formalismu mohou být přidány další předpoklady, což umožňuje snadnější řešení problému. Často předpokládáme, že množiny S a $A(s)$ jsou konečné a že prostředí se chová jako tzv. Markovský rozhodovací proces (Markov decision process). Pro Markovský rozhodovací proces platí tzv. Markovská podmínka[5], totiž že pravděpodobnost, že akce a_t ve stavu s_t povede ke stavu s_{t+1}, není závislá na historii předchozích stavů, vstupů a odměn:

$$Pr\{s_{t+1} = s \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0\} = Pr\{s_{t+1} = s \mid s_t, a_t\} \overset{ozn.}{=} Pr_{a_t}\{s_t, s\} \tag{2.3}$$

[4]anglicky cumulative reward nebo též return

[5]Markov property

Platí-li Markovská podmínka, agent nepotřebuje žádnou paměť, protože by pro jeho rozhodování neměla žádný přínos.

Příklad: Agent je program hrající dámu. Stavem s je pozice kamenů na hrací desce. Možné akce jsou přípustné tahy. Agent dostává menší odměnu za vzetí soupeřova kamene nebo získání dámy, zápornou odměnu (trest) za ztrátu kamene, velkou odměnu za výhru partie a velký trest za prohru partie. Jedná se o Markovský rozhodovací proces, neboť informace o tom, jak vypadala hrací deska v předchozích tazích, nemá žádný přínos pro předpověď dalšího vývoje partie. To ale platí pouze v případě, kdy neuvažujeme možnost odhadnout z historie chování soupeře.

Pro řešení Markovského rozhodovacího procesu lze použít metody tzv. dynamického programování[2], které postupně zlepšují nejlepší nalezenou strategii π, až nakonec konvergují k optimální strategii π^*. Tyto metody pracují s funkcí V, která pro danou strategii π ohodnocuje každý stav s. $V_\pi(s)$ je očekávaná kumulativní odměna, kterou agent získá, pokud začne ve stavu s a pokračuje podle strategie π. $V^*(s) = V_{\pi^*}(s)$ je funkce, která každému stavu přiřazuje kumulativní odměnu, kterou agent získá, pokud bude vycházet ze stavu s a řídit se optimální strategií. Platí tzv. Bellmanova rovnice:

$$V_\pi(s) = r(s) + \gamma \sum_{s' \in S} (Pr_{\pi(s)}\{s, s'\} V_\pi(s')) \tag{2.4}$$

Metody dynamického programování si typicky uchovávají nejlepší dosud nalezenou strategii π a jí odpovídající funkci V v tabulce hodnot. Tyto hodnoty iterativně upravují s využitím 2 předpisů:

$$\pi(s) \leftarrow \arg\max_a \left(\sum_{s' \in S} Pr_a\{s, s'\} V(s') \right) \tag{2.5}$$

$$V(s) \leftarrow r(s) + \gamma \sum_{s' \in S} Pr_{\pi(s)}\{s, s'\} V(s') \tag{2.6}$$

Tyto kroky se opakují v pořadí, které se různí mezi jednotlivými metodami tak, aby byly aktualizovány hodnoty V a π pro všechny stavy. Aktualizace probíhá do té doby, dokud se aktualizované hodnoty neustálí. Potom π konverguje k π^* a V konverguje k V^*. Mezi tyto metody patří tzv. "Policy iteration", při kterém střídavě provádíme jednou první krok pro všechny stavy a poté opakujeme krok 2, dokud se hodnoty V mění. Dále sem patří metoda "Value iteration". Při "Value iteration" si nepotřebujeme ukládat strategii π do tabulky, neboť se její hodnota vždy odvodí podle potřeby. Upravujeme tedy pouze funkci V, a to dle předpisu:

$$V(s) \leftarrow r(s) + \gamma \max_a \left(\sum_{s' \in S} Pr_a\{s, s'\} V(s') \right) \tag{2.7}$$

Použití těchto metod však vyžaduje, abychom předem znali pravděpodobnosti výsledků akcí agenta. V případě, že je neznáme, můžeme použít alternativní metodu, která se nazývá Q-učení[20]. Q-učení používá místo funkce V funkci Q, která ohodnocuje všechny dvojice stav-akce. Hodnota $Q_\pi(s, a)$ označuje očekávanou kumulativní odměnu,

kterou agent získá, pokud nejprve vykoná ve stavu s akci a, a potom pokračuje dle strategie π. Hodnota $Q^*(s,a) = Q_{\pi^*}(s,a)$ označuje očekávanou kumulativní odměnu při použití optimální strategie ve stavu s.

Podobně jako pro $V_\pi(s)$ platí i pro $Q_\pi(s,a)$ Bellmanova rovnice:

$$Q_\pi(s,a) = r(s) + \gamma \sum_{s' \in S} (Pr_a\{s,s'\}.V_\pi(s')) = r(s) + \gamma \sum_{s' \in S} (Pr_a\{s,s'\}.Q_\pi(s',\pi(s'))) \quad (2.8)$$

Agent se potom učí přímo během (simulované) interakce s prostředím, takzvaně online - viz dále. Ve stavu s_t získá odměnu r_t vykoná akci a, která ho přenese do stavu s_{t+1}, a poté upraví hodnotu Q podle vzorce:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha_t(r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)), \quad (2.9)$$

kde α_t je parametr učení, $0 \leq \alpha_t \leq 1$. Je dokázáno[20], že s rostoucím t konverguje Q k Q^* pokud platí všechny z následujících podmínek:

- Pro každou akci a použitelnou ve stavu s je k dispozici neomezený počet pozorování.
- Posloupnost α_t jde monotónně shora k nule.
- $\sum_t \alpha_t = \infty$

2.3 Omezení dynamického programování

Popsané metody zpětnovazebného učení reprezentují odhady funkcí V respektive Q pomocí tabulky hodnot, jejich paměťové nároky jsou proto úměrné počtu možných stavů a počtu proveditelných akcí agenta. I když je tento počet konečný, bývá často tak velký, že vylučuje praktické použití těchto metod. Má-li například robot n senzorů, z nichž každý může být v některém z m možných stavů, je celkový počet možných stavů robota m^n. Počet možných stavů agenta tedy roste exponenciálně s počtem senzorů, a velmi snadno se tak dostaneme k počtu stavů, při kterém budou zmiňované algoritmy dynamického programování nepoužitelné. Navíc kromě paměťové náročnosti s počtem stavů a proveditelných akcí roste i čas potřebný ke konvergenci algoritmu. Tento problém se někdy označuje jako Bellmanovo prokletí dimensionality.

Existují nicméně algoritmy, které používají kompaktní reprezentaci znalostí - snaží se predikované hodnoty $V(s)$ přiblížit pomocí vhodné dostatečně obecné, ale krátce popsatelné funkce, např. pomocí nějaké kombinace příznaků odvozených z agentových vstupů nebo pomocí neuronové sítě. Příkladem takového algoritmu je Suttonův TD(λ). Tento algoritmus použil Gerald Tesauro ve svém TD-Gammonu[19], programu pro hraní hry Backgammon, který se na začátku devadesátých let téměř vyrovnal lidským šampionům (viz níže).

Předpoklady pro použití výše uvedených algoritmů jsou navíc velmi omezující. Agent obvykle nedostává na vstupu kompletní reprezentaci stavu prostředí, ale "vidí jen jeho část". Ke správné predikci výsledku agentovy akce proto může být potřebná informace

o vstupech, které agent pozoroval v minulosti. V tom případě už ale neplatí Markovská podmínka a agent potřebuje pro kvalitní rozhodování také nějaký vnitřní stav, který zachycuje informaci o minulých rozhodnutích a pozorováních. Právě techniky využívající umělé neuronové sítě by ale mohly být vhodné i v situacích, kde Markovská podmínka neplatí.

Příklad: Uvažme robota, který se učí hrát fotbal. Nahraje spoluhráči a otočí se směrem k soupeřově bráně. V tu chvíli ztratí z dohledu míč. Pokud si v paměti neuchovává, že míč drží spoluhráč, těžko se může rozhodnout, zda pokračovat směrem k soupeřově bráně, a nebo se stáhnout k obraně brány vlastní.

2.4 Závěrečné poznámky

Některé problémy zpětnovazebného učení je v praxi možné řešit tak, že úlohu necháme mnohokrát odsimulovat, abychom agenta naučili vhodnou strategii. Když je pak naučená strategie uspokojivá, učení zastavíme a nalezenou strategii nasadíme. Tento tzv. offline způsob učení ale nemusí být použitelný pro jiné problémy. Potom musí dojít k učení online (za běhu) a my musíme rozhodnout, kolik prostředků vynaloží agent na tzv. exploraci (tj. učení, průzkum nových možností) a exploataci (tj. využití nejlepšího nalezeného postupu) tak, aby kumulativní odměna za celou dobu činnosti agenta byla maximální.

Pro zajímavost ještě poznamenejme, že nedávno byl objeven obecný způsob, jak optimálně predikovat na základě dosud pozorovaných skutečností. Na něm je založen algoritmus optimálně se rozhodujícího agenta AIXI Marcuse Huttera[8]. Tento algoritmus je ale velmi neefektivní a prakticky nerealizovatelný, neboť funguje tak, že se snaží modelovat prostředí pomocí všech použitelných Turingových strojů. AIXI byl popsán ve Švýcarském institutu IDSIA, kde dále probíhá výzkum z cílem najít podobný algoritmus, který by byl prakticky použitelnější.

Kapitola 3

Neuronové sítě

3.1 Umělý neuron

Základní výpočetní jednotkou umělé neuronové sítě je neuron. Neuron má n vstupních a jedno výstupní spojení. Tato spojení se nazývají synapse. Každá vstupní synapse $i, 0 \leq i < n$ má svoji váhu $w_i \in \mathbb{R}$. Při každém kroku výpočtu vstupuje do neuronu n vstupních hodnot $x_0, x_1, \ldots, x_{n-1}$. Neuron vypočítá vážený součet vstupních hodnot a odpovídajících vah snížený o tzv. práh b. Tato hodnota se nazývá potenciál neuronu a budeme ji značit ξ.

$$\xi = \left(\sum_{i=0}^{n-1} x_i w_i \right) - b \tag{3.1}$$

Na potenciál je aplikována tzv. přenosová funkce $f(\xi)$. Výsledná funkční hodnota je pak přiřazena výstupu y. Tato hodnota se nazývá výstup neuronu. Přenosová funkce je typicky neklesající s hodnotami z intervalu $[0, 1]$, méně často z intervalu $[-1, 1]$. Někdy se používá funkce signum, vracející 0 pro záporný potenciál a 1 pro nulu nebo kladný potenciál. Častěji se ale setkáme s funkcí sigmoidální:

$$f(\xi) = \sigma(\xi) = \frac{1}{1 + e^{-\lambda \xi}}, \tag{3.2}$$

kde $\lambda > 0$ je konstanta vyjadřující tzv. strmost přenosové funkce.

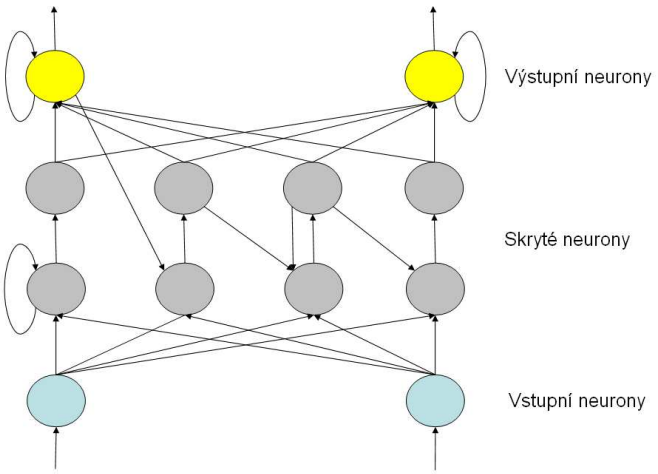
Výhodou této funkce je, že je spojitě derivovatelná podle svých vstupních vah, a navíc lze tyto derivace velmi efektivně vypočítat:

$$\begin{aligned}
\sigma'(\xi) &= \left(\frac{1}{1 + e^{-\lambda \xi}} \right)' (\xi) = \\
&= -\frac{1}{(1 + e^{-\lambda \xi})^2} (e^{-\lambda \xi}.(-\lambda)) = \\
&= \lambda \frac{e^{-\lambda \xi}}{(1 + e^{-\lambda \xi})^2} = \\
&= \lambda \frac{1 + e^{-\lambda \xi} - 1}{(1 + e^{-\lambda \xi})^2} =
\end{aligned}$$

$$
\begin{aligned}
&= \lambda\left(\frac{1}{1+e^{-\lambda\xi}} - \frac{1}{(1+e^{-\lambda\xi})^2}\right) = \\
&= \lambda(\sigma(\xi) - \sigma^2(\xi)) = \\
&= \lambda\sigma(\xi).(1-\sigma(\xi))
\end{aligned} \tag{3.3}
$$

Pokud nebude uvedeno, jinak, budeme v dalším textu předpokládat, že hodnota parametru λ je 1. V části 4.4 se setkáme také s parametrem λ, ale ve zcela jiném významu. Je třeba tyto parametry důsledně odlišovat.

Vzájemným propojením více neuronů vzniká neuronová síť (obrázek 3.1). Neuronová síť má obvykle nějaké vstupní a výstupní neurony, které tvoří její vnější rozhraní. Výstupy vstupních neuronů představují vstupy pro celou síť a výstupy výstupních neuronů představují naopak výstupy celé sítě. Vstupní neurony nemají žádné vstupní synapse ani práh a přenosovou funkci. Neurony, které nejsou vstupní ani výstupní, se nazývají skryté.



Obrázek 3.1: **Příklad neuronové sítě.** *Synaptická spojení mohou tvořit cykly a dokonce mohou existovat reflexivní vazby neuronu se sebou samotným.*

Definice: Neuronová síť je uspořádaná sedmice $Net = (N, C, I, O, w, b, F)$, kde:

- N je neprázdná konečná množina neuronů
- $C \subseteq N \times N$ je neprázdná množina orientovaných spojů - synapsí mezi neurony
- $I \subseteq N$ je neprázdná množina vstupních neuronů
- $O \subseteq N$ je neprázdná množina výstupních neuronů
- $w : C \to \mathbb{R}$ je váhová funkce
- $b : (N \setminus I) \to \mathbb{R}$ je prahová funkce
- $F : (N \setminus I) \to (\mathbb{R} \to \mathbb{R})$ přiřazuje každému neuronu jeho přenosovou funkci

Značení

- Váhu synaptického spoje $w((i,j))$ mezi neurony i a j budeme značit zkráceně w_{ij}.
- Práh $b(i)$ neuronu i budeme též značit b_i.
- Přenosovou funkci $F(i)(\xi)$ neuronu i budeme značit $f_i(\xi)$. V případě, že mají všechny neurony stejnou přenosovou funkci, budeme psát jen $f(\xi)$.
- Výstup neuronu i budeme značit y_i.
- Při nějakém daném seřazení vstupních neuronů $i_0, i_1, \ldots, i_{l-1}$ budeme daný vektor jejich výstupů $(y_{i_0}, y_{i_1}, \ldots, y_{i_{l-1}})$ značit $\vec{x} = (x_0, x_1, \ldots, x_{l-1})$ a nazývat vstupní vektor.
- Chceme-li vyznačit, že se hodnoty mění v čase, budeme psát např. $y_i(t), w_{ij}(t), \vec{x}(t)$ apod.

Na místo toho, aby měly neurony práh, bývá síti často přidán jeden fiktivní vstupní neuron, který má konstantní výstup -1. Tento neuron je propojen se všemi nevstupními neurony a hodnota váhy těchto synapsí odpovídá hodnotě původních prahů.

Existuje mnoho různých modelů neuronových sítí lišících se způsobem propojení jednotlivých neuronů a svojí funkcí. Dle uspořádání synapsí rozlišujeme tzv. dopředné sítě, které neobsahují cyklická spojení, a naopak tzv. rekurentní (anglicky též feedback) sítě, které cykly obsahují. Cyklická spojení neuronů, jak uvidíme dále, mohou sloužit k delšímu uchovávání informace v síti, rekurentní síť tak může mít "paměť". Dle způsobu použití můžeme neuronové sítě rozdělit na statické a dynamické. Výpočty statické sítě jsou jednorázové. Výsledek předchozího výpočtu statické sítě nemá vliv na výpočet následující. Naproti tomu dynamická neuronová síť bude obvykle rekurentní a bude uchovávat informace z předchozího výpočtu pro jejich využití ve výpočtech následujících.

Příklad: Naším cílem je předvídat pomocí neuronové sítě počasí v dané lokalitě na základě údajů naměřených před jedním, dvěma a třemi dny. Budeme - li používat statickou síť, budeme ji muset vybavit zvláštními vstupními neurony pro údaje ze všech tří předchozích dnů. Použijeme - li rekurentní síť, stačí nám pouze vstupní neurony pro údaje z předchozího dne, protože starší údaje si síť mezi jednotlivými výpočty může uchovávat ve svých skrytých neuronech.

3.2 Síť zpětného šíření

Popíšeme nyní široce používaný model sítě zpětného šíření[1]. Síť zpětného šíření je příkladem tzv. vrstevnaté sítě (viz obrázek 3.2).

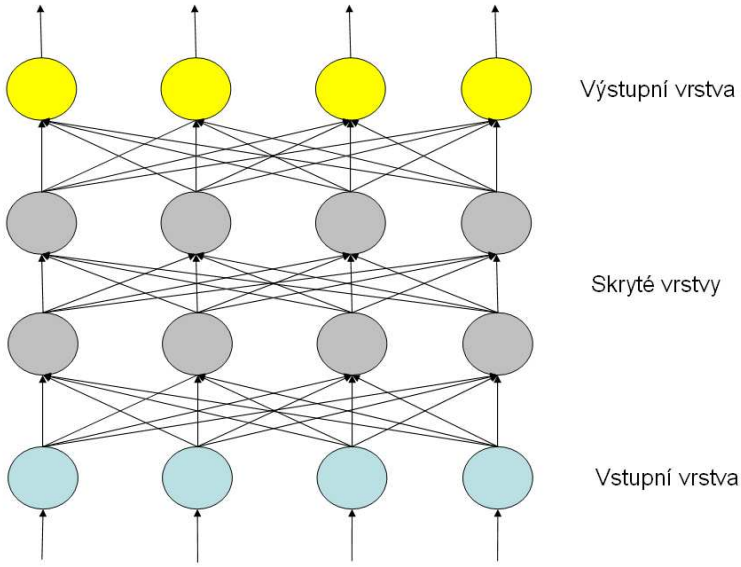
Vrstevnatá síť je dopředná síť, jejíž neurony jsou rozděleny do navzájem disjunktních množin nazývaných vrstvy. Nultá vrstva je vrstva vstupní, obsahuje vstupní neurony sítě. Neurony v každé další vrstvě jsou propojeny se všemi neurony z vrstvy předchozí. Vrstva

[1] anglickým názvem backpropagation

$k-1$ obsahuje výstupní neurony sítě a nazývá se výstupní. Všechny vrstvy kromě vstupní a výstupní se nazývají vrstvy skryté.

Definice: Neuronová síť $Net = (N, C, I, O, w, b, F)$ se nazývá vrstevnatá, pokud existují neprázdné množiny $N_0, N_1, \ldots, N_{k-1} \subseteq N$(vrstvy) takové, že:

- $\bigcup_{i=0}^{k-1} N_i = N$
- pro $i \neq j : N_i \cap N_j = \emptyset$ (vrstvy jsou navzájem disjunktní)
- $N_0 = I$ je tzv. vstupní vrstva
- $N_{k-1} = O$ je tzv. výstupní vrstva
- $C = \{(i,j) \mid \exists l : 0 \leq l \leq k-2 \wedge i \in N_l \wedge j \in N_{l+1}\}$ (Synapse vedou mezi po sobě následujícími vrstvami.)



Obrázek 3.2: **Vrstevnatá neuronová síť.** Zobrazená síť má vstupní vrstvu, 2 skryté vrstvy a jednu výstupní vrstvu. Každá z vrstev má čtyři neurony.

Výpočet vrstevnaté sítě (tzv. vybavování) probíhá po vrstvách. Nejprve je na vstupní vrstvu předložen vstupní vektor $\vec{x}$. Poté jsou vrstvu po vrstvě počítány výstupy neuronů pomocí již spočítaných výstupů neuronů z předchozí vrstvy.

Díky tomu, že jsou přenosové funkce nelineární, jsou vrstevnaté neuronové sítě schopny počítat nelineární funkce svých vstupů. Funkce, kterou síť počítá, je dána její strukturou a hodnotami jejích vah a prahů. Vrstevnaté neuronové sítě mají velmi dobré aproximační vlastnosti. Bylo teoreticky dokázáno, že při dostatečném počtu skrytých neuronů je vrstevnatá neuronová síť schopna aproximovat libovolnou borelovskou funkci s libovolnou přesností[7].

Značení

- Při nějakém daném seřazení výstupních neuronů $o_0, o_1, \ldots, o_{|O|-1}$ vrstevnaté sítě Net budeme vektor jejich výstupů $(y_{o_0}, y_{o_1}, \ldots, y_{o_{|O-1|}})$ vypočítaný pro vstupní vektor $\vec{x}$ značit $Net(\vec{x})$ a nazývat výstupní vektor Net pro $\vec{x}$.

Sítí zpětného šíření (chyby) budeme nazývat vrstevnatou síť, jejíž činnost se řídí algoritmem[13], který popíšeme níže. Pro síť zpětného šíření platí $\forall i \in N \setminus N_0 : F(i) = \sigma$ (všechny přenosové funkce jsou sigmoidy), a tudíž pro ni můžeme psát $Net = (N, C, I, O, w, b)$ a vynechat ze zápisu F.

Typické použití sítí zpětného šíření je tzv. učení s učitelem. V takovém případě máme k dispozici neuronovou síť Net a množinu trénovacích vzorů $T = \{(\vec{x}^0, \vec{d^0}), (\vec{x}^1, \vec{d^1}), \ldots, (\vec{x}^n, \vec{d^n})\}$, kde všechna $\vec{d^m}$ označují vektor požadovaných výstupů sítě Net pro vstupní vektor $\vec{x}^m$. Naším cílem je tedy, aby se pro každé $m \in \{0, 1, \ldots, n\}$ hodnota skutečného výstupu sítě $Net(\vec{x}^m)$ co nejvíce blížila $\vec{d^m}$. Pro tento účel zavádíme tzv. chybovou funkci

$$E_T(\vec{w}, \vec{b}) = \frac{1}{2} \sum_{m=0}^{n} \sum_{l=0}^{|O|-1} (Net(\vec{x}^m)_l - d_l^m)^2, \tag{3.4}$$

kde $\vec{w}$ značí vektor všech vah sítě a $\vec{b}$ značí vektor všech prahů sítě.

Naším cílem je tuto funkci minimalizovat. Pokud budou předkládané trénovací vzory dostatečně reprezentativním vzorkem množiny všech vzorů, se kterými se síť může v průběhu své činnosti setkat, síť by se měla minimalizací chybové funkce naučit generalizovat. To znamená, že setká-li se s novým vstupním vzorem, který se podobá některému z trénovacích vzorů, vydá výstup podobný požadovanému výstupu pro dotyčný trénovací vzor.

K minimalizaci chybové funkce nám slouží algoritmus zpětného šíření chyby. Tento algoritmus patří mezi tzv. gradientní metody. To znamená, že postupně snižuje hodnotu chybové funkce $E_T(\vec{w}, \vec{b})$ úpravami vah a prahů proti směru gradientu chybové funkce. To vyžaduje, abychom byli schopní co nejefektivněji vypočítat hodnoty parciálních derivací $E_T(\vec{w}, \vec{b})$ podle jednotlivých vah a prahů. Hodnota této parciální derivace udává, jak je třeba upravit hodnotu určité váhy, respektive prahu, abychom snížili hodnotu chybové funkce.

Uvažme nejprve váhy w_{ij} těch synapsí, které vedou do výstupní vrstvy ($j \in O, i \in N_{k-2}$) a trénovací množinu s jednou dvojicí vstupního vzoru a požadovaného výstupu $T = \{(\vec{x}, \vec{d})\}$.

$$\begin{aligned}
\frac{\partial E_T}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \left(\frac{1}{2} \sum_{l=0}^{|O|-1} (Net(\vec{x})_l - d_l)^2 \right) = \\
&= \frac{\partial}{\partial w_{ij}} \left(\frac{1}{2} (y_j - d(j))^2 \right) = \\
&= (y_j - d(j)) \frac{\partial y_j}{\partial w_{ij}} = \\
&= (y_j - d(j)) \sigma'(\xi_j) \xi_j'(w_{ij}) =
\end{aligned}$$

$$
\begin{aligned}
&\overset{3.3}{=} (y_j - d(j))\lambda\sigma(\xi_j).(1-\sigma(\xi_j))\xi'_j(w_{ij}) = \\
&\overset{3.1}{=} (y_j - d(j))\lambda\sigma(\xi_j).(1-\sigma(\xi_j))\frac{\partial}{\partial w_{ij}}\left(\left(\sum_{l\in N_{k-2}} y_l w_{lj}\right) - b_j\right) = \\
&= (y_j - d(j))\lambda y_j(1-y_j)y_i \\
&= -\delta_j y_i, \qquad (3.5)
\end{aligned}
$$

kde $d(j)$ pro $j \in O$ značí požadovanou hodnotu výstupu j (tedy $d(o_l) = d_l$) a kde

$$\delta_j = -(y_j - d(j))\lambda y_j(1-y_j) = -\frac{\partial E_T}{\partial \xi_j} \qquad (3.6)$$

nazýváme chybový člen neuronu j pro $j \in O$.

Předpokládejme nyní, že známe chybové členy všech neuronů ve vrstvě N_K. Pro w_{ij} splňující $j \in N_{K-1}, i \in N_{K-2}$ platí:

$$
\begin{aligned}
\frac{\partial E_T}{\partial w_{ij}} &= \sum_{l\in N_K} \frac{\partial E_T}{\partial \xi_l}\frac{\partial \xi_l}{\partial y_j}\frac{\partial y_j}{\partial \xi_j} y_i = \\
&= \sum_{l\in N_K} \delta_l w_{jl}\sigma'(\xi_j) y_i = \\
&= \sum_{l\in N_K} \delta_l w_{jl}\lambda y_j(1-y_j) y_i = \\
&= -\delta_j y_i, \qquad (3.7)
\end{aligned}
$$

kde podobně jako v 3.6 platí

$$\delta_j = -\sum_{l\in N_K} \delta_l w_{jl}\lambda y_j(1-y_j) = -\frac{\partial E_T}{\partial \xi_j} \qquad (3.8)$$

pro $j \in N_{K-1}$.

Podobně pro práh $b_j, j \in N \setminus I$ můžeme odvodit

$$\frac{\partial E_T}{\partial b_j} = \delta_j \qquad (3.9)$$

Všimněme si, že všechna δ_j pro $j \in O$ je možné vypočítat v čase úměrném počtu neuronů ve výstupní vrstvě. Při výpočtu δ_j pro skryté neurony j násobíme váhami všech synapsí jdoucích z neuronu j a každou vahou násobíme právě jednou. Tedy δ_j pro všechny skryté neurony j jsme schopni vypočítat v čase úměrném počtu vah. Z toho plyne, že jsme schopni v čase lineárním vzhledem k délce zápisu neuronové sítě vypočítat parciální derivaci chybové funkce dle všech vah a prahů sítě.

Algoritmus zpětného šíření chyby neboli backpropagation postupuje tak, že předkládá síti trénovací vzory jeden po druhém. Pro každý vypočítá skutečný výstupní vektor sítě. Rozdíl mezi požadovaným výstupem a skutečným výstupem použije k vypočítání

parciálních derivací chybové funkce dle vah a prahů sítě. Váhy a prahy následně adaptuje tak, aby snížil hodnotu chybové funkce pro daný vzor:

$$w_{ij}(t+1) \leftarrow w_{ij}(t) + \alpha\delta_j y_i \tag{3.10}$$

$$b_j(t+1) \leftarrow b_j(t) - \alpha\delta_j, \tag{3.11}$$

kde α je malá konstanta - parametr učení, který udává velikost změny v každém kroku. Poté je předložen další vzor a cyklus se opakuje.

Algoritmus 1 (Backpropagation)

1. Inicializace všech vah a prahů náhodným malým číslem.

2. Předložme na vstup sítě vstupní vzor $\vec{x}^m$ a postupně vypočtěme výstupy y_i všech neuronů od první skryté vrstvy až k vrstvě výstupní $Net(\vec{x})$ podle vzorce:

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

3. Vypočtěme všechny chybové členy δ_i počínaje výstupní vrstvou až po první skrytou vrstvu dle vzorce

$$\delta_j = -(y_j - d(j))\lambda y_j(1 - y_j)$$

pro výstupní vrstvu a dle vzorce

$$\delta_j = -\sum_{l \in N_K} \delta_l w_{jl} \lambda y_j (1 - y_j)$$

pro ostatní vrstvy.

4. Upravme váhy dle vzorce

$$w_{ij}(t+1) \leftarrow w_{ij}(t) + \alpha\delta_j y_i$$

a prahy dle vzorce

$$b_j(t+1) \leftarrow b_j(t) - \alpha\delta_j$$

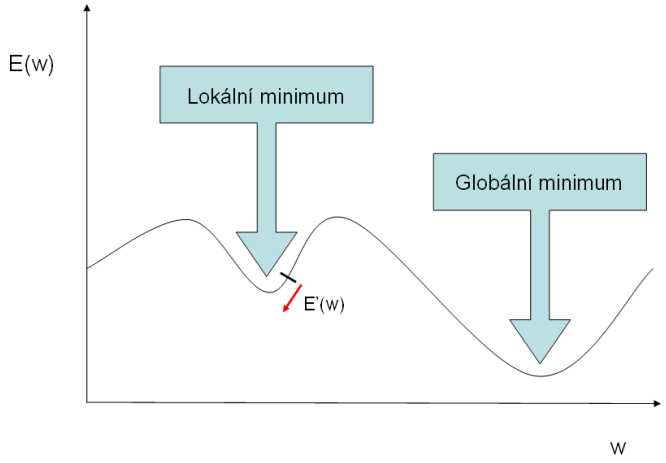
5. Pokud nejsou splněny podmínky ukončení, pokračujeme krokem 2 pro další vstupní vzor.

Podmínka ukončení může být například předem daný počet iterací, které se mají provést. Můžeme též skončit, pokud zjistíme, že chybová funkce se již dále nesnižuje. Poté, co je učení ukončeno, se síť používá k výpočtu výstupů pro nové vstupy. Její podoba se přitom už nemění. Síť se tedy učí offline. Síť zpětného šíření je také statická, tj. pro výpočet výstupu síť nepracuje s předchozími vstupy. Algoritmus má řadu modifikací, které ale přesahují rámec této práce.[2]

[2] Jednou z obvyklých modifikací je přidání takzvaného momentu, což je určitá "setrvačnost" ve změnách vah. Možnost použít tuto modifikaci je obsažena v knihovně Flexinets popisované v příloze A. Více o použití momentu při učení neuronových sítí viz [3].

Algoritmus zpětného šíření sice v obecném případě najít globální minimum chybové funkce, ale často dokáže najít postačující minimum lokální. Poznamenejme zde, že najít takové váhy a prahy sítě, pro které bude chybová funkce ve svém globálním minimu[3], je NP-těžký problém[9].

Gradientní metody a obecně metody, které se snaží síť postupně adaptovat malými úpravami, z nichž každá sama o sobě je zlepšující, (viz též Algoritmy 2 a 3 níže), mohou trpět problémem lokálních extrémů (obrázek 3.3). Problém nastane, pokud nastavíme hodnoty vah a prahů tak, že kvalita sítě bude ve svém lokálním maximu (V případě učení s učitelem to znamená, že chybová funkce bude ve svém lokálním minimu.). Toto maximum ale nemusí být globální a kvalita této sítě nemusí být postačující. Přitom je ale možné, že algoritmus už nenabízí žádnou změnu vah a prahů, která by tuto síť dokázala zdokonalit. To způsobí že pomocí zvolené metody už nedokážeme lepší řešení najít.



Obrázek 3.3: **Problém lokálního minima.** *Pokud bude váha odpovídat vyznačenému bodu, její úpravy proti směru gradientu chybové funkce ji přivedou do minima, které není globální.*

Pro tento problém neznáme žádné univerzální řešení. V některých případech může ale pomoci adaptace velikosti přípustných změn vah a prahů sítě. Toho v případě algoritmu zpětného šíření docílíme zvýšením parametru učení. V této práci se tímto problémem ale nebudeme hlouběji zabývat.

Dalším problémem, se kterým se při používání algoritmu zpětného šíření někdy setkáváme, je tzv. přeučení. Dochází k němu v situacích, kdy neuronovou síť příliš dlouho učíme na nepříliš velké množině trénovacích vzorů, a způsobuje zhoršení schopnosti sítě generalizovat. Tímto problémem a možnostmi, jak jej omezit, se podrobně zaobírá L. Civín v [3].

Příklad: Vraťme se k úloze s předpovědí počasí z předchozího příkladu s použitím statické sítě. Vstupní vrstvu sítě tvoří neurony obsahující informaci o teplotě, rychlosti

[3]viz tzv. loading problem, česky též tréninkový problém nebo problém učení

a směru větru, intenzitě srážek, tlaku vzduchu atd. pro každý ze tří předchozích dnů. Výstupní vrstva bude obsahovat předpověď těchto údajů pro den následující. V množině trénovacích dat by měla být zastoupena co nejrozmanitější počasí (když bylo jasno a teplo, když padaly kroupy, ...) v tomto dni a údaje naměřené v jim předcházejících třech dnech. Poté, co předložíme síti tyto trénovací vzory, můžeme předložit síti na vstup nějaký dosud neznámý vzor. Síť by měla být schopná k tomuto vzoru vydat výstup blízký požadovaným výstupům trénovacích vzorů, které byly podobné. Toto je pochopitelně velmi zjednodušený příklad. Předpovídání chování tak složitého systému, jako je zemská atmosféra, je v praxi velmi komplikované a je otázka, nakolik by se tato predikce sítě blížila skutečnému počasí.

Kapitola 4

Použití neuronových sítí ve zpětnovazebném učení

4.1 Specifika zpětnovazebného učení

Umělé neuronové sítě jsou inspirovány nervovou soustavou biologických organismů, ale způsob práce sítí zpětného šíření a obecně učení s učitelem se od činnosti nervové soustavy organismů v mnohém zásadně odlišuje. Činnost neuronových sítí určených pro zpětnovazebné učení by se v jistém smyslu měla činnosti nervové soustavy organismů více podobat. Takováto neuronová síť by se měla učit online, tj. fáze učení a vybavování by neměla být striktně oddělena. Naopak, síť by se měla učit v celém průběhu své existence.

Pokud bychom zcela trvali na biologické opodstatněnosti zvoleného modelu neuronové sítě, měli bychom požadovat též vlastnost, kterou J. Schmidhuber nazývá časovou a prostorovou lokalitou algoritmu[14]. Časová lokalita je požadavek, aby změny vah a prahů v každém kroku závisely pouze na informacích o síti z určitého pevného časového okna předcházejícího ten který krok algoritmu, kterým se neuronová síť řídí. Časová lokalita úzce souvisí s online způsobem učení. Algoritmus je lokální v prostoru, pokud změna každé váhy a prahu v každém kroku závisí pouze na hodnotách výstupů sousedních neuronů. Splňuje-li algoritmus oba tyto požadavky, nazveme jej lokálním. Příkladem lokálního učícího pravidla je Hebbovo pravidlo. Schmidhuber též navrhuje jeden lokální algoritmus učení v [14]. Zejména požadavek prostorové lokality je ale velmi omezující (nepřipouští například použití parciálních derivací pro úpravy vah), a proto jej nebudeme vždy dodržovat.

Pokud má být síť schopna zvažovat pro rozhodnutí v daném okamžiku i vstupy pozorované dříve a dříve vykonané akce, pak musí síť obsahovat zpětné vazby, které umožní, aby se v ní nějakým způsobem uchovaly hodnoty z předcházejících výpočtů, a musí být dynamická. U rekurentní dynamické sítě se v jednom kroku počítá nová hodnota výstupu každého neuronu pouze z předešlých hodnot bezprostředně sousedících neuronů (na rozdíl od vrstevnatých sítí, kde se nová hodnota vstupu šíří přes všechny vrstvy až na výstup). Z hlediska implementace tak bude pro synchronně aktualizované rekurentní sítě třeba zvlášť uchovávat staré a nově vypočítané hodnoty výstupů neuronů, abychom si nově vypočítanými výsledky nepřepsali hodnoty výstupů neuronů, které ještě budeme potřebovat k výpočtu jiných hodnot. Pokud v rekurentní síti není přímé spojení vstupních

a výstupních neuronů a přitom chceme, aby síť dokázala na nové vstupy okamžitě reagovat, je potřeba pro každý časový krok prostředí počítat více kroků rekurentní sítě.

Pro účely zpětnovazebného učení mohou přicházet v úvahu i další, méně obvyklé modely neuronových sítí (spiking sítě, kompetiční sítě aj.). Techniky klasických vrstevnatých sítí typu zpětného šíření nám nicméně mohou být velmi užitečné. Jsou totiž poměrně jednoduché a jejich vlastnosti jsou dobře prozkoumané, a proto se jimi můžeme inspirovat při stavbě sítí pro zpětnovazebné učení.

V následujícím jsou diskutovány různé modely neuronových sítí použitelných pro zpětnovazebné učení. Úspěšnost těchto modelů byla testována na úloze kořisti a dravce v simulátoru, který jsme speciálně pro tento účel vyvinuli. Agent zde bude vystupovat v roli kořisti, která se musí vyhýbat dravcům a zároveň vyhledávat pro sebe potravu, aby přežila co nejdéle. Úloha je přesněji popsána spolu s experimenty v části 5.

4.2 Síť s náhodnými změnami

Nejprve představíme poměrně naivní model vrstevnaté sítě, jejímiž vstupy jsou senzorické vstupy agenta a jejíž výstupy kódují agentovy akce. Agentova odměna do sítě přímo nevstupuje. Síť je zpočátku náhodně inicializována a učí se za běhu agenta. V každém kroku proběhne drobná náhodná změna vah a prahů. Pokud změna přinese v příštím kroku zvýšení agentovy odměny, tato změna se zafixuje. Síť si tedy musí uchovávat pro každou váhu a práh jejich aktuální hodnotu b_j, w_{ij} a předchozí hodnotu b_j^{old}, w_{ij}^{old}. Dále algoritmus musí uchovávat odměnu r_{old} z předchozího kroku pro porovnání s odměnou v následujícím kroku r.

Algoritmus 2

1. (INICIALIZACE PŘEDCHOZÍ ODMĚNY NULOU A VŠECH VAH A PRAHŮ NÁHODNÝM ČÍSLEM)
   $r^{old} \leftarrow 0$
   $\forall i, j : w_{ij} \leftarrow w_{ij}^{old} \leftarrow random()$
   $\forall j : b_j \leftarrow b_j^{old} \leftarrow random()$

2. načtení nových vstupů z prostředí $\vec{x}$ a odměny r

3. (VYHODNOCENÍ PŘÍNOSU NAPOSLED VYKONANÝCH ÚPRAV VAH A PRAHŮ)
   pokud $r > r_{old}$
   $\quad \forall i, j : w_{ij}^{old} \leftarrow w_{ij}$
   $\quad \forall j : b_j^{old} \leftarrow b_j$
   jinak
   $\quad \forall i, j : w_{ij} \leftarrow w_{ij}^{old}$
   $\quad \forall j : b_j \leftarrow b_j^{old}$

4. (UCHOVÁNÍ ODMĚNY PRO POZDĚJŠÍ POROVNÁNÍ)
   $r^{old} \leftarrow r$

5. (NÁHODNÁ ZMĚNA VŠECH VAH A PRAHŮ)
   $\forall i,j : w_{ij} \leftarrow w_{ij} + random(2MaxChange) - MaxChange$
   $\forall j : b_j \leftarrow b_j + random(2MaxChange) - MaxChange$

6. spočítat výstup sítě $Net(\vec{x})$

7. vykonat akci kódovanou výstupem sítě $Net(\vec{x})$

8. pokud agentova činnost neskončila, přejít na krok 2

Konstanta $MaxChange$ zde označuje v absolutní hodnotě maximální přípustnou změnu vah a prahů. Idea motivující tento algoritmus je, že po změně přinášející "zlepšení" sítě je pravděpodobné, že se zvýší získaná odměna v bezprostředně následujícím kroku. Ponecháním změn zvyšujících okamžitou odměnu by se pak síť mohla v čase zlepšovat. V této základní podobě ale, jak ukazují experimenty provedené v 5.1.3, algoritmus nefunguje.

V iteraci následující po vykonání akce zlepšující schopnost sítě řešit úlohu, je pravděpodobnost vzrůstu získané odměny větší než 50%. Aby však algoritmus fungoval dostatečně dobře, potřebovali bychom, aby platilo jiné tvrzení. Totiž, že vzrostla-li získaná odměna oproti poslední iteraci, potom je pravděpodobnost, že poslední změna přinesla zlepšení sítě, vyšší než 50%. Toto ale zřejmě neplatí, protože zlepšující změny sítě jsou méně pravděpodobné než změny zhoršující, a čím lepší je již dosažené řešení, tím je pravděpodobnější, že náhodná změna síť zhorší.

Naopak, pokud dojde ke změně zlepšující síť, zlepšení nemusí přinést výsledek už v následující iteraci algoritmu. V netriviálních úlohách obvykle dochází k tomu, že důsledky agentovy akce se v podobě odměny nebo trestu projeví až později. Tyto tzv. opožděné odměny[1] se ale tento "naivní" algoritmus nesnaží vysledovat (viz experimenty popsané v části 5.1.3).

Uvedený problém s opožděnými odměnami bychom mohli částečně vyřešit tím, že bychom náhodnou změnu vah a prahů umožnili pouze jednou za n iterací (kroků simulace). Po vykonání n iterací bychom porovnali celkovou odměnu za těchto n iterací r^{last} s odměnou získanou za n předchozích iterací r^{old}. V případě, že r^{last} je vyšší než r^{old}, bychom poslední úpravu vah a prahů ponechali, v opačném případě bychom ji odvolali. Tato modifikace je zahrnuta v algoritmu 3.

Algoritmus 3

1. (INICIALIZACE ČÍTAČE A PŘEDCHOZÍ ODMĚNY NULOU A VŠECH VAH A PRAHŮ NÁHODNÝM ČÍSLEM)
   $r_{old} \leftarrow 0$, $r_{last} \leftarrow 0$
   $count \leftarrow 0$
   $\forall i,j : w_{ij} \leftarrow w_{ij}^{old} \leftarrow random()$
   $\forall j : b_j \leftarrow b_j^{old} \leftarrow random()$

2. načtení nových vstupů z prostředí $\vec{x}$ a odměny r

[1] v angličtině delayed rewards

3. (ITERACE ČÍTAČE A PŘIPSÁNÍ NOVÉ ODMĚNY) $count \leftarrow count + 1$
   $r^{last} \leftarrow r^{last} + r$

4. pokud $count = n$

   (a) (VYHODNOCENÍ PŘÍNOSU NAPOSLED VYKONANÝCH ÚPRAV VAH A PRAHŮ)
   pokud $r^{last} > r^{old}$
   $\forall i, j : w_{ij}^{old} \leftarrow w_{ij}$
   $\forall j : b_j^{old} \leftarrow b_j$
   jinak
   $\forall i, j : w_{ij} \leftarrow w_{ij}^{old}$
   $\forall j : b_j \leftarrow b_j^{old}$

   (b) (VYNULOVÁNÍ ČÍTAČE A RESET PROMĚNNÝCH PRO ODMĚNU)
   $count \leftarrow 0$
   $r^{old} \leftarrow r^{last}$
   $r^{last} \leftarrow 0$

   (c) (NÁHODNÁ ZMĚNA VŠECH VAH A PRAHŮ)
   $\forall i, j : w_{ij} \leftarrow w_{ij} + random(2MaxChange) - MaxChange$
   $\forall j : b_j \leftarrow b_j + random(2MaxChange) - MaxChange$

5. spočítat výstup $Net(\vec{x})$

6. vykonat akci kódovanou výstupem sítě $Net(\vec{x})$

7. pokud agentova činnost neskončila, přejít na krok 2

V mnoha experimentech, které jsme provedli (viz 5.1.3), se nakonec úspěšnost sítě ustálila na určité suboptimální hodnotě. V případě nízké maximální přípustné změny vah může síť uvíznout v lokálním minimu. Ve většině případů se spíš jedná o to, že čím je síť kvalitnější, tím je nižší pravděpodobnost, že zvolená změna ji bude zlepšovat. To vede k ustálení kvality sítě v určité rovnováze.

4.3 Sítě s modelem světa

4.3.1 Modelující sítě

Síť popsaná v předchozí části nemá přístup k hodnotě odměny a nijak explicitně si nemodeluje zákony, jakými se řídí prostředí. Mohlo by ale být užitečné, kdyby agent znal, jaký dopad mají jeho akce na stav prostředí a především, jak závisí získaná odměna na tom, v jaké situaci vykonal kterou akci. Z této znalosti by pak bylo možné odvodit ideální chování agenta vedoucí k dosažení maximální odměny. Zřejmá je analogie tohoto postupu s lidským myšlením. Člověk se nejprve naučí, jak svět funguje, k čemu slouží různé nástroje, atd., a potom využívá této znalosti k dosažení svých cílů.

Agentovu interní reprezentaci znalostí o tom, jak se chová prostředí, nazveme model světa (nebo též model prostředí). Je-li touto reprezentací neuronová síť, nazveme ji modelující[2]. Naproti tomu síti, která na základě vstupů generuje agentovy akce, budeme říkat řídicí[3].

Uvažme nyní velmi zjednodušený hypotetický model neuronové sítě, který budeme postupně zdokonalovat. Předpokládejme v prvním přiblížení, že modelující síť je vrstevnatá síť, jejíž vstupy reprezentují stav prostředí a agentovu akci a jejíž výstupy reprezentují odměnu, ke které tato akce povede. Tato síť uchovává znalost o tom, jakou odměnu přinese vykonání určité akce v určitém stavu prostředí. Máme-li dostatečnou historii pozorování následků různých akcí v různých stavech, můžeme použít k učení takové sítě například algoritmus backpropagation.

Tato pozorování můžeme získat selektivním výběrem dostatečně reprezentativního počtu situací a vyzkoušením různých akcí agenta v těchto situacích. Je možné též nechat agenta automaticky prozkoumat odměny za různé akce v různých situacích tak, že vyčleníme nějaký čas, ve kterém se bude agent chovat náhodně a přitom se učit. Jelikož se ale chceme zaměřit na online učení, budeme se zaobírat především možností učit agenta v celém průběhu jeho činnosti, přičemž se setkáme s již zmíněným dilematem mezi explorací nových možností a exploatací dosavadních znalostí.

Máme-li připravenu modelující síť, můžeme ji zkusit využít pro efektivnější řízení agenta. Nechť se agent nachází ve stavu s. Pro každou proveditelnou akci $a \in A(s)$ můžeme pomocí připravené modelující sítě predikovat odměnu. Pokud by možných akcí byl malý konečný počet, mohli bychom ohodnotit modelující sítí všechny tyto akce a vybrat tu, která přinese největší okamžitou odměnu. Možných akcí bude ale často velký počet, potenciálně nekonečno (například v případě, že agentem je robot, jehož motor se může pohybovat jakoukoli rychlostí v rozmezí 0 - 10 km/hod).

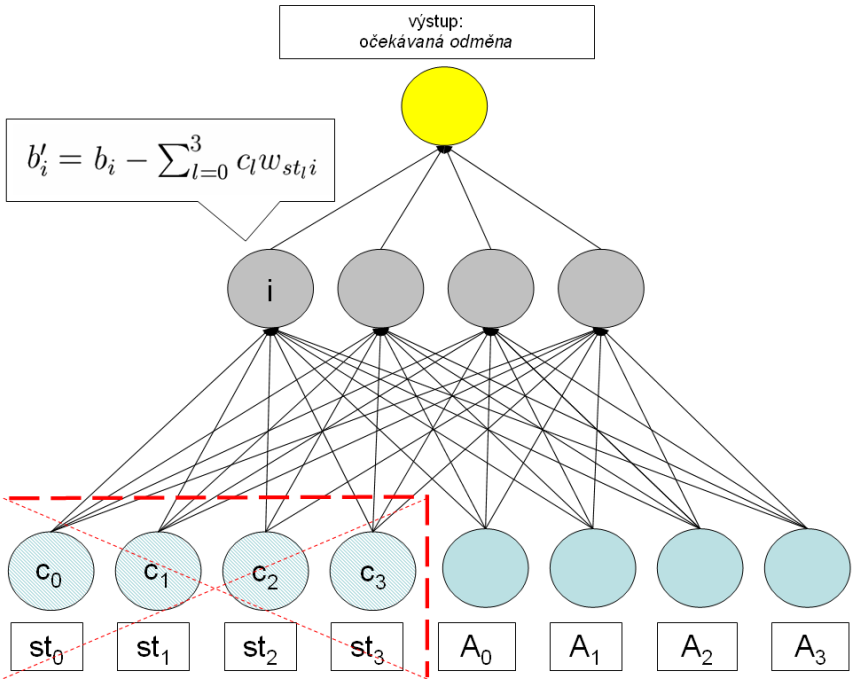
V takovémto případě bychom velmi ocenili algoritmus, který by pro libovolnou vrstevnatou síť s jedním výstupním neuronem dokázal najít vstupní vektor $\vec{x}$, který maximalizuje výstup y výstupního neuronu. Ukažme si, jak by se takovýto algoritmus dal snadno zobecnit, aby pro daný pevně zvolený stav s našel akci a, která povede k získání maximální odměny modelující sítě.

Mějme vrstevnatou neuronovou síť $Net1 = (N, C, I, O, w, b)$. Označme ty její vstupní neurony, které reprezentují stav prostředí, jako $st_0, st_1, \ldots, st_m$. Nechť výstupy těchto neuronů ve stavu s jsou $c_0, c_1, \ldots, c_m$. Vstupní neurony sítě $Net1$, které reprezentují agentovu akci, označíme jako $A_0, A_1, \ldots, A_n$. Definujme novou neuronovou síť $Net2 \stackrel{def.}{=} Net1_{c_0,c_1,\ldots,c_m} = (N', C', I', O', w', b')$ (viz obrázek 4.1), kde

- $N' = N \setminus \{st_0, st_1, \ldots, st_m\}$
- $C' = C \cap (N' \times N')$, kde $\times$ značí kartézský součin množin
- $I' = I \setminus \{st_0, st_1, \ldots, st_m\}$
- $O' = O$

[2]anglicky model network
[3]anglicky control network

- $w'_{ij} = w_{ij}$ pro $(i,j) \in C'$
- $b'_i = \begin{cases} b_i - \sum_{l=0}^{m} c_l w_{st_l i} & : \quad i \in N_1 \\ b_i & : \quad i \in N_2, \ldots, N_{k-1} \end{cases}$



Obrázek 4.1: **Síť pro pevný stav prostředí.** *Neurony reprezentující stav prostředí odstraníme tak, že jejich pevné příspěvky k potenciálu neuronů první skryté vrstvy zahrneme do prahů těchto neuronů. Obrázek znázorňuje síť s jednou skrytou vrstvou a čtyřmi vstupními neurony obou typů.*

Z neuronové sítě $Net1$ jsme odstranili ty neurony, které kódovaly stav prostředí, a prahy neuronů ve vrstvě nad vstupní vrstvou jsme upravili tak, aby obsahovaly zafixovaný stav prostředí s. Všimněme si, že algoritmicky jsme pro danou síť $Net1$ schopni sestrojit $Net2$ v lineárním čase vzhledem k $|N| + |C|$[4].

Předpokládejme, že jsme použili algoritmus, který našel takový vstupní vektor $\vec{x}^2 = (y_{A_0}, y_{A_1}, \ldots, y_{A_n})$ sítě $Net2$, který maximalizuje výstup $Net2(\vec{x}^2)$. Aktivujme vstupní neurony $(st_0, st_1, \ldots, st_m, A_0, A_1, \ldots, A_n)$ sítě $Net1$ vstupním vektorem $\vec{x}^1 = (c_0, c_1, \ldots, c_m, y_{A_0}, y_{A_1}, \ldots, y_{A_n})$. Chceme dokázat, že hodnota výstupu $Net1(\vec{x}^1)$ bude maximální pro daný stav prostředí s. K tomu nám postačí, když pro libovolný vstupní vektor $\vec{x} = (x_0, x_1, \ldots, x_n)$ sítě $Net2$ bude platit:

$$Net1(c_0, c_1, \ldots, c_m, x_0, x_1, \ldots, x_n) = Net1_{c_0, c_1, \ldots, c_m}(x_0, x_1, \ldots, x_n) \tag{4.1}$$

Předložme tyto dva vektory na vstup sítě $Net1$ respektive $Net2$. Pro libovolný neuron $i \in N_1$ platí:

$$y_i = \sigma\left(\left(\sum_{l=0}^{n} x_l w_{A_l i}\right) + \left(\sum_{l=0}^{m} c_l w_{st_l i}\right) - b_i\right) = \sigma\left(\left(\sum_{l=0}^{n} x_l w_{A_l i}\right) - b'_i\right) = y'_i \tag{4.2}$$

[4]Zde předpokládáme, že hodnoty, se kterými počítáme, nejsou libovolná reálná čísla, ale čísla s pevně omezeným počtem desetinných míst, která lze reprezentovat Turingovým strojem.

Tedy výstupy všech neuronů první (skryté) vrstvy sítí $Net1$ a $Net2$ se rovnají. Výstupy neuronů každé následující vrstvy závisí jen na výstupech neuronů předchozí vrstvy, vahách synapsí a prazích neuronů. Tedy i výstupy obou sítí jsou shodné.

Ukázali jsme tedy, že problém nalezení agentovy akce, která v daném stavu prostředí povede k nejvyšší očekávané okamžité odměně, lze v lineárním čase převést na problém nalezení globálního maxima vrstevnaté neuronové sítě. Jelikož druhý problém je speciální případ toho prvního, funguje převoditelnost i v opačném směru. Oba problémy mají tedy asymptoticky stejnou složitost, neboť zřejmě nemůže existovat algoritmus, který by řešil tyto problémy rychleji než v lineárním čase, a složitost převedení úloh tak můžeme zanedbat. Jak ale plyne z dalšího, pro řešení těchto problémů naneštěstí není znám efektivní algoritmus.

4.3.2 Zvolit akci je těžké

Abychom mohli úlohu formálně definovat pro Turingův stroj, omezíme se v této části pouze na práci s celými čisly. Hodnoty všech vah a prahů budou celá čísla, povolíme pouze vstupy sítě z množiny $\{0, 1\}$ a budeme používat přenosovou funkci definovanou takto:

$$f(\xi) = \begin{cases} 1 & : \quad \xi \geq 0 \\ 0 & : \quad \xi < 0 \end{cases} \tag{4.3}$$

Takovouto síť budeme nazývat sítí prahových hodnot. Položme si otázku, zda existuje algoritmus, který by pro neuronovou síť s jedním výstupním neuronem dokázal najít vstupní vektor $\vec{x}$, který maximalizuje výstup y tohoto neuronu. V případě sítě prahových hodnot je tato úloha ekvivalentní nalezení takového vstupního vektoru, pro který bude výstup sítě roven jedné, pokud takovýto vstupní vektor existuje. Nejprve zformulujme otázku jako rozhodovací problém.

Problém maximálního výstupu:

Instance: Celočíselná vrstevnatá neuronová síť Net s jedním výstupním neuronem, dvěma skrytými vrstvami a n vstupními neurony
Otázka: Existuje vstupní vektor $\vec{x} = (x_0, x_1, \ldots, x_{n-1})$ takový, že $x_j \in \{0, 1\}$ pro $0 \leq j < n$ a $Net(\vec{x}) = 1$?

Tento rozhodovací problém je z hlediska řešitelnosti v polynomiálním čase nejvýše tak těžký, jako zjistit, jak vypadá dotyčný vstupní vektor. Známe - li totiž vektor, ve kterém výstup sítě nabývá svého maxima, dokážeme pro něj v polynomiálním čase vypočíst hodnotu výstupu sítě a porovnat ji s číslem 1.

Věta 1 *Problém maximálního výstupu je NP-úplný.*

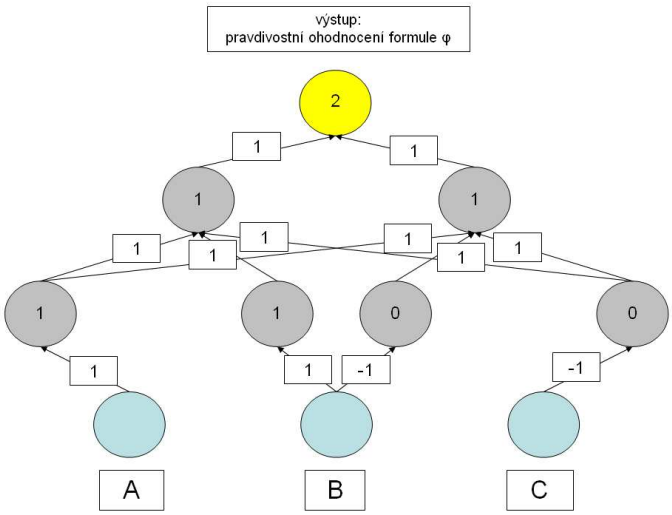
Důkaz: Nejprve dokážeme, že problém je NP-těžký a potom, ze patří do množiny NP. Dokážeme, že problém 3-SAT lze v polynomiálním čase převést na problém maximálního výstupu. Protože o tomto problému je již dokázáno, že je NP-těžký, dokážeme tímto převodem, že i problém maximálního výstupu je NP-těžký. Připomeňme, že problém 3-SAT odpovídá na otázku, zda libovolná daná formule výrokové logiky φ v konjunktivní

normální formě, která se skládá z klausulí o právě třech literálech, je splnitelná, tj. zda existuje pravdivostní ohodnocení jejích výrokových proměnných ϵ takové, že $\epsilon \models \varphi$.

Mějme formuli $\varphi \equiv (A_{11} \vee A_{12} \vee A_{13}) \wedge (A_{21} \vee A_{22} \vee A_{23}) \wedge \ldots \wedge (A_{v1} \vee A_{v2} \vee A_{v3})$ v konjunktivní normální formě, kde všechna $A_{\bullet\bullet}$ jsou literály, tj. jsou ve tvaru X nebo $\neg X$, kde X označuje výrokovou proměnnou. Klausulemi nazýváme disjunkce literálů $(A_{\bullet 0} \vee A_{\bullet 1} \vee A_{\bullet 2})$.

Zkonstruujme celočíslenou vrstevnatou neuronovou síť $Net = (N, C, I, O, w, b, F)$ (obrázek 4.2), tak, že vstupní vrstva bude reprezentovat výrokové proměnné, první skrytá vrstva literály, druhá skrytá vrstva klausule a výstupní neuron celou formuli φ:

- $N = I \cup N_1 \cup N_2 \cup O$ (dvě skryté vrstvy)
- $I = \{X |\ X$ je výroková proměnná obsažená v $\varphi\}$
- $N_1 = \{Y_L |\ L$ je literál obsažený ve $\varphi, L_i \not\equiv L_j \rightarrow Y_{L_i} \neq Y_{L_j}\}$ - Zde z technických důvodů literály samotné neztotožňujeme s neurony, neboť by tím neurony pro pozitivní literály splynuly s neurony pro výrokové proměnné z I. Budeme říkat, že neuron Y_L odpovídá literálu L.
- $N_2 = \{Z |\ Z$ je klausule obsažená ve $\varphi\}$
- $O = \{o\}$ (pouze jeden výstupní neuron)
- $C = (I \times N_1) \cup (N_1 \times N_2) \cup (N_2 \times O)$
- $w_{ij} = 0$ pro $i \in I, j \in N_1$ pokud literál odpovídající neuronu j neobsahuje výrokovou proměnnou i.
- $w_{iY_i} = 1$ pro $i \in I, Y_i \in N_1$ - synapse mezi neuronem pro pozitivní literál a v něm obsaženou výrokovou proměnnou
- $w_{iY_j} = -1$ pro $j \equiv \neg i, i \in I, Y_j \in N_1$ - synapse mezi neuronem pro negativní literál a v něm obsaženou výrokovou proměnnou
- $w_{ij} = 1$ pro $i \in N_1, j \in N_2$ pokud je literál příslušející i obsažen v klausuli j.
- $w_{ij} = 0$ pro $i \in N_1, j \in N_2$ pokud literál příslušející i není obsažen v klausuli j.
- $w_{io} = 1$ pro $i \in N_2$.
- $b_i = 1$ pro $i \in N_1$, pokud i přísluší pozitivnímu literálu.
- $b_i = 0$ pro $i \in N_1$, pokud i přísluší negativnímu literálu.
- $b_i = 1$ pro $i \in N_2$.
- $b_o = v$



Obrázek 4.2: **Příklad sítě zkonstruované popsaným postupem pro formuli** $\varphi \equiv (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee \neg C)$. Synapse s nulovou váhou pro přehlednost nejsou zakresleny.

Mějme ohodnocení ϵ výrokových proměnných formule φ. Vstupní vektor $\vec{x}(\epsilon)$ vytvoříme přímočaře tak, aby přiřazoval vstupním neuronům $i \in I$ sítě Net hodnotu 1, pokud $\epsilon(i) \equiv TRUE$ a hodnotu 0 pokud $\epsilon(i) \equiv FALSE$.

Pro neuron $i \in N_1$ odpovídající pozitivnímu literálu j, který je totožný s neuronem - výrokovou proměnnou j platí:

$$y_i = 1 \Leftrightarrow \sum_{k \in I} y_k w_{ki} \geq b_i \Leftrightarrow y_j = 1 \Leftrightarrow \epsilon(i) \equiv TRUE \tag{4.4}$$

Pro neuron $i \in N_1$ odpovídající negativnímu literálu $\neg j$, kde $j \in I$ platí:

$$y_i = 1 \Leftrightarrow \sum_{k \in I} y_k w_{ki} \geq b_i \Leftrightarrow -(y_j) \geq 0 \Leftrightarrow y_j = 0 \Leftrightarrow \epsilon(i) \equiv FALSE \tag{4.5}$$

Tedy výstupy neuronů v první skryté vrstvě odpovídají pravdivostní hodnotě odpovídajících literálů.

Pro neuron $i \in N_2, i \equiv A \vee B \vee C$, kde A, B, C jsou literály, kterým odpovídají neurony $j, k, l \in N_1$ platí:

$$y_i = 1 \Leftrightarrow \sum_{m \in N_1} y_m w_{mi} \geq b_i \Leftrightarrow y_j + y_k + y_l \geq 1 \tag{4.6}$$

Tedy neurony pro klausule mají jedničkové výstupy, právě když aspoň jeden z jejich literálů je při ohodnocení ϵ splněn. Tedy výstupy neuronů v druhé skryté vrstvě odpovídají pravdivostní hodnotě klausulí.

Pro výstupní neuron o platí:

$$y_o = 1 \Leftrightarrow \sum_{i \in N_2} y_i w_{io} \geq b_i \Leftrightarrow \sum_{i \in N_2} y_i \geq v \;\Leftrightarrow \sum_{i \in N_2} y_i \geq |N_2| \tag{4.7}$$

Tedy výstup sítě neuronu y_o je roven jedné, právě když jsou všechny klausule splněny. Platí tedy $\epsilon \models \varphi$ právě když $Net(\vec{x}(\epsilon)) = 1$. Tedy existuje splňující ohodnocení formule φ, právě když existuje vstupní vektor $\vec{x} \in \{0,1\}^{|I|}$, pro který $Net(\vec{x}) = 1$. Převedli jsme problém 3-SAT v polynomiálním čase na problém maximálního výstupu. Tedy problém maximálního výstupu je NP-těžký.

Zbývá dokázat, že problém maximálního výstupu patří do množiny NP, tedy že je v polynomiálním čase řešitelný pomocí nedeterministického Turingova stroje. Turingův stroj vybere kandidáta $\vec{x}$ na maximalizující vstup sítě v lineárním čase vzhledem k počtu vstupů sítě. K ověření, zda $Net(\vec{x}) = 1$, stačí odsimulovat jeden výpočet neuronové sítě, což lze v lineárním čase vzhledem k počtu synapsí sítě. Celý problém tedy lze vyřešit nedeterministickým Turingovým strojem v lineárním čase vzhledem k délce instance úlohy.

□

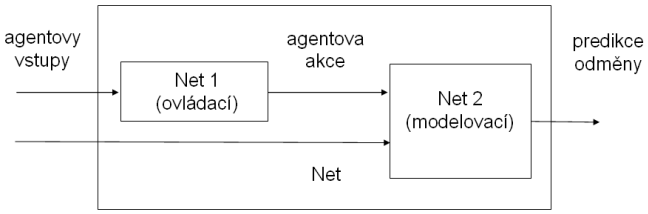
Víme, že problém maximálního výstupu je NP-úplná úloha pro případ sítě prahových hodnot. V praxi ale obvykle používáme neuronovou síť se sigmoidální přenosovou funkcí, která pracuje s vybranou reprezentací pseudoreálných čísel. Můžeme používat například čísla s pevným počtem desetinných míst nebo čísla s plovoucí desetinnou čárkou a pevnou délkou mantisy a exponentu. Přenosová funkce pak není přesně rovna sigmoidě, ale nějaké její aproximaci. Na takovouto síť se výše uvedený důkaz nevztahuje, lze ale předpokládat, že i pro tuto síť bude v obecném případě nalezení vstupu, který maximalizuje výstup sítě, velmi težké.

4.3.3 Backpropagation pro řízení

Vraťme se zpět k popsané hypotetické modelující síti. V obecném případě nejsme schopni pro určitý stav prostředí efektivně najít agentovu akci, která přinese nejvyšší okamžitou odměnu. Můžeme ale s pomocí algoritmu zpětného šíření spočítat parciální derivace výstupního neuronu podle vstupů celé sítě. To můžeme použít k tomu, abychom vylepšovali zvolenou agentovu akci, dokud nebude modelující síť v lokálním maximu vzhledem k této akci.

Podobným směrem se vydal ve své práci Paul Munro [11]. Jeho ideou je použít dvě neuronové sítě, $Net1$ a $Net2$ (obrázek 4.3). Tyto sítě propojíme do jedné sítě Net tak, že vstupy sítě $Net1$ budou vstupy celé sítě Net, výstupy sítě $Net1$ tvoří vstupy sítě $Net2$ a výstupy sítě $Net2$ jsou výstupy celé sítě Net. Zatímco váhy a prahy sítě $Net2$ jsou pevně dány, síť $Net1$ můžeme učit vytvářet vhodné vstupy pro $Net2$ tak, aby celá síť Net dávala požadované výstupy.

Za $Net2$ můžeme vzít modelující síť naučenou klasickým algoritmem zpětného šíření, která předvídá okamžitou odměnu v závislosti na agentově akci (a stavu prostředí). Síť $Net1$ je v našem případě řídicí síť - popisuje agentovo chování. Můžeme ji učit pomocí upravené gradientní metody. K tomu je zapotřebí znát hodnoty parciálních derivací



Obrázek 4.3: **Schéma Munroovy sítě.** *modelující síť Net2 může být učena algoritmem zpětného šíření a řídicí síť Net1 ve směru gradientu odměny predikované sítí Net2.*

výstupu sítě Net (očekávané odměny) podle vah a prahů sítě $Net1$. Pomocí algoritmu zpětného šíření můžeme vypočíst parciální derivace výstupu Net podle vstupů $Net2$. Tyto parciální derivace pak můžeme propagovat zpět do sítě $Net1$, abychom získali parciální derivace výstupu Net podle vah a prahů $Net1$. V případě, když Net má jediný výstup představující očekávanou odměnu, budeme upravovat všechny váhy a prahy sítě $Net1$ ve směru gradientu výstupu podle těchto vah, čímž budeme zvyšovat odměnu dosaženou v následujícím kroku.

Takto můžeme v principu naučit agenta požadovanému chování, aniž bychom k tomu potřebovali jakoukoli informaci od vnějšího učitele. Celá síť Net pak bude reprezentovat chování soustavy agent - prostředí. Autor toto demonstroval na jednoduchém příkladu. Nejprve byla síť $Net2$ naučena počítat funkci sinus. Poté byla úspěšně učena síť $Net1$ tak, aby celá síť Net počítala funkci cosinus. Síť $Net1$ se tak musela naučit transformaci, která pro každý vstup x vypočítá takový výstup y, že $\sin(y) = \cos(x)$.

Bohužel má tento přístup řadu problémů:

1. Použijeme-li k učení sítí $Net1$ a $Net2$ algoritmus zpětného šíření, budeme narážet na problém s lokálními extrémy, který má každá gradientní metoda.

2. Pokud jsou obě sítě, modelující i řídicí, sítě vrstevnaté, jak je popsáno výše, agent nemá možnost si cokoli pamatovat. To může znemožnit algoritmu vyvinout účinnou strategii, pokud se agent pohybuje v nemarkovském prostředí.

3. Výše používaná modelující síť predikuje pouze okamžitou odměnu, tj. odměnu, kterou agent získá v bezprostředně následujícím kroku. Pokud se agent chová tak, aby maximalizoval okamžitou odměnu, nedokáže získávat opožděné odměny a jeho strategie může být velmi neúčinná.

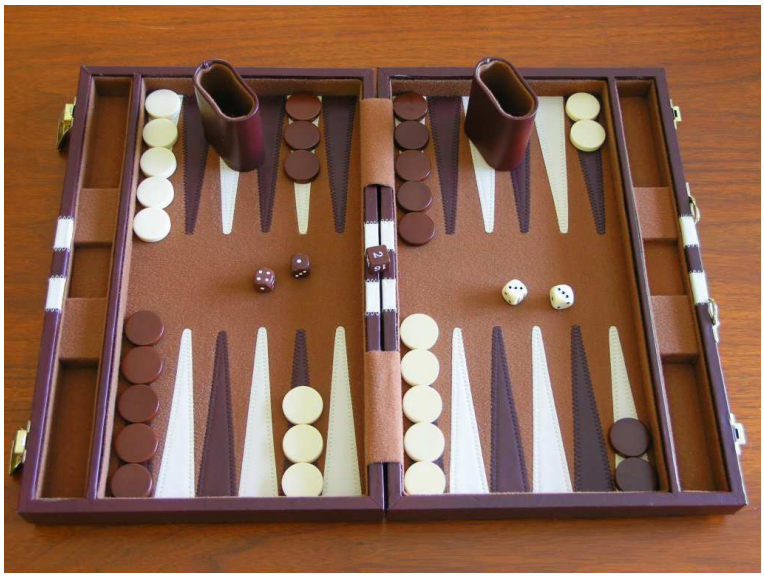
Problému číslo 1 se v této práci věnovat nebudeme. Problém 2 se pokusíme překonat s pomocí rekurentních sítí později. Ukažme si nyní možnost, jak se lze vypořádat se problémem třetím s pomocí vrstevnaté sítě.

4.4 Algoritmus TD(λ)

Algoritmus pro učení neuronových sítí TD(λ) byl navrhnut R. Suttonem v [18] a proslavil se především díky G. Tesaurovi, který jej s úspěchem použil k vytvoření algoritmu

TD-Gammon pro hru Backgammon (česky též vrhcáby)[19], jehož verze 2.0 se v roce 1992 téměř vyrovnala lidským šampionům. Představme si algoritmus TD(λ) na příkladu programu TD-Gammon.

Backgammon je starobylá desková hra pro 2 hráče (obrázek 4.4). Cílem hráčů je dopravit do úkrytu všechny své kameny dříve, než tak učiní soupeř. Hráči postupují kameny po jednorozměrné dráze směrem proti sobě. Možnosti pohybu jsou dány hodem dvou kostek. Jedná se tedy o hru nedeterministickou. V každém tahu hráč pohybuje dvěma kameny (lze i dvakrát tím stejným), o tolik polí, kolik je na kostkách. Pokud hráč přemístí svůj kámen na pole, kde se nachází právě jeden kámen soupeře, vyhodí ho a soupeřův kámen se vrací na začátek dráhy. Zvítězí-li hráč, aniž by mu soupeř vyhodil jediný kámen, dosáhl tzv. gammon a vyhrává dvojnásobek sázky. Vstoupit na pole, kde má soupeř více než jeden kámen, je zakázáno, díky čemuž ve hře vzniká prostor pro řadu zajímavých strategií. Navíc existuje tzv. násobící kostka (doubling cube), která umožňuje hráči zdvojnásobit sázku hry, pokud myslí, že je jeho výhra pravděpodobná. Zdvojnásobí-li jeden z hráčů sázku, kostka je předána soupeři, který může kdykoli později sázku opět zdvojnásobit a předat kostku prvnímu hráči.



Obrázek 4.4: **Hra Backgammon.** (Obrázek pořízen z www.wikipedia.org, kde lze též nalézt veřejnou licenci k jeho používání.)

Hra Backgammon je z pohledu umělé inteligence poměrně složitá. Počet možných stavů desky je přibližně 10^{20}, což znemožňuje v současné době vyřešení hry prohledáním všech možných cest. Prohledávání na několik tahů dopředu používané s úspěchem např. u šachu zde také není vhodné, neboť náhodnost způsobená používáním kostek velmi zvyšuje počet možných přechodů mezí stavy (tzv. branching factor). Hod dvěma kostkami může mít 21 různých výsledků a na každý z nich připadá v průměru 20 možných tahů. V každém půltahu se tedy hra větví v průměru několiksetkrát (pro srovnání, u šachů je to třicetkrát až čtyřicetkrát). Rozhodneme-li se ke hře přistoupit pomocí paradigmatu zpětnovazebného učení, složitost hry nám znemožní použití algoritmů dynamického programování a Q-učení (Bellmanovo prokletí dimensionality). V naší terminologii, budeme nuceni použít nějakou formu kompaktní reprezentace.

Algoritmus TD(λ), je založen na predikční metodě "Temporal Difference learning". Tento princip, často využívaný pro zpětnovazebné učení, se dá stručně charakterizovat jako postupné zpřesňování modelu pro nějakou predikci pomocí jiné, dokonalejší predikce. Sutton[18] uvádí tento příklad.

Příklad: Předpokládejme, že naším cílem je předpovědět počasí na sobotu a že máme modely, které předpovídají sobotní počasí na základě počasí různých dnů v týdnu. Při klasickém učení s učitelem bychom počkali do soboty a podle skutečného sobotního počasí opravili všechny modely tak, aby lépe predikovaly. Například v pátek už ale máme k dispozici mnohem lepší odhad počasí na sobotu, a tak již můžeme zdokonalit pondělní model podle výsledků pátečního modelu.

Pro zajímavost dodejme, že "Temporal Difference learning" se též těší zájmu neurologů, neboť bylo zjištěno, že "Temporal Difference learning" nápadně připomíná chování některých systémů ve zvířecím mozku[17].

Tesaurův TD-Gammon používá vrstevnatou neuronovou síť (N, C, I, O, w, b) se sigmoidální přenosovou funkcí a gradientní metodu pro její učení. Derivaci výstupu neuronu $l \in N$ podle váhy w_{ij} budeme značit p^l_{ij} a derivaci podle prahu b_i budeme značit p^l_i. Vstupní neurony této sítě kódují polohu všech kamenů na desce a informaci o tom, který hráč je na řadě. Výstupní vrstva má čtyři neurony, které předpovídají čtyři možné výsledky partie - vítězství prvního / druhého hráče a gammon pro prvního / druhého hráče. Síť se tudíž učí počítat funkci, která ohodnotí výhodnost situace na desce pro prvního a druhého hráče.

Síť je zpočátku inicializována náhodně. Postup učení sítě je potom následující: V každém časovém okamžiku t (okamžiky odpovídají tahům obou stran, tj. co půltah, to jeden časový okamžik) je síti předložen vstupní vektor $\vec{x}(t)$ kódující stav hrací desky. Síť pro tento vstup vypočítá (v našem případě čtyřsložkový) výstupní vektor $\vec{y}(t)$. Poté jsou vypočteny parciální derivace výstupních neuronů sítě podle hodnot vah p^l_{ij} a prahů p^l_j pro $l \in O$ a $(i, j) \in C$, respektive $i \in N \setminus I$ pro případ prahů. V každém kroku jsou pak upraveny váhy a prahy sítě tak, aby se predikce výsledku partie v předchozím kroku přiblížila té aktuální:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \sum_{l \in O} (y_l(t+1) - y_l(t)) \sum_{m=0}^{t} \lambda^{t-m} p^l_{ij}(m) \tag{4.8}$$

$$b_j(t+1) = b_j(t) + \alpha \sum_{l \in O} (y_l(t+1) - y_l(t)) \sum_{m=0}^{t} \lambda^{t-m} p^l_j(m), \tag{4.9}$$

kde $\alpha > 0$ je parametr učení, který udává velikost úpravy v každém kroku (podobně jako u algoritmu zpětného šíření). Parametr λ udává, nakolik se chyba predikce propaguje zpět do minulosti. Pro $\lambda = 0$ se chyba propaguje pouze do předchozího okamžiku, zatímco pro $\lambda = 1$ se propaguje do minulosti bez omezení. Hodnoty mezi 0 a 1 pak představují kompromis mezi těmito dvěma extrémy.

Jedinou odměnu respektive trest dostává agent na závěr partie v podobě čtyřsložkového vektoru $\vec{z}$, jehož složky odpovídají čtyřem možným výsledkům zápasu. To je zajištěno tím, že váhy sítě jsou na konci partie upraveny místo podle rovnice 4.8 a

4.9 dle vzorců:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \sum_{l \in O} (z_l - y_l(t)) \sum_{m=0}^{t} \lambda^{t-m} p_{ij}^l(m) \tag{4.10}$$

$$b_j(t+1) = b_j(t) + \alpha \sum_{l \in O} (z_l - y_l(t)) \sum_{m=0}^{t} \lambda^{t-m} p_j^l(m) \tag{4.11}$$

Jak zjistil Sutton[18], pro výpočet $\sum_{m=0}^{t} \lambda^{t-m} p_{ij}^l(m)$ není nutné si v paměti uchovávat celou historii $p_{ij}^l(t)$ pro všechna i, j, l. Je-li totiž hodnota této sumy pro nějaká i, j, l v čase t rovna $\mathcal{S}(t)_{ij}^l$, potom:

$$\begin{aligned} \mathcal{S}(t+1)_{ij}^l &= \sum_{m=0}^{t+1} \lambda^{t+1-m} p_{ij}^l(m) = \\ &= p_{ij}^l(t+1) + \lambda \sum_{m=0}^{t} \lambda^{t-m} p_{ij}^l(m) = \\ &= p_{ij}^l(t+1) + \lambda \mathcal{S}(t)_{ij}^l \end{aligned} \tag{4.12}$$

Podobně lze pro všechny prahy sítě odvodit

$$\mathcal{S}(t+1)_j^l = p_j^l(t+1) + \lambda \mathcal{S}(t)_j^l, \tag{4.13}$$

kde $\mathcal{S}(t)_j^l = \sum_{m=0}^{t} \lambda^{t-m} p_j^l(m)$.

Během učení ovládá stejná neuronová síť oba hráče. V každém půltahu ohodnotí síť všechny možné tahy. Hráč, který je na řadě, pak zvolí ten tah, který pro něj má nejvyšší očekávaný výnos. Program se takto učí hraním sama proti sobě. Jelikož síť samotná nereprezentuje činnost násobící kostky, v průběhu učení s násobící kostkou nepracujeme. Při samotné hře pak využívá program teoretický vzorec, který z odhadovaného výsledku partie, který poskytne neuronová síť, rozhodne, zda se vyplatí zdvojnásobit sázku.

O výsledcích tohoto algoritmu Tesauro píše, že:
"Poměrně překvapivé bylo, že dokonce i při experimentech bez jakýchkoli počátečních znalostí s přímým kódováním stavu hrací desky probíhalo významné množství učení. Během prvních několika tisíc trénovacích her se síť naučila množství základních strategií a taktik, kupříkladu vyhazovat soupeřovy kameny nebo upřednostňovat bezpečné pozice. Po několika tisících trénovacích her se pak objevily komplikovanější koncepty. Pravděpodobně nejpovzbudivější bylo zjištění, že síť měla dobrou schopnost kategorizace v tom smyslu, že pokud jsme zvyšovali velikost sítě a dobu učení, výkonnost sítě se zásadně zvyšovala. Nejvyšší výkonnosti jsme dosáhli se sítí o 40 skrytých neuronech, která byla trénována 200 000 her. Tato síť dosáhla úrovně nadprůměrného hráče přibližně stejné kvality jako program Neurogammon (předchozí autorův program založený na neuronové síti učené s učitelem, který zvítězil v roce 1989 na mezinárodní počítačové olympiádě - pozn. autor). Podrobná analýza vah jdoucích ze vstupních do skrytých neuronů v této sítí odhalila zajímavé vzory pozitivních a negativních vah hrubě odpovídajících tomu, co by znalostní

inženýr nazval příznaky užitečné pro hru. Síť byla tedy schopná automaticky objevovat podstatné elementy, což je jeden z dlouhodobých cílů výzkumu učení v oblasti her již od Samuelova programu pro hraní dámy."

Poté se Tesauro pokusil program ještě zdokonalit tím, že na vstup neuronové sítě přidal ručně vybrané příznaky odpovídající podstatným informacím o stavu hry (verze 1.0). To přineslo zásadní zvýšení výkonnosti, kterým program překonal všechny tehdejší programy pro hraní Backgammonu a stal se důstojným soupeřem lidských šampiónů. Pozdější verze 2.0 byla dále zdokonalena tím, že program prohledával na dva půltahy dopředu a zohledňoval při výběru nejlepšího tahu i soupeřův protitah.

4.4.1 Přínos TD-Gammonu

TD(λ) tedy nedělají problém opožděné odměny, dokonce s časovým odstupem i stovek kroků. Tento model neuronové sítě řeší problém 3 popsaný v části 4.3.3 tak, že modelující síť nepředvídá pouze okamžitou odměnu v příštím kole, ale učí se predikovat celkovou kumulativní odměnu za partii. Dále poznamenejme, že v terminologii zpětnovazebného učení vstupem této neuronové sítě není celý popis stavu $s \in S$, který má agent k dispozici, když se rozhoduje vykonat nějakou akci. Chybí zde totiž informace o hodnotách, které jsou na kostkách. To nám ale nevadí, protože modelující síť nám ohodnocuje "mezistav", který nastane poté, co agent vykoná určitou akci, ale předtím, než se znovu hází kostkami. A výsledek hodu kostkami je zcela náhodný, nezávisí na žádné předchozí události. Díky tomu můžeme predikovat kumulativní odměnu pro každou možnou agentovu akci. Algoritmus umožňuje učení online i offline - učení můžeme v určitém okamžiku přerušit, a nebo se síť může učit celém průběhu svého života.

Použití TD(λ) v Backgammonovém programu TD-Gammon byl bezesporu velký úspěch v oblasti zpětnovazebného učení. Zvažme, čím je úspěch TD-Gammonu způsoben a nakolik je tento přístup aplikovatelný i pro jiné úlohy.

Vlastnosti hry Backgammon způsobují, že je tato hra v mnoha ohledech ideální příležitostí pro aplikaci algoritmu TD(λ) [19].

1. Jedná se o pravděpodobnostní Markovský rozhodovací problém - znalost o aktuálním stavu hrací desky je vším, co agent potřebuje znát ke správnému rozhodování. Proto zde vystačíme s vrstevnatou neuronovou sítí bez zpětných vazeb.

2. Počet aplikovatelných akcí v každém stavu je konečný a poměrně nízký. To umožňuje vypočítat očekávané výnosy pro každý možný tah (ve verzi 2.0 dokonce pro každou kombinaci tah - protitah) a zvolit tu nejlepší. Díky tomu se také obejdeme bez řídicí sítě, která by volila pro každou situaci agentovu akci.

3. Je-li agent-hráč v určitém stavu (tj. kameny jsou v určité poloze na desce a na kostkách jsou určité dvě hodnoty), pak následek každé akce (tj. stav desky po vykonání určitého tahu) je deterministický a je nám předem známý. Agent se tedy nemusí učit, jak se projevují jeho zásahy do prostředí (modelující síť nemusí predikovat, jak agentovy akce mění prostředí), a může ohodnotit výsledek každé své možné akce.

4. Použití kostek do hry vnáší náhodnost, díky které ta stejná strategie může v různých partiích vést do různých situací. Díky tomu dochází od počátku učení automaticky k prohledávání (exploraci) velkého množství stavů. To pomáhá vyhnout se situaci, která může nastat např. pokud se program učí hrát hru Dáma sám proti sobě, a to, že oba soupeři dosáhnou pouze nízkou úroveň a přestanou se zdokonalovat, protože oba v každé partii stále procházejí stejnými stavy.

5. I při velmi špatném způsobu hry nakonec vždy jeden z hráčů zvítězí a získá odměnu. Díky tomu se i počáteční agenti řízení náhodně inicializovanou sítí mohou naučit, jak zvítězit (náhodně se chovající agenti obvykle dohráli v několika tisících tahů, zatímco normální hra trvá v průměru 50-60 tahů). Kdybychom nechali hrát proti sobě dva náhodně se chovající algoritmy například šachy, hra by pravděpodobně stále končila remízou a agenti by se nikdy nemusili naučit, za co lze odměnu získat.

4.4.2 TD(λ) v nemarkovském prostředí

Rádi bychom aplikovali algoritmus TD(λ) i na složitější úlohy zpětnovazebného učení, jako je například úloha kořisti a dravce popsaná v 5.1. Tato úloha už nemá vhodné vlastnosti popsané výše, které má hra Backgammon. Algoritmus budeme tedy muset pro naši úlohu přizpůsobit.

1. Nejedná se o Markovský rozhodovací problém, protože pro předpověď budoucího chování prostředí jsou podstatné informace z minulosti, které agent v daném okamžiku nemá na vstupech (Například pokud zkonzumuje agent všechnu potravu na určitém místě a popojde o jedno pole vpřed, nemá již na vstupu informaci o tom, že se na poli za ním nenachází potrava. Tato informace je přitom relevantní pro predikci stavu, do kterého se může agent v budoucnosti dostat.).

   Možným řešením tohoto problému by bylo přidat rekurentní spoje, díky kterým by síť získala určitou krátkodobou paměť. Rekurentním sítím se budeme věnovat v následující části. V některém případě můžeme tento problém též řešit tak, že bychom agentovi uměle přidali nové vstupy, které by obsahovaly užitečné informace z historie (například zda v posledních pěti krocích spatřil dravce). My jsme se ale rozhodli ponechat vrstevnatou síť se vstupy tak, jak jsou popsány v 5.1 s tím, že lze očekávat nižší úspěšnost algoritmu oproti jeho použití v Markovském rozhodovacím procesu.

2. Jelikož agentovy výstupy v naší úloze jsou spojité (každý ze tří vstupů může být libovolné číslo z intervalu $[0, 1]$), nebude možné ohodnotit všechny možné akce a najít tu, které odpovídá největší očekávaná odměna. Budeme moci pouze heuristicky vybrat nějakou suboptimální akci. První přístup, který zvolíme, bude diskretizovat možné hodnoty vstupů rovnoměrným rozdělením intervalu $[0, 1]$ na množinu l hodnot. Takto například pro $l = 3$ budeme uvažovat pouze vstupy z množiny $\{0, 0.5, 1\}$. Musíme zde ale dát pozor na výpočetní složitost, neboť vyzkoušení všech variací k vstupů, každého z množiny l hodnot bude úměrné l^k.

   Dalším možným řešením je například vygenerovat náhodně 100 možných akcí a poté vykonat tu, pro kterou je očekávaná odměna nejvyšší.

3. V případe naší úlohy agent nezná předem, jaký dopad jeho akce zanechá na prostředí. To je zásadní odlišnost oproti hře Backgammon, která si vyžádá změnu způsobu práce s neuronovou sítí. Vrstevnatá neuronová síť, kterou použijeme, nebude reprezentovat odhad odměny pro daný stav, ale pro kombinaci stav × akce. Funkci, kterou síť počítá, tedy interpretujeme jako odhad kumulativní odměny získané během zbytku životního cyklu agenta, který je v určité situaci a vykoná v ní určitou akci.

4. Náhodnost, vnesená do hry Backgammon házením kostek, v naší úloze chybí, což nám bude působit určité problémy. Do prostředí zasahují nepředvídatelné elementy (například start na náhodném místě, pohyb dravců atd.), tudíž simulace probíhají pokaždé jinak, i když se agenti chovají stále stejně. Přesto je ale tendence algoritmu ke zkoušení nových možností (exploraci) v našem případě nižší než u Backgammonu. Může, jak popíšeme níže, dojít k tomu, že agent se mylně naučí, že nějaká akce je nevýhodná, a proto tuto akci přestane zkoušet. Tím ztratí možnost naučit se, že v jiné situaci by tato akce mohla být přínosnou. Budeme zvažovat více řešení tohoto problému, která se budou snažit vnést do úlohy náhodnost "uměle".

5. Jak jsme uvedli, u Backgammonu je téměř jisté, že po určitém čase náhodného hraní jeden z hráčů zvítězí. Předpokládejme, že bychom ponechali v úloze dravce a kořisti trvání jedné simulace 500 kroků. Přitom bychom agenta odměňovali podobně jako v TD-Gammonu pouze na konci simulace jedničkou, pokud by přežil, a nulou, pokud by umřel před koncem simulace. Narazili bychom na problém, že počáteční náhodně inicializované sítě by řídily agenta velmi neúčinně, a agent by se tak pravděpodobně konce simulace nikdy nedožil. Nemohl by se tak naučit, jaká situace a akce je výhodná, protože vede k přežití, a jaká ne.

   Jednoduchou modifikací by bylo na konci simulace udělit agentovi odměnu úměrnou tomu, jak dlouho žil. Zvažme ale, že od neuronové sítě budeme požadovat, aby v každém okamžiku predikovala celou získanou odměnu za simulaci pouze s využitím informace o tom, jaký signál má momentálně agent na senzorech a pro jakou akci se rozhodl. Pokud bychom ponechali délku jedné simulace 500 kroků, síť by neměla informaci o tom, kolik času ještě zbývá do konce simulace, a nemohla by tak smysluplně předvídat, jaká bude odměna. Když při stejném způsobu odměňování zrušíme časové omezení jedné simulace (simulace potrvá, dokud všechna kořist nezahyne), síť opět nebude mít dost informací k tomu, aby odhadovala odměnu, neboť nebude znát, jak dlouho již agent žil.

   Z toho plyne, že budeme potřebovat agenta odměňovat průběžně. V každém časovém kroku t dostane agent určitou odměnu dle toho, jak je situace příznivá pro jeho přežití. Síť pro daný stav prostředí a zvolenou akci predikuje, kolik odměny takto agent ještě získá, než zahyne. V každém kroku tak budeme síť adaptovat tak, aby předchozí predikce předvídaly predikci v čase t plus okamžitou odměnu získanou v čase t. Vzorce 4.8 a 4.9 nahradíme:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \sum_{l \in O} (y_l(t+1) + r_{t+1} - y_l(t)) \sum_{m=0}^{t} \lambda^{t-m} p_{ij}^{l}(m) \qquad (4.14)$$

$$b_j(t+1) = b_j(t) + \alpha \sum_{l \in O} (y_l(t+1) + r_{t+1} - y_l(t)) \sum_{m=0}^{t} \lambda^{t-m} p_j^l(m), \tag{4.15}$$

kde r_t je okamžitá odměna v čase t. Je možné, že bude nutné tuto okamžitou odměnu podělit nějakým koeficientem, aby velikost celkové získané odměny za simulaci nepřesahovala hodnotu 1. Vyšší hodnotu by síť s výstupy z intervalu $[0, 1]$ nedokázala predikovat.

V okamžiku agentovy smrti je pak okamžitá odměna nulová a predikce do budoucna též, vzorce 4.10 a 4.11 tedy nahradíme:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \sum_{l \in O} (0 - y_l(t)) \sum_{m=0}^{t} \lambda^{t-m} p_{ij}^l(t-m) \tag{4.16}$$

$$b_j(t+1) = b_j(t) + \alpha \sum_{l \in O} (0 - y_l(t)) \sum_{m=0}^{t} \lambda^{t-m} p_j^l(t-m) \tag{4.17}$$

Algoritmus bude tedy pracovat následovně:

Algoritmus 4

1. (NÁHODNÁ INICIALIZACE VAH A PRAHŮ A VYNULOVÁNÍ VŠECH SUM PARCIÁLNÍCH DERIVACÍ A VEKTORU STARÝCH VÝSTUPŮ)
   $\forall i, j : w_{ij} \leftarrow random()$
   $\forall j : b_j \leftarrow random()$
   $\forall i, j, l : \mathcal{S}_{ij}^l \leftarrow 0$
   $\forall j, l : \mathcal{S}_j^l \leftarrow 0$
   $\vec{y}^{old} \leftarrow 0$

2. Načti z prostředí vstup $\vec{s}$ a odměnu r

3. Ohodnoť vybranou podmnožinu množiny možných akcí $A(\vec{s})$

4. $\vec{a} \leftarrow$ akce s nejlepším nalezeným hodnocením

5. (VÝPOČET AKTUÁLNÍ PREDIKCE KUMULATIVNÍ ODMĚNY)
   $\vec{y} \leftarrow Net(\vec{s}, \vec{a})$

6. (VÝPOČET VŠECH SUM PARCIÁLNÍCH DERIVACÍ $\mathcal{S}$)
   $\forall i, j, l : \mathcal{S}_{ij}^l \leftarrow p_{ij}^l + \lambda \mathcal{S}_{ij}^l$
   $\forall j, l : \mathcal{S}_j^l \leftarrow p_j^l + \lambda \mathcal{S}_j^l$

7. (ZPŘESNĚNÍ PŘEDCHOZÍCH PREDIKCÍ SÍTĚ ÚPRAVOU VAH A PRAHŮ)
   $\forall i, j : w_{ij} \leftarrow w_{ij} + \alpha \sum_{l \in O} (y_l + r - y_l^{old}) \mathcal{S}_{ij}^l$
   $\forall j : b_j \leftarrow b_j + \alpha \sum_{l \in O} (y_l + r - y_l^{old}) \mathcal{S}_j^l$

8. (ULOŽENÍ VÝSTUPŮ PRO POZDĚJŠÍ POROVNÁNÍ)
   $\vec{y}^{old} \leftarrow \vec{y}$

9. Vykonej akci $\vec{a}$

10. Pokud agent stále žije, přejdi na 2

11. (PROVEDENÍ ZÁVĚREČNÉ ÚPRAVY VAH A PRAHŮ)
$\forall i,j : w_{ij} \leftarrow w_{ij} + \alpha \sum_{l \in O}(0 - y_l^{old})\mathcal{S}_{ij}^l$
$\forall j : b_j \leftarrow b_j + \alpha \sum_{l \in O}(0 - y_l^{old})\mathcal{S}_j^l$

Některé úlohy (viz například úloha z robotiky popsaná v 5.2) nejsou rozděleny na oddělené časové úseky nebo simulace, jako je tomu u backgammonu nebo v úloze dravce a kořisti popsané v 5.1. Učení pak probíhá po teoreticky neomezenou dobu a v celém průběhu tohoto učení agent získává odměnu. Celková kumulativní odměna jde tedy s časem k nekonečnu. Neupravíme-li vzorce 4.14 a 4.15, budeme po síti požadovat, aby dávala stále vyšší a vyšší výstup. Proto je třeba uvedené vzorce obohatit o konstantu $\gamma \in (0,1)$, jejíž význam je podobný významu parametru γ z části 2.1.

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \sum_{l \in O}(y_l(t+1) \cdot \gamma + r_{t+1} - y_l(t)) \sum_{m=0}^{t} \lambda^{t-m} p_{ij}^l(m) \tag{4.18}$$

$$b_j(t+1) = b_j(t) + \alpha \sum_{l \in O}(y_l(t+1) \cdot \gamma + r_{t+1} - y_l(t)) \sum_{m=0}^{t} \lambda^{t-m} p_j^l(m), \tag{4.19}$$

Výše této konstanty bude udávat, nakolik upřednostňujeme odměnu v blízké budoucnosti oproti odměně v budoucnosti vzdálené. Vzorce 4.16 a 4.17 nebudou v tomto případě vůbec použity. I v tomto případě je nutné zajistit, aby výše okamžité odměny nebyla tak velká, že bychom síť učili predikovat kumulativní odměnu vyšší než 1.

Aby mohl algoritmus 4 najít řešení, kdy agent bude schopen nějakým způsobem diferencovat své akce podle toho, jaké má vstupní informace o prostředí, budeme potřebovat minimálně jednu skrytou vrstvu. Síť bez skryté vrstvy by totiž nedokázala reprezentovat informaci typu "v jedné situaci je výhodná jedna akce a v jiné situaci je výhodná jiná akce". Rozhodující pro agentův výběr akce by byla pouze váha synapse vedoucí od vstupního neuronu, který reprezentuje tuto akci, do neuronu výstupního. Pokud by váha byla kladná, agent by se vždy rozhodoval pro akci, která je kódována hodnotou 1 tohoto neuronu. Naopak v případě, že by tato váha byla záporná, agent by vždy zvolil takovou akci, která je kódována hodnotou 0.

4.4.3 Problém "strachu z neznáma"

Popišme nyní jeden zásadní problém, se kterým se budeme muset vypořádat při této úpravě TD(λ), ale na který narazíme i později při aplikaci rekurentních neuronových sítí pro zpětnovazebné učení.

Předpokládejme, že aplikujeme výše uvedenou modifikaci algoritmu TD(λ) na úlohu dravce a kořisti z 5.1. Použijeme nějaké λ mezi 0 a 1, např. $\lambda = 0.5$. Agent bude v každém kroku svoji akci volit tak, že vyzkouší všech možných $3^3 = 27$ variací pro povolené hodnoty každého vstupu z množiny $\{0, 0.5, 1\}$ vzniklých diskretizací popsanou výše. Vstupy a výstupy jsou kódovány tak, jak je popsáno v příloze 5.1, tedy speciálně nulová hodnota

výstupu 0 značí, že agent zrovna nekonzumuje potravu, a nulová hodnota výstupu 2 značí, že se nepohybuje. Jedničkové hodnoty výstupů mají opačný význam.

Předpokládejme dále, že síť po počáteční inicializaci malými náhodnými hodnotami dává výstup blízký 0.5. Nechť je ale skutečná kumulativní odměna za simulaci podstatně nižší než 0.5. Použitá gradientní metoda pak způsobí, že výstupní neuron začne zvyšovat svůj práh a naopak snižovat vazby na cestách vedoucích od aktivních vstupů do výstupního neuronu. Uvažme nyní vstupní neurony kódující agentovu akci. Pokud cesty vedoucí z nich do výstupního neuronu přispívaly kladně k výstupu tohoto neuronu, potom budou v tomto procesu změněny tak, že budou přispívat záporně. Tak se dostaneme do situace, že modelující síť považuje tyto akce za špatné a agent je přestane vykonávat.

Poté, co je síť učením rámcově nastavena a její predikce se blíží průměrné skutečné odměně, měla by se začít učit diferencovat, které akce ve které situaci ovlivňují odměnu pozitivně a které negativně. Jelikož ale z neuronů pro všechny akce vedou negativní cesty do výstupu, agent tyto akce nevykonává. Modelující síť proto nemá příležitost se naučit, jaký dopad by mělo vykonání těchto akcí. Výstupy odpovídajících vstupních neuronů budou vždy 0, a tím pádem již nikdy během algoritmu nedojde ke změně vah synapsí jdoucích z těchto neuronů (parciální derivace výstupu podle těchto vah bude totiž stále nula).

Toto chování sítě jsme skutečně při uvedených podmínkách pozorovali. Po několika simulacích se agenti zastavili, a nedocházelo tak ke zdokonalování sítě. Agenti přestali konat jakékoli akce a pouze čekali na místě, dokonce aniž by konzumovali potravu. Když jsme ale změnili kódování hodnot tak, že hodnota 1 vstupního neuronů znamenala, že se agent bude pohybovat, agenti se po krátkém čase naučili chovat tak, že neustále běhali, až umřeli hladem.

Pomohlo by, kdybychom použili jiné kódování vstupů, například hodnoty z intervalu $[-1, 1]$? Pak by se nemohlo snadno stát, že by hodnoty vstupů pro akce byly nulové a odpovídající parciální derivace též nulové. Bohužel bychom tímto uvedený problém neodstranili. Uvažme, co se stane, pokud budeme agentovu odměnu násobit vyšším koeficientem tak, aby její hodnota přesahovala 0.5. Analogicky prvnímu případu, síť začne nejprve zvyšovat váhy všech cest od aktivních vstupních neuronů k výstupu. Tím se vytvoří v síti cesty, které budou pro vyšší hodnoty vstupů pro akce zvyšovat hodnoty výstupu. Agent pak bude v každém kroku tyto akce konat a síť již nikdy nebude mít možnost se naučit, k čemu by vedlo, kdyby agent tuto akci nevykonal a vstupní hodnota by byla nulová.

Uvedený problém může nastat vždy, když se zároveň agent učí modelovat svět a přitom se rozhoduje vykonávat svoje akce podle tohoto ještě nehotového modelu. V terminologii sekce 2.4, problém je v tom, že agent věnuje všechnu svoji činnost na exploataci již známého. Naším cílem ale je, aby explorace a exploatace byly v určité rovnováze.

Pokusili jsme se problém vyřešit tak, že jsme agentovo chování do určité míry znáhodnili, aby měl agent vždy možnost pozorovat důsledky svých různých akcí a učit se z nich. V experimentální části srovnáváme výsledky, kterých jsme dosáhli pomocí různých nastavení parametrů, různého způsobu odměňování a také různých metod znáhodnění. Možné metody znáhodnění jsou například:

1. Nejprve agenta učíme, přičemž jeho chování je zcela náhodné. Po určitém čase znáhodnění vypneme a agent se dále rozhoduje vykonávat jen tu akci, která predi-

kuje maximální kumulativní odměnu. Poté lze ještě agenta doučovat bez znáhodnění. Faktorem snižujícím úspěšnost této metody je to, že v první fázi se modelující síť naučí predikovat kumulativní odměnu za předpokladu, že agent se bude do konce života chovat náhodně. Tento model pak používáme k selekci nejlepší akce v situaci, kdy se již agent náhodně nechová. Predikce tak může být zkreslena. V tomto případě se nejedná o online učení.

2. Jednou za několik (např. 10) kroků, nebo s určitou pravděpodobností donutíme agenta vykonat náhodnou akci.

3. Abychom zabránili tomu, že se agent vlivem nehotového modelu bude chovat stále stejně, můžeme zakázat opakování stejné akce mnohokrát v řadě za sebou. Toho jsme docílili pomocí zavedení speciální proměnné actionLimit pro každý z agentových výstupů (akcí) - konzumaci, rotaci a pohyb. Hodnota těchto proměnných se v každém kroku sníží o výstup neuronu, který přísluší odpovídající agentově akci. Poté se zvýší o 0.8. Pokud hodnota actionLimit pro nějakou akci klesne pod 0, agentovi je zakázáno tuto akci vykonat (hodnota výstupu je změněna na 0) a naopak pokud tato hodnota překročí výchozí hodnotu, agent je nucen akci vykonat (odpovídající výstup pak bude roven jedné). Postup nejlépe osvětlí citace kódu napsaného v jazyce C++:

```
int i;
// for all action neurons:
for(i=0; i<netInputs.size()-stateInputs; i++)
{
    // temporary variable for the action neuron's output:
    float f = this->getAction(i);

    // the action neuron's value has been high for too long
    // it's forced to be 0:
    if(f>reserves[i]) f=0;

    if(reserves[i] > 2*actionLimit)
    // the action neuron's value has been low for too long:
    {
        f = 1; // the action is forced to be 1
        reserves[i]= 1;
    }

    // substract the current action neuron's output from the reserve:
    reserves[i]-=f;

    // the reserve is refilling each step by 0.8:
    reserves[i]+=0.8f;

    // alter the action neuron's output:
    setOldInput(i+stateInputs,f);
}
```

4. Dalším možným způsobem znáhodnění je již výše zmíněná možnost v každém kroku pokusně náhodně vygenerovat určitý počet akcí. Vygenerované akce necháme sítí ohodnotit, a poté z nich vybereme tu nejlepší.

Navržená modifikace algoritmu TD(λ), především s metodou znáhodnění 4, se při řešení zadané úlohy dravce a kořisti osvědčila - viz část 5.1.4.

4.5 Použití rekurentní sítě

Doposud uvedené modely neobsahují zpětné vazby a agentovi tak v případě nemarkovského prostředí chybí možnost rozhodovat se na základě vstupů pozorovaných v minulosti. Nejedná se tedy o dynamické sítě. Modelující sítě, které jsme zatím popisovali, měly za cíl pouze predikovat agentovu odměnu. Pokud bychom ale umožnili síti též reprezentovat to, jak se mění stav prostředí (respektive agentova vstupní informace o tomto stavu) v závislosti na agentových akcích či na předchozích stavech prostředí, agent by měl možnost využívat kompletní znalost dynamiky prostředí. To by mu mohlo například umožnit implicitně plánovat užitečné mezicíle.

V této části se budeme zabývat gradientními metodami práce s rekurentními neuronovými sítěmi, přičemž se zaměříme na jeden konkrétní model, a ten zkusíme aplikovat. Použitím algoritmu zpětného šíření v případě rekurentní sítě se zabýval již Rumelhart s Hintonem a Williamsem v [13]. Ukázali, jak lze algoritmus zpětného šíření v této situaci přizpůsobit tak, že rekurentní síť převedeme na síť vrstevnatou, která v každém kroku naroste o jednu vrstvu (tzv. Backpropagation through time). Algoritmus bude mít v tomto případě ale prostorovou složitost úměrnou počtu časových kroků, což je většinou prakticky neúnosné.

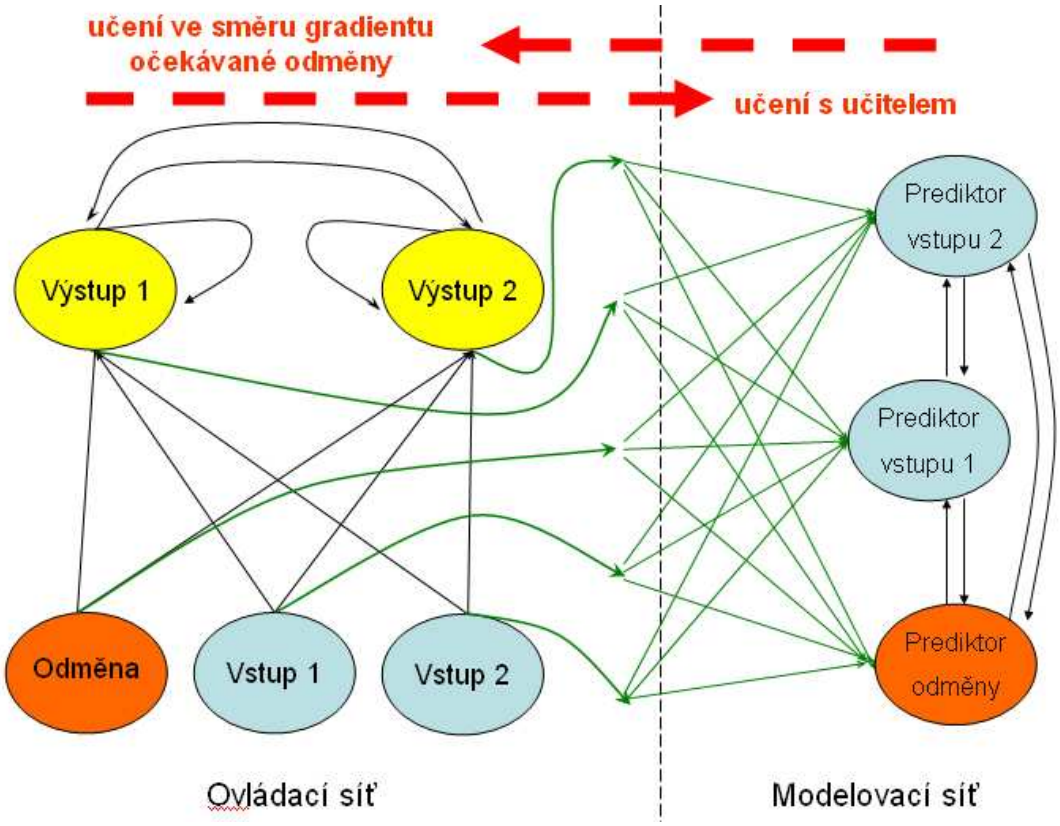
Efektivní algoritmus pro případ rekurentní dynamické neuronové sítě, ovšem pro případ učení s učitelem, navrhuje Robinson a Fallside [12] nebo Williams a Zipser [21]. Mimo jiné z těchto prací vycházel Schmidhuber při návrhu algoritmu pro použití rekurentních sítí v online zpětnovazebném učení v nemarkovském prostředí[15], [16], který v dalším popíšeme.

Schmidhuber, podobně jako Munro, navrhuje neuronovou síť skládající se ze dvou vzájemně propojených částí - řídicí a modelující sítě. Obě sítě jsou ale plně rekurentní (s tou výjimkou, že do vstupů řídicí sítě nevedou žádné synapse). Vstupy řídicí sítě kódují stav prostředí. Mezi nimi je jeden nebo více speciálních neuronů, které kódují hodnotu odměny. Výstupem řídicí sítě je akce, kterou agent v příslušné situaci vykoná. Vstupy modelující sítě jsou všechny vstupní a výstupní neurony řídicí sítě (včetně odměny). Modelující síť obsahuje pro každý agentův vstup jeden neuron, tzv. prediktor, jehož hodnota předpovídá hodnotu odpovídajícího vstupu v příštím okamžiku. Přenosové funkce všech neuronů jsou sigmoidální.

Použitá neuronová síť (obrázek 4.5) tedy vypadá takto:
$Net = (H \cup I \cup M, C, I, O, w, b, F)$, kde

- H je množina všech neuronů řídicí sítě, které nejsou vstupní.

- I je množina vstupů řídicí sítě. Platí $I = I_N \cup I_R$, kde I_N je množina "normálních" vstupních neuronů a I_R je množina vstupních neuronů pro odměnu. I_N a I_R jsou disjunktní.
- M je množina všech neuronů modelující sítě. H, I a M jsou po dvou disjunktní.
- $O \subseteq H$ je množina výstupů řídicí sítě.
- $P \subseteq M$ je množina všech prediktorů. $P_R \subseteq P$ je množina všech prediktorů odměn. Platí $P = \{pred_k \mid k \in I\}$, kde $pred_k$ označuje prediktor neuronu k.
- $C = C_C \cup C_M$, kde $C_C = \{(i,j) \mid i \in H \cup I, j \in H, i \neq j\} \cup (O \times O)$ je množina synapsí vedoucích do řídicí sítě a $C_M = \{(i,j) \mid i,j \in M, i \neq j\} \cup \{(i,j) \mid i \in I \cup O, j \in M\}$ je množina synapsí vedoucích do modelující sítě. Tedy synapse vedou mezi všemi různými nevstupními neurony řídicí sítě, mezi všemi různými neurony modelující sítě a ze všech vstupů a výstupů řídicí sítě do všech nevstupních neuronů. Jedině výstupní neurony řídicí sítě jsou tedy spojeny samy se sebou.
- $\forall i \in N \setminus I \ : \ F(i) = \sigma$



Obrázek 4.5: **Příklad Schmidhuberovy rekurentní sítě se dvěma vstupními a dvěma výstupními neurony a bez skrytých neuronů.**

Oproti předchozím modelům budeme zde pro každou odměnu mít dánu konstantu udávající její požadovanou hodnotu. Cílem algoritmu pak bude minimalizovat hodnotu

$$\sum_t \sum_{i \in I_R} (c_i - y_i(t))^2, \tag{4.20}$$

kde c_i je požadovaná hodnota odměny i a $y_i(t)$ hodnota této odměny v čase t. Pro odměnu, tak jak jsme s ní pracovali doposud, bude hodnota c_i typicky rovna jedné. Můžeme ale např. místo odměny používat trest, jehož požadovaná hodnota bude nulová. Ukážeme si později ještě jeden zvláštní případ odměny - umělou zvědavost.

V algoritmu si pro každý neuron k budeme uchovávat starou hodnotu jeho výstupu $y_{k_{old}}$ a novou hodnotu $y_{k_{new}}$, abychom mohli v rekurentní síti počítat nové výstupy všech neuronů z předchozích výstupů jejich sousedů. Proměnná $p^k_{ij_{new}}$ bude uchovávat odhad parciální derivace výstupu neuronu k podle w_{ij} a proměnná $p^k_{ij_{old}}$ její předcházející hodnotu.

V průběhu algoritmu se zároveň modelující síť učí de facto pomocí učení s učitelem co nejlépe predikovat reakci prostředí na agentovy zásahy. Paralelně s tím se řídicí síť učí chovat tak, aby vyvíjený model predikoval co nejvyšší odměnu. Učení modelující sítě probíhá tak, že adaptujeme váhy proti gradientu chyby predikce. Učení řídicí sítě probíhá (v duchu Munroova přístupu) adaptací vah řídicí sítě proti gradientu $\sum_{i \in I_R}(c_i - pred_i)^2$, tedy chybu budeme propagovat zpět z modelující sítě do řídicí sítě. K tomu je třeba umět v rekurentní sítí vypočítat hodnoty parciálních derivací výstupů neuronů dle vah. V zájmu efektivity algoritmu ale obětujeme přesnost výpočtů a budeme používat vzorec

$$p^k_{ij_{new}} = y_{k_{new}}(1 - y_{k_{new}}) \left(\sum_l w_{lk} p^l_{ij_{old}} + \delta_{jk} y_{i_{old}} \right), \tag{4.21}$$

kde δ je Kroneckerovo delta, $\delta_{ii} = 1$ a $\delta_{ik} = 0$ pro $i \neq k$.

Nepřesnost výpočtu parciálních derivací je způsobena tím, že použité hodnoty $p^l_{ij_{old}}$ jsou počítány pro určité hodnoty vah sítě, které se ale v průběhu algoritmu pozvolna mění. Tato nepřesnost je ale, jak ukázal Williams a Zipser, únosná. Pro jednoduchost je považujeme v popisu algoritmu prahy neuronů za váhy synapse vedoucí z fiktivního vstupního neuronu řídící sítě, jehož výstupem je konstantně -1.

Algoritmus 5 (Schmidhuber)

1. (INICIALIZACE)
   Pro každé $(i,j) \in C$:
   $w_{ij} \leftarrow random$
   Pro každé k: $p^k_{ij_{old}} \leftarrow 0, p^k_{ij_{new}} \leftarrow 0$
   Pro každé $k \in M \cup H : y_{k_{old}} \leftarrow 0, y_{k_{new}} \leftarrow 0$
   Pro každé $k \in I$: načti $y_{k_{old}}$ z prostředí, $y_{k_{new}} \leftarrow 0$

2. (VYBAVENÍ OVLÁDACÍ SÍTĚ)
   Pro každé $i \in H : y_{i_{new}} \leftarrow \sigma(\sum_j w_{ji} y_{j_{old}})$
   Pro každé $(i,j) \in C_C$ a $k \in H$: $p^k_{ij_{new}} = y_{k_{new}}(1 - y_{k_{new}})(\sum_{l \in H \cup I} w_{lk} p^l_{ij_{old}} + \delta_{jk} y_{i_{old}})$
   Pro každé $k \in H$:
   $y_{k_{old}} \leftarrow y_{k_{new}}$
   Pro každé $(i,j) \in C_C : p^k_{ij_{old}} \leftarrow p^k_{ij_{new}}$

3. (INTERAKCE S PROSTŘEDÍM)
   Vykonej všechny agentovy akce dle výstupů neuronů v O.
   Aktualizuj prostředí.
   Pro každé $k \in I$: načti $y_{k_{new}}$ z prostředí.

4. (VYBAVENÍ MODELOVACÍ SÍTĚ)
   Pro každé $i \in M : y_{i_{new}} \leftarrow \sigma(\sum_j w_{ji} y_{j_{old}})$
   Pro každé $(i,j) \in C$ a $k \in M$: $p^k_{ij_{new}} = y_{k_{new}}(1 - y_{k_{new}})(\sum_{l \in M \cup I \cup O} w_{lk} p^l_{ij_{old}} + \delta_{jk} y_{i_{old}})$
   Pro každé $k \in M$:
   $y_{k_{old}} \leftarrow y_{k_{new}}$
   Pro každé $(i,j) \in C : p^k_{ij_{old}} \leftarrow p^k_{ij_{new}}$

5. (UČENÍ MODELOVACÍ A OVLÁDACÍ SÍTĚ)
   Pro každé $(i,j) \in C_M : w_{ij} \leftarrow w_{ij} + \alpha_M \sum_{k \in I} (y_{k_{new}} - y_{pred_{k\ old}}) p^{pred_k}_{ij_{old}}$
   Pro každé $(i,j) \in C_C : w_{ij} \leftarrow w_{ij} + \alpha_C \sum_{k \in I_R} (c_k - y_{k_{new}}) p^{pred_k}_{ij_{old}}$
   Pro každé $k \in I$:
   $y_{k_{old}} \leftarrow y_{k_{new}}$
   $y_{pred_{k\ old}} \leftarrow y_{k_{new}}$
   Pro každé $(i,j) \in C_M : p^{pred_k}_{ij_{old}} \leftarrow 0$
   Pro každé $(i,j) \in C_C : p^k_{ij_{old}} \leftarrow p^{pred_k}_{ij_{old}}$

6. Jdi na 2.

Nejvíce pozornosti zasluhuje blok 5, kde dochází k učení obou sítí. $\alpha_C > 0$ zde označuje parametr učení pro řídicí síť, $\alpha_M > 0$ je parametr učení pro modelující síť. Používáme zde tzv. "teacher forcing" [21], při kterém na konci časového kroku nahrazujeme výstupy prediktorů skutečnými hodnotami predikovaných vstupů, což usnadňuje učení modelující sítě. Nakonec aproximujeme derivace agentových vstupů derivacemi výstupů odpovídajících prediktorů, což opět zkvalitňuje proces učení.

Časová složitost algoritmu je $O(n^4)$, kde n je počet neuronů, kvůli výpočtům parciálních derivací v krocích 2 a 4. Prostorová složitost je pak $O(n^3)$ kvůli nutnosti uchovávat v paměti všechna p^k_{ij}.

Povšimněme si dále, že popsaný algoritmus předpokládá, že chování prostředí lze popsat modelem, který síť vypočítá v jednom kroku. Tedy předpokládáme, že závislost nového stavu prostředí na předchozím stavu lze popsat tzv. lineárně separabilní funkcí (tj. vlastně výpočtem jednovrstvé sítě). Tento předpoklad není v mnoha případech splněn, ale naštěstí v takové situaci stačí algoritmus upravit tak, že bloky 2 (pro zvýšení výpočetní síly řídicí sítě) a 4 (pro zvýšení výpočetní sily modelující sítě) algoritmu opakujeme v každém kroku vícekrát.

Stejně jako model popsaný v 4.4.2, i tato síť může trpět problémem "strachu z neznáma", který autor nazývá "deadlock". Pro jeho řešení zavádí pravděpodobnostní výstupní neurony. Pravděpodobnostní neuron k se skládá z konvenčního neuronu k_μ, který udává střední hodnotu k, a konvenčního neuronu k_σ, který generuje rozptyl. Výstup $y_{k_{new}}$ pravděpodobnostního neuronu je pak dán vzorcem

$$y_{k_{new}} = y_{k\mu_{new}} + y_{k\sigma_{new}}, \tag{4.22}$$

kde z je náhodně rozděleno například podle normálního rozdělení. Odpovídající p^k_{ij} pak musíme počítat dle následující formule:

$$p^k_{ij_{new}} \leftarrow p^{k\mu}_{ij_{new}} + \frac{y_{k_{new}} - y_{k\mu_{new}}}{y_{k\sigma_{new}}} p^{k\sigma}_{ij_{new}} \tag{4.23}$$

My však budeme v experimentech předcházet problému "strachu z neznáma" tím, že s pravděpodobností 5% budeme výstup sítě nahrazovat náhodným výstupem, podobně jako u modifikace algoritmu TD(λ) výše.

Další možností, jak vnést do algoritmu větší explorativní schopnosti a zefektivnit učení modelující sítě, je autorem navrhovaná umělá zvědavost a umělá nuda [16]. Pro zdokonalování modelu je důležité, aby se agent dostával do situací, ve kterých zatím selhávají predikce jeho modelující sítě. Modelující síť pak má možnost se z těchto chyb poučit. Jak autor tvrdí, pro zdokonalení modelu jsou nejzajímavější situace, kde se model pouze "lehce mýlí".

Zvědavost a nudu budeme reprezentovat jedním z odměňovacích neuronů $l \in I_R$, jehož hodnota bude odpovídat v každém okamžiku eukleidovské vzdálenosti predikce vstupů a jejich skutečných hodnot. Jeho požadovaná hodnota c_l bude odpovídat zmíněnému "lehkému omylu". Zvědavost se tak stane jednou z agentových odměn a agent tak bude motivován, aby vyhledával situace, ve kterých se může "něco naučit".

Jaká je schopnost uvedeného algoritmu získávat opožděné odměny? Modelující síť predikuje pouze hodnotu okamžité odměny, i když v závislosti na historii stavů a akcí. V každém kroku pak upravujeme řídicí síť tak, aby série jejich předchozích akcí přinesla v daném kroku co největší odměnu. Chyba řídicí sítě se tedy propaguje do minulosti a síť díky tomu do určité míry umí získávat opožděné odměny. Algoritmus lze ale ještě zdokonalit tak, že by modelující síť predikovala kumulativní odměnu s využitím principu "Temporal Difference learning".

Předpokládejme pro jednoduchost, že pracujeme s obvyklou odměnou, tak jako v předchozím, která nemá požadovanou hodnotu, ale jejíž kumulativní hodnotu chceme maximalizovat. Nechť $k \in I_R$ je vstupní neuron pro odměnu. Chyba prediktoru $pred_k \in P_R$ v čase t bude rovna

$$(y_{pred_k}(t) - \gamma y_{pred_k}(t+1) - y_k(t+1))^2, \tag{4.24}$$

kde $0 < \gamma < 1$ má podobný význam jako v 2.1. Změnu použité chybové funkce provedeme úpravou prvního cyklu v bloku 5 algoritmu. Abychom ale byli schopni tuto chybu počítat, musíme mít k dispozici již predikci z následujícího kroku, a tedy budeme muset adaptaci vah modelující sítě odložit o jeden krok. Požadované hodnotu c_k odměny k nastavíme na dostatečně velké číslo. Modelující síť pak bude predikovat kumulativní odměnu za celý život agenta a řídicí síť bude upravována podle gradientu predikce této kumulativní odměny tak, aby agent tuto kumulativní odměnu maximalizoval.

Schmidhuber popisuje experiment[15], kdy síť byla úspěšně trénována, aby zareagovala výstupem na první výskyt vstupního vzoru B po posledním výskytu vzoru A, kdy mezi nimi mohl být libovolněkrát vložen vzor X. Dále popisuje částečně úspěšný experiment s vyvážením simulované tyče stojící na vozíku. Při tomto experimentu byl algoritmus upraven tak, že všechny parciální derivace byly pravidelně po 8 krocích nulovány. Podobnou úpravu používáme i v našich experimentech s tímto modelem, které jsou popsány v experimentální části.

Kapitola 5

Experimentální část

5.1 Úloha dravce a kořisti

5.1.1 Zadání úlohy

Naším cílem bylo porovnat úspěšnost popsaných modelů neuronových sítí při různých nastaveních jejich parametrů při řešení neprimitivní úlohy zpětnovazebného učení. Pro tento účel byla zvolena varianta simulace kořisti a dravce. Popis použitého simulátoru lze nalézt v příloze B.

V dvourozměrném diskrétním prostoru o velikosti 100×100 se pohybuje určitý počet jedinců kořisti a určitý počet dravců. Agent v roli kořisti řídící se testovaným algoritmem zpětnovazebného učení se snaží co nejdéle přežít. Kořist umírá, pokud je sežrána dravcem nebo pokud zůstane dlouho bez potravy (vyhladoví). V každém kroku simulace ztrácí kořist část své vnitřní zásoby potravy, kterou může doplnit příjmem potravy z prostředí. Potrava pro kořist se zpočátku nachází všude v prostoru a pomalu dorůstá, pokud je sežrána. Pokud se kořist pohybuje, spotřebuje více potravy, než pokud zůstává v klidu. V každém případě ale spotřebovává potravu větší rychlostí, než potrava dorůstá, a proto je nucena se pohybovat. Dravci, řídící se jednoduchým napevno daným algoritmem, se snaží co nejrychleji sežrat kořist. Pro zjednodušení jsme vyloučili možnost, že dravec umře hladem.

Vstupy:
Každý agent (kořist i dravec) má 4 vstupy (senzory), číslované od 0:

Vstup 0 - Hlad:
Informace o tom, jak dlouho ještě agent vydrží bez potravy. Pokud je tato hodnota nulová, agent je zcela sytý. Pokud tato hodnota vystoupí na 1, jedinec vyhladoví a umře.

Vstup 1 - Čich:
Čich napovídá agentovi, kolik je v okolí jedinců jiného druhu (kořisti nebo dravců), než je on sám. Každý jedinec jiného druhu, který je v dosahu čichu, zvyšuje hodnotu tohoto vstupu o příspěvek odpovídající rozdílu svojí vzdálenosti od agenta a dosahu agentova čichu.

Vstup 2 - Zrak:
Zrak napovídá agentovi, zda se nachází nějaký jedinec jiného druhu ve směru před

ním. Hodnota tohoto senzoru odpovídá vzdálenosti agenta od nejbližšího jedince jiného druhu, který se nachází před agentem.

Vstup 3 - Přítomnost potravy:
Tento vstup agentovi říká, kolik je v daném místě potravy pro kořist (Pro dravce tak tato hodnota není příliš důležitá.).

V každém kroku se může agent otočit až o 90° v libovolném směru a poté vykonat jednu ze dvou možných akcí - konzumovat potravu nebo se pohybovat. Agent se pohybuje vždy ve svém aktuálním směru o počet kroků, který je omezen maximální rychlostí příslušnou jeho druhu. Kořist se nemůže pohybovat, pokud je na stejném místě i dravec (je chycena). Pokud se dravec rozhodne konzumovat potravu, zabije tím jeden z kusů kořisti, která se nachází na stejném místě jako on. Pokud zde žádná kořist není, dravcova akce nemá žádný výsledek. Pokud se kořist rozhodne žrát, zkonzumuje všechnu potravu v daném místě a zvýší svojí zásobu úměrně tomu, kolik potravy se v daném místě vyskytovalo.

Výstupy
Agent ovládá kořist prostřednictvím čtyř výstupů.

Výstup 0 - Příjem potravy:
Pokud je hodnota tohoto výstupu větší než hodnota výstupu 2, agent se rozhodl konzumovat potravu.

Výstupy 1 a 2 - Rotace:
Každý jedinec v simulaci je v každém okamžiku otočen v určitém směru. Výstupy 1 a 2 udávají zda a na kterou stranu se agent otočí (změní směr). Výstup 1 udává rotaci vpravo (ve směru hodinových ručiček), výstup 2 rotaci vlevo (proti směru hodinových ručiček). Rozdíl těchto dvou hodnot je vynásoben 90° a o výslednou hodnotu se agent pootočí vpravo nebo vlevo, podle toho, který výstup měl vyšší hodnotu.[1]

Výstup 3 - Pohyb:
Pokud je hodnota tohoto výstupu větší než hodnota výstupu 0, agent se bude pohybovat. V takovém případě se posune ve svém aktuálním směru o vzdálenost rovnou součinu hodnoty výstupu 2 a agentovy maximální rychlosti.

Odměna
V průběhu experimentů jsme porovnávali výsledky jednotlivých algoritmů při třech různých způsobech odměňování.

Odměňovací funkce 1:
V každém kroku života je agentovi přidělena odměna velikosti 1. V okamžiku smrti pak dostává odměnu v hodnotě 0.

Odměňovací funkce 2:
V každém kroku života je agentovi přidělena odměna velikosti 1 mínus agentův hlad. Tuto hodnotu budeme nazývat agentova zásoba potravy *Food*. V okamžiku smrti pak agent dostává odměnu v hodnotě 0.

[1]Tento způsob kódování výstupů se osvědčil mnohem lépe, než použití jediného neuronu pro rotaci, jehož krajní hodnoty by kódovaly maximální rotaci vpravo a vlevo. V takovém případě měla kořist přílišnou tendenci se točit na jednu či druhou stranu, podle toho zda příspěvek neuronu pro rotaci k predikci odměny byl záporný nebo kladný.

Odměňovací funkce 3:

Odměna 3 zavádí obdobu "strachu z nepřítele". Agentova odměna bude pak minimum ze zbývající zásoby potravy a tohoto "strachu":

$$r = \min(Food, Mood), \tag{5.1}$$

kde $Food \in [0, 1]$ je agentova zásobě potravin, a $Mood \in [0, 1]$ je přímo úměrné vzdálenosti nejbližšího dravce, který je v dosahu agentova čichu. $Mood$ bude rovno 1, pokud nebude žádný dravec v dosahu agentova čichu, a bude rovno 0, pokud se bude nacházet dravec na stejném místě jako agent. V okamžiku smrti pak ještě agent dostává odměnu v hodnotě 0.

Úloha má všechny vlastnosti, které potřebujeme k prověření algoritmů. Prostředí je nemarkovské a pravidla, jakými se řídí, je agentům zpočátku neznámé. Agenti nemají žádnou apriorní informaci o správném řešení problému. Agenti musejí umět vyhledávat odměnu opožděnou aspoň o jeden krok, neboť, aby mohli vyhledat potravu, musí se pohybovat, což je nejprve stojí část jejich zásoby potravy, a přinese výnos až po dvou krocích, a to jen pokud v následujícím kroku agent konzumuje nalezenou potravu. Optimální způsob, jak se vyhýbat dravcům, není zřejmý, a vzniká tak prostor pro sofistikované strategie. Agenti se učí online způsobem. Pro jejich úspěšnost bude též významné, kolik aktivity vynaloží na exploraci a kolik na exploataci.

5.1.2 Měření úspěšnosti

Úspěšnost jednotlivých algoritmů budeme měřit dvěma typy testů. Každý test se skládá ze série simulací. V průběhu celého testu jsou sítě učeny. V každé simulaci jsou jedinci rozmístěni na nová náhodná místa v náhodném směru. Simulace končí, pokud je všechna kořist mrtvá, nejdéle však po 500 krocích. Poté jsou jedinci oživeni a začíná další simulace. Každý jedinec kořisti bude mít svou vlastní neuronovou síť, která mezi jednotlivými simulacemi bude zachována (bude si pamatovat, co se naučil).

Při prvním typu testu bude v prostředí 20 jedinců kořisti a 5 dravců. Úspěšnost algoritmu ohodnotíme body tak, že v každém kroku přidáme tolik bodů, kolik je žijících jedinců kořisti. Potom budeme počítat průměrný počet bodů za každých 100 po sobě jdoucích simulací a sledovat vývoj v čase. Celková délka testu se bude v různých případech lišit dle toho, zda se ještě síť zdokonaluje, a v závislosti na časové náročnosti testů. Nicméně srovnávat výsledky různých konfigurací budeme vždy za stejnou dobu testování.

Druhý typ testu bude test bez dravců, který budeme provádět u vybraných nejlepších konfigurací. V tomto typu testu budeme zkoušet, zda se jeden jedinec kořisti dokáže naučit strategii, jak vyjít s jídlem tak, aby přežil do konce simulace. Příkladem takové strategie je pravidelné střídání jednoho kroku, ve kterém se agent bude pohybovat, a jednoho kroku, ve kterém bude konzumovat potravu. Protože naučené řešení nemusí být stabilní, budeme jednak měřit počet simulací, než se agent tuto strategii naučí poprvé, jednak počet simulací nutných k tomu, aby se tuto strategii naučil a dokázal ji udržet ve dvaceti následujících simulacích. Test budeme opakovat při stejném nastavení parametrů a různé náhodné inicializaci desetkrát, pokaždé až 10000 simulací, nebo dokud algoritmus nenajde stabilní řešení.

Každý algoritmus s každou konfigurací testovanou testem prvního typu jsme očíslovali číslem testu, kterým se pak odkazujeme v grafech a tabulkách. Poté, co jsou popsány výsledky všech testů prvního typu, je uvedena jejich shrnující tabulka. Následně je popsán vzhled naučené sítě v nejúspěšnějším z experimentů z daným modelem. Tato konfigurace je pak testována v testu typu 2 bez dravců. Testy tohoto typu jsou značeny velkými písmeny. Nakonec je prezentováno porovnání výhod a nevýhod všech testovaných modelů.

Pro srovnání jsme vytvořili jednoduchý pevný program skládající se z rozhodovacích pravidel.

Algoritmus 6

1. Pokud je dravec za tebou, utíkej
2. Pokud je dravec před tebou, otoč se o 90° u utíkej
3. Pokud je na místě malé množství potravy, popojdi a lehce se pootoč
4. Jinak zůstaň na místě a konzumuj potravu

Test 0: Agenti řídící se tímto programem byli velmi úspěšní a získali v průměru okolo 721000 bodů za 100 simulací. Pokud jsme v simulaci vypnuli dravce, byli tito agenti schopni stabilně přežít celé trvání simulace (500 kroků).

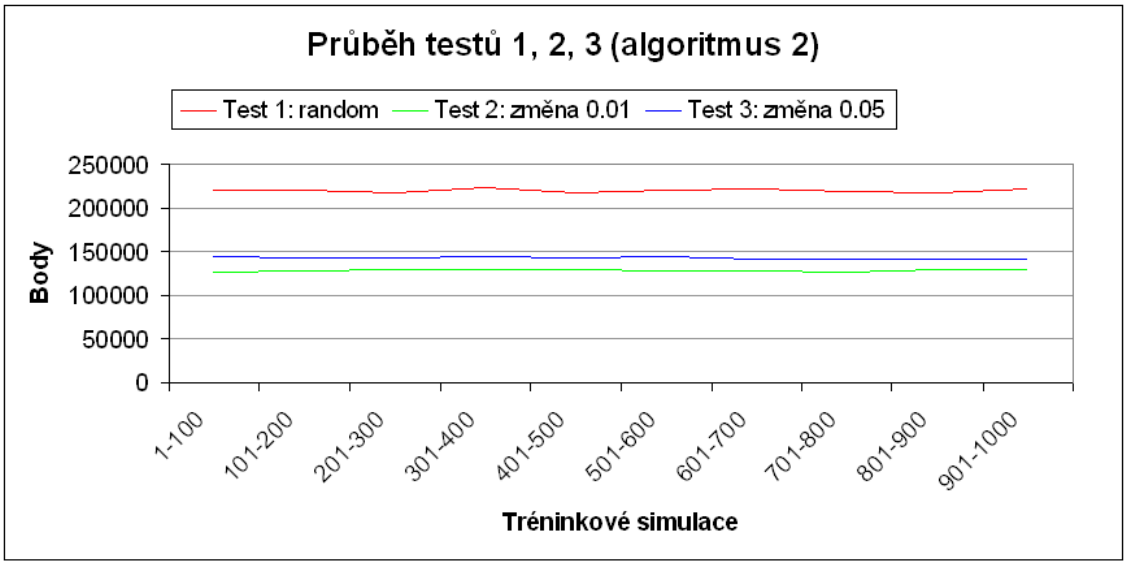
Test 1: Dále jsme změřili úspěšnost náhodně chovajícího se algoritmu, opět pro účel porovnání s následujícími experimenty. Náhodný algoritmus byl též poměrně úspěšný, získával v průměru okolo 220000 bodů za 100 simulací. V testu bez agentů se často povedlo náhodně se chovajícímu agentu přežít celou simulaci, ale nikdy nesplnil kriterium stability. Uvědomme si, že náhodný algoritmus je v řešení této úlohy poměrně úspěšný z toho důvodu, že jedinec rovnoměrně rozděluje čas mezi konzumaci potravy a vyhledávání nové potravy. Navíc pro dravce není snadné chytit náhodně se pohybující kořist. Naproti tomu neuronová síť s neměnnou náhodnou konfigurací by si vedla mnohem hůře, protože by většinou opakovala stále stejnou akci.

5.1.3 Síť s náhodnými změnami

V této části popisujeme experimenty s algoritmy 2 a 3 popsanými v části 4.2. Síť řízená těmito algoritmy byla testována v simulátoru dravce a kořisti. Síť dostávala na vstupu vjemy simulovaného organismu a měla za úkol vykonat takové akce, aby se organismus dokázal vyhnout predátorům a najít potravu. Použili jsme síť bez skrytých neuronů, která měla pouze 4 vstupní a 4 výstupní neurony. Nejprve jsme použili odměňovací funkci 2, tj. úměrnou agentově zásobě potravy.

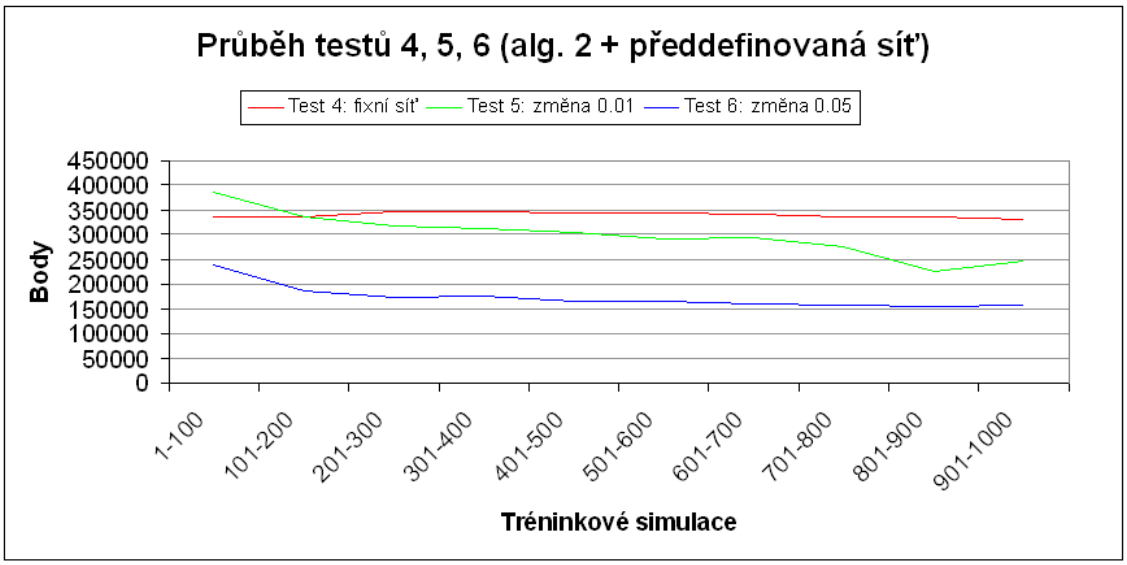
Test 2 a 3 (obrázek 5.1): Nejprve byl testován algoritmus 2 při dvou různých hodnotách omezujících maximální náhodnou změnu váhy. V testu 2 to bylo ± 0.01 a v testu 3 ± 0.05. Sítě se v tomto případě učily velmi málo nebo vůbec. Test 3 byl o něco úspěšnější než test 2. Při ještě vyšší maximální možné změně vah byly výsledky ještě o něco lepší. To je ale pravděpodobně způsobeno jen tím, že při vyšších změnách vah se algoritmus více podobá náhodnému algoritmu z testu 1. Sítě nakonec dospěly do takového

stavu, že se jedinci přestali pohybovat, a pouze konzumovali potravu na místě, dokud jim nedošla. To je příkladem toho, že algoritmus neumí vyhledávat opožděné odměny, neboť dává přednost okamžitému výnosu za konzumaci, před potravou, kterou by musel hledat.



Obrázek 5.1: **Testy 1, 2, 3.**

Testy 4, 5 a 6 (obrázek 5.2): Poté byla místo náhodné inicializace použita ručně přednastavená síť, která již zpočátku dosahovala slušných výsledků. Pokud jsme ji zafixovali a otestovali (test 4), získala v průměru necelých 340000 bodů za 100 simulací. Algoritmus s náhodnými změnami ale ani při této inicializaci nefungoval a síť dosahovala v každé další simulaci stále horších výsledků. Podoba sítě nekonvergovala a spíše se náhodně měnila. A to při nastavení maximálních změn ±0.01 (Test 5) i ±0.05 (Test 6).

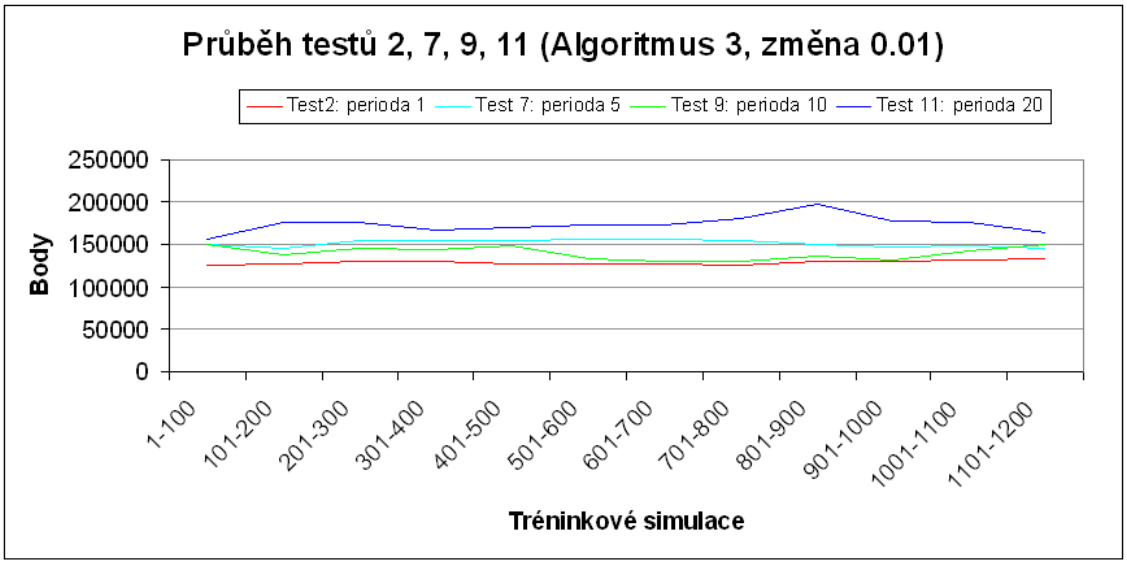


Obrázek 5.2: **Testy 4, 5, 6.**

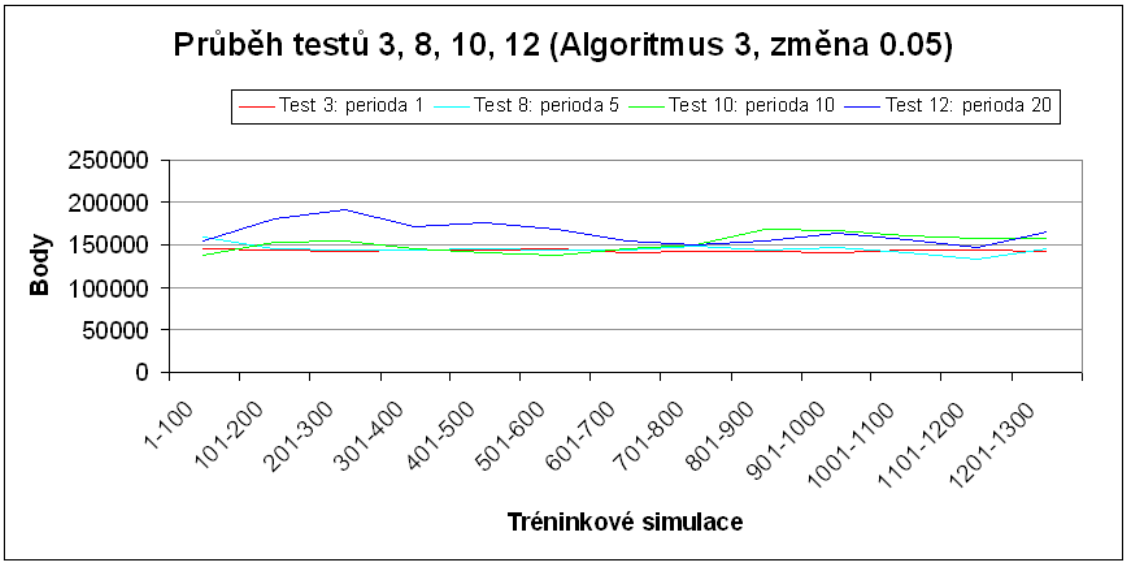
Během testu 5 se v prvních 100 simulacích algoritmu povedlo překonat výsledky původní sítě, síť tak vylepšil a dosáhl necelých 385000 bodů. Jedinci se stali o něco

aktivnějšími a vyhledávali novou potravu i když ještě jejich zásoba nedocházela, na což při výchozím nastavení sítě čekali. V následných simulacích ale úspěšnost již jen klesala a agenti se opět stávali stále pasivnější.

Diskuze nedostatků algoritmu 2 viz 4.2.



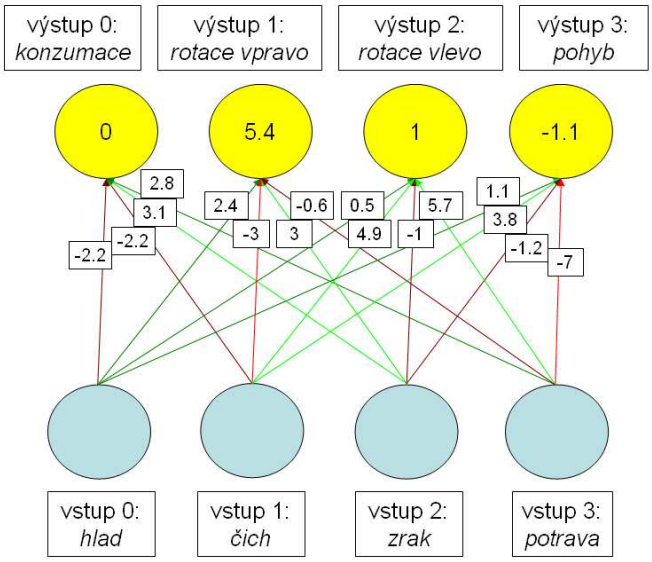
Obrázek 5.3: **Testy 2, 7, 9, 11.**



Obrázek 5.4: **Testy 3, 8, 10, 12.**

Testy 7 - 12 (obrázky 5.3 a 5.4): S algoritmem 2 jsme porovnávali algoritmus 3 pro délku periody změny 5, 10 a 20 kroků. Jelikož je v tomto algoritmu náhodná změna prováděna jen jednou za n kroků (n je délka časové periody), vynásobili jsme v každém experimentu velikost maximální možné změny číslem n, aby rozsah změn zůstal srovnatelný. Použité velikosti maximálních změn tedy byly $\pm(n \times 0.01)$ respektive $\pm(n \times 0.05)$. Toto byly testy v pořadí 7, 9, 11 respektive 8, 10, 12. Algoritmus 3 znatelně překonal algoritmus

2, především v případě periody 20 (obrázek 5.5), a to u obou zkoušených maximálních hodnot přípustných změn. Mezi jedinci se objevili takoví, kteří střídavě konzumovali a vyhledávali potravu, a "inteligentně" reagovali útěkem na přítomnost dravců. A to přestože použitá odměna nezohledňuje blízkost dravce. Projevily se tak účinky negativní odměny, kterou agent dostává při úmrtí. Problémy algoritmu 2 však v nižší míře stále přetrvávají.

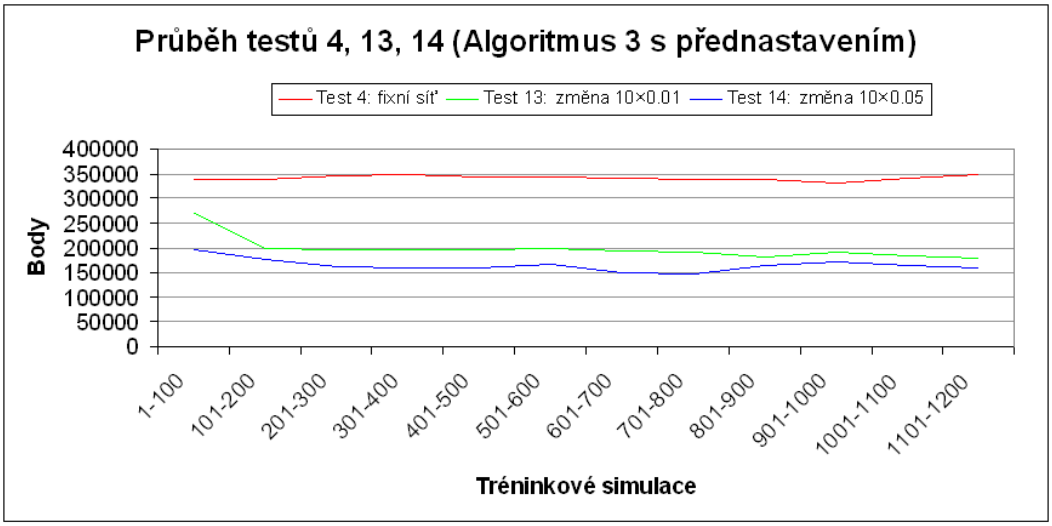


Obrázek 5.5: **Váhy a prahy sítě učené v testu 11.** *Na obrázku jsou váhy a prahy sítě po 900 tréninkových simulacích (zaokrouhlené na jedno desetinné místo). Synapse s kladnými vahami jsou vyznačeny zeleně (synapse s nejvyššími vahami světle zeleně), synapse s negativními vahami naopak červeně (světle červeně pro váhy s nejnižšími hodnotami). Příslušný agent uměl vyhledávat potravu a do jisté míry se vyhýbat dravcům. Váhy zajišťují, že agent se vydá hledat potravu, pouze když má hlad a zkonzumoval potravu na místě, kde se nachází. Pokud agent vidí dravce, neutíká, ale otočí se na místě. Tím se dostane do situace, kdy dravce cítí, ale nevidí. V této situaci pak utíká. Agent má tendenci se točit vlevo. I při útěku před dravcem agent pravidelně konzumuje potravu.*

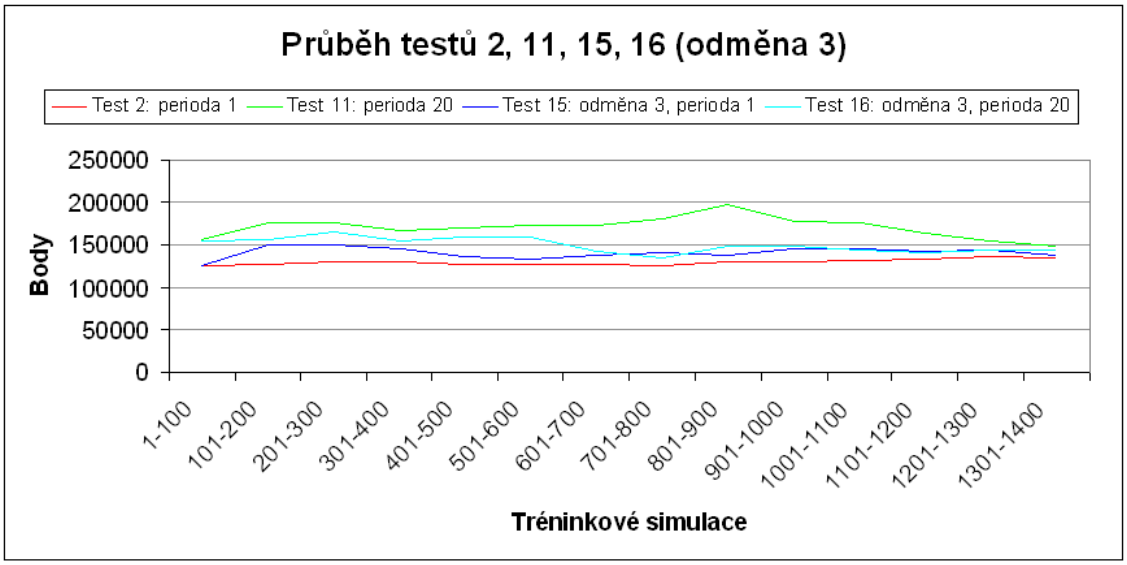
Testy 13 a 14 (obrázek 5.6): Algoritmus 3 s periodou změn 5 a maximálními povolenými změnami vah nejprve $\pm(10 \times 0.01)$ a poté $\pm(10 \times 0.05)$ jsme aplikovali též na síť přednastavenou jako v testu 4. Výsledek nepřinesl zdokonalení.

Testy 15, 16 a 17 (obrázky 5.7 a 5.8): Nejúspěšnější z testů výše jsme zkusili zopakovat s odměňovací funkcí 3, která zahrnuje strach z dravců. Maximální změna vah byla $\pm(n \times 0.01)$. Testovali jsme náhodně inicializovaný algoritmus 2 (test 15) a algoritmus 3 s periodou 20 (test 16) a algoritmus 3 přednastavený jako v předchozích testech (test 17). Změna odměňování nepřinesla markantní zlepšení.

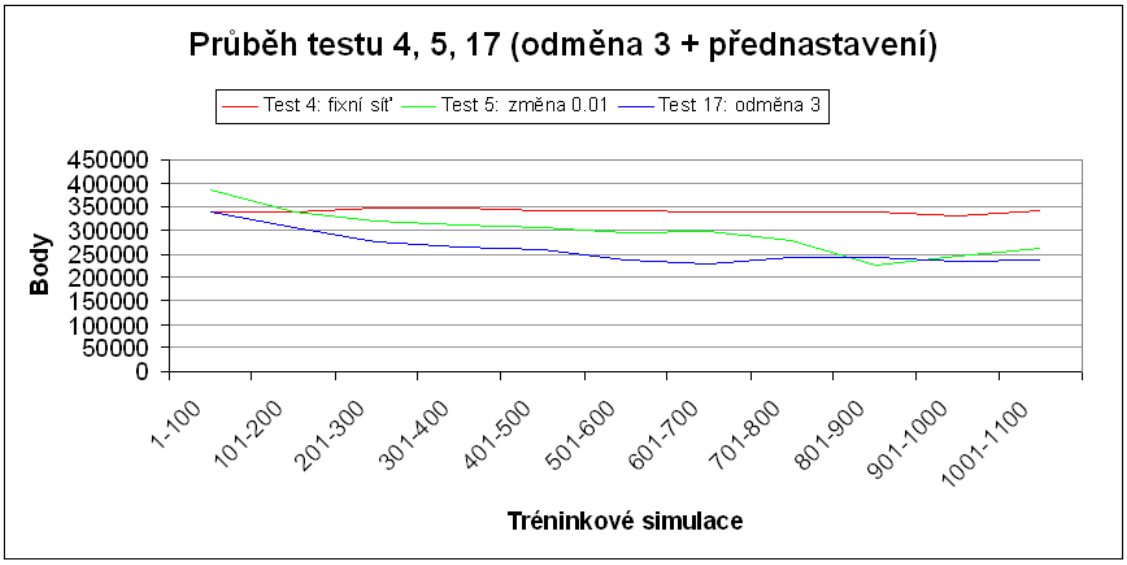
Výsledky algoritmu 2 a 3 v testu s dravci s náhodnou inicializací shrnuje tabulka 5.1, s přednastavením pak tabulka 5.2.



Obrázek 5.6: **Testy 4, 13, 14.**



Obrázek 5.7: **Testy 2, 11, 15, 16.**



Obrázek 5.8: **Testy 4, 5, 17.**

Test	Odměna	Perioda změn	Max. změna	Nej. výsledek	Prům. výsledek
2	2	1	±0.01	129354	128100
3	2	1	±0.05	144907	142849
7	2	5	$\pm(5 \times 0.01)$	156520	152720
8	2	5	$\pm(5 \times 0.05)$	159940	146356
9	2	10	$\pm(10 \times 0.01)$	149876	138636
10	2	10	$\pm(10 \times 0.05)$	168060	149958
11	**2**	**20**	$\pm(\mathbf{20 \times 0.01})$	**196024**	**174555**
12	2	20	$\pm(20 \times 0.05)$	164802	158255
15	3	1	±0.01	151334	140685
16	3	20	$\pm(20 \times 0.01)$	164698	152093

Tabulka 5.1: **Výsledky experimentů s algoritmy 2 a 3, ve kterých byla použita náhodná inicializace.** *V sloupci "Nej. výsledek" je nejvyšší počet bodů získaný za sto simulací z prvních deseti po sobě jdoucích set simulací. Ve sloupci "Prům. výsledek" je průměrný počet bodů na sto simulací měřený ve stejném období. Nejlepší výsledky byly dosaženy v testu 11.*

Test	Odměna	Perioda změn	Max. změna	Nej. výsledek	Prům. výsledek
4	-	1	±0.01	347542	340160
5	2	1	±0.01	384843	299716
6	2	1	±0.05	240072	173787
13	2	10	$\pm(10 \times 0.01)$	268697	200941
14	2	10	$\pm(10 \times 0.05)$	195122	164943
17	3	1	±0.01	339505	263215

Tabulka 5.2: **Výsledky experimentů s algoritmy 2 a 3, ve kterých byla síť přednastavena ručně vybranými hodnotami.** *Ve sloupci "Nej. výsledek" je nejvyšší počet bodů získaný za sto simulací z prvních deseti po sobě jdoucích set simulací. Ve sloupci "Prům. výsledek" je průměrný počet bodů na sto simulací měřený ve stejném období. Pouze v testu 5 byla úspěšnost výchozí sítě krátkodobě zdokonalena.*

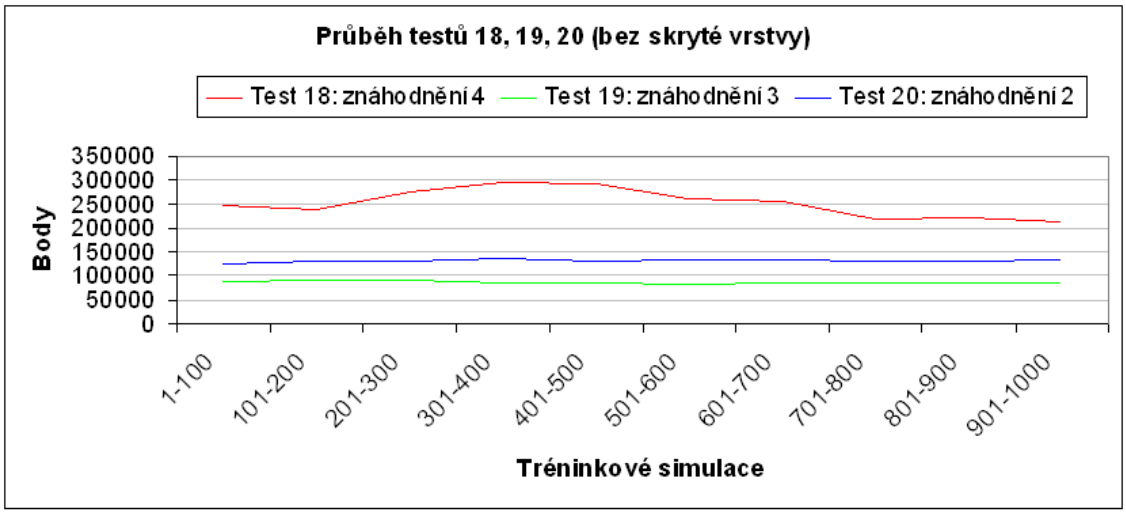
Test A: Síť se stejnou konfigurací, jako v testu 11 jsme testovali druhým typem testu. V deseti náhodných pokusech jsme testovali, zda se agent bez přítomnosti dravců naučí konzumovat a vyhledávat potravu a přežít tak celou simulaci. Ve všech deseti případech se toto agentovi povedlo v průměru po 11.5 simulacích. Stabilní řešení se povedlo najít v osmi z deseti případů a to v průměru po 1006 simulacích se směrodatnou odchylkou 980.

5.1.4 Modifikované TD(λ)

V následujícím popisujeme experimenty s algoritmem 4 popsaným v 4.4.2 - upravenou verzí TD(λ). Používali jsme vždy parametr učení $\alpha = 0.5$ a získanou odměnu jsme v každém kroku dělili číslem 50, aby celková odměna za partii nemohla příliš přesáhnout číslo 1. Použitá hodnota λ, pokud není uvedeno jinak, byla 0.5. V experimentech, ve kterých jsme používali metodu znáhodnění číslo 2 a 3, jsme hodnoty každého ze 4 vstupů pro akci diskretizovali do množiny 0, 0.5, 1, čili možných akcí bylo $3^4 = 81$.

Testy 18, 19 a 20 (obrázek 5.9): Použili jsme odměňovací funkci číslo 2 a síť bez skrytých neuronů. Porovnávali jsme výsledky pro různé způsoby znáhodnění popsané v 4.4.3. V testu 18 jsme použili znáhodnění, při kterém je vygenerováno 50 akcí a z nich vybrána ta nejlepší (metoda znáhodnění č. 4). V testu 19 jsme použili řešení, které zakazuje agentovi používat stejnou akci mnohokrát za sebou, přičemž hodnota $ACTIONLIMIT$ byla rovna třem (metoda znáhodnění číslo 3). V testu 20 jsme pak s pravděpodobností 5% nahradili nejlepší akci náhodnou akcí (metoda znáhodnění 2).

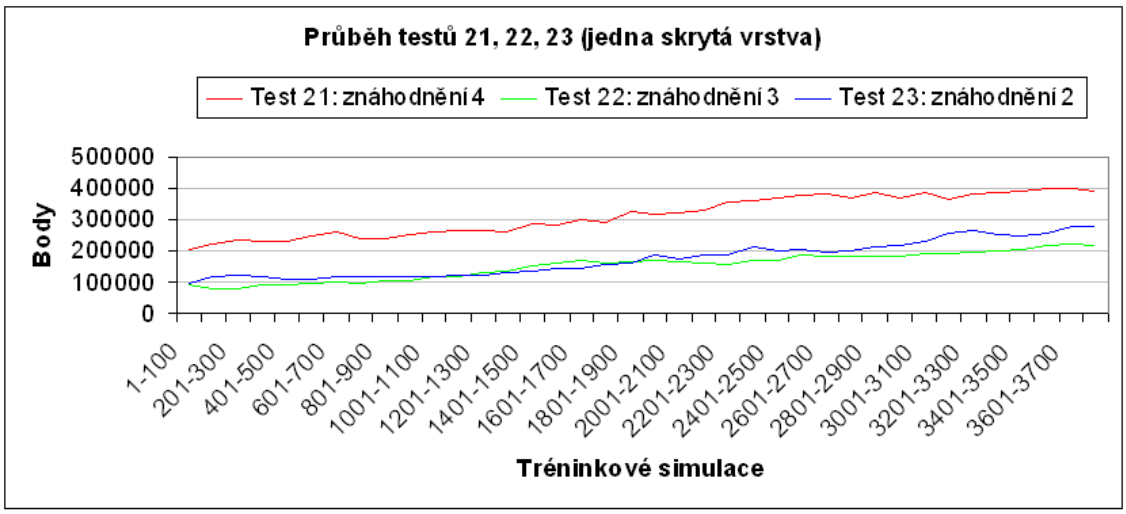
V testu se prokázalo, že pro reprezentaci vlivu agentovy činnosti na získání odměny je třeba sítě se skrytými neurony. Sítě si nevedly zcela špatně, ale velmi brzy se přestaly zdokonalovat a jejich úspěšnost stagnovala. Zdaleka nejlepší výsledky dosáhla síť v testu 18 používající znáhodnění 4. Síť používající znáhodnění 2 v testu 20 si vedla lépe než síť se znáhodněním 3 v testu 19.



Obrázek 5.9: **Testy 18, 19, 20.**

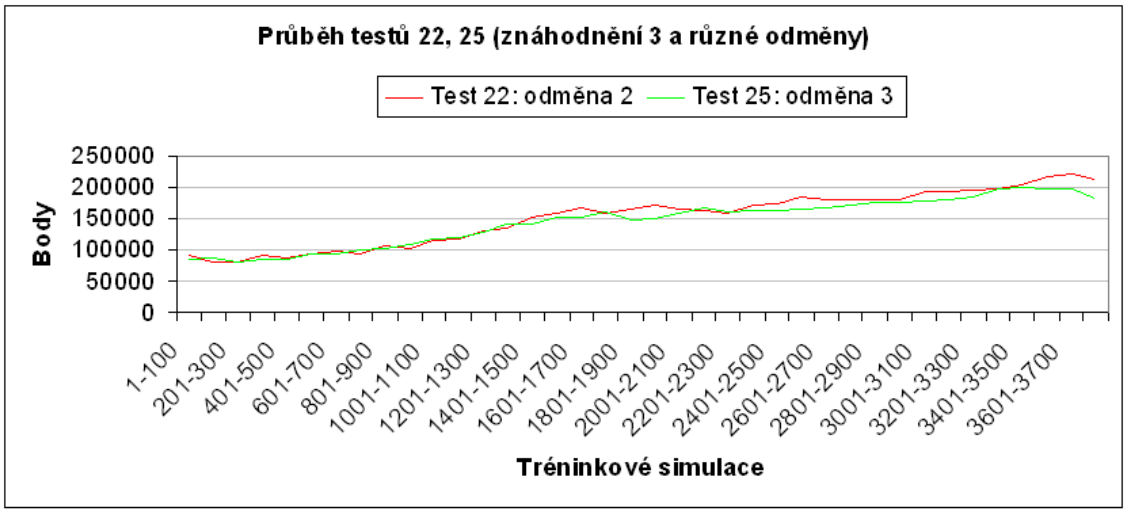
Testy 21, 22 a 23 (obrázek 5.10): Stejné testy jsme poté opakovali pro síť s jednou skrytou vrstvou o čtyřech neuronech. V testu 21 jsme použili znáhodnění 4 s nejlepší z 50

náhodných akcí, v testu 22 jsme omezovali opakování stejných akcí a v testu 23 jsme 5% akcí generovali náhodně. Test 21 dopadl velmi úspěšně. Síť se pomalu, ale nepřetržitě zdokonalovala a po 10000 trénovacích simulací dosahovala v průměru přes 460000 bodů za 100 simulací, čímž významně překonala i ručně nastavenou síť z testu 4. Síť z testu 22 s metodou znáhodnění 3 se též učila, ale ze třech testovaných sítí nejpomaleji. Síť v testu 23 se v prvních 1000 simulacích téměř nezdokonalovala, ale později se začala učit a dosáhla lepších výsledků než v testu 22.



Obrázek 5.10: **Testy 21, 22, 23.**

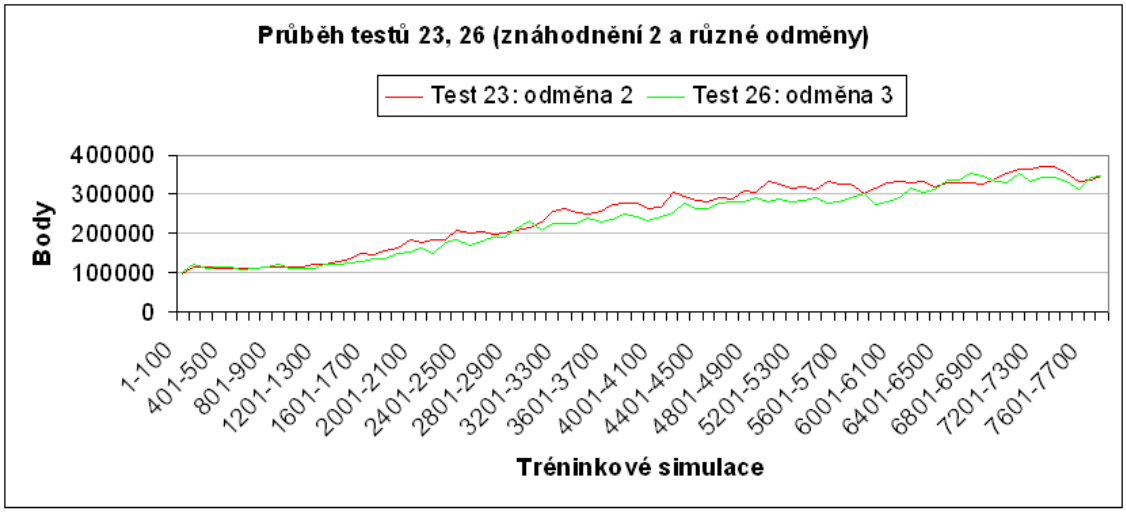
Lze říci, že všechny tři metody fungovaly. Agenti se dokázali naučit výhodným způsobem volit mezi konzumací nalezené potravy a vyhledáváním nové potravy. Agenti se také naučili reagovat útěkem na přítomnost dravců.



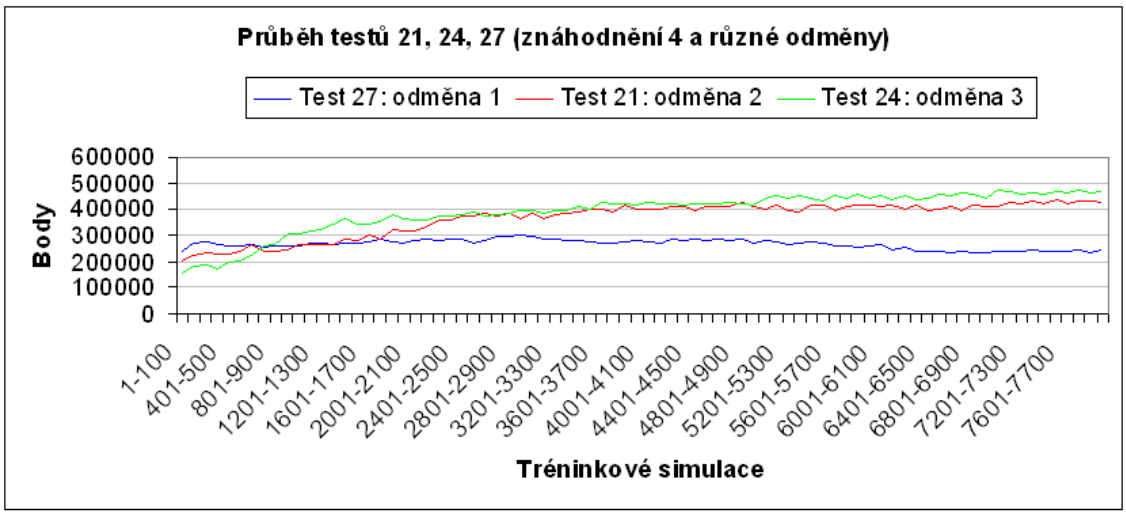
Obrázek 5.11: **Testy 22, 25.**

Testy 24, 25 a 26 (obrázky 5.11, 5.12 a 5.13): Stejné testy s jednou skrytou vrstvou jsme opakovali při nahrazení odměňovací funkce 2 odměňovací funkcí 3. V testu 24 bylo použito znáhodnění 4, v testu 25 znáhodnění 3 a v testu 26 znáhodnění 2.

Změna odměňovací funkce přinesla v kombinaci se znáhodněním 4 v testu 24 podstatné zlepšení úspěšnosti výsledné sítě (obrázek 5.14). Přitom v prvních cca 1000 simulacích měla lepší výsledky síť učená s odměnou 2, která byla později během učení sítí s odměnou 3 předstihnuta. Tento jev je možné vysvětlit tak, že jednodušší odměňovací funkce pomohla agentovi na počátku naučit se základní taktiku přežití, zatímco v pozdější fázi pomáhala agentovi sofistikovanější odměna 3 se inteligentněji vyhýbat nebezpečí. Při zbývajících dvou typech znáhodnění ke zdokonalení s použitím odměny 3 nedošlo. Vzájemné pořadí jednotlivých metod znáhodnění bylo stejné jako v předchozích experimentech při odměně číslo 2.

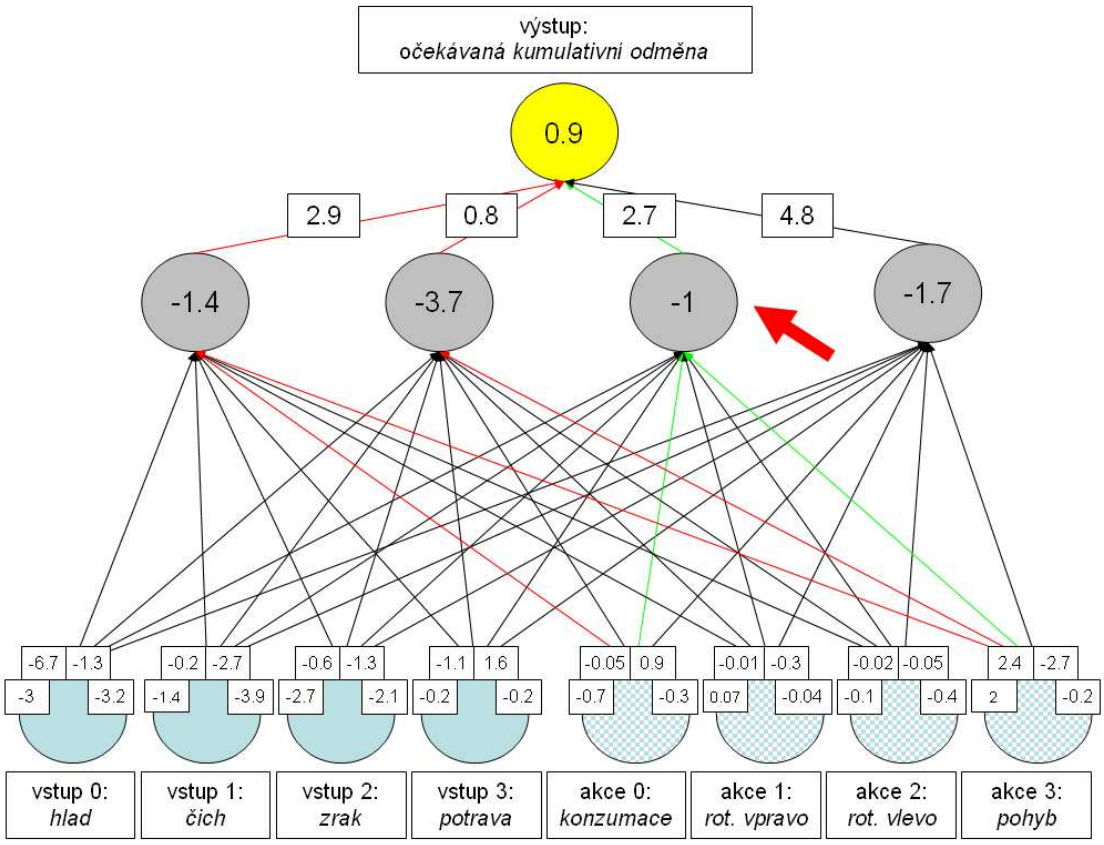


Obrázek 5.12: **Testy 23, 26.**



Obrázek 5.13: **Testy 21, 24, 27.**

Test 27 (obrázek 5.13): Výsledky z předchozích testů jsme zkusili též porovnat s výsledky sítě při metodě znáhodnění 4 a nejjednodušší odměňovací funkce 1. Síť s touto odměnou pouze z počátku učení předstihovala sítě odměňovací funkcí 2 a 3, velmi záhy se ale přestala zdokonalovat a její výsledná úspěšnost byla mnohem nižší. Potvrdilo se tak, že jemnější odměňovací funkce zlepšuje proces zpětnovazebného učení.

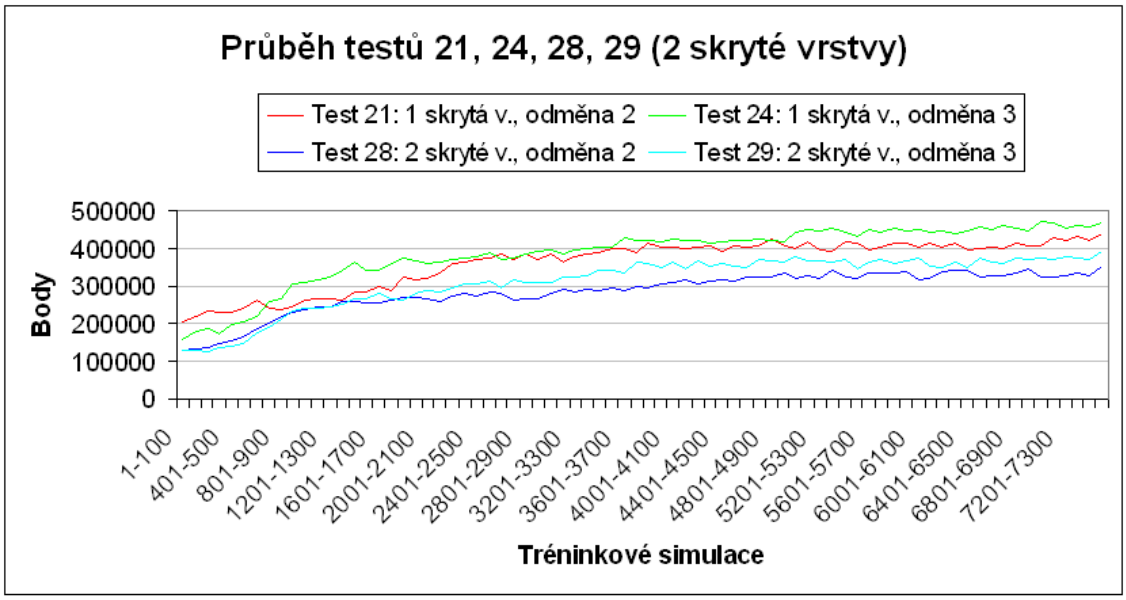


Obrázek 5.14: **Síť učená algoritmem 4.** *Síť jednoho z nejúspěšnějších agentů v testu 24 po 4000 tréninkových simulacích. Váhy a prahy byly zaokrouhleny. Agent uměl vyhledávat potravu a velmi úspěšně unikat dravcům. Sítě různých agentů v testu si vzájemně byly velmi podobné. Ze všech neuronů ve skryté vrstvě vedou kladné synapse do výstupního neuronu, mají tedy kladný příspěvek k očekávané kumulativní odměně. Síti dominují dráhy, které snižují predikci odměny, pokud je pozorován dravec nebo pokud agent trpí hladem. Váhy zachycující vliv agentových akcí jsou méně výrazné. Zeleně jsou vyznačeny dráhy, které zachycují, že je výhodné konzumovat potravu. Červeně jsou naopak označeny váhy, které zachycují výhody pohybu. Agent se rozhodne konzumovat potravu a zůstat na místě, pokud je v takovém stavu, že výstup skrytého neuronu označeného šipkou je rozhodující pro hodnotu výstupu celé sítě.*

Jak ukázaly experimenty výše, metoda znáhodnění 4 přináší podstatně lepší výsledky než ostatní uvažované metody. V dalším již experimentujeme pouze s metodou znáhodnění 4.

Testy 28 a 29 (obrázek 5.15): V těchto experimentech bylo testováno, zda druhá skrytá vrstva může být přínosem v řešení dané úlohy. V obou případech měla síť dvě skryté vrstvy po čtyřech neuronech. V experimentu 28 byla použita odměňovací funkce číslo 2 a v experimentu 29 odměňovací funkce číslo 3. V obou testech byly dosaženy horší výsledky než v odpovídajících variantách s jednou skrytou vrstvou. Další skrytá vrstva tedy není přínosem při řešení dané úlohy.

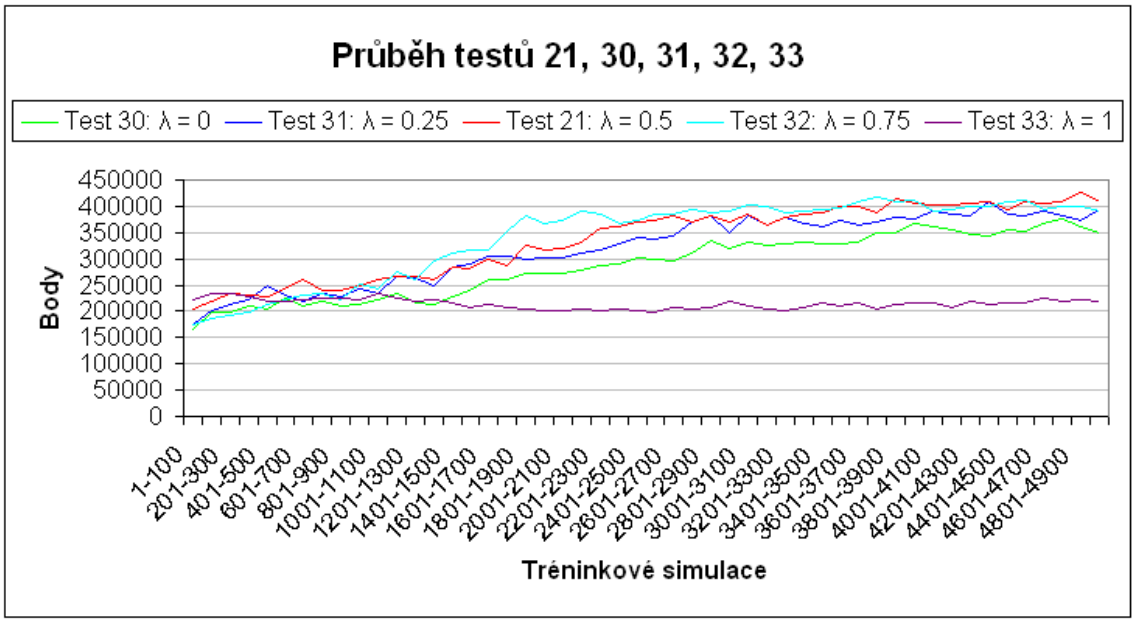
Testy 30 - 33 (obrázek 5.16): V těchto testech jsme se pokusili najít nejlepší hodnoty parametru λ pro danou úlohu. V každém testu jsme použili jednu skrytou vrstvu



Obrázek 5.15: **Testy 21, 24, 28, 29.**

a odměňovací funkci číslo 2. V testech 30, 31, 32 a 33 byly použity hodnoty λ v pořadí $0, 0.25, 0.75, 1$ a výsledky byly srovnány s testem 21, ve kterém byla použita hodnota $\lambda = 0.5$.

Při hodnotě $\lambda = 1$ se síť zpočátku učila nejrychleji, ale velmi brzy se přestala zdokonalovat a výsledky byly znatelně horší než pro všechny ostatní hodnoty. Pro ostatní hodnoty byly výsledky srovnatelné, nicméně nejlépe se síť učila při hodnotě $\lambda = 0.75$, méně dobře pro $\lambda = 0.5$, potom pro $\lambda = 0.25$ a ještě hůře pro $\lambda = 0$. Tedy od určité hodnoty rostla úspěšnost sítě s velikostí λ, patrně díky lepší schopnosti předvídat opožděné odměny.



Obrázek 5.16: **Testy 21, 30, 31, 32, 33.**

Výsledky experimentů s modifikovaným TD(λ) shrnuje tabulka 5.3.

Test	Odměna	Znáhodnění	λ	Skryté neurony	Nej. výsledek	Prům. výsledek
18	2	4	0.5	0	293703	251125
19	2	3	0.5	0	91675	86156
20	2	2	0.5	0	135955	132399
21	2	4	0.5	4	399840	313315
22	2	3	0.5	4	220962	151198
23	2	2	0.5	4	274455	172748
24	**3**	**4**	**0.5**	**4**	**427641**	**330954**
25	3	3	0.5	4	200613	144956
26	3	2	0.5	4	247435	160655
27	1	4	0.5	4	297242	273909
28	2	4	0.5	4 + 4	299929	244587
29	3	4	0.5	4 + 4	360709	260887
30	2	4	0	4	348474	266309
31	2	4	0.25	4	382142	298424
32	2	4	0.75	4	418372	323127
33	2	4	1	4	234959	213849

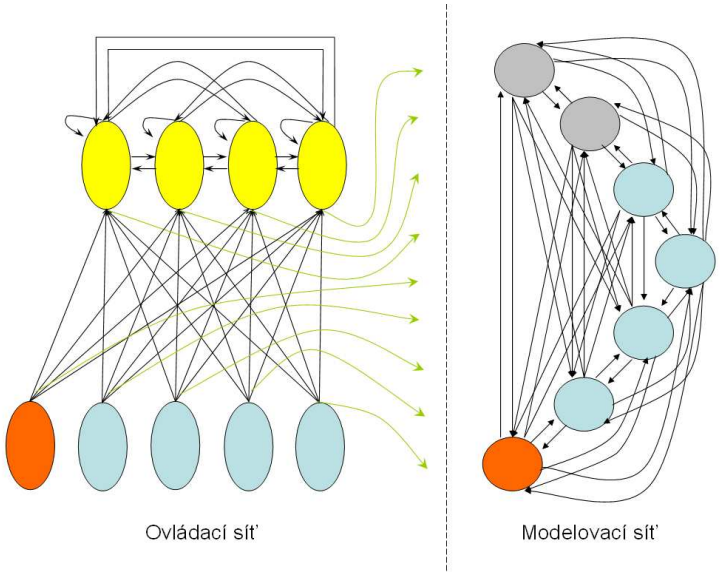
Tabulka 5.3: **Výsledky experimentů s algoritmem 4.** *Použité metody znáhodnění jsou očíslovány tak, jak je popsáno v části 4.4.3. V sloupci "Nej. výsledek" je nejvyšší počet bodů získaný za sto simulací z prvních 38 po sobě jdoucích set simulací. Ve sloupci "Prům. výsledek" je průměrný počet bodů na sto simulací měřený ve stejném období. Nejlepší výsledky byly dosaženy v testu 24.*

Test B: Pro test bez dravců byla vybrána konfigurace použitá v testu 24 - odměňovací předpis 3 se znáhodněním číslo 4, jednou skrytou vrstvou a parametrem $\lambda = 0.5$. V každém z deseti testů se agentovi povedlo přežít celou simulaci, průměrně k tomu docházelo po 77 simulacích. Stabilní řešení se povedlo najít pouze ve třech z těchto pokusů, v průměru po 282 simulacích se směrodatnou odchylkou 97. Avšak i během pokusů, při kterých nebylo stabilní řešení nalezeno, se síť už po několika stech simulacích dostala do takového stavu, že agent přežíval v průměru každou druhou testovací simulaci.

5.1.5 Schmidhuberova rekurentní síť

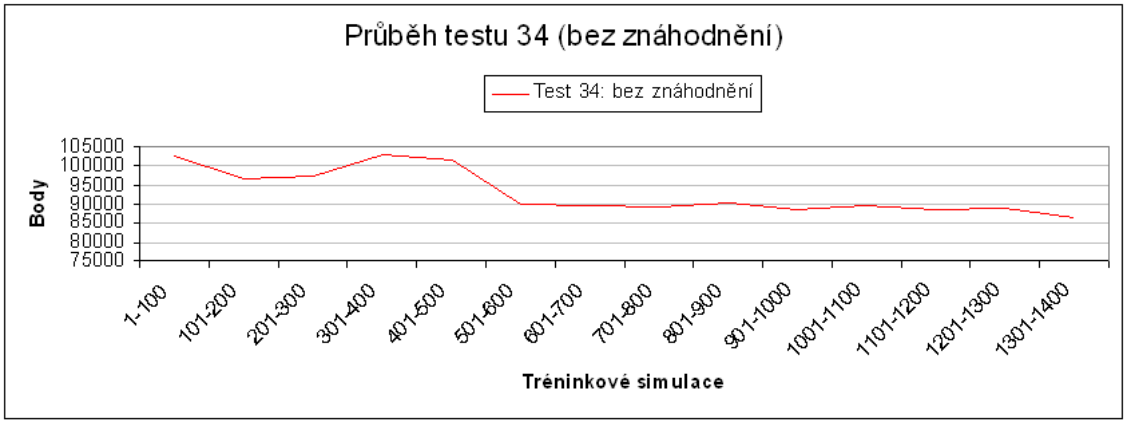
V této části jsou popsány experimenty s algoritmem 5 z části 4.5. Pro účely úlohy dravce a kořisti byla řídicí síť vybavena čtyřmi normálními vstupy (hlad, čich, zrak a přítomnost potravy), jedním odměňovacím vstupem a čtyřmi výstupy (konzumace potravy, rotace vpravo, rotace vlevo a pohyb vpřed). Modelující síť obsahovala prediktory pro všechny vstupy řídicí sítě. Počet skrytých neuronů se měnil mezi různými testy. Síť je znázorněna na obrázku 5.17. Použité parametry učení pro řídicí i modelující síť byly rovné 0.5. Všechny aproximace parciálních derivací byly jednou za dvacet kroků vynulovány. Pokud jsme totiž tyto parciální derivace nevynulovali, narůstaly jejich hodnoty během učení k nekonečnu a s nimi i váhy řídicí sítě. V testech nebyly použity zmiňované modifikace rozšiřující algoritmus o umělou zvědavost, pravděpodobnostní výstupy ani techniky "Temporal Difference learning".

Test 34 (obrázek 5.18): Nejprve jsme zkusili použít algoritmus bez jakékoli formy znáhodnění výstupů. Pro odměňování byl použit předpis 2 zohledňující agentovu zásobu



Obrázek 5.17: **Schmidhuberova síť pro úlohu dravce a kořisti.** *Na obrázku je schéma Schmidhuberovy rekurentní sítě pro řešení úlohy kořisti a dravce se dvěma skrytými neurony modelující sítě tak, jak byla použita například v testu 36. Modře jsou vybarveny normální vstupní neurony a jejich prediktory, oranžově odměna a její prediktor, žlutě výstupní neurony a šedě skryté neurony. Ze všech vstupních a výstupních neuronů řídicí sítě vedou synapse do všech neuronů modelující sítě.*

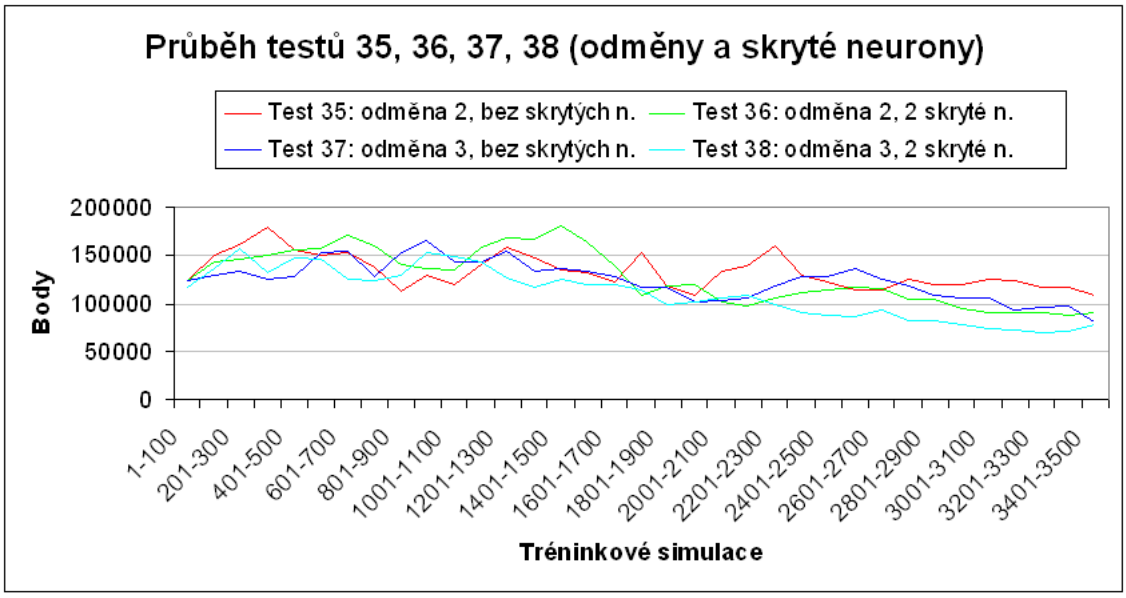
potravy ale nikoli nebezpečí od dravců. Řídicí síť neměla žádný skrytý neuron a modelující síť měla 2 skryté neurony. Aby byla modelující síť schopna reprezentovat složitější znalosti typu "v jedné situaci přinese odměnu akce A a v jiné situaci přinese odměnu akce B", opakovali jsme krok 4 algoritmu v každé iteraci dvakrát. Vybavení řídicí sítě jsme v tomto experimentu prováděli pouze jednou.



Obrázek 5.18: **Testy 34.**

Síť bez znáhodnění v testu 34 se nedokázala vyrovnat s problémem "strachu z neznáma" popsaným v části 4.4.3. Sítě se neučily a všichni agenti po určitém počtu simulací dospěli do stavu, kdy neustále opakovali stejnou akci. Buď se nepřetržitě pohybovali a nekonzumovali potravu, a nebo stále zůstávali na místě a konzumovali potravu, dokud jim nedošla či dokud je nedostihl dravec.

Testy 35 - 38 (obrázek 5.19): Test 34 prokázal nutnost určité formy znáhodnění. V těchto a následujících testech proto byla v každé iteraci algoritmu s pravděpodobností 5% agentova akce nahrazena akcí náhodnou. V těchto testech jsme experimentovali s použitou odměňovací funkcí a počtem skrytých neuronů modelující sítě. V testech 35 a 36 byla použita odměňovací funkce 2, zatímco v testech 37 a 38 odměňovací funkce 3. V testech 35 a 37 byla použita modelující síť bez skrytých neuronů, v testech 36 a 38 měla modelující síť 2 skryté neurony. Ostatní podmínky byly stejné, jako v testu 34.



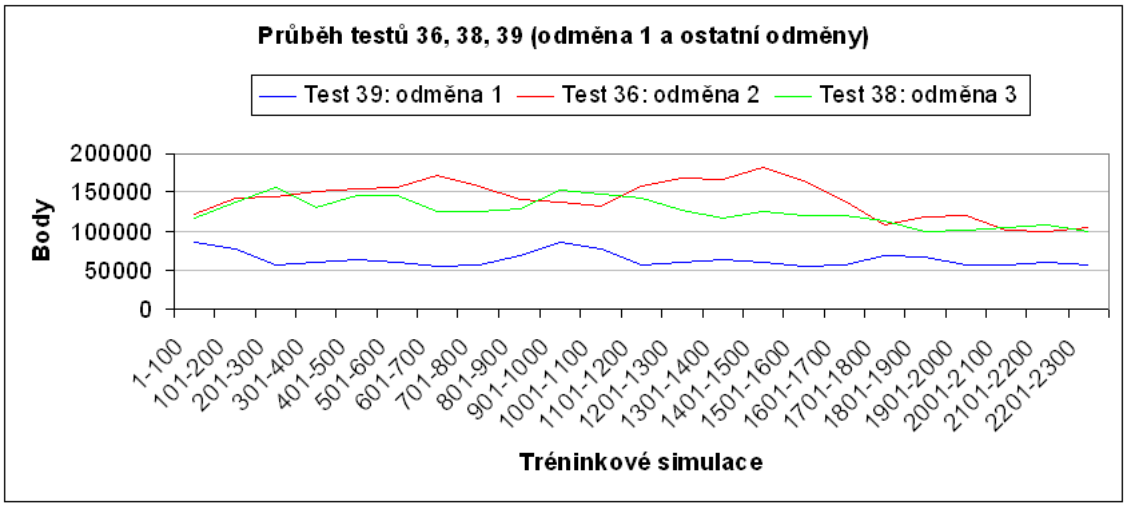
Obrázek 5.19: **Testy 35, 36, 37, 38.**

Všechny tyto čtyři sítě se dokázaly úlohu zpočátku rychle naučit. Dokázaly vyhledávat a konzumovat potravu i vyhýbat se dravcům. S příliš dlouhým učením se pak ale jejich kvalita zhoršovala a přibližně po 2500 simulacích velmi klesla a agenti měli tendenci pasivně zůstávat na jednom místě. Je možné, že tato síť podléhá obdobě tzv. přeučení, se kterým se setkáváme při učení s učitelem - viz konec části 3.2. O něco lépe dokázaly úlohu řešit sítě v testech 41 a 42 při použití odměňovací funkce 2. V tomto případě se tedy osvědčila spíše jednodušší odměňovací funkce 2. Skryté neurony v modelující síti neměly na úspěšnost příliš velký vliv.

Test 39 (obrázek 5.20): Při dvou skrytých neuronech modelující sítě jsme též provedli test s odměňovací funkcí číslo 1. Při použití této odměňovací funkce však nedocházelo k učení a agenti po krátké době vykonávali neustále stejné akce. Tedy primitivní odměňovací funkce, která rozlišuje pouze zda je agent živý nebo mrtvý, nestačí k tomu, aby algoritmus fungoval.

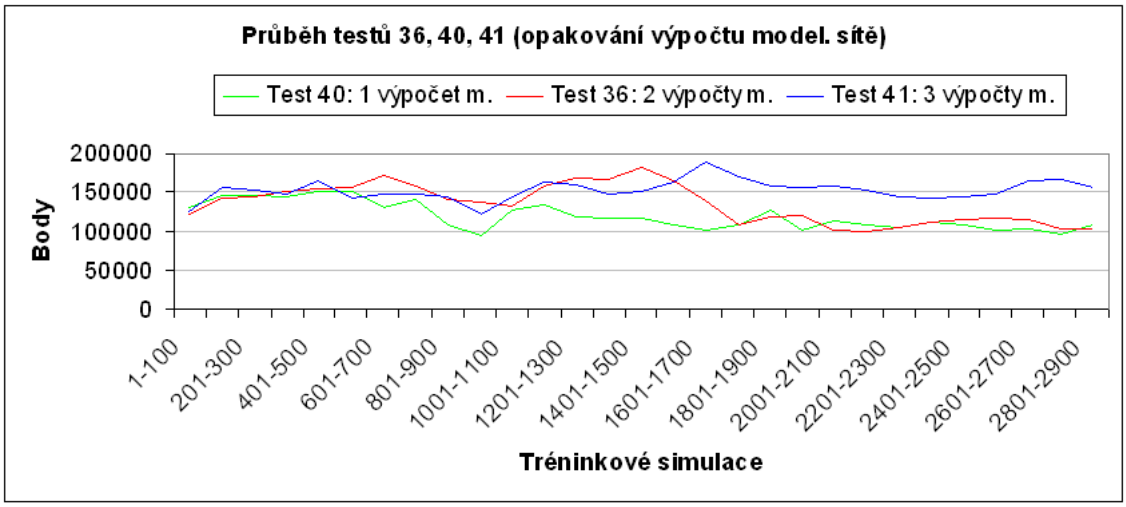
Testy 40 a 41 (obrázek 5.21): Testována byla úspěšnost neuronové sítě při různém počtu opakování kroku 4 použitého algoritmu 5, tedy při různém počtu vybavení modelující sítě na jeden krok simulace. V testu 40 byl počet opakování 1 a v testu 41 byl roven 3. Výsledky byly porovnány s testem 36, ve kterém byl krok 4 opakován dvakrát.

Testy ukazují, že s přibývajícím počtem opakování výpočtu modelující sítě na jednu iteraci algoritmu roste úspěšnost sítě. Síť v testu 41 předstihla síť z testu 36, která byla



Obrázek 5.20: **Testy 36, 38, 39.**

úspěšnější než síť v testu 40. Úspěšnost sítí po určitém počtu iterací opět začala výrazně klesat, přičemž ale při počtu iterací 3 klesala znatelně pomaleji.

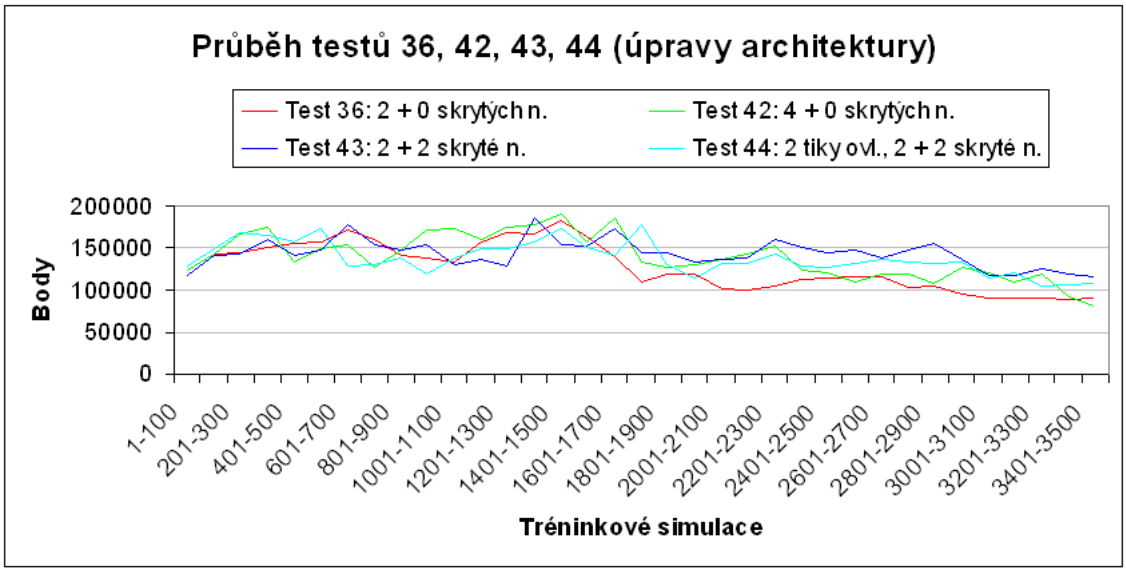


Obrázek 5.21: **Testy 36, 40, 41.**

Testy 42, 43 a 44 (obrázek 5.22): V těchto testech jsme se dále snažili zvýšit sílu sítě. Vycházeli jsme opět z konfigurace použité v testu 36. V testu 42 byly místo 2 skrytých neuronů modelující sítě použity 4 skryté neurony. V testu 43 jsme ponechali 2 skryté neurony v modelující síti, ale řídicí síť též obsahovala 2 skryté neurony. Nastavení testu 44 pak bylo stejné jako pro test 43 s tou výjimkou, že nejen modelující síť, ale též řídicí síť vybavovala dvakrát v jednom kroku simulace (Dvakrát jsme opakovali krok 2 algoritmu 5.).

Úspěšnost sítí byla velmi podobná. Závěrem je, že zvýšení počtu skrytých neuronů ani opakování výpočtu řídicí sítě navíc nepřináší výrazné zvýšení úspěšnosti.

Výsledky algoritmu 5 v experimentu s dravci jsou zaznamenány v tabulce 5.4.



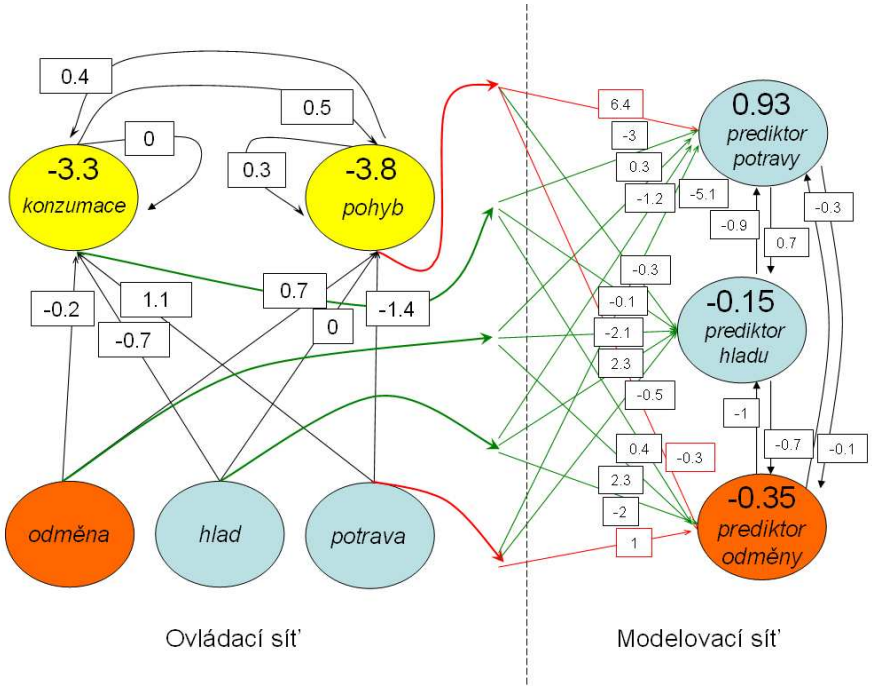
Obrázek 5.22: **Testy 36, 42, 43, 44.**

Test	Odměna	Kroky o + m	Skryté o + m	Náhodnost	Nej. výsl.	Prům. výsl.
34	2	1 + 2	0 + 2	-	102777	92910
35	2	1 + 2	0 + 0	5%	179324	135352
36	**2**	**1 + 2**	**0 + 2**	**5%**	**181512**	**133507**
37	3	1 + 2	0 + 0	5%	165068	129025
38	3	1 + 2	0 + 2	5%	156824	116359
39	1	1 + 2	0 + 2	5%	86026	63893
40	2	1 + 1	0 + 2	5%	150783	119371
41	**2**	**1 + 3**	**0 + 2**	**5%**	**188129**	**152446**
42	**2**	**1 + 2**	**0 + 4**	**5%**	**190217**	**145156**
43	**2**	**1 + 2**	**2 + 2**	**5%**	**184423**	**147094**
44	2	2 + 2	2 + 2	5%	177350	142303

Tabulka 5.4: **Výsledky experimentů s algoritmem 5.** *Sloupec "Kroky o + m" označuje, kolik výpočtů řídicí respektive modelující sítě bylo opakováno v jednom kroku algoritmu. Sloupec "Skryté o + m" obsahuje počet skrytých neuronů v řídicí a modelující síti. V sloupci "Nej. výsl." je nejvyšší počet bodů získaný za sto simulací z prvních 30 po sobě jdoucích set simulací. Ve sloupci "Prům. výsl." je průměrný počet bodů na sto simulací měřený ve stejném období. Nejlepší výsledky byly dosaženy v testech 36, 41, 42 a 43.*

Test C: Pro test bez dravců byla zvolena konfigurace z testu 36. Ve všech z deseti pokusů bylo poměrně rychle nalezeno stabilní řešení, při kterém agent přežil dvacet simulací v řadě. Poprvé agent přežil celou simulaci v průměru po 58 tréninkových simulacích, stabilní řešení nacházel v průměru po 101 simulacích se směrodatnou odchylkou 99. Výslední jedinci obvykle pravidelně střídali krok, ve kterém konzumovali potravu, a krok, ve kterém se přesouvali na jiné místo.

Na obrázku 5.23 je zobrazena síť podobná té, která byla výsledkem testu C. V modelující síti lze pozorovat kódování mnoha základních znalostí, například, že pokud má agent hlad a nízkou odměnu, je pravděpodobné, že bude mít hlad a odměna bude nízká i v příštím kroku. Síť též zachycuje znalost o tom, že konzumace potravy přináší po-



Obrázek 5.23: **Schmidhuberova síť učená v testu bez predátorů.** *Agent řízený touto sítí uměl přežít v prostředí bez predátorů neomezeně dlouhou dobu. Narozdíl od testu C zde pro jednoduchost nebyly použity skryté neurony (použita konfigurace z testu 35). Pro přehlednost jsme též nezakreslili vstupní neurony pro čich a zrak, ani jejich prediktory, neboť v tomto testu nebyly tyto vstupy nikdy aktivní. Obrázek neobsahuje ani výstupní neurony pro rotaci, které nejsou příliš důležité. Pro pochopení funkce modelující sítě je třeba vzít v úvahu, že výpočet modelující sítě probíhal dvakrát v každé iteraci algoritmu. Váhy byly zaokrouhleny na jedno desetinné místo.*

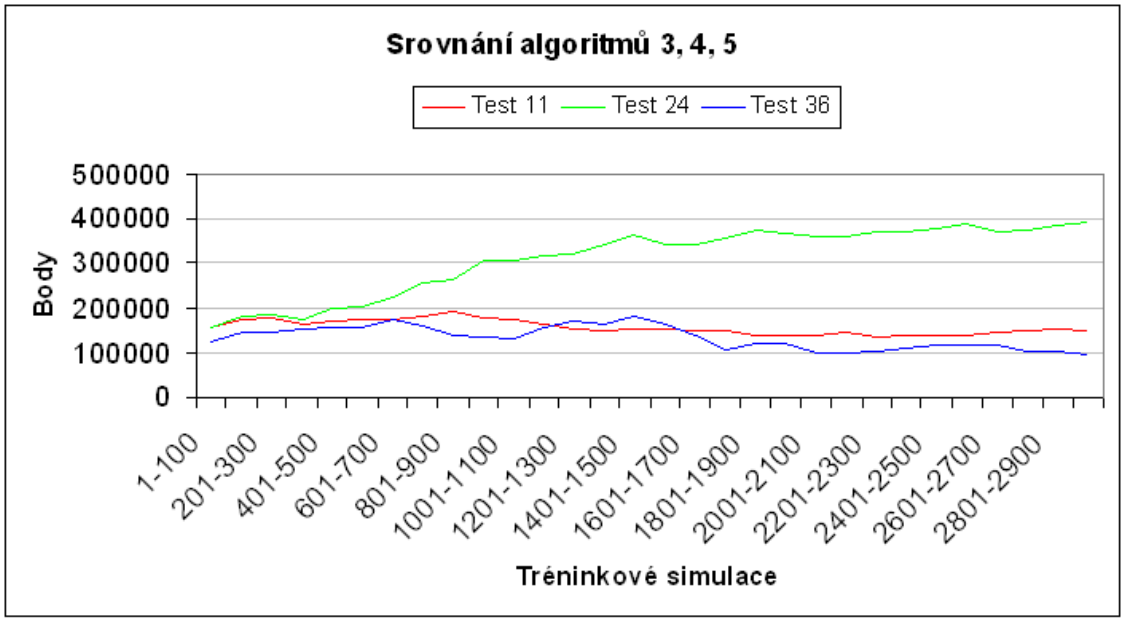
travu, ale snižuje dostupnost potravy v příštím kroku. Za pozornost stojí dvě dráhy, které jsou vyznačeny červeně. Jedna obsahuje znalost o tom, že pohyb snižuje získanou odměnu v následujícím kroku (neboť agent nekonzumuje potravu a naopak pohybem jí více spotřebuje). Druhá dráha reprezentuje znalost, že pohyb přináší potravu a potrava přináší odměnu. Tyto dvě znalosti působí protichůdně při trénování vah synapsí řídicí sítě, které vedou do výstupního neuronu pro pohyb. To nutí řídicí síť rovnoměrně střídat mezi pohybem a konzumací potravy. Návaznost červených synapsí z neuronu pro pohyb do prediktoru potravy a ze vstupu pro potravu do prediktoru odměny při učení řídicí sítě je dána tím, že na posledním řádku bloku 5 algoritmu 5 jsou parciální derivace výstupů vstupních neuronů nahrazovány parciálními derivacemi aktivací jejich prediktorů.

Zobrazené nastavení řídicí sítě vede agenta k tomu, aby konzumoval potravu, pokud je přítomna, a pohyboval se v opačném případě. Kladné váhy křížových vazeb mezi výstupními neurony pro konzumaci potravy a pohyb navíc vedou agenta k tomu, aby akce střídal (Dochází zde k využití rekurentní dráhy coby paměti.).

5.1.6 Srovnání testovaných algoritmů

Pro srovnání výsledků algoritmů 3, 4 a 5 jsme vybrali jejich nejúspěšnější konfigurace z testů 11, 24 a 36 (obrázek 5.24). Pro každé z těchto tří nastavení jsme opakovali test s dravci desetkrát při různých náhodných inicializacích. Pro každé opakování jsme zaznamenali dosažené body v nejlepší stovce z prvních 3000 trénovacích simulací. Pro každý z algoritmů jsme spočítali průměr bodů za nejlepší stovku a směrodatnou odchylku přes všech 10 opakování.

Jak ukazuje tabulka 5.5, v testech prvního typu byl jednoznačně nejúspěšnější algoritmus 4, který se v celém průběhu testování dokázal zdokonalovat a dosahoval přibližně dvojnásobný počet bodů oproti ostatním dvěma algoritmům. Proto lze se statistickou spolehlivostí 95% a dokonce i 99% tvrdit, že tento algoritmus dosahuje lepších výsledků než ostatní dva algoritmy. Sofistikovanější algoritmus 5, který oproti oběma zbývajícím algoritmům vybavuje agenta též pamětí, se po 1500 simulacích přestal zdokonalovat. Měření ukazují, že úspěšnost tohoto algoritmu byla velmi podobná úspěšnosti algoritmu 3. V testech druhého typu pak nejlépe uspěl algoritmus 5, který se vždy poměrně rychle naučil přežít celé trvání simulace.



Obrázek 5.24: **Srovnání algoritmů 3, 4, 5**

Možnosti algoritmu 3 jsou zřejmě omezené a nebyla očekávána jeho přílišná úspěšnost v řešení úlohy. Výsledky jsou nicméně zajímavou ukázkou toho, čeho lze dosáhnout s velmi jednoduchým algoritmem. Vysoká úspěšnost algoritmu 4 je poměrně překvapivá, neboť algoritmus byl původně určen pro odlišný typ úloh a síť nemá možnost si uchovávat informace z minulosti. Co se týče algoritmu 5 pro rekurentní síť, výsledky dosažené v testech bez dravců jsou povzbudivé. Naproti tomu v testech prvního typu se algoritmus příliš neosvědčil, a vezmeme-li v úvahu jeho vysokou výpočetní náročnost, algoritmus neprokázal svoji užitečnost v řešení dané úlohy. Je ale pravděpodobné, že existují rezervy ve zdokonalení tohoto algoritmu, například v použití propracovanější metody znáhodnění.

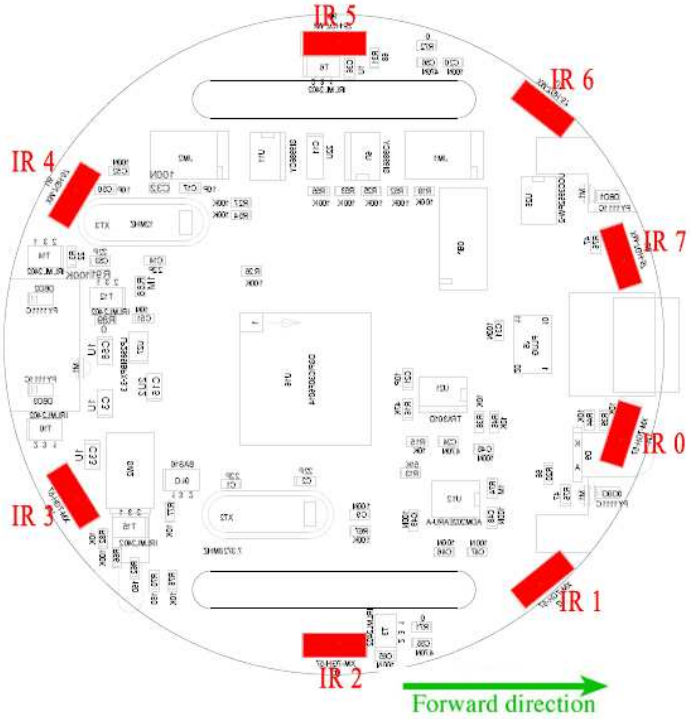
Algoritmus	Test	Test bez dravců	Průměrné body	Směr. odch.
3	11	80%	$1.805 \cdot 10^5$	$1.06 \cdot 10^4$
4	24	30%	$3.929 \cdot 10^5$	$1.78 \cdot 10^4$
5	36	100%	$1.815 \cdot 10^5$	$1.64 \cdot 10^4$

Tabulka 5.5: **Srovnání úspěšnosti algoritmů 3, 4 a 5 při nejlepším nalezeném nastavení jejich parametrů.** *Sloupec "Test bez dravců" označuje podíl testů druhého typu (bez dravců), při kterých bylo nalezeno stabilní řešení. Sloupec "Průměrné body" označuje průměr z bodů získaných algoritmem v nejlepší stovce z prvních 3000 trénovacích simulací přes všech deset opakování. Sloupec "Směr. odch. udává příslušnou směrodatnou odchylku."*

Na disku CD přiloženém k této práci se nachází spustitelná verze simulátoru přednastavená pro spuštění experimentu s dravci i bez dravců pro vybrané konfigurace každého ze tří popsaných algoritmů společně se svým zdrojovým kódem.

5.2 Aplikace v robotice

Jak již bylo uvedeno, vhodnou oblastí pro aplikaci zpětnovazebného učení je robotika. Rozhodli jsme se proto otestovat nejúspěšnější z popsaných algoritmů - algoritmus 4 (modifikovaný TD(λ)) na úloze robotiky. Použitým robotem byl e-puck [2], navržený týmem okolo Francesca Mondady z École Polytechnique Fédérale de Lausanne ve Švýcarsku. E-puck je jednoduchý robot ve tvaru kotouče o průměru cca 8 cm, který je vybaven dvěma nezávisle poháněnými koly a osmi infračervenými senzory schopnými detekovat blízkost překážky. Robot je též vybaven dalšími senzory a efektory, které v testu nebyly použity. Hledání vhodného nastavení a učení neuronové sítě bylo prováděno z důvodu úspory času na simulátoru Webots 5.1.13 vytvořeného taktéž Švýcarskou firmou Cyberbotics [3]. Výsledná neuronová síť byla nakonec otestována na fyzickém robotu.

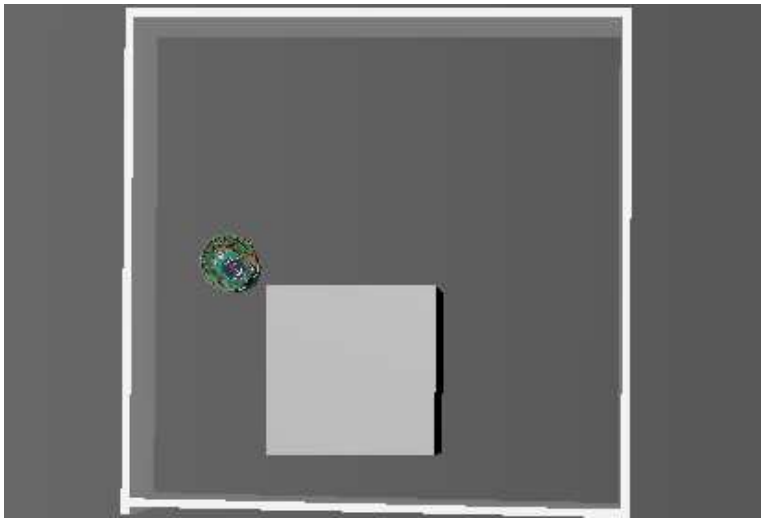


Obrázek 5.25: Robot e-puck je vybaven osmi infračervenými senzory, které mohou snímat vzdálenost nejbližší překážky ve směru, kterým míří. Obrázek byl přejat z webové stránky http://www.e-puck.org.

5.2.1 Řešená úloha robotiky

Cílem bylo naučit agenta - robota pohybovat se ve vymezeném prostoru a přitom se vyhýbat překážkám (obrázek 5.26). Tímto prostorem je ohrazené čtvercové pole velikosti 60 × 60 cm. Aby nebylo úlohu možné řešit příliš snadno jízdou po stále stejné trase,

[2] http://www.e-puck.org
[3] http://www.cyberbotics.com



Obrázek 5.26: **Prostředí, ve kterém se pohybuje robot e-puck v průběhu experimentu.** *Obrázek byl vygenerován programem Webots.*

v prostoru se též nachází čtvercová překážka velikosti 20 × 20 cm, která je v průběhu experimentu přemísťována. Jak ohrádka, tak překážka mají světle šedivou barvu.

Vstupy:

Jak je patrné z obrázku 5.25, robot je infračervenými senzory hustěji osazen vepředu. To umožňuje účinněji zabránit kolizi s překážkou, pokud se robot pohybuje vpřed. Hodnoty snímané senzory ve stejné situaci se mezi různými fyzickými roboty liší, protože každý robot je jedinečný. Dosah senzorů je v řádu centimetrů a záleží také na barvě překážky. Světlé těleso lépe odráží infračervené paprsky a je tedy lépe vidět. Lze říci, že pokud není v dosahu senzoru žádná překážka, hodnota senzoru se pohybuje pod hodnotou 300 [4]. S blížící se překážkou roste tato hodnota až k několika tisícům, a to nelineárně vzhledem ke vzdálenosti překážky.

Abychom zjednodušili agentovu neuronovou síť, omezíme počet jeho vstupních neuronů kódujících stav prostředí na 4, které budeme značit st_0, st_1, st_2, st_3. Každý z těchto vstupů bude odpovídat dvěma z infračervených senzorů, přesněji tomu z nich, který je v daném okamžiku aktivnější. Vstup st_0 odpovídá dvojici senzorů na levé straně robota (IR 5, IR 6), vstup st_1 odpovídá dvojici předních senzorů (IR 7, IR 0), vstup st_2 odpovídá dvojici senzorů na pravé straně robota (IR 1, IR 2) a vstup st_3 odpovídá dvojici zadních neuronů (IR 3, IR 4). Hodnoty senzorů jsou jako takové pro vstup neuronové sítě nevhodné, proto je třeba je předzpracovat. Jsou-li signály dvojice senzorů odpovídajících určitému vstupnímu neuronu st_i rovny S_1, respektive S_2, potom bude pro hodnotu y_i tohoto neuronu platit:

$$y_{st_i} = \frac{\sqrt{\max(\max(S_1 - 300, 0), \max(S_2 - 300, 0))}}{70} \tag{5.2}$$

Hodnoty vstupů nejprve snížíme o 300, abychom se zbavili šumu. Potom vybereme aktivnější z obou neuronů, jeho hodnotu odmocníme, abychom omezili nelinearitu vzhledem

[4] Pro fyzické roboty platí, že každý senzor dává poněkud různé hodnoty, a senzory je třeba pro každého zvlášť kalibrovat .

ke vzdálenosti překážky, a výsledek nakonec vydělíme číslem 70, aby hodnota spadala do intervalu $[0, 1]$, nebo jen mírně převyšovala 1.

Výstupy (akce):

Obě z robotových kol se mohou nezávisle na sobě pohybovat vpřed nebo vzad. Agent bude mít v pojetí zpětnovazebného učení 4 výstupy, z pohledu algoritmu 4 se bude jednat o 4 vstupní neurony pro akce A_0, A_1, A_2, A_3. Pro usnadnění učení jsme se rozhodli kódovat zvláštním neuronem pohyb každého kola vpřed a vzad. Výstup A_0 odpovídá pohybu levého kola vpřed, výstup A_1 pohybu levého kola vzad, výstup A_2 pohybu pravého kola vpřed a výstup A_3 pohybu pravého kola vzad[5]. Robot povoluje rychlost každého kola z intervalu $[-1000, 1000]$, kde 1000 označuje maximální rychlost vpřed, -1000 maximální rychlost vzad a 0 situaci, kdy je kolo v klidu. My omezíme rychlosti obou kol na interval $[-500, 500]$. Pro rychlost levého kola $Speed1$ tedy bude platit:

$$Speed1 = 500(y_{A_0} - y_{A_1}) \tag{5.3}$$

Podobně pro rychlost $Speed2$ pravého kola platí:

$$Speed2 = 500(y_{A_2} - y_{A_3}) \tag{5.4}$$

Odměna:

Abychom nemuseli v každém kroku posílat nějakým způsobem robotu zvnějšku informaci o tom, jak velkou získal odměnu, byla odměňovací funkce navržena tak, aby robot sám byl schopen svoji odměnu spočítat. To komplikuje fakt, že robot sám nezná svou přesnou polohu v prostředí. Proto bude, poněkud nepřesně, počítat robot ujetou vzdálenost a detekovat kolizi s překážkou pouze z hodnot svých senzorů (Jedná se o tzv. interní ohodnocovací funkci.). Použitý vzorec pro výpočet odměny v čase t bude:

$$r_t = r_t^{(1)} \cdot \left(1 - \sqrt{r_t^{(2)}}\right) \cdot (1 - r_t^{(3)}), \tag{5.5}$$

kde

$$r_t^{(1)} = \frac{|y_{A_0}(t-1) - y_{A_1}(t-1)|}{2} + \frac{|y_{A_2}(t-1) - y_{A_3}(t-1)|}{2} \tag{5.6}$$

je člen odměňující za pohyb kol (motivuje robota k pohybu),

$$r_t^{(2)} = \frac{|y_{A_0}(t-1) - y_{A_1}(t-1) - y_{A_2}(t-1) + y_{A_3}(t-1)|}{2} \tag{5.7}$$

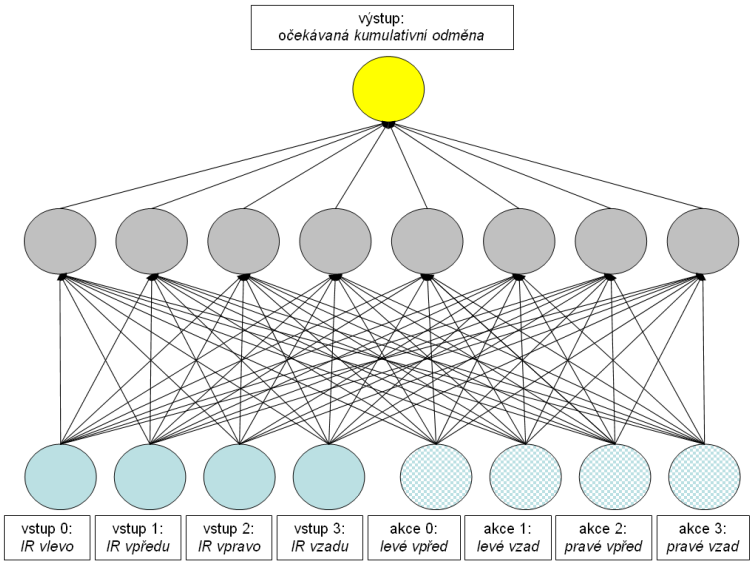
je člen penalizující pohyb kol proti sobě (odrazuje robota od točení se na místě) a

$$r_t^{(3)} = \min\left(1, \frac{|y_{st_0}(t) + y_{st_1}(t) + y_{st_2}(t) + y_{st_3}(t)|}{2}\right) \tag{5.8}$$

je člen penalizující za aktivaci infračervených senzorů (odrazuje robota od bourání do překážek). Povšimněme si, že odměnu v čase t určujeme z hodnoty stavových vstupů v čase t (z toho, co momentálně zaznamenávají senzory) a z hodnoty akčních vstupů z času $t - 1$ (z předchozí agentovy akce). Hodnota odměny je navíc v každém kroku podělena speciálním koeficientem, jak je uvedeno v následující části.

Neuronová síť použitá pro řízení robota je na obrázku 5.27.

[5]Podobně jako v experimentu s dravcem a kořistí, i zde se osvědčilo použít více výstupních neuronů. Kdybychom pro každý z motorů použili pouze jeden výstupní neuron, agent by se velmi těžko učil vykonávat akce, při kterých zůstává jedno z kol nehybné.



Obrázek 5.27: **Neuronová síť pro experiment s robotem.** *Neuronové síť má 4 vstupní neurony pro stav, 4 vstupní neurony pro akci, 8 skrytých neuronů a jeden výstup predikující kumulativní odměnu.*

5.2.2 Specifika úlohy

Popsaná úloha je netriviální a je stejně jako úloha popsaná v 5.1 nemarkovská. Navíc, stejně jako předchozí úloha i tato vyžaduje od algoritmu schopnost naučit se získávat opožděné odměny. Taková situace nastává například, pokud se před agentem nachází nějaká překážka, kterou je třeba objet. Během zatáčení se totiž kola točí proti sobě, což snižuje agentovu okamžitou odměnu. Ta přijde ale později, poté co se agentovi podaří vzdálit se od překážky. Uvědomme si, že řešením takové situace pro robota řízeného popsanou sítí není couvat zpět. To by sice přineslo odměnu okamžitě (agent se pohybuje a navíc se vzdaluje od překážky), ale dostává se do stejného stavu, v jakém byl předtím, než se dostal do blízkosti překážky (nemá paměť). V tomto stavu pak vykoná znovu tu stejnou akci a opět se dostane do blízkosti překážky. Takovéto chování, kdy se robot opakovaně přibližoval k překážce a opět od ní vzdaloval, bylo v průběhu experimentů několikrát přechodně pozorováno.

Ukázalo se také, že tato úloha klade vyšší nároky na explorativní schopnost algoritmu, než úloha z části 5.1. Odměna v předchozí úloze byla typicky opožděna nejvýše o jeden časový krok, protože agentovi stačilo se pohnout o jedno pole, aby našel nový zdroj potravy. Proto k objevení cesty, jak tuto odměnu získat, stačilo jen slabé znáhodnění agentova chování. Naproti tomu je v této úloze někdy třeba k tomu, aby se robot vzdálil od překážky, vykonat delší posloupnost kroků, které odměnu nepřinášejí. Modifikovaný algoritmus je schopen se takovouto posloupnost kroků naučit, ovšem pouze v případě, že bude disponovat dostatečnou schopností prozkoumávat nové možnosti.

Uvažme případ, že by algoritmus agentovi tuto schopnost nedával. Potom by se agent dokázal pouze naučit to, že rychlý pohyb rovným směrem přináší odměnu. Snažil by se tedy stále rovně pohybovat a velmi brzo by narazil na překážku. Jelikož nikdy předtím žádnou překážku úspěšně neobjel, neměl příležitost se naučit, že taková posloupnost akcí

přinese v budoucnu odměnu. Proto se o to ani nepokusí. Nastal by tak problém "strachu z neznáma" diskutovaný v 4.4.3.

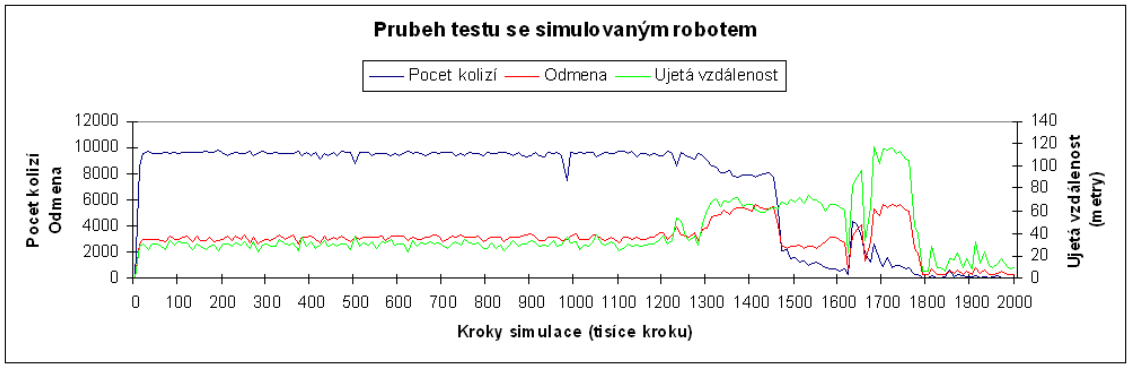
V algoritmu bylo provedeno několik drobných modifikací, abychom tento problém omezili. Přestože v předchozí úloze bylo úspěšnější znáhodnění číslo 4 popsané v části 4.4.3, použili jsme zde metodu znáhodnění 2. Tato metoda dává agentovi větší explorativní schopnosti, neboť agent občasně vykonává zcela náhodnou akci. Pravděpodobnost vykonání náhodné akce byla zvýšena na 10%. Povolené hodnoty každého neuronu pro akci byly 0, 0.5 a 1. Nastavili jsme také poměrně velkou délku časového kroku, za který může robot změnit rychlost pohybu svých kol - 256 ms. Díky tomu má jeden náhodný tah delší účinnost. Aby se nestávalo, že se účinek náhodného otočení na jednu stranu vyruší následným náhodným otočením robota na opačnou stranu, generujeme pouze takové náhodné akce, při kterých se levé kolo točí vpřed více než kolo pravé (platí $y_{A_0} - y_{A_1} \geq y_{A_2} - y_{A_3}$). Tím jsme na jednu stranu omezili agentovu schopnost prozkoumávat důsledky rotace vlevo, na druhou stranu posílili schopnost poznat, jaký výsledek přináší rotace vpravo. Jelikož otáčením vpravo se robot může dostat do každé pozice, do které se lze dostat otáčením vlevo, nemělo by toto omezení působit přílišnou újmu.

Další odlišností této úlohy oproti úloze předchozí je fakt, že tato úloha se neskládá z jednotlivých oddělených simulací. Budeme tedy v algoritmu 4 místo vzorců 4.14 a 4.15 používat vzorce 4.18 a 4.19 tak, jak je popsáno na konci části 4.4.2, aby nedosahovala kumulativní odměna nekonečných hodnot. Parametr γ bude mít hodnotu 0.9. Vzorce 4.16 a 4.17 pro poslední učící krok nebudou použity vůbec. Hodnota okamžité odměny bude v každém kroku podělena číslem 10, aby síť nebyla učením nucena vybavovat výstup o hodnotě vyšší než 1.

5.2.3 Průběh experimentu s robotem

Simulace a učení bude probíhat nepřetržitě. Po každých 200 krocích simulace je čtvercová překážka velikosti 20 × 20 cm náhodně přemístěna (tak, aby nekolidovala s robotem), a robot je náhodně pootočen, ale zůstává na stejném místě. Robota pootáčíme, abychom zamezili tomu, že v průběhu učení uvízne v nějaké pozici (například v rohu ohrádky), ze které by se těžko dostával. Za každých 2000 kroků simulace změříme celkovou získanou odměnu, celkovou ujetou vzdálenost a celkový počet robotových nárazů. Použitá neuronová síť bude mít jednu skrytou vrstvu o osmi neuronech, nastavení parametrů bylo $\alpha = 0.5$, $\lambda = 0.75$.

Vývoj experimentu zachycuje obrázek 5.28. Během prvního tisíce kroků simulace se robot po určitém váhání naučil jezdit rovně maximální rychlostí. Aspektem komplikujícím se naučit přímou jízdu mohlo být to, že některý náhodný tah naznačil neuronové síti, že pohyb kupředu přináší odměnu, a jiný naopak, že odměnu přináší pohyb vzad. Výsledné úpravy vah sítě se tím mohly částečně rušit a učení prodloužit. Zajímavé je, že se robot naučil jezdit pozadu. Důvodem je, že robotova interní odměňovací funkce 5.5 identifikuje kolizi pomocí aktivací infračervených senzorů ve členu 5.8. Jelikož je robot senzory hustěji osazen vepředu, pociťoval agent kolize při pohybu zpět jako mírnější trest. Robot se tak pohyboval kupředu, dokud nenarazil na stěnu, podél které klouzal až do rohu ohrádky, kde vždy uvízl až do doby, než byl náhodně otočen vnějším zásahem. Robot sám se jen občas lehce pootočil vpravo (respektive ve směru hodinových ručiček). Preference rotace



Obrázek 5.28: **Průběh experimentu se simulovaným robotem**

tímto směrem je patrně způsobena tím, že náhodné kroky, která síť mohla pozorovat, měly vždy rotaci tímto směrem.

Toto chování robotu zůstalo velmi dlouho, cca až do 1.3-miliontého kroku učení. Poté se jeho chování lehce změnilo tak, že stále jezdil plnou rychlostí vzad, ale když se ocitnul v rohu, otočil se a zamířil do středu prostoru. Někdy se také dokázal odvrátit od rovné zdi. Na grafu 5.28 je patrné zvýšení odměny i ujeté vzdálenosti, a naopak snížení počtu kolizí. K další změně chování došlo po cca 1.46 milionech kroků simulace. Robot se naučil jezdit opatrněji, poloviční rychlostí, a po nárazu do překážky se otočil vpravo a překážce se vyhnul. Schopnost vyhýbat se překážkám se postupně zdokonalovala až se robot naučil jím vyhýbat ještě před kolizí, tak, že couvnul, přičemž se pootočil ve směru hodinových ručiček. Na grafu 5.28 je tato změna opět dobře patrná. Kolizí ubývá a ujetá vzdálenost zůstává přibližně stejná, neboť robot se na jednu stranu nezasekává o překážky, ale na druhou stranu se pohybuje poloviční rychlostí. Pozoruhodné ale je, že hodnota získané odměny poklesla. To je způsobeno nepřesností použité interní odměňovací funkce 5.5, která určuje robotovu rychlost dle nastavené rychlosti kol, a proto nedostatečně trestá náraz plnou zpětnou rychlostí do překážky.

Toto robotovo úspěšné, i když příliš opatrné chování trvalo přibližně do 1.6-miliontého kroku. Poté se síť dostala do nestabilního stavu a robotovo chování se velmi rychle měnilo. Nějaký čas se otáčel na místě. Po dalších cca 20000 krocích se začal rychle pohybovat směrem vpřed přičemž zpočátku hodně boural, ale brzy se naučil překážkám vyhýbat rotací vpravo, později opět i s malým couvnutím zpět. Zajímavé je, že v této době se robot někdy, především za nepřítomnosti překážky, stáčel i proti směru hodinových ručiček. Poté se znovu začal chovat proměnlivě. To skončilo po cca 1.68-miliontém kroku, kdy se robot opět začal pohybovat zpět, tentokrát plnou rychlostí, přičemž se úspěšně vyhýbal kolizím s překážkami. Našel tedy téměř ideální řešení. Po přibližně 1.77-miliontém kroku se robot náhle přestal pohybovat a dále již nejevil známky učení se.

Konfigurace vyvíjející se neuronové sítě byly průběžně ukládány a nejúspěšnější z nich byly později otestovány na fyzickém robotu e-puck (obrázek 5.29), kde též fungovaly. Robot se i zde dokázal úspěšně vyhýbat překážkám.

Na přiloženém CD jsou uloženy videa a fotografie experimentu na reálném i simulovaném robotu, použité zdrojové kódy i všechny historické konfigurace neuronové sítě z průběhu experimentu.

Obrázek 5.29: Experiment s fyzickým robotem e-puck - robot se pohybuje ve vymezeném prostoru, přičemž se úspěšně vyhýbá překážkám.

Kapitola 6

Meze zpětnovazebného učení

V této části (nepříliš formálně) srovnáme úlohu zpětnovazebného učení s prohledáváním stavového prostoru. Díky tomu budeme moci navrhnout určité doporučení pro stanovení vhodné odměňovací funkce a také vyvodíme některé obecné závěry o mezích praktického použití zpětnovazebného učení.

6.1 Zpětnovazebné učení a stavový prostor

Formalismus zpětnovazebného učení má poměrně blízko k formalismu prohledávání stavového prostoru [10], který bývá často používán například při řešení her. Podobně jako v úloze zpětnovazebného učení (viz část 2.1), i zde je dána konečná množina stavů $\mathcal{S}$ a konečná množina akcí (operátorů) $\mathcal{A}$, z nichž každá je parciálním zobrazením z množiny stavů $\mathcal{S}$ do množiny stavů $\mathcal{S}$. Tedy, podobně jako při zpětnovazebném učení, akce vede z jednoho stavu do druhého stavu, přičemž ne všechny akce musí být aplikovatelné v každém stavu. Dále je dán počáteční stav $\int_0$ a množina koncových stavů $\mathcal{Z} \subseteq \mathcal{A}$. Cílem úlohy je najít posloupnost akcí vedoucí ze stavu $\int_0$ do libovolného z koncových stavů, případně najít nejkratší takovou posloupnost akcí.

Úlohu prohledávání stavového prostoru lze přímočaře převést na úlohu zpětnovazebného učení. Množina stavů, množina akcí i počáteční stav zůstane beze změny a odměňovací funkci definujeme takto:

$$r(s) = \begin{cases} 1 & : \quad s \in \mathcal{Z} \\ -1 & : \quad s \notin \mathcal{Z} \end{cases} \tag{6.1}$$

Maximalizací získané kumulativní odměny bychom pak donutili agenta, aby co nejkratší cestou dorazil do koncového stavu, a přestal tak získávat záporné odměny. Povšimněme si, že v obecném případě je úloha zpětnovazebného učení tak, jak jsme ji definovali v 2.1, složitější, než úloha prohledávání stavového prostoru. V případě prohledávání stavového prostoru máme zaručenu konečnost množiny stavů i množiny akcí. Prostředí se chová deterministicky, markovsky a navíc zákony jeho dynamiky jsou přesně známé (Agent se nemusí učit, jaké mají jeho akce dopad na prostředí.). V případě prohledávání stavového prostoru předpokládáme typicky offline učení. Úloha prohledávání stavového prostoru je vyřešena, když je dosažen cílový stav, zatímco při zpětnovazebném učení nemusí koncový

stav existovat a činnost agenta nemusí být časově omezená. Tyto předpoklady umožňují k řešení úlohy prohledávání stavového prostoru používat algoritmy, jako je například prohledávání do hloubky (backtracking) nebo prohledávání do šířky.

Lze tedy říci, že prohledávání stavového prostoru je speciálním případem úlohy zpětnovazebného učení. Úlohu prohledávání stavového prostoru umíme v obecném případě pomocí prohledávacích algoritmů řešit, to ale může být prakticky neschůdné v případě příliš velké množiny $\mathcal{S}$. Úloha zpětnovazebného učení je v obecném případě ještě složitější a například pro nekonečně velkou množinu S nemusíme být vůbec schopni nalézt takovou strategii, aby agent dosáhnul maximální odměny. Přístup zpětnovazebného učení se ale oproti prohledávání stavového prostoru liší i v tom, že často netrváme na nalezení optimálního řešení - strategie která agentovi získá maximální kumulativní odměnu, ale spokojíme se s řešením suboptimálním - strategií, která získá "dostatečně velkou" kumulativní odměnu.

Přestože úloha zpětnovazebného učení je v obecném případě efektivně neřešitelná, úlohy, se kterými se setkáme v praxi, nemusí být takto složité. Faktory, které mohou řešení úlohy usnadnit, jsou především:

1. Použitý způsob odměňování
2. Složitost pravidel dynamiky prostředí

Tyto faktory jsou rozebrány v následujících dvou částech.

6.2 Jak stanovit odměnu

Dle naší definice úlohy zpětnovazebného učení je již mechanismus odměňování pevně součástí zadání. V reálné situaci však obvykle mámě nějaký cíl, kterého chceme dosáhnout, a pro tento účel se snažíme odměňovací funkci nadefinovat co nejvhodněji tak, aby se maximalizací celkové odměny agent blížil dosažení cíle. Takto například v části 5.1 bylo naším cílem, aby agent v roli kořisti přežil co nejdéle, a v části 5.2, aby se robot rychle pohyboval, aniž by narážel do překážek. Odměňovací funkci jsme zkoušeli definovat tak, aby agenta vedla k dosažení tohoto cíle. Jiným příkladem jsou živočichové, kteří se zřejmě v biologické evoluci vyvinuli tak, že jejich cílem je přežití a reprodukce, a jejich mozky je odměňují za akce, které k tomuto cíli vedou. Vhodná definice odměňovacího mechanismu je zásadním faktorem ovlivňujícím, zda použití zpětnovazebného učení bude úspěšné. Přesto této problematice nebývá v literatuře věnováno příliš pozornosti.

Předpokládejme opět, že úlohu máme zadánu ve formě prohledávání stavového prostoru. Dána je množina stavů $\mathcal{S}$, množina akcí $\mathcal{A}$, počáteční stav $\int_o$ a množina koncových stavů $\mathcal{Z}$. Prostředí je tedy deterministické, markovské, jeho dynamika je známá a počet možných stavů a akcí je konečný. V předchozí části definujeme formulí 6.1 jednu možnou variantu odměňování, která, pokud bude odpovídající úloha zpětnovazebného učení optimálně vyřešena, povede též k vyřešení původního problému prohledávání stavového prostoru. Navrhovaná odměňovací funkce ale agentovi neposkytuje žádnou užitečnou informaci o tom, zda jeho jednotlivé akce vedou k cíli či nikoliv. Pouze ho informuje o jeho

úspěchu až v případě nalezení řešení. K tomu může vést ale velmi dlouhá sekvence agentových akcí, kterou může být těžké nalézt. To bude klást příliš vysoké nároky na použitý algoritmus zpětnovazebného učení, který se bude muset potýkat s velmi komplikovaným problémem opožděných odměn popsaným v 4.2. Vhodnou úpravou odměňovací funkce může být ale možné tento problém omezit až eliminovat.

Předpokládejme, že používáme algoritmus zpětnovazebného učení, který v každém kroku vybírá tu akci, která přinese nejvyšší okamžitou odměnu[1]. Tento algoritmus nazvěme hladovým. Díky uvedeným předpokladům na vlastnosti prostředí nebude těžké hladový algoritmus implementovat. Hladový algoritmus je primitivní v tom smyslu, že nedává agentovi vůbec žádnou schopnost získávat opožděné odměny. Pokud by byl použit s odměnou 6.1, výsledky by byly velmi špatné. Přesto je hypoteticky možné zvolit odměňovací funkci r tak, aby tento algoritmus dokázal úlohu optimálně řešit.

Pro $\int \in S$ definujme $\mathcal{D}(\int)$ jako nejnižší možný počet akcí vedoucí ze stavu $\int$ do libovolného koncového stavu. Pokud neexistuje cesta z $\int$ do žádného koncového stavu, položíme $\mathcal{D}(\int) = \infty$ (nebo dostatečně vysoké reálné číslo).

Odměňovací funkci potom definujeme takto:

$$r(\int) = -\mathcal{D}(\int) \tag{6.2}$$

Je zřejmé, že při takovéto odměňovací funkci najde hladový algoritmus strategii, která maximalizuje kumulativní odměnu, a tím také najde nejkratší posloupnost akcí vedoucí z počátečního do koncového stavu pro odpovídající úlohu prohledávání stavového prostoru. Abychom dosáhli přirozeného vztahu, že hodnota dosud získané kumulativní odměny bude přímo úměrná (s koeficientem -1) vzdálenosti, která zbývá k dosažení cíle v odpovídající úloze prohledávání stavového prostoru, a zároveň zaručili další příjemnou vlastnost, že hodnota okamžité odměny v průběhu činnosti hladového algoritmu neustále neporoste, ale bude rovna jedné, upravíme 6.2 na:

$$r(\int_t) = \mathcal{D}(\int_{t-1}) - \mathcal{D}(\int_t) = -\Delta\mathcal{D}(\int_t) \tag{6.3}$$

Okamžitá odměna tak bude odpovídat úbytku vzdálenosti od nejbližšího koncového stavu v prostoru stavů[2] [3]. Při takovéto odměňovací funkci bude hladový algoritmus fungovat stejně dobře jako při odměňovací funkci 6.2. Uvědomme si, že v tomto případě ale není okamžitá odměna funkcí stavu, ale přechodu mezi stavy (viz poznámka 2 z části 2.1). Díky tomu, že při daných předpokladech a odměňovací funkci 6.2, respektive 6.3 dosahuje optimálních výsledků i nejprimitivnější hladový algoritmus, lze tyto odměňovací funkce považovat za ideální.

Odměňovací funkce 6.2 a 6.3 jsou narozdíl od odměňovací funkce 6.1 definovány tak, že okamžitá odměna také velmi dobře agentovi napovídá, jak získat maximální kumulativní odměnu. Lze očekávat, že i pro takové úlohy zpětnovazebného učení, které nebyly

[1]Použití tohoto algoritmu pří řešení úlohy zpětnovazebného učení odvozené z úlohy prohledávání stavového prostoru s odměňovací funkcí r bude analogické použití tzv. hill-climbing algoritmu[10] na původní úloze prohledávání stavového prostoru používající r coby hodnotící funkci stavů.

[2]Uvedená odměňovací funkce tedy neodměňuje za dosažený stav ale za zlepšení. Tuto situaci lze volně připodobnit k člověku, který nemá potěšení z toho, co má, a smutek z toho, co nemá, ale potěšení z toho, co získal, a smutek z toho, co ztratil.

[3]Pokud bychom uvažovali spojitý čas, okamžitá odměna by byla derivací kumulativní odměny podle času a psali bychom $r(\int_t) = -\frac{\partial\mathcal{D}(\int)}{\partial t}$.

odvozeny od prohledávání stavového prostoru, a při použití jiného než hladového algoritmu budou výsledky pro odměňovací funkce s touto vlastností lepší. Odměňovací funkce s touto vlastností agentovi v případě, že mu dynamika světa není předem známa, usnadní postupné zdokonalování. Lze tedy v tomto smyslu hovořit o určité kvalitě odměňovací funkce. Tento princip podporují též experimenty s různými odměňovacími funkcemi popsané v části 5.1. Výsledky experimentů při použití tamtéž popsané odměňovací funkce 1 dokládají, že tato odměňovací funkce není kvalitní (Je to obdoba zde popsané odměňovací funkce 6.1.), neboť úspěšnost při použití této odměňovací funkce byla vždy nízká. Naproti tomu s odměňovacími funkcemi 2 a 3 byly dosaženy podstatně lepší výsledky, přičemž v některých případech byla výhodnější odměna číslo 2 a v jiných naopak odměna číslo 3.

Méně kvalitní odměňovací funkce klade vyšší nároky na schopnost použitého algoritmu zpětnovazebného učení získávat opožděné odměny. Pokud bychom místo hladového algoritmu například použili takový algoritmus, při kterém agent vždy vykoná dvojici po sobě následujících akcí, které v součtu přinesou nejvyšší možnou odměnu ze všech možných dvojic akcí proveditelných v daném stavu, pravděpodobně by k dosažení stejně dobrých výsledků stačila méně kvalitní odměna. Kdybychom použili algoritmus TD(λ) popisovaný v 4.4, stačila by nám pravděpodobně i méně kvalitní odměňovací funkce. Návrh kvalitnější odměňovací funkce však vyžaduje od jejího tvůrce vyšší znalost řešené úlohy. Proto nebude obvykle v praxi možné definovat odměňovací funkci tak, aby i hladový algoritmus dosahoval optimálních výsledků tak, jako tomu je u odměňovací funkce 6.3. Jedná se tedy o určitý kompromis mezi kvalitou odměňovací funkce a nároky na algoritmus zpětnovazebného učení, který je třeba hledat zvlášť pro každou konkrétní úlohu.

6.3 Složitost dynamiky prostředí

Druhým faktorem, který má zásadní dopad na složitost řešení určité úlohy zpětnovazebného učení, je složitost zákonů, kterými se řídí prostředí. V předchozí části jsme naznačili, že při kvalitní odměňovací funkci může k řešení úlohy stačit poměrně jednoduchý algoritmus. K tomu jsme ale předpokládali, že prostředí je konečné a chová se deterministicky, markovsky, a že máme dánu úplnou znalost o tom, jak se toto prostředí chová. Ve mnoha úlohách zpětnovazebného učení, například v úlohách robotiky, tyto předpoklady ale splněny nejsou. V takové situaci mohou být velmi užitečné právě neuronové sítě. V části 4.3 se zabýváme možnostmi, jak lze neuronovou síť naučit kompaktně reprezentovat zákonitosti prostředí. I tyto možnosti ale mají své meze. Při vymezení úloh zpětnovazebného učení jsme připustili prostředí, která se mohou řídit zcela jakýmikoli zákonitostmi, tedy i taková prostředí, jejichž reprezentaci nedokážeme žádným algoritmem neuronovou síť efektivně naučit. Navíc, jak bylo dokázáno v 4.3.2, i při dané síti, která dokonale predikuje okamžitou odměnu, je nalezení akce, která ji maximalizuje, obecně NP-těžké. Je tedy velice pravděpodobné, že existují natolik složité úlohy zpětnovazebného učení, že je prakticky řešit nedokážeme.

Například lidské bytosti jsou ale schopny v průběhu života pochopit poměrně složité přírodní zákony a využívat je k tvorbě komplikovaných plánů akcí a lze předpokládat, že podstatná část tohoto učení se odehrává procesem podobným zpětnovazebného učení. Známé filosofické pravidlo Occamovy břitvy říká, že bychom měli dávat přednost tomu nejjednoduššímu vysvětlení pozorovaných jevů. Jinými slovy, dynamika prostředí, se kterými

se v praxi budeme setkávat, může být často jednodušší, než v obecném případě. Proto lze očekávat, že při úlohách z reálného světa dokáží umělé neuronové sítě se svojí schopností generalizace vytvořit použitelný model reprezentující zákonitosti prostředí. Tento model je možné využít k naplánování akcí, které povedou k získání i do jisté míry opožděných odměn. Spíše než s řešením přinášejícím maximální kumulativní odměnu se ale v případě komplikovaných úloh budeme muset spokojit se strategií, která přinese "dostatečně vysokou" kumulativní odměnu, tedy se suboptimálním řešením.

Je třeba uvážit, že pokud lidské bytosti používají ke svému učení nějaký druh algoritmu zpětnovazebného učení, tento algoritmus se musel vyvíjet v evoluci miliony let. Kdyby existoval nějaký jednoduchý algoritmus, který by byl použitelný pro inteligentní řešení problémů reálného světa, zrodily by se v evoluci pravděpodobně inteligentní živočichové mnohem dříve. Navíc chování lidí stejně jako ostatních živočichů z velké části určují též vrozené vzorce chování. Přehnaná očekávání možností řešit komplikované problémy vyžadující inteligentní uvažování a tvorbu plánů pouze pomocí vhodného algoritmu zpětnovazebného učení tedy nejsou na místě. V tomto ohledu je ale velmi zajímavá možnost kombinace zpětnovazebného učení s evolučními algoritmy.

Kapitola 7

Závěr

V úvodu této práce byl vymezen přístup zpětnovazebného učení a naznačen možný přínos metody zpětnovazebného učení k řešení reálných problémů. Byly formálně definovány pojmy používané v oblasti zpětnovazebného učení a též zavedeny definice týkající se neuronových sítí. Popsali jsme odlišnosti klasické neuronové sítě pro učení s učitelem a neuronové sítě určené pro zpětnovazebné učení. Diskutován byl přínos neuronových sítí modelujících zákony prostředí a bylo též formálně dokázáno, že pro danou síť prahových hodnot modelující odměnu je nalezení akce maximalizující tuto odměnu NP-úplný problém. Poté byly detailně popsány tři konkrétní vybrané modely neuronové sítě. První z nich je poměrně primitivní neuronová síť zdokonalující se vhodnými náhodnými změnami, která byla autorem navrhnuta spíše jen pro demonstrační účely. Druhým z modelů je námi navržená modifikace Suttonova algoritmu TD(λ), která úspěšně rozšiřuje doménu použití původního algoritmu i na nemarkovská nedeterministická prostředí s agentovi neznámou dynamikou a potenciálně nekonečnou množinou možných stavů a akcí. Posledním popsaným modelem byla dvojitá rekurentní síť Juergena Schmidhubera, která narozdíl od předchozích poskytuje agentovi paměť. Všechny uvedené modely byly porovnány pomocí experimentu se simulátorem dravce a kořisti, nejúspěšnější z nich - modifikované TD(λ) bylo též otestováno v robotickém experimentu pomocí simulátoru Webots, nejprve na simulovaném, poté na fyzickém robotu e-puck.

Experimenty prokázaly vhodnost přístupu zpětnovazebného učení k řešení některých složitých úloh umělé inteligence. Experimenty potvrdily, že velmi zásadním kriteriem úspěšnosti algoritmu v případě složitějších úloh je jeho schopnost získávat opožděné odměny, tedy odměny, které agent získá teprve až s určitým časovým odstupem od vykonání první z akcí, která způsobuje získání této odměny. Ukázalo se, jak důležité je v tomto ohledu vybavit algoritmus schopností explorace, tj. schopností vykonávat dosud nevyzkoušené sekvence akcí. Dále experimenty potvrdily, že vhodný způsob odměňování může tento problém velmi usnadnit. V neposlední řadě také experimenty naznačily, že řešení problému může být usnadněno vhodným kódováním agentových vstupů a výstupů.

Nakonec byl porovnán přístup zpětnovazebného učení s přístupem prohledávání stavového prostoru. Z tohoto srovnání byla vyvozena určitá doporučení vhodného způsobu odměňování a některé spekulativní závěry o mezích použití zpětnovazebného učení na reálné úlohy.

Popisované modely neuronových sítí nejsou zcela biologicky opodstatněné a netvrdíme, že tyto konkrétní procesy probíhají v nervové soustavě živých organismů. Ani modifiko-

vaný algoritmus TD(λ), ani Schmidhuberův algoritmus pro rekurentní síť nejsou lokální v prostoru, speciálně k učení používají výpočet parciálních derivací. Není pravděpodobné, že by takové výpočty probíhaly v nervové soustavě živých organismů. Dalším rozdílem oproti činnosti nervové soustavy vyšších živočichů je to, že námi popsané neuronové sítě mají speciální a pevnou architekturu, zatímco architektura přirozených neuronových sítí je velmi komplikovaná a proměnlivá - dovoluje vznik a zánik nových synapsí i neuronů v průběhu života. Otázkou je, zda a případně jak by tyto odlišnosti změnily výpočetní schopnosti umělých neuronových sítí.

Lze očekávat, že bychom mohli dosáhnout ještě lepších výsledků zkombinováním výhod dvou popsaných algoritmů - schopnosti algoritmu TD(λ) předvídat opožděné odměny a paměti Schmidhuberovy rekurentní sítě.

Zde popisované algoritmy jsou jen výběrem z mnoha možných technik, jak lze aplikovat neuronové sítě ve zpětnovazebném učení. Řada dalších, které by mohly být v tomto směru přínosné, je již mimo rozsah této práce. Mezi jinými jsou to například různé kompetiční sítě inspirované teorií neurálního darwinismu G. Edelmana[4] nebo Schmidhuberova Long Short-Term Memory[5], která údajně dosahuje velmi dobrých výsledků při sledování opožděných odměn. Dále zmíníme Suttonovo Adaptive Heuristic Critics[1], sítě s tzv. spiking neurony a především různé možnosti kombinace neuronových sítí s genetickými algoritmy[6]. Genetické algoritmy mohou být například použity k vývoji vhodné odměňovací funkce, architektury sítě, nebo k částečnému naučení sítě, která bude doučena pomocí zpětnovazebného učení.

Seznam obrázků

Seznam tabulek

Literatura

[1] Barto A. G., Sutton R. S., Anderson C. W. (1983): Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems Man and Cybernetics* 13, 834-846.

[2] Bertsekas D. P., Tsitsiklis J. N. (1995): Neuro-Dynamic Programming: An overview. *Proceedings of the 34th IEEE Conference on Decision and Control* , 560-564.

[3] Civín L. (2006): Vrstevnaté neuronové sítě a jejich aplikace při dobývání znalostí. Diplomová práce, Matematicko-fyzikální fakulta Univerzity Karlovy v Praze.

[4] Edelman G. M. (1978): The mindful brain. VB Mountcastle, MIT.

[5] Hochreiter J., Schmidhuber J. (1997): Long Short-Term Memory. *Neural Computation*, 9(8), 1735-1780.

[6] Holland J. H. (1975): Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor.

[7] Hornik K., Stinchcombe M., White H. (1989): Multilayer feedforward networks are universal approximators. *Neural Networks 2*, 359-366.

[8] Hutter M. (2001): General loss bounds for universal sequence prediction. *Proceedings of the 18th International Conference on Machine Learning (ICML-2001).*

[9] Judd J. S. (1988): Neural Network Design and the Complexity of Learning. Dissertation, Computer Science Department, California Institute of Technology, USA.

[10] Mařík V., Štěpánková O., Lažanský J. (1993): Umělá inteligence (1), kapitola 2. Praha, Academia.

[11] Munro P. W. (1987): A dual back-propagation scheme for scalar reinforcement learning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, Seatle, WA.

[12] Robinson A. J., Fallside F. (1987): Static and dynamic error propagation networks with application to speech encoding. *Proceedings of Neural Information Processing Systems*, American Institute of Physics.

[13] Rumelhart D. E., Hinton G. E., Williams R. J. (1986): Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructure of cognition*, volume I, 318-362.

[14] Schmidhuber J. H. (1989): A Local Learning Algorithm for Dynamic Feedforward and Recurrent Networks. *Connection Science*, 1(4), 403-412.

[15] Schmidhuber J. H. (1990): Making the world differentiable: On using supervised learning fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical Report FKI-126-90 (revised), Insitut für Informatik, Technische Universität München.

[16] Schmidhuber J. H. (1991): A possibility for implementing curiosity and boredom in model-building neural controllers. *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats* 222-227, MIT Press/Bradford Books.

[17] Schultz W., Dayan P., Montague P. (1997): A neural substrate of prediction and reward. *Science 275* , 1593-1599.

[18] Sutton R. S. (1988): Learning to predict by the methods of temporal differences. *Machine learning 3* , 9-44.

[19] Tesauro G. (1995): Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, Vol. 38, No. 3.

[20] Watkins C. J. C. H. (1989): Learning from Delayed Rewards. PhD thesis, Cambridge University, Cambridge, England.

[21] Williams R. J., Zipser D. (1989): A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural computation 1* , 270-280.

Dodatek A

Knihovna neuronových sítí Flexinets

Součástí diplomové práce je i knihovna objektů a metod pro práci s neuronovými sítěmi Flexinets. V této knihovně byly implementovány všechny výše popsané modely neuronových sítí (síť zpětného šíření, síť s náhodnými změnami, modifikované TD(λ) a Schmidhuberova rekurentní síť). Všechny experimenty popsané v experimentální části práce byly provedeny s využitím knihovny Flexinets. Knihovna umožňuje poměrně jednoduchými úpravami přidávat nové modely neuronových sítí nebo rozšiřovat funkcionalitu stávajících modelů.

A.1 Principy knihovny Flexinets

Knihovna byla napsána v C++ s použitím Microsoft Visual C++ 6.0 a je objektově orientována. Skládá se ze dvou souborů - neurons.h a neurons.cpp. První z nich je hlavičkový soubor obsahující deklarace všech tříd a položek, druhý z nich obsahuje těla všech metod. V těchto souborech jsou popsány postupně všechny implementované modely neuronových sítí. Byla použita konvence, že jména objektů a dalších typů začínají velkým písmenem, jména instancí objektů, proměnných a položek objektů písmenem malým a jména konstant preprocesoru jsou psána celá velkým písmem. Oba soubory jsou podrobně okomentovány (v Angličtině), přičemž v neurons.h lze nalézt podrobněji okomentované parametry metod, zatímco v neurons.cpp lze nalézt i komentáře dílčích výpočtů a příkazů, které to vyžadují.

Knihovna Flexinets definuje třídu `Synapse` pro spojení mezi neurony, třídu `Neuron` pro obecný neuron a třídu `NNetwork` pro obecnou neuronovou síť. Obecný neuron a obecná neuronová síť jsou vybaveny datovými položkami a metodami, které jsou potřeba pro práci s většinou různých modelů neuronových sítí. Konkrétní modely neuronových sítí jsou pak v knihovně přidány vytvořením potomků třídy `NNetwork` a třídy `Neuron`, které nesou datové položky a metody charakteristické pro ten který model. Knihovna ve velké míře využívá standardní šablony jazyka C++ (tzv. STL - Standart templates library) pro práci se seznamy neuronů, seznamy synapsí apod.. Vnitřně zachází knihovna s Neurony především pomocí ukazatelů, ale veřejné metody určené pro uživatelské použití v externích programech odkazují na neurony pomocí jejich indexů v síti. Popíšeme zde stručně všechny v současnosti implementované třídy knihovny Flexinets. Omezíme se přitom na popis jejich datových položek, který vystihuje základní funkcionalitu těchto tříd. Jejich metod je

mnoho, a proto čtenáře odkazujeme na jejich okomentovaný kód v knihovně. Interní metody knihovny Flexinets jsou typicky deklarovány jako chráněné (protected nebo private), metody, které bývají volány z externího programu, jako veřejné. Datové položky tříd jsou vesměs chráněné.

A.2 Třídy knihovny Flexinets

`Class Synapse`
Tato třída popisuje spojení mezi dvěma neurony. Požadavky na podobu synapse se mezi implementovanými modely příliš neliší, a proto v současnosti není implementován žádný potomek této třídy a tato třída je používána v každém modelu přímo. Jelikož je seznam synapsí položkou neuronu, do kterého synapse vstupuje, není třeba, aby synapse obsahovala položku pro její cílový neuron. Obsahuje tedy pouze ukazatel na zdrojový neuron `Neuron* source` a aktuální hodnotu váhy `float weight`. Některé algoritmy (například ty, které používají moment učení) potřebují znát též předchozí hodnotu váhy. Ta je uložena v datové položce `float oldWeight`.

`Class Neuron`
Třída Neuron reprezentuje umělý neuron s jeho základními položkami. Model sítě s náhodnými změnami vah popsaný v 4.2 používá tuto třídu přímo, v ostatních modelech pracujeme s potomky této třídy. Třída je vybavena ukazatelem na přenosovou funkci `static float(* tFunction)(float)`, což umožňuje pracovat s libovolnou přenosovou funkcí (Typicky používáme sigmoidu.) Protože `tFunction` je statická položka, všechny neurony mají stejnou přenosovou funkci. Další statickou položkou je `float maxRandomWeight`, která udává maximální absolutní hodnotu prahů a vah při náhodné inicializaci. Položka `ConnectionList inputs` je seznam vstupních synapsí neuronu, `float bias` a `float oldBias` obsahují aktuální, respektive předchozí hodnoty prahu. Položky `float outputVal` a `float oldVal` pak obsahují aktuální a předchozí výstup neuronu.

`Class NNetwork`
Tato třída představuje obecnou neuronovou síť. Jejími datovými položkami je seznam ukazatelů na všechny neurony sítě `PtrList neurons`, jeho podmnožina ukazující pouze na vstupní neurony `PtrList netInputs` a jeho podmnožina ukazující pouze na výstupní neurony `PtrList netOutputs`. Neuron této sítě bude obvykle instance třídy `Neuron`. Model sítě s náhodnými změnami vah popsaný v 4.2 používá tuto třídu přímo, v ostatních modelech pracujeme s potomky této třídy. Třída `NNetwork` je vybavena řadou metod manipulujících s neurony sítě, které využijeme v různých algoritmech pracujících s neuronovými sítěmi.

`Class BPNeuron, Class BPNetwork`
Tyto třídy definují neuron a síť neuronů pro algoritmus 1 - backpropagation popsaný v 3.2. Třída `BPNeuron` je oproti třídě `Neuron` obohacena o položku `double delta`, která obsahuje hodnotu chybového členu daného neuronu, ze kterého se počítají parciální derivace výstupů sítě podle jednotlivých vah. Hlavní metodou třídy `BPNetwork`, která vykoná jednu smyčku algoritmu zpětného šíření, je `void teach(float* input, float* desired, float alpha, float moment)`. Aby metody třídy `BPNetwork` správně fungovaly, je nutné, aby neurony sítě byly seřazeny dle vah, počínaje vstupní vrstvou a výstupní

vrstvou konče. Pokud síť vytvoříme metodou třídy NNetwork `void createLayers(int layersNum, ...)`, jsou tyto podmínky zaručeny.

`Class TDNetwork`

Tato třída implementuje modifikovaný algoritmus TD(λ) se třemi druhy znáhodnění používanými v popsaných experimentech. Zmiňovaný druh znáhodnění 1 pro offline učení, který v experimentech nebyl použit, lze snadno dodat též. Tento model neuronové sítě využívá stejně jako algoritmus zpětného šíření neurony třídy `BPNeuron`. Jejich položku `double delta` ale narozdíl od tohoto algoritmu využívá k ukládání parciální derivace výstupu jediného výstupního neuronu sítě podle potenciálu neuronu, jehož je položkou. Staré hodnoty vah synapsí využívá algoritmus k uložení časově vážené sumy parciálních derivací $\mathcal{S}(t)$ definovaného vzorcem 4.12 v části 4.4. Kromě seřazení neuronů dle vrstev zde navíc pro správnou funkčnost metod třídy `TDNetwork` požadujeme, aby v seznamu neuronů všechny vstupní neurony pro stav předcházely všechny vstupní neurony pro akce. Třída `TDNetwork` zavádí navíc čtyři datové položky. Položka `float lambda` obsahuje hodnotu parametru λ, položka `int stateInputs` udává, kolik ze vstupních neuronů kóduje stav prostředí, `BPNeuron* predictor` je ukazatel na jediný výstupní neuron, který predikuje odměnu, a `float rand_prob` je pravděpodobnost, že vybraná akce bude nahrazena náhodně zvolenou akcí při použití znáhodnění typu 2.

`Class SNeuron, Class SNetwork`

Třídy představují neuron a neuronovou síť pro algoritmus 5 - Schmidhuberovu rekurentní síť. Třída `SNeuron` je oproti třídě `Neuron` pro účely algoritmu 5 obohacena o datové položky `float* derivatives, float* oldDerivatives, float* derBias, float* oldDerBias`, které ukazují na dynamicky vytvořená pole starých a nových hodnot parciálních derivací neuronu dle vah a prahů celé sítě. `SNeuron` si dále udržuje v položce `SNetwork* netPointer` ukazatel na síť, ve které je umístěn, a svůj index v této síti v položce `int neuronIndex`. V případě, že se jedná o predikční neuron z modelující sítě, položka `SNeuron* predicts` ukazuje na predikovaný neuron, v opačném případě má tato položka hodnotu nulového ukazatele.

Třída `SNetwork` klade pro správnou funkčnost svých metod na uspořádání neuronů v síti další podmínky. Všechny neurony řídicí sítě musí v seznamu neuronů předcházet všechny neurony modelující sítě, přičemž první musí být vstupní neurony a poslední z neuronů řídicí sítě musí být neurony výstupní. Derivace dle neexistujících vah musí mít nastaveny nulovou hodnotu. Tyto podmínky jsou při používání dodaných metod zaručeny. Třída `SNetwork` obsahuje dále datové položky `int stepNo` udávající, zda je algoritmus ve svém prvním cyklu, `int controlSize` udávající velikost řídicí sítě, ukazatel na odměňovací vstup `SNeuron* reward`, ukazatel na prediktor odměny `SNeuron* rewardPredictor` a pole ukazatelů na všechny prediktory `std::vector<SNeuron*> predictors`.

Pro rozšíření knihovny o další model je obvykle třeba napsat nové potomky tříd `Neuron` a `NNetwork`, definovat nové datové položky a metody a předefinovat virtuální metody třídy `NNetwork`.

Knihovnu Flexinets včetně příkladu jejího použití při učení sítě zpětného šíření logickou funkci XOR lze nalézt na CD přiloženém k této práci.

Dodatek B

Použitý simulátor kořisti a dravce

Za účelem testování algoritmů popsaných v této práci jsme vyvinuli jednoduchý simulátor dravce a kořisti, který je přílohou této práce.

B.1 Uživatelské rozhraní

Uživatelské rozhraní simulátoru je poměrně strohé, skládá se z textové konzole a jednoho grafického okna. V grafickém okně se zobrazuje průběh simulace nebo uživatelská nápověda. V prvním případě okno znázorňuje dravce v podobě červených bodů a kořist v podobě bodů modrých, které se postupně přebarvují do zelena, pokud kořist hladoví. V grafickém okně najdeme dále informace o tom, kolik kořisti bylo sežráno dravci a kolik jí vyhladovělo, kolik bodů a jakou odměnu zatím získala kořist v právě běžící simulaci a kolikátá simulace právě probíhá. Konzole slouží k výpisu informací o průběhu experimentů. Po skončení každé série simulací je zde vypsán počet simulací, které proběhly, celkový počet bodů a odměna získané v sérii.

Po spuštění je v grafickém okně zobrazena stručná nápověda k ovládání programu, kterou lze skrýt a opětovně zobrazit klávesou F1. Jeden krok simulace provedeme klávesou enter, kontinuální běh simulace spustíme a zastavíme mezerníkem. Klávesa 0 slouží k vypnutí nebo zapnutí znáhodnění v případě, že je tato volba relevantní (tedy při použití Schmidhuberova modelu či modifikovaného TD(λ) se znáhodněním číslo 2). Klávesou 1 v grafickém okně naprogramujeme jednu simulaci, klávesou 2 naprogramujeme sérii simulací a klávesou 3 naprogramujeme nekonečný sled simulací. Klávesou A lze zakázat a opět povolit učení. Simulace může běžet ve dvou módech - se zobrazením a bez zobrazení. Mód bez zobrazení je rychlejší a hodí se pro experimenty sestávající z dlouhé série simulací. Módy přepínáme klávesou D. Klávesou R přerušíme právě probíhající simulaci a připravíme novou. Volba S uloží strukturu a váhy neuronových sítí řídících kořist do očíslovaných souborů v podadresáři /results. Klávesa escape ukončí celý simulátor.

Potřebujeme-li při experimentu zjistit číslo některého z agentů, můžeme na něj kliknout levým tlačítkem myši. Na konzolu se pak vypíše jeho číslo a případně další údaje o něm. Uložíme-li agenty pomocí klávesy S, můžeme takto poklikáním na agenta zjistit, které číslo souboru odpovídá kterému agentu na obrazovce. Pro některé modely vyvolá poklepání myší na agenta též výpis dalších informací užitečných pro analýzu příslušné neuronové sítě (například hodnoty jeho vstupů a výstupů).

B.2 Stručně o implementaci simulátoru

Simulátor byl stejně jako knihovna pro práci s neuronovými sítěmi Flexinets naprogramován v C++ s použitím Microsoft Visual C++ 6.0. Uživatelské rozhraní bylo vytvořeno pomocí knihovny OpenGL. Program využívá objektově orientovaných technik tak, aby byl program přehledný a snadno modifikovatelný. Simulátor poskytuje rozhraní pro použití dodaného algoritmu (funkce v jazyce C / C++) pro ovládání kořisti nebo dravce. Stejně jako v knihovně Flexinets, i zde byla použita konvence, že jména objektů a dalších typů začínají velkým písmenem, jména instancí objektů, proměnných a položek objektů písmenem malým a jména konstant preprocesoru jsou psána celá velkým písmem. Kód je pro větší čitelnost podrobně okomentován.

B.2.1 Hierarchie objektů

Základní použité objekty jsou třída `Environment` a třída `Creature`. Třída `Environment` poskytuje abstraktní rozhraní pro prostředí (svět) obecného simulátoru a deklaruje nejdůležitější datové položky a metody, které konkrétní simulátor musí obsahovat. Jednou z položek třídy `Environment` je spojový seznam `creatureList` instancí třídy `Creature`, která poskytuje abstraktní rozhraní pro obecného agenta. Kromě seznamu agentů obsahuje dále třída `Environment` abstraktní deklaraci základních metod pro přidávání a ubírání agentů a jejich interakci s prostředním.

Třída `Creature` popisuje základní datové položky, které musí mít každý druh agenta (např. souřadnice udávající jeho polohu) a základní virtuální metodu `action(inputs, outputs, reward)`, která posílá agentovi informaci o jeho aktuálních vstupech a odměně a požaduje informaci o jeho výstupech. Tato metoda by sama neměla měnit stav prostředí, pouze vrátit informaci o tom, co agent chce vykonat.

Simulátor, který používáme pro popisované experimenty, využívá třídu `Environment1`, která je potomkem třídy `Environment`. Třída definuje konkrétní vlastnosti světa s diskrétním prostorem a časem odpovídajícího výše popsané úloze. Speciálně třída `Environment1` předpokládá, že agenti komunikují s prostředím prostřednictvím 4 vstupů, 3 výstupů a odměny, tak jak jsou popsány v zadání úlohy.

Třídě `Environment1` odpovídá třída agentů `Creature1`, která je potomkem třídy `Creature`. Popisuje stále ještě abstraktní třídu agentů pro toto prostředí. Jejím potomkem je třída `Tiger` představující dravce a třída `Rabbit` představující kořist. Agenti řídící se různými dodanými algoritmy (např. nějakým modelem neuronové sítě) jsou pak potomci třídy `Rabbit` lišící se mezi sebou definicí virtuální metody `action`.

B.2.2 Struktura programu

Základem činnosti simulátoru je metoda třídy `Environment1` `oneStep`, která představuje jeden krok simulace. Metoda postupně projde seznam agentů creatureList. Pro každého z nich vypočítá hodnotu vstupů a odměnu a zavolá jeho metodu `action`, aby získala jeho výstupy, které potom vykoná pomocí metody `perform`.

Zdrojový kód simulátoru se skládá z těchto souborů:

genericSim.h
- obsahuje definice abstraktních tříd `Environment` a `Creature`

environment.h
- obsahuje definice konstant použitých v simulátoru a potomků třídy `Environment` a `Creature`

environment.cpp
- kód metod určujících logiku simulátoru

SurvivalSim.cpp
- funkce main a uživatelské rozhraní simulátoru

Chceme-li tedy experimentovat s novým algoritmem, je třeba vytvořit nového potomka třídy `Rabbit`, upravit v souboru environment.cpp metodu `Environment1::spawn` tak, aby vytvářela v prostředí agenty řízené tímto algoritmem, a případně v témže souboru upravit metodu `Environment1::restart`, pokud je potřeba agenty na začátku simulace inicializovat.

Počet kroků na jednu simulaci a počet simulací v experimentu můžeme nastavit změnou konstanty `TICKS` respektive `SESSIONS` v souboru SurvivalSim.cpp.

Simulátor kořisti a dravce je přiložen na disku CD k této práci.

Dodatek C

Obsah přiloženého CD

Součástí této práce je i CD, na kterém lze nalézt tuto práci v elektronické verzi, zdrojové kódy knihovny neuronových sítí Flexinets, zdrojové kódy použité pro experimenty popsané výše a demonstrační videa a programy umožňující snadno opakovat nejúspěšnější z experimentů popsaných výše. Následuje popis obsahu CD:

- Precti.txt - popis obsahu CD
- Readme.txt - popis obsahu CD v Angličtině
- Adresář Thesis
  Tento adresář obsahuje elektronickou verzi této práce.
  - NS a RL.ps - tato práce ve formátu postscript
  - NS a RL.pdf - tato práce ve formátu Adobe reader
- Adresář Flexinet
  Tento adresář obsahuje zdrojový kód použité knihovny neuronových sítí Flexinets.
  - neurons.h - hlavička s definicemi knihovny Flexinets
  - neurons.cpp - těla metod a funkcí knihovny Flexinets
  - xor.cpp - ukázkový příklad aplikace knihovny Flexinets pro naučení sítě zpětného šíření výpočet logické funkce XOR
- Adresář PredatorPrey
  Tento adresář obsahuje soubory týkající se experimentů v simulátoru dravce a kořisti popsaných v části 5.1. Adresář obsahuje podadresáře odpovídající experimentům s dravci a bez dravců pro vybranou konfiguraci každého ze tří testovaných algoritmů (celkem 6 experimentů). Každý z adresářů má obdobný obsah:
  - SurvivalSim.exe - demonstrační program spouštějící experiment
  - opengl.dll, glu.dll, glut.dll - potřebné dynamické knihovny openGL
  - results - adresář pro ukládání konfigurací sítí během experimentu
  - source - adresář se zdrojovým kódem simulátoru včetně souboru environment.h s parametry nastavenými pro spuštění příslušného experimentu

- Adresář ExperimentRobot
  Tento adresář obsahuje soubory týkající se experimentu se simulovaným i reálným robotem e-puck v programu Webots popsaného v části 5.2. Názvy některých souborů v jeho podadresářích odpovídají stavu experimentu v určitém čase. Potom jejich názvy obsahují číslo, které udává, po kolika časových periodách tento stav nastal. Časová perioda se skládá z 200 kroků robota. Tedy například soubor real_8000.avi zachycuje chování fyzického robota po 1600000 krocích simulovaného učení. Obsah adresáře je následující:
  - real_epuck01.jpg, real_epuck02.jpg - fotografie z experimentu s reálným robotem
  - video_simulation - filmy zachycující průběh experimentu se simulovaným robotem
  - video_real - filmy zachycující průběh experimentu s fyzickým robotem
  - controler_source - zdrojový kód použitý v experimentu
    * neurons.h - hlavičkový soubor knihovny Flexinets s parametrem EPUCKTURNRIGHT, který pro účely tohoto experimentu povoluje pouze náhodné pohyby, při nichž se robot točí vpravo
    * ThesisController.cpp - zdrojový kód programu ovládajícího robota (tzv. kontroleru)
    * ThesisSupervisor.cpp - zdrojový kód obslužného programu pro experiment (tzv. supervisoru)
  - experiment_log - adresář s výsledky experimentu
    * results - adresář obsahující všechny mezikonfigurace neuronové sítě z průběhu experimentu, které mohou být opětovně načteny upravením parametru LOADFILE ve zdrojovém souboru kontroleru
    * rewards.csv - soubor obsahující historii získaných odměn za každých 200 kroků experimentu
    * testWebots.csv - soubor obsahující historii ujeté vzdálenosti a počtu robotových kolizí za každých 200 kroků experimentu
  - Webots
    Tento adresář obsahuje soubory potřebné pro spuštění experimentu v simulátoru Webots. Pro spuštění experimentu je třeba mít nainstalovánu verzi simulátoru Webots umožňující použití uživatelsky definovaných kontrolerů. V takovém případě stačí překopírovat tento podadresář do adresáře, kde je simulátor nainstalován, otevřít soubor se světem a ujistit se že je propojen se správným supervisorem a kontrolerem. Soubory s příponou .nn obsažené v podadresářích tohoto adresáře obsahují uložené konfigurace neuronové sítě v určitém stádiu experimentu.
    * thesisWorld.wbt - soubor se světem pro program Webots použitým v experimentu
    * ThesisSupervisor - adresář se supervisorem přeloženým pro použití v simulátoru Webots
    * ThesisController - adresář s kontrolerem přeloženým pro použití v simulátoru Webots

* 4000, 7000, 8000, 8210, 8660 - adresáře s kontrolery pro simulátor Webots, které demonstrují chování robota, pokud zastavíme jeho učení v určitém stádiu experimentu