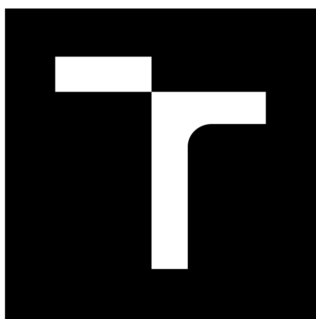
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2018 Jana Daňhelová



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

ZPĚTNOVAZEBNÍ UČENÍ PRO ŘEŠENÍ HERNÍCH ALGORITMŮ

REINFORCEMENT LEARNING FOR SOLVING GAME ALGORITHMS

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

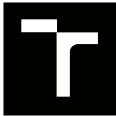
AUTOR PRÁCE Jana Daňhelová

AUTHOR

VEDOUCÍ PRÁCE Ing. Martin Kolařík

SUPERVISOR

BRNO 2018



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Teleinformatika**
Ústav telekomunikací

Studentka: Jana Daňhelová ***ID:*** 185967
Ročník: 3 ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Zpětnovazební učení pro řešení herních algoritmů

POKYNY PRO VYPRACOVÁNÍ:

Nastudujte existující metody zpětnovazebního učení. Proveďte rešerši těchto metod, jejich vlastností a použití. Po konzultaci s vedoucím práce poté vyberte testovací hru a ze zkoumaných metod vhodný postup, který použijete pro její řešení. Výběr zdůvodněte a metodu otestujte pro použití na daný problém. Výsledky testování vhodně prezentujte.

DOPORUČENÁ LITERATURA:

[1]MNIH, Volodymyr, Koray KAVUKCUOGLU, David SILVER, et al., Human-level control through deep reinforcement learning [online]. [cit. 2017-09-13]. DOI: 10.1038/nature14236. ISBN 10.1038/nature14236. Dostupné z: http://www.nature.com/doifinder/10.1038/nature14236

[2]SCHMIDHUBER, Jürgen, Deep learning in neural networks: An overview [online]. [cit. 2017-09-13]. DOI: 10.1016/j.neunet.2014.09.003. ISBN 10.1016/j.neunet.2014.09.003. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/S0893608014002135

Termín zadání: 5.2.2018 ***Termín odevzdání:*** 29.5.2018

Vedoucí práce: Ing. Martin Kolařík
Konzultant:

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

UPOZORNĚNÍ:
Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Bakalářská práce Zpětnovazební učení pro řešení herních algoritmů je rozdělena do dvou částí. V teoretické části jsou popsány a srovnávány základní metody zpětnovazebního učení, přičemž zvláštní pozornost je věnována metodám aktivního učení – Q-učení a hlubokému učení. Praktická část je zaměřena na aplikaci metody deep learning na hru Had. Výsledky jsou prezentovány ve formě programu napsaného v programovacím jazyku Python, který se skládá z herního prostředí vytvořeného v PyGame, modelu konvoluční neuronové sítě zkonstruovaného v knihovně Keras a herního agenta. Výstupem programu je několik typů datasetů ve formátu csv. Získaná data, obsahující hodnoty jednotlivých parametrů jako počet epoch, přesnost, ztráta nebo výše odměny, mohou být následně použita jako podklady pro další zpracování.

KLÍČOVÁ SLOVA

zpětnovazební učení, Had, hluboké učení, konvoluční neuronová síť, herní algoritmus, PyGame, Keras, Python 3.5

ABSTRACT

The bachelor thesis Reinforcement learning for solving game algorithms is divided into two distinct parts. The theoretical part describes and compares the fundamental methods of reinforcement learning with special attention to the methods of active learning – Q-learning and deep learning. In the practical part the deep q-learning technique is chosen for testing and applied to the case of the Snake game. The results are presented in the form of program written in Python programming language, which consists of the game environment created in PyGame, the model of convolutional neural network designed in Keras and agent playing the game. As an output of the program there are several types of datasets in CSV format. The gained data containing the values of parameters like number of epochs, accuracy, loss or the amount of the reward can later be used for further processing.

KEYWORDS

reinforcement learning, Snake, deep learning, convolutional neural network, game algorithm, PyGame, Keras, Python 3.5

DAŇHELOVÁ, Jana. *Zpětnovazební učení pro řešení herních algoritmů*. Brno, 2018, 48 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Martin Kolařík

Vysázeno pomocí balíčku thesis verze 2.63; http://latex.feec.vutbr.cz

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Zpětnovazební učení pro řešení herních algoritmů" jsem vypracoval(a) samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno . . . .

podpis autora(-ky)

PODĚKOVÁNÍ

Za poskytnuté rady i vedení bakalářské práce děkuji panu Ing. Martinu Kolaříkovi.

Brno . . . .

podpis autora(-ky)

OBSAH

Úvod **9**

1 Teoretická část studentské práce **12**

1.1 Zpětnovazební učení a neuronové sítě 12

1.1.1 Markovův rozhodovací proces 13

1.2 Neuronové sítě pro hluboké zpětnovazební učení 15

1.2.1 Neuronové sítě a určování rysů 15

1.2.2 Hluboké neuronové sítě (deep neural networks) 16

1.3 Metody zpětnovazebního učení 17

1.3.1 Temporální diference TD(0) 18

1.3.2 Metoda hrubé síly 19

1.3.3 Přímé hledání strategie 19

1.3.4 Monte Carlo 20

1.3.5 Dynamické programování 21

1.3.6 Q-learning a deep learning 22

1.3.7 Paralelizace zpětnovazebního učení 23

1.3.8 SARSA 27

2 Výsledky studentské práce **29**

2.1 Testovací hra Had 29

2.1.1 Princip hry a herní prostředí 29

2.1.2 Návrh a implementace programu 30

2.1.3 Popis programu 34

3 Závěr **40**

Literatura **41**

Seznam symbolů, veličin a zkratek **46**

Seznam příloh **47**

A Obsah přiloženého CD **48**

SEZNAM OBRÁZKŮ

1 Systematické procházení herního prostoru s jistotou projití každého políčka . . . 10
2 Vyvážení strategií přežití a zvyšování skóre . . . 11
1.1 Princip zpětnovazebního učení . . . 12
1.2 Graf sigmoidální funkce . . . 14
1.3 Agent s Q-učením . . . 24
2.1 Hodnoty parametru Loss při trénování sítě . . . 33
2.2 Had volí cestu přežití na úkor sběru jídla [39] . . . 34
2.3 Struktura použité konvoluční neuronové sítě . . . 35
2.4 Průběh tréninku při spuštění z příkazového řádku . . . 35
2.5 Zdrojový kód souboru neuronova_sit.py ve formě UML diagramu . . . 36
2.6 Předzpracování herního screenshotu pro vstup do neuronové sítě . . . 37

ÚVOD

Tato bakalářská práce se zaměří na využití umělé inteligence (*Artificial Intelligence*, AI) v oblasti her. Jednou z hlavních výzev na tomto poli je vytvářet inteligentní agenty, kteří budou měnit své chování na základě interakcí s okolním prostředím a postupem času se v hraní hry stávají zdatnějšími, stejně jako v adaptaci na nové stavy, do nichž se dostanou. Techniky vyvinuté v oblasti her s umělou inteligencí mohou být využity i v jiných sférách, jako je např. robotika (výše zmíněné schopnosti agentů jsou klíčové mj. při zavádění robotů do lidského prostředí), optimalizace zdrojů nebo inteligentní asistenti. Umožňují také vytvářet nové metody umělé inteligence, měřit jejich výkonnost apod. [1] Počítačové hry jsou typické velkým stavovým a akčním prostorem a vyžadují od agentů rozmanité typy chování, rychlou adaptaci na prostředí a také zapamatování minulých prožitých stavů.

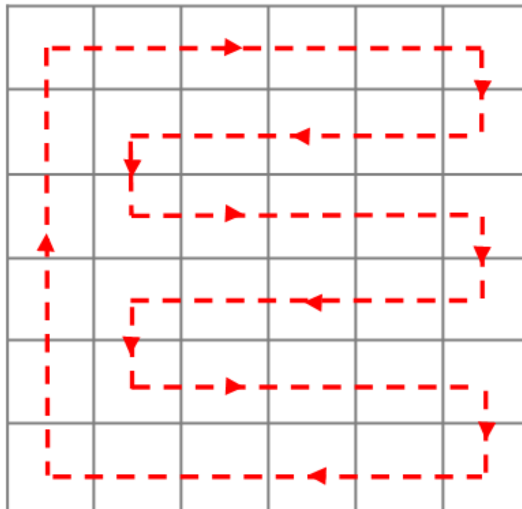
Pozornost bude věnována jedné z odnoží AI, tzv. zpětnovazebnímu učení (*Reinforcement Learning*, RL). Právě hraní her je jedna z oblastí, kde je aplikace RL nejvhodnější, protože prostředí, v němž se ústřední charakter – agent nachází, je komplexní a nenabízí žádné nebo jen obtížně naprogramovatelné řešení. Určení nejvýhodnější strategie je závislé na mnoha různých faktorech, z čehož vyplývá velké množství stavů, do nichž se agent může během hry dostat. Pokrýt všechny tyto stavy pevně danými pravidly, jak se má agent v různých situacích rozhodovat, se jeví jako nemožné. RL řeší tento problém absencí potřeby manuálně nastavovaných pravidel, agent se rozhoduje a učí samotným hraním hry.

Teoretická část práce bude rozdělena do tří částí: první kapitola 1.1 se bude věnovat nástinu historie RL a neuronových sítí. Následující část 1.2 popíše architekturu hlubokých neuronových sítí. Třetí část 1.3 bude charakterizovat a vzájemně srovnávat jednotlivé metody zpětnovazebního učení, jejich vlastnosti a využití. Zvláštní pozornost bude věnována metodám Q-learning a deep q-learning. Aplikace metod hlubokého učení (především využití konvolučních neuronových sítí) na zpětnovazební učení vedla ke vzniku hlubokého zpětnovazebního učení (*Deep Reinforcement Learning*), které bude využito při řešení praktické části. V praktické části 2.1 tedy budou teoretické předpoklady aplikovány na zvolenou hru a otestovány. Výstupem praktické části a prezentací výsledků aplikovaného postupu bude funkční testovací program, jehož zdrojový kód bude uveden v příloze.

Pro testování byla zvolena hra Had. Co se týče strategie, jako stěžejní se zde jeví dvě dovednosti klíčové pro úspěšné hraní hry. Agent může v této hře upřednostnit buď rychlost (najít co nejvíce jídla – nejvyšší skóre) nebo spolehlivost (zůstat naživu co nejdéle). [2] Správná kombinace obou přístupů by hypoteticky mohla vést k vysoké spolehlivosti a současně dosahování vysokého skóre. K otestování této domněnky bude využit algoritmus deep q-learning, který je založen na trénování

konvolučních neuronových sítí pomocí Q-učení. Vliv na výkonnost neuronové sítě může mít mimo jiné i velikost herního prostředí nebo nastavení hyperparametrů neuronové sítě. Učení neuronové sítě pokračuje až do momentu, kdy se přesnost jejích odhadů nebude dále zlepšovat. Jde tedy o hledání určité rovnovážné kombinace strategií, kdy agent-hráč dosáhne stavu, v němž nebude mít důvod svou strategii nadále měnit. [3]

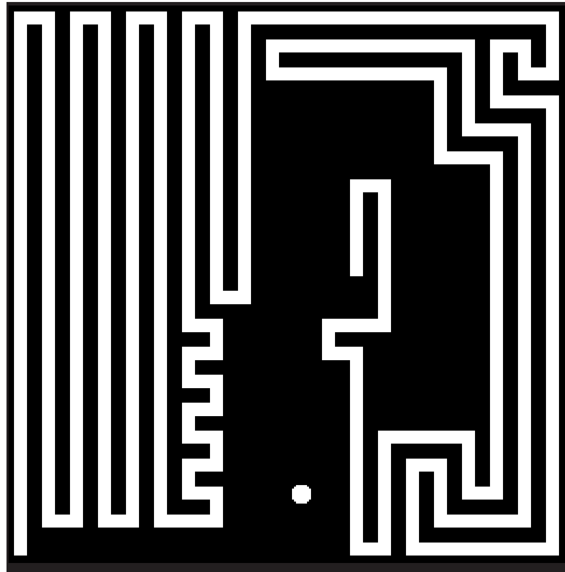
Chování agentů učících se pomocí zpětnovazebního učení však může být mnohdy překvapivé a v rozporu s intuicí, někdy i subverzivní (konkrétním příkladem může být špatné nastavení odměn ve hře CoastRunners, kde je cílem hry dokončit závod člunů co nejrychleji a s co nejlepším umístěním, přičemž agent může navyšovat skóre zasahováním terčů umístěných podél trasy. Při hře se však RL agent zaměřil na projíždění terčů, aniž by se snažil dojet do cíle a ačkoliv přitom narážel do ostatních člunů a několikrát začal hořet, dokázal načasovat svůj pohyb tak, že se mu paradoxně dařilo dosahovat o 20 % vyššího skóre než soupeřům v cíli, jak je vidět ve videu [1]. [4]) Nepředvídatelné chování bylo sledováno i při učení agenta v rámci této práce, kdy např. agent i v pozdějších fázích opakovaně dokola objížděl těsně kolem jídla, aniž by ho sebral, bylo pozorováno i chování, kdy agent systematicky prozkoumával prostředí nejprve vodorovnými pohyby, po určité době přešel do kolmého směru a pokračoval svislými pohyby, což by mohlo znamenat, že had jídlo ignoroval a upřednostnil strategii přežití s jistotou nalezení jídla „cestou“ (připomínalo to způsob, který by byl zvolen při řešení problému hrubou silou (*brute-force search*, *exhaustive search*), a to i s jeho charakteristickou velkou časovou náročností) 1.



Obr. 1: Systematické procházení herního prostoru s jistotou projití každého políčka

[1] https://youtu.be/tlOIHko8ySg

Jak vypadá ideální hráč Hada (ať už lidský nebo počítačový) lze vidět po pár desítkách sekund na následujícím odkazu: http://datagenetics.com/blog/april42013/s1.gif 2 [5].



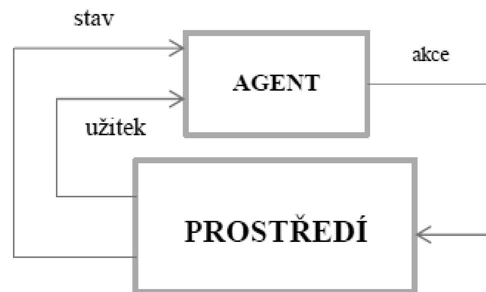
Obr. 2: Vyvážení strategií přežití a zvyšování skóre

V souvislosti s uvedeným příkladem a výše zmíněným záměrem přenést získané poznatky do reálného života vyvstává i otázka bezpečnosti systémů založených na RL a potřeba zvnějšku sledovat chování agenta během učení a vytvářet nástroje pro jeho kontrolu – viz např. *Reinforcement Learning Control Center* [6].

1 TEORETICKÁ ČÁST STUDENTSKÉ PRÁCE

1.1 Zpětnovazební učení a neuronové sítě

Zpětnovazební učení (jeho princip ukazuje obrázek 1.1) je typem strojového učení (*machine learning*), kdy se agent nacházející se v neznámém prostředí rozhoduje, jaké akce zvolí, aby z nich maximalizoval svůj užitek. Pro usnadnění rozhodování je mu v daný okamžik dáno k dispozici jen omezené množství informací, ostatní nebere v úvahu. [7] Tím, že je agentovi při jeho rozhodování poskytnuta jen částečná zpětná vazba, se RL odlišuje od učení s učitelem (*supervised learning*). V RL jsou pouze dva zdroje zpětné vazby: odměna a trest (negativní odměna). Proces učení se pak zjednodušuje na procházení agenta daným prostředím a najití nejvýhodnější cesty do cíle, na níž agent nasbírá pokud možno maximální možnou kumulativní odměnu.



Obr. 1.1: Princip zpětnovazebního učení

Počátky zpětnovazebního učení sahají do konce 70. let 20. století, konkrétně do roku 1979, kdy byl zahájen jeden z prvních projektů zaměřujících se na využití neuronových prvků se schopností adaptace v rozvoji umělé inteligence. [8] Výchozím bodem projektu byla tzv. heterostatická teorie adaptivních systémů A. Harryho Klopfa. [1] Podle Klopfa by se k neuronům mělo přistupovat jako k „hédonistům“, entitám vyhledávajícím rozkoš a požitky. Depolarizace neuronu (excitace, posun hodnoty membránového napětí směrem k méně negativním až pozitivním hodnotám)

[1] Na základě Klopfovy teorie zformulovali Richard Sutton a Andrew Barto myšlenku zpětnovazebního učení – ideu učícího se hédonistického systému, který má nějaký cíl a adaptuje svoje chování tak, aby maximalizoval užitek z prostředí, v němž se nachází, ve snaze dosáhnout stanoveného cíle. Systém přitom nemá žádnou podporu zvnějšku, učí se bez učitele. Jediná odezva, kterou dostává, jsou změny v okolním prostředí, které vyvolává svým jednáním. Musí tedy metodou experimentu zkoušet, které stavy jsou pro něj výhodné a které naopak jeho užitek snižují. Jeho učení může být buď pasivní nebo aktivní. U pasivní formy má systém předurčenou strategii a učí se pouze zjišťováním, jakou hodnotu mu přinesou stavy, do nichž se dostává. Naproti tomu aktivní učení spočívá v samostatném rozhodování systému, jaké akce má zvolit a zjišťování, jaký užitek mu přinesou.

je přitom brána jako „rozkoš“ a hyperpolarizace (inhibice, proces opačný k depolarizaci, posun hodnoty membránového napětí do zápornějších až záporných hodnot) jako „bolest“. V tomto pojetí je cílem neuronu maximalizovat depolarizaci na úkor hyperpolarizace. V teorii H. Klopfa neurony využívají tzv. heterostatickou adaptaci; při dostatečně vysoké depolarizaci se zvýší membránové napětí natolik, že vznikne akční potenciál a neuron „vystřelí“. V takovém stavu si neuron všimne, zda se během určitého (několikavteřinového) časového intervalu mění rozdíl mezi jeho excitacemi a inhibicemi a podle toho pak reaguje v následných stavech, v nichž se ocitne. [9]

Charakteristickou vlastností neuronů byla jejich dovednost učit se – učení v tomto smyslu znamená správné přenastavování vah w při konfrontaci s neznámou informací na základě informací neuronu již známých. V té době již existovalo několik modelů neuronu. Jedním z nich byl adaptivní lineární neuron Adaline, který je zároveň jednou z prvních, jednovrstvých umělých neuronových sítí. Vstupy – podněty jsou u Adaline libovolná čísla v rozsahu $x_1,\ldots,x_\mathrm{m}$, která jsou vynásobena vahou w_i a z jednotlivých součinů je vytvořen vážený součet jednotlivých částí (1.1):

$$y = \mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{m} w_\mathrm{i} \cdot x_\mathrm{i}, \tag{1.1}$$

kde m je počet vstupů, w váhový vektor a x vstupní vektor. Výstupem Adaline mohou být dvě hodnoty. Pokud vážený součet nepřekročí práh w_0, je výstup roven 0 (případně -1), jinak je roven 1. Když součet překročí danou hranici, neuron se aktivuje. Nejpoužívanější aktivační funkcí je tzv. sigmoidální 1.2:

$$f(y) = \frac{1}{1 + e^{-y}}, \tag{1.2}$$

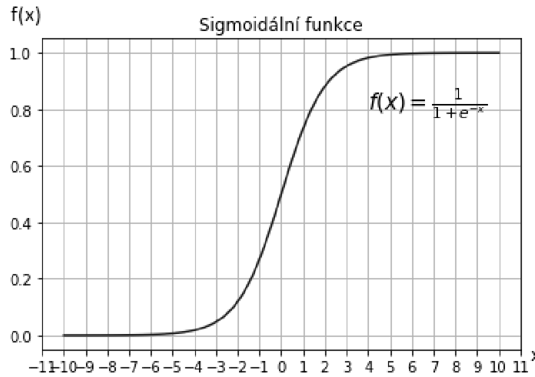
kdy výstupem neuronu jsou hodnoty z intervalu [0, 1]. [10] Graf sigmoidální funkce je na obrázku 1.2. K učení používal Adaline tzv. gradientní metodu. Tento jednoduchý model neuronu byl v pozdějších letech nahrazen složitějšími modely, které místo skokové funkce používají hladké funkce nabývající hodnot v rámci určitého intervalu. Právě podmínka hladkosti funkce je klíčová v algoritmu nastavování vah během učícího procesu. [8] Aktualizace vah probíhá dle vztahu 1.3:

$$w \leftarrow w + \eta(o - y)x, \tag{1.3}$$

kde η je kladná konstanta určující rychlost učení, y je výstup modelu a o je požadovaný výstup.

1.1.1 Markovův rozhodovací proces

V RL se agent snaží optimalizovat dlouhodobé zisky prostřednictvím pochopení neznámého prostředí a rozhodnutí učiněných v jeho rámci. Na začátku učení jsou mu



Obr. 1.2: Graf sigmoidální funkce

užitky a pravděpodobnosti vyplývající z jeho kroků v tomto prostředí neznámé. Procesem učení však postupně přestává být jeho rozhodování náhodné a dostává situaci částečně pod svou kontrolu. Vzhledem k tomu, že úkoly RL jsou často formulovány jako Markovovy rozhodovací problémy, je pro jejich řešení vhodné aplikovat tzv. Markovův rozhodovací proces (*Markov Decision Process*, MDP), který umožňuje matematicky popsat rozhodování v situacích, kdy má entita – agent situaci částečně pod kontrolou a částečně je jeho rozhodování dílem náhody. Princip MDP je tedy analogický s výše popsanou situací agenta. [11] [2]

Pojmem *stav* je míněna jakákoliv informace, která je agentovi dostupná a kterou je schopen prostřednictvím svých senzorů vnímat. Neznamená to ale, že by agent byl informován o všech detailech okolního prostředí, případně o všem, co by mu usnadnilo jeho rozhodování. Řada informací mu zůstává skryta, i když by mu přinesly užitek. [8] Pokud agent zjistí informace o okolním prostředí, jeho primárním úkolem je si tyto informace uložit a zapamatovat pro pozdější použití. MDP se v každém časovém okamžiku nachází ve stavu s, který nabízí možné akce a, z nichž agent jednu zvolí. Proces agenta ocení užitkem $R_a(s, s')$ a jeho volbou se přesune do následujícího časového okamžiku a stavu *s'*. Výběr tohoto konkrétního stavu *s'* je tedy závislý na vybrané akci a a na předchozím stavu s, které jsou závislé na všech minulých s a a. Proces však drží v paměti jen minulý stav a akci a ne kompletní historii všech

[2]Učení agenta může nabývat dvou forem. Explorace spočívá v aktivním průzkumu okolního terénu, kdy systém zatím nezná důsledky svých akcí a náhodně vybírá následnou akci s pravděpodobností úměrnou částečné odměně. U exploatace už agent zná výsledky svých akcí a jen využívá svých nabytých znalostí ke zvyšování užitku. [12] Pokud v minulosti vyzkoušel nějakou akci a obdržel za ni patřičnou odměnu, tak zopakování stejné akce v budoucnu bude znamenat i opakovanou odměnu. Naproti tomu pokud bude agent riskovat a rozhodne se i pro jiné možnosti než doposud vyzkoušené, může získat ještě vyšší odměnu. Optimální strategie spočívá v najití kompromisu mezi oběma alternativami a je klíčová pro úspěšné učení. Nejjistější taktikou je tak vyzkoušení širokého spektra akcí a současné upřednostňování těch, které vedou k nejvyšším odměnám.

minulých stavů a akcí, má tzv. Markovovu vlastnost. MDP se skládá z pěti částí:

- množiny stavů S,
- množiny akcí A,
- pravidel určující odměnu agenta R,
- pravděpodobností přechodů do dalších stavů $T(s, a, s')$,
- diskontního faktoru γ.

Každému stavu přísluší odměna a tzv. Bellmanův princip optimality (viz 1.3.5) lze pro stav $V^*(s)$ formulovat jako získání odměny za výběr stavu s a optimální akce a podle vztahu 1.4:

$$V^*(s) = R(s) + \gamma max_{\mathrm{a}} \sum_{s'} T(s, a, s') V^*(s'), \tag{1.4}$$

Optimální strategie π je pak definována jako 1.5:

$$\pi^*(s) = argmax_{\mathrm{a}} \sum_{s'} T(s, a, s') V^*(s'), \tag{1.5}$$

Výše uvedené však platí na teoretické úrovni. Problematické je, že MDP předpokládá (ale zároveň to není podmínkou), že se množiny možných stavů a akcí nemění v průběhu času (také R. Sutton a A. Barto se pro zjednodušení omezili na předpoklad konečných množin stavů a akcí [8]). Řada reálných problémů zpětnovazebního učení však nemá tento charakter a nelze je modelovat a řešit jako konečné Markovovy rozhodovací procesy, nelze na ně klást požadavky konvergence či optimality, u některých dokonce ani nelze určit, kolik učení je potřeba pro to, aby agent dosáhl stanovené výkonnostní úrovně či vůbec definovat nejlepší výkon, kterého lze dosáhnout. [13] Tento problém bude blíže popsán v kapitole věnující se metodám zpětnovazebního učení. Další problém vyvstává při výběru vhodné metody, kterou použijeme při řešení problému, protože na rozdíl od oblasti supervizovaného učení, kde lze čerpat z mnoha studií provedených na téma jejich metodologie, RL tolik vodítek v tomto smyslu neposkytuje.

1.2 Neuronové sítě pro hluboké zpětnovazební učení

1.2.1 Neuronové sítě a určování rysů

Zpětnovazební učení je součástí tzv. strojového učení fungujícího na principu trénování modelu, kterému jsou poskytnuta data na učení. Vytrénovaný model má pak při poskytnutí nových dat činit na jejich základě predikce a podle nich řídit své chování. Model se v podstatě učí na základě svých předchozích znalostí a při jeho vystavování novým podmínkám (datům) dokáže na jejich základě krok za krokem zlepšovat své chování v daném prostředí. V každém kroku činí model odhady

a dostává zpětnou vazbu o tom, jak (ne)přesné jeho predikce byly. Podle míry shody odhadu a reálného výsledku pak model upraví své predikce tak, aby míru této shody pokud možno maximalizoval v příštím kroku procesu učení. [14] Učení pak končí ve chvíli, kdy se odhady modelu zpřesní natolik, že se již dále nezlepšují. Z výše uvedeného vyplývá, že na vytrénování kvalitního modelu s přesnými odhady může být zapotřebí velké množství iterací učícího procesu. Klíčová je dovednost výběru malého počtu nejdůležitějších rozpoznávacích znaků – *features* (atributů nebo vlastností sdílených všemi jednotkami, na nichž má být predikce uskutečněna), které zajistí rychlou a efektivní klasifikaci a současně omezení délky trvání učícího procesu. Tato dovednost (*feature engineering*) je však v praxi velmi obtížně dosažitelná, kvůli rozmanitým typům dat, s tím související nutností rozdílných přístupů k jejich klasifikaci a zároveň snahou proces zautomatizovat a nalézt algoritmy, které budou schopny klíčové znaky nalézt.

S *feature engineering* je spojeno *feature learning* – automatizované vyhledávání rozpoznávacích znaků prostřednictvím algoritmů, s primárním cílem nalézt společné vzory (*patterns*) a automaticky je extrahovat k procesu klasifikace či regrese. U hlubokého učení (*deep learning*) se pro tento účel s úspěchem využívá konvolučních vrstev, které jsou schopné vybírat kvalitní znaky v rámci jednotlivých vrstev a posléze vytvářet jejich komplexní hierarchii. Poslední vrstva používá všechny vygenerované znaky ke klasifikaci nebo regresi. [14] Znaky jsou předány klasifikátoru, který je zkombinuje a následně činí odhady, jejichž přesnost závisí na množství nashromážděných znaků z co nejhlubších vrstev hierarchie. Generování znaků, které obsahují komplexnější informace, lze dosáhnout transformací znaků v předchozích hierarchických vrstvách.

1.2.2 Hluboké neuronové sítě (deep neural networks)

Pokusy o učení hlubokých neuronových sítí (především sítí s více než pěti skrytými vrstvami) a trénování schopnosti generalizace u vícevrstvých sítí nevedlo až do nedávné doby k uspokojivým výsledkům. [15] Učení založené na tzv. gradientním sestupu s náhodným počátkem (náhodnou inicializací, viz také 1.3.7) nevedlo k nalezení optimálního řešení ani při dostatečném počtu iterací; nebylo v nich totiž nalezeno globální maximum, ale pouze lokální a poté se algoritmus ukončil. [16] Lepších výsledků bylo dosaženo, když byly jednotlivé vrstvy postupně předtrénovány pomocí tzv. hlubokých autoenkodérů.

Předtrénování spočívalo v efektivním nastavení počátečních vah prostřednictvím vytrénování všech vrstev neuronové sítě (počínaje první) jen na základě úpravy jejích vstupů (nahrazuje tedy náhodnou inicializaci). Pozitivní efekt měl také nárůst výpočetních schopností počítačů (s až stovkami jader CPU či GPU) a využití al-

goritmu zpětné propagace (*backpropagation*). [17] Přitom právě algoritmus zpětné propagace, propagující chyby z výstupní vrstvy na vstupní, byl původně problematický. Postupným přibližováním reálných výstupů k požadovaným se totiž zmenšoval gradient chyby a ztrácela informace o tom, jak správně nastavovat váhy. Správné uspořádání každé vrstvy závisí na vrstvě předchozí, paradoxně při *backpropagation* nejsou informace o konfiguraci předchozích vrstev přímo dostupné. [18] Jak již bylo zmíněno výše, důsledkem je uvíznutí algoritmu v lokálním minimu, aniž by bylo nalezeno globální. Řešením by bylo manuální nastavování vah, což je na druhou stranu v rozporu s ideou hlubokého učení.

Při použití autoenkodéru lze využít jeho jednoduché struktury. Postup učení neuronové sítě je následující: nejprve je pomocí algoritmu pro nesupervizované učení trénována nižší vrstva a vytvořen počáteční soubor parametrů pro první vrstvu sítě. Výstup první vrstvy pak představuje vstup pro vrstvu následující, která je opět trénována prostřednictvím algoritmu pro nesupervizované učení. [15] Takto může být vytrénován požadovaný počet vrstev. V případě autoenkodéru první vrstva vstupy sítě kóduje a ta následující dekóduje tak, aby vznikla co nejmenší chyba. Pokud chceme síť trénovat ve schopnosti generalizace, nastavíme nižší počet neuronů ve vstupní vrstvě (pokud bychom tak neučinili, síť by pouze kopírovala informace ze vstupu na výstup). [18]

1.3 Metody zpětnovazebního učení

Zpětnovazební učení zahrnuje řadu různých metod, které se vzájemně nevylučují, lze vytvářet jejich kombinace (například metody temporální diference s Q-učením, nelineární aproximací či metodami Monte Carlo atp.) a také implementovat jednotící řešení, které jejich výhody spojí do jediného algoritmu (jedním z pokusů je např. algoritmus $Q(\sigma)$). Tato kapitola bude analyzovat jednotlivé metody pasivního i aktivního RL, pozornost bude věnována i jejich asynchronním variantám, které využívají vícevláknové programování a pro zvýšení pravděpodobnosti jejich konvergence, a zrychlení a stabilizaci učícího procesu nabízejí možnost implementace *experience replay*, kdy se Q hodnoty aktualizují na základě náhodného vzorku minulých zkušeností; umožňují také fixaci sítě (*target network*), kdy je neuronová síť ustálena po stanovený počet iterací (obvykle několik tisíc) v neměnném stavu, než jsou aktualizovány její váhy.

V teoretické oblasti zaznamenala oblast zpětnovazebního učení pokroky v upřesnění odhadů, za jakých podmínek a jak rychle konkrétní algoritmy konvergují k dosažení optimální strategie. V praxi jsou však teoretické předpoklady málokdy naplněny. Kvůli rozmanitosti učících metod může být jejich klasifikace obtížná. Řešením by mohla být jejich parametrizace, která jednak redukuje data a také je pomůže

uspořádat a rozdělit, aby bylo možné z pokud možno minimálního množství dat extrahovat co nejkvalitnější informace. Parametrizace tedy učící problémy rozliší a ty pak budou charakteristické právě svou odlišností. Shivaram Kalyanakrishnan a Peter Stone navrhují učící proces, který je realizovatelný za relativně krátký časový interval, čímž umožňuje rozsáhlé experimentování a porovnávání naučené strategie s optimálním chováním. To je v souladu s hlavním cílem RL – vytvořit efektivní učící algoritmus, který povede k maximalizaci užitků za minimálních omezení. Jako prvořadé faktory při parametrizaci identifikují částečnou pozorovatelnost a funkční aproximaci. [13] Hlavní důraz kladou na metody Sarsa(λ), Q-learning(λ) a Expected Sarsa(λ) neboli ExpSarsa(λ) a ze zástupců metod přímého hledání strategie (*policy search methods*) na metodu CMA-ES. Uvedené metody však nejsou jediné, které lze využít při řešení problémů RL. Na následujících řádcích budou analyzovány a vzájemně porovnány i další metody zpětnovazebního učení a budou popsány způsoby jejich využití. Lze je rozčlenit do několika skupin:

- metoda hrubé síly (*Brute Force*),
- přímé hledání strategie (*Direct Policy Search*),
- metody Monte Carlo,
- metody dynamického programování,
- metody založené na temporální diferenci.

1.3.1 Temporální diference TD(0)

Metody založené na temporální diferenci (TD) kombinují ideje metod Monte Carlo a dynamického programování. Co se týče jejich vzájemných podobností, metoda TD umožňuje, stejně jako Monte Carlo, učení z přímé zkušenosti a nevyžaduje model dynamiky prostředí (u metod MC navíc řeší problém časové náročnosti při vyhodnocování suboptimální strategie). Od metody dynamického programování převzala zásadu pravidelné aktualizace odhadů na základě již naučených odhadů, bez čekání na konečný výsledek. [8] Díky tomu lze TD využít jako metodu pro odhad i kontrolu. Hlavní výhodou je, že užitek jednotlivých stavů lze přímo upravovat tak, aby byl v souladu s Bellmanovými rovnicemi. Algoritmus Q-učení představuje off-policy implementaci TD jako kontrolní metody. Q-učení je reprezentantem učení off-policy, neboť strategie generující chování agenta a cílová strategie, kterou se agent naučí, se liší. [19]. Pro on-policy kontrolu je metoda TD použita u algoritmu Sarsa 1.6 1.7:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)], \tag{1.6}$$

$$a' = argmax_{\mathrm{a}}\{Q(s',a)\}. \tag{1.7}$$

Sarsa bere v úvahu poslední stav-akci, příslušnou odměnu a následující stav-akci. Následná akce a' je zvolena vždy dle hodnoty současné $Q(s,a)$ a na základě toho je upravena daná strategie 1:

Algorithm 1 On-Policy Temporal Difference Control

1: $Q(s,a)$ = arbitrary, $Q(terminal, a) = 0$
2: for $t = 1...N$
3: $s = start_state$, $a = epsilon_greedy_from(Q(s))$
4: while not game over:
5: s', $r = do_action(a)$
6: $a' = epsilon_greedy_from(Q(s'))$
7: $Q(s,a) = Q(s,a) + \alpha[r + \gamma\, Q(s',a') - Q(s,a)]$
8: $s = s'$, $a = a'$

Výhody metody TD(0):

- na rozdíl od metody Monte Carlo, kde je nutné čekat, až skončí celá epizoda, u TD se aktualizace hodnoty pro stav v čase t provádí hned v následujícím časovém okamžiku $t+1$,
- umožňuje agentovi učit se přímo během epizody, což je výhodné např. u dlouhotrvajících epizod,
- na rozdíl od metody Monte Carlo ji lze použít i pro úkoly neepizodického charakteru.

1.3.2 Metoda hrubé síly

Metoda hrubé síly funguje na principu zjištění užitku pro všechny možné strategie a následném výběru strategie s maximálním užitkem. Počet možných strategií však může být enormní (až nekonečný); navíc mezi výší užitků mohou být značné rozdíly a aby bylo možné stanovit přesnou výši užitku pro každdou strategii, je potřeba velké množství vzorků. Tyto nevýhody omezují využitelnost metody hrubé síly pro řešení úkolů RL. Konkrétním příkladem využití metody v oblasti RL je trénování herního agenta v sebeobraně proti nepříteli, kdy je generováno mnoho protivníků a agent se učí, jak se jim ubránit. [20]

1.3.3 Přímé hledání strategie

Nevýhody metody hrubé síly by bylo možné eliminovat pomocí metody přímého hledání strategie. Přímé hledání strategie se totiž zaměřuje jen na určitý úsek strategického prostoru a problém pak redukuje na tzv. stochastickou optimalizaci. Lze využít dvou druhů stochastických optimalizačních metod: jednak metod bez využití znalosti gradientu (*zero-order methods*), druhým typem jsou pak metody s využitím znalosti gradientu. Reprezentantem metod přímého vyhledávání jsou heuristiky, které se používají v případě, kdy je preferováno zkrácení doby řešení problému na

úkor dosažení optimální strategie, případně když není znám způsob, jak najít optimální strategii (např. metoda pokus-omyl či algoritmus Nelder-Mead). [21]

1.3.4 Monte Carlo

Mezi stochastické optimalizační metody, konkrétně mezi tzv. metaheuristiky, patří i metody typu Monte Carlo. Stejně jako metoda hrubé síly implikují několik problémů, které znesnadňují jejich aplikaci na úkoly RL: jsou použitelné pouze pro konečné Markovovy rozhodovací procesy v prostředích, u nichž lze odhady hodnot všech stavů uchovávat v přímo v paměti počítače ve formě tabulky (*tabular case*) nebo pole (*array*). [22] Dalším omezením metod Monte Carlo je jejich použitelnost jen na problémy epizodického charakteru a problematická může být i značná časová náročnost při vyhodnocování suboptimální strategie. Na rozdíl od metod dynamického programování nepředpokládají úplnou znalost prostředí, vyžadují pouze zkušenosti – vzorové posloupnosti akcí, stavů a odměn získaných pomocí online interakce s prostředím. Je to výhodné z toho pohledu, že od agenta není vyžadována řádná apriorní znalost fungování prostředí, a přesto může dosáhnout optimálního chování. Agent se může učit i pomocí simulované zkušenosti; tento přístup nicméně vyžaduje model, ale jednoduššího typu než u dynamického programování (kde musí model generovat kompletní distribuce pravděpodobností všech možných tranzic – přechodů do dalších stavů). [8] Úkolem agenta je tady odehrát určitý počet kol, nashromáždit odměny za všechny stavy, které agent vyzkoušel a určit jejich výběrový průměr pro každý z těchto stavů dle vztahu 1.8. Tím získáme hodnotu výběrové funkce:

$$V(s) = E[G(t)|S(t) = s] \approx \frac{1}{N}\sum_{i=1}^{N} G_{\mathrm{i,s}}. \tag{1.8}$$

Jsou tedy známy jen hodnoty stavů, které agent prošel, hodnoty ostatních stavů jsou neznámé. Postup je následující:

1. Inicializace náhodné strategie
2. Pokud nekonverguje:
   - zahrát jednu herní epizodu a spočítat zisky za každý stav – vyhodnotit současnou strategii
   - stejně jako u dynamického programování upravit strategii na základě současné hodnoty $Q(s, a)$

Metoda má tyto nevýhody:

- vyžaduje zahrání mnoha epizod, je nepoužitelná u úkolů neepizodického charakteru, případně agenta dovede do situace, kdy např. chodí v kruhu a epizoda

by pokračovala donekonečna. Řešením jsou úpravy, které zamezí vzniku nekonečných smyček a epizodu v takovém případě ukončí, případně je za ni agent oceněn velkou negativní odměnou a příště už ji nezvolí,

- řada možných stavů může zůstat neprozkoumána a agent tak nezjistí, zda by jejich navštívení jeho kumulativní odměnu zvýšilo či snížilo. Řešením by mohlo být začít každou novou epizodu z nového, náhodně zvoleného místa a zvýšit tak míru explorace. Jinou možností by mohla být strategie výběru akce ϵ-greedy, kdy je ve většině případů zvolena akce s nejvyšší odhadovanou hodnotou („nejchamtivější akce“), ale existuje i malá pravděpodobnost ϵzvolení náhodné akce, a to nezávisle na předchozí zkušenosti. Tím by bylo zajištěno, že při dostatečném počtu pokusů (epizod) bude vyzkoušena každá akce a nalezeny ty optimální, [23]
- vyžaduje plnou kontrolu nad prostředím, což není vždy proveditelné.

TD(λ) Metoda TD(λ) představuje kombinaci metod Monte-Carlo a TD(0). Pokud se parametr $\lambda = 0$, metoda je výše popsaná TD(0), pokud $\lambda = 1$, pak je TD(1) ekvivalentní metodě Monte Carlo. [22]

1.3.5 Dynamické programování

Metoda dynamického programování používá pro výpočet užitku Bellmanovy rovnice. [3] Bellman navrhl opustit klasický přístup k řešení matematických problémů – vypočítat zisk z každé jednotlivé uskutečnitelné strategie a následně vybrat maximální hodnotu – který považoval za nepraktický, zvláště v případě vícefázových rozhodovacích procesů. Vyslovil hypotézu, že není nutné znát kompletní sekvenci rozhodnutí, nýbrž stačí mít k dispozici obecný předpis k rozhodování (z hlediska aktuálního stavu systému) použitelný v kterékoli fázi. Formuluje tzv. princip optimality: „Optimální strategie má tu vlastnost, že ať jsou počáteční stav a rozhodnutí jakákoli, další rozhodnutí musí vytvořit optimální strategii s ohledem na stav vyplývající z počátečních rozhodnutí.“ [24] Dynamické programování je v praxi nevýhodné tím, že vyžaduje při každé iteraci projít všechny stavy, což může znamenat problém, vezmeme-li v úvahu exponenciální nárůst počtu stavů nebo nekonečný počet možných stavů. Další nevýhodou je, že vyžaduje znalost modelu dynamiky prostředí, a to především s ohledem na pravděpodobnosti přechodu do dalších stavů. Agent se v tomto případě navíc neučí ze zkušenosti. Na rozdíl od dynamického programování metody TD a Monte Carlo umožňují agentovi učit se ze zkušeností, nevyžadují

[3] Richard Bellman zavedl ve své práci *The Theory of Dynamic Programming* z roku 1954 terminologii používanou i v teorii RL: termínem strategie označuje sekvenci rozhodnutí a strategii, která je dle určitých předem stanovených kritérií nejvýhodnější, jako optimální strategii. [24]

znalost modelu dynamiky prostředí a aktualizují funkční hodnoty jen pro stavy, ve kterých se agent již nacházel.

1.3.6 Q-learning a deep learning

Následující podkapitoly se zaměří na metody aktivního učení. Q-učení je společně s algoritmem Sarsa hlavním zástupcem metod založených na aktivním učení, kdy si agent sám určuje svou strategii a při tom zkoumá okolní prostředí. Metoda Q-learning na rozdíl od ostatních RL algoritmů, jejichž cílem je naučit agenta strategii a RL algoritmů zaměřujících se na hodnotové funkce (*value functions*), propojuje současný stav (rozhodování se, která akce bude mít pro agenta největší hodnotu – udává tzv. *value function*) a akci, která bude následovat (strategii – udává tzv. *policy function*), jinými slovy strategie agenta je určena tím, co má pro něj hodnotu, a to posléze určí akci, kterou provede. Toto propojení se nazývá *Q-function* a jejím výsledkem je částečná odměna za provedenou dvojici stav-akce (s, a), tedy jinými slovy $Q(s,\ a)$ udává, jakou odměnu obdrží agent nacházející se ve stavu s, pokud zvolí akci a. [25] Výhodné je, že při porovnávání užitků z možných akcí Q-učení nepotřebuje znát model prostředí. Q-učení je off-policy algoritmus, kdy nezáleží na tom, jakou akci zvolíme v příštím kroku, cíl zůstává stejný. Zisk je určován jako maximální q ze všech možných následných akcí, tzn. nezáleží na tom, jakou akci agent zvolí v příštím kroku, protože vždy zvolí tu, která bude znamenat maximální možnou. Prostřednictvím metod off-policy se lze učit různé strategie na základě chování a odhadu, přičemž aktualizace je možná i na základě hypotetických odhadů o doposud nevyzkoušených akcích (naproti tomu on-policy metody provádí aktualizace výhradně na základě již vyzkoušených akcí a neumožňují separovat kontrolu od explorace). Vzhledem k tomu, že strategie chování je obvykle *soft*, je v chování agenta zajištěna určitá míra explorace 1.9:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{\mathrm{a'}} Q(s',a') - Q(s,a)]. \tag{1.9}$$

To ale platí jen pro nejjednodušší případ; v reálných, tedy složitějších situacích se při predikci dalšího stavu bere v úvahu více možných stavů. Vyvstává ovšem problém komplexnosti možných stavů-akcí. Kvůli jejich množství je nemožné v rozumném čase propočítat všechny částečné odměny, které z nich vyplývají. Q-funkce řeší tento problém omezením na prozkoumání všech možných stavů, které mohou nastat v příštím kroku (na rozdíl od *value function*, která bere v úvahu všechny možné kroky) a identifikaci té akce, která povede k nejlepšímu výsledku – největší odměně. Strukturu agenta s Q-učením ukazuje obrázek 1.3 Vzhledem k tomu, že mnoho stavů může být analogických, agent může uplatnit své znalosti nabyté předchozím rozhodováním.

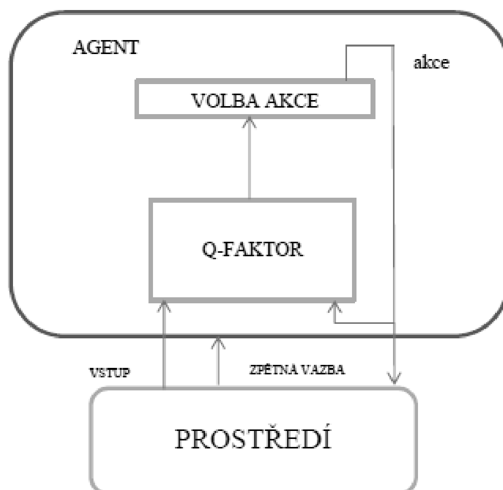
Algorithm 2 Pseudokód – Q-Learning

```
1: Initialize Q(s,a) arbitrarily
2: Repeat (for each episode):
3:     Initialize s
4:     Repeat (for each step of episode):
5:         Choose a from s using policy derived from Q
6:         Take action a, observe r, s'
7:         Q(s,a) ← Q(s,a) + α [r + γ max_a' Q(s',a') − Q(s,a)]
8:         s ← s'
9:     until s is terminal
```

Pomocí deep learning lze kombinovat podobné stavy tak, že budou mít obdobné Q-hodnoty. Pro to lze využít neuronové sítě, ale i ty je nutné naučit, jak uložené poznatky využívat a odhadovat ocenění jednotlivých akcí. Pokud uložíme stávající stav – obraz do konvoluční neuronové sítě (CNN) jako novou zkušenost (*experience*), ta bude schopna, pokud v budoucnu uvidí stejný nebo podobný obraz, zvolit stejnou akci jako při prvním setkání, tzn. bude schopna generalizace. [26] Podmínkou je, aby síť měla uložen obraz společně s příslušnou akcí a oceněním, a zapamatovala si je jako jednolitý celek, nikoli jako separované vědomosti. Síť si přitom nemusí pamatovat své začátečnické zážitky, po určité době ukládání se z doposud uložených zážitků vytvoří reprezentativní náhodný výběr a aktualizují váhy, v nichž má síť informace uloženy a vyberou ty, které maximalizují užitky (Q-hodnoty) ze všech akcí provedených během doby ukládání těchto zážitků. [25] Informace se v neuronových sítích nacházejí v jednotlivých vahách rozdělených po celé síti, nikoli na jednom místě jako v případě klasického počítače. Neuronovou síť to přibližuje fungování lidského mozku (a to i v tom smyslu, že pokud se s některými informacemi síť setká opakovaně, propojení mezi neurony se posilují díky násobení dat prostřednictvím vah). Algoritmus deep learning používá hluboké neuronové sítě pro nelineární aproximaci. Pro zvýšení pravděpodobnosti konvergence metody se využívá *experience replay* a váhy neuronové sítě jsou aktualizovány až po uplynutí stanoveného počtu iterací, což vede ke stabilizaci tréninkových dat i celého učícího procesu.

1.3.7 Paralelizace zpětnovazebního učení

Algoritmus založený na využití asynchronní metody pro hluboké zpětnovazební učení (přesněji jde o úpravu standardních RL algoritmů do jejich asynchronní varianty) využívá pro optimalizaci regulátorů hluboké neuronové sítě asynchronní gradientní sestup (tzv. algoritmus slézání kopce, *hillclimbing*). Výhodou algoritmu je jeho jednoduchost; nepoužívá vyhledávací strom, stačí jen uložit hodnotu aktuálního stavu. Gradientní sestup je typem algoritmu s iterativním vylepšováním, který je založen na předpokladu, že daný stav již sám o sobě poskytuje veškeré informace nutné pro vyřešení úlohy. Vyjdeme tedy z této konfigurace a jejími postupnými úpravami



Obr. 1.3: Agent s Q-učením

se snažíme dosáhnout lepšího výsledku – optimálního řešení. Algoritmus si pamatuje jen data o aktuálním stavu a sousedních stavech bezprostředně následujících. Aktér-agent ve smyčce stoupá ke stále vyšším hodnotám (do kopce). Pokud je mu nabídnuto více nejlepších stavů, náhodně zvolí jeden z nich. Nevýhodou daného přístupu je, že nemusí vést k optimálnímu řešení i přes značný počet iterací. Není totiž dosaženo globálního maxima, ale pouze lokálního, po jehož nalezení se algoritmus ukončí. [16] Řešením je úprava algoritmu, která umožní jeho restart po nalezení lokálního maxima a zahájení cesty z jiného místa (tzv. slézání kopce s náhodným počátkem). Ukládá se pouze hodnota nejlepšího nalezeného řešení. Algoritmus je ukončen ve chvíli, kdy se nejlepší řešení po určitý počet iterací nezmění, případně uplyne předem stanovený počet iterací. Vzhledem ke složitosti úkolů RL a tedy velkému množství lokálních maxim nelze očekávat rychlé nalezení řešení. Dalším problémem je stav, kdy aktér-agent nestoupá nahoru, ale nachází se na rovině. V tomto případě vrací algoritmus stále stejnou hodnotu, což vede ke zvolení náhodné cesty prostředím. [27]

Tato implementace řeší problém nestability, kterou přinášela kombinace RL algoritmů a hlubokých neuronových sítí. Původní řešení limitovala RL na off-policy učení. RL nesplňuje jeden ze základních požadavků učení neuronové sítě – nezávislost vstupních dat. Řešení spočívá v podobě ukládání vstupů jako čtveřice (aktuální stav, akce, stav po vykonání akce, odměna za akci) do vyrovnávací paměti (*experience replay*), z níž se pak náhodně vybere dávka dat (*batch*) případně jsou data náhodně vzorkována z různých časových okamžiků a poté poskytnuta síti k učení. [28] Tento postup je však nevýhodný z hlediska paměťových nároků a výpočetní náročnosti. Autoři proto navrhují místo ukládání vstupů do bufferu asynchronně řídit

více agentů paralelně se učících ve více prostředích. Tím by byl vyřešen problém korelace vstupů, protože agenti paralelně existující v jednotlivých prostředích by v každém okamžiku získávali širokou škálu nových stavů; a byla by možná efektivní aplikace on-policy (SARSA) i off-policy (Q-learning) algoritmů v kombinaci s hlubokými neuronovými sítěmi. Z praktického hlediska by již pro implementaci nebyl potřeba speciální hardware, stačila by běžná vícejádrová centrální procesorová jednotka (CPU). [29]

Při měření výkonnosti paralelních algoritmů je třeba použít jiná kritéria než u sekvenčních algoritmů. Především musíme vzít do úvahy počet použitých procesorů. V ideálním případě paralelní algoritmus povede k lineárnímu zrychlení učícího procesu, tzn. stoupne-li počet procesorů n-krát, doba učení se n-krát zmenší a při použití p procesorů je výsledku dosaženo p-krát rychleji. Dalším kritériem je paralelní čas – doba od zahájení výpočtu do okamžiku zakončení výpočtu posledním procesorem (měří se inkrementací výpočetních a komunikačních kroků, tzn. záleží na výpočetní a komunikační složitosti). Komunikační složitost a náklady lze snížit použitím jednoho stroje s více současně prováděnými vlákny – samostatně prováděnými výpočetními toky – CPU (platí pro vícejádrové a víceprocesorové systémy), vedlejším efektem by mohlo být zrychlení provádění vypočetních operací a lepší efektivita programu (odpadá nutnost posílání parametrů mezi jednotlivými počítači a parametrovým serverem, kde jsou uchovávány hodnoty jednotlivých parametrů). U paralelně se učících agentů bylo pozorováno, že budou s velkou pravděpodobností prozkoumávat různé části prostředí. Pro agenty v jednotlivých vláknech by tedy šlo navrhnout strategii „na míru“. To by pomohlo snížit pravděpodobnost korelace při paralelních aktualizacích parametrů. [30]

Jednokrokové Q-učení a jeho asynchronní varianta Asynchronní úprava algoritmu Deep Q-Learning spočívá v současném paralelním trénování v několika vláknech s kopiemi herního prostředí, přičemž každému vláknu je přiřazena jiná explorační strategie, což zlepšuje robustnost i výkonnost. [29] Výhodou tohoto přístupu je z hardwarového pohledu možnost použití vícejádrového procesoru CPU namísto grafického GPU a z programového hlediska zrychlení konvergence. Díky tomu, že agenti operují ve vzájemně nezávislých prostředích, se učící proces stabilizuje. [31] Dalším pozitivem je, že si samostatně se učící agent nemusí ukládat své minulé zkušenosti, parametry θ užitkové funkce $Q(s, a, \theta)$ jsou aktualizovány iterativně dle vztahu 1.10. Pseudokód pro asynchronní jednokrokové učení je uveden na 3:

$$L_i(\theta_i) = E(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))^2. \tag{1.10}$$

Pro srovnání *on-policy* asynchronní jednokroková SARSA aktualizuje parametry θ

dle vztahu 1.11:

$$L_i(\theta_i) = E(r + \gamma Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))^2. \tag{1.11}$$

Algorithm 3 Asynchronous one-step Q-learning – pseudocode for each actor-learner thread.

1: Assume global shared θ, θ^-, and counter $T = 0$.
2: Initialize thread step counter $t \leftarrow 0$
3: Initialize target network weighs $\theta^- \leftarrow \theta$
4: Initialize network gradients $d\theta \leftarrow 0$
5: Get initial state s
6: **repeat**
7: Take action α with ϵ-greedy policy based on $Q(s, a; \theta)$
8: Receive new state s' and reward r

$$f(x) = \begin{cases} r & \text{for terminal s'} \\ r + \gamma max_{\mathrm{a}'} Q(s', a'; \theta^-) & \text{for non-terminal s'} \end{cases}$$

9: Accumulate gradients wrt $\theta : d\theta \leftarrow d\theta + \frac{\delta(y - Q(s,a;\theta))^2}{\delta\theta}$
10: $s = s'$
11: $T \leftarrow T + 1$ and $t \leftarrow t + 1$
12: **if** T mod $I_{\mathrm{target}} == 0$ **then**
13: Update the target network $\theta^- \leftarrow \theta$
14: **end if**
15: **if** t mod $I_{\mathrm{AsyncUpdate}} == 0$ or s is terminal **then**
16: Perform asynchronous update of θ using $d\theta$.
17: Clear gradients $d\theta \leftarrow 0$.
18: **end if**
19: **until** $T > T_{\max}$

N-krokové Q-učení a jeho asynchronní varianta Na rozdíl od jednokrokového Q-učení tato metoda aktualizuje parametry po n krocích učícího procesu θ dle vztahu 1.12:

$$r_t + \gamma r_{t+1} + ... + \gamma^{\mathrm{n}-1} r_{t+n} + max_{\mathrm{a}} \gamma^{\mathrm{n}} Q(s_{t+n+1}, a; \theta_i). \tag{1.12}$$

Algoritmus čeká $t_{\max}$ časových okamžiků nebo až je dosažen konečný stav. Za tento časový interval (počínaje poslední aktualizací parametrů až do $t_{\max}$ nebo terminálního stavu) agent obdrží odměnu. Algoritmus následně určí gradienty pro každý pár stav-akce za minulých n-kroků a aplikuje je na jeden gradientní krok. [29] Pseudokód pro asynchronní n-krokové učení viz 4:

Algorithm 4 Asynchronous n-step Q-learning – pseudocode for each actor-learner thread.

1: Assume global shared parameter vector θ.
2: Assume global shared target parameter vector θ^{-}.
3: Assume global shared counter $T = 0$.
4: Initialize thread step counter $t \leftarrow 1$
5: Initialize target network parameters $\theta^{-} \leftarrow \theta$
6: Initialize thread-specific parameters $\theta^{'} = \theta$
7: Initialize network gradients $d\theta \leftarrow 0$
8: **repeat**
9: Clear gradients $d\theta \leftarrow 0$
10: Synchronize thread-specific parameters $\theta^{'} = \theta$
11: $t_{\text{start}} = t$
12: Get state s_t
13: **repeat**
14: Take action a_t according to the ϵ-greedy policy based on $Q(s_t, a; \theta^{'})$
15: Receive reward r_t and new state s_{t+1}
16: $t \leftarrow t + 1$
17: $T \leftarrow T + 1$
18: **until** terminal s_t or $t - t_{\text{start}} == t_{\max}$

$$R = \begin{cases} 0 & for\ terminal\ s_t \\ max_{\text{a}} Q(s_t, a; \theta^{-}) & for\ non-terminal\ s_t \end{cases}$$

19: Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \frac{\sigma(y - Q(s,a;\theta))^2}{\sigma\theta}$
20: $s = s'$
21: $T \leftarrow T + 1$ and $t \leftarrow t + 1$
22: **for** $i \epsilon \{t-1, ..., t_{\text{start}}\}$ **do**
23: $R \leftarrow r_i + \gamma\ R$
24: Accumulate gradients wrt $\theta^{'}$: $d\theta \leftarrow d\theta + \frac{\sigma(R - Q(s_t, a_t; \theta^{'}))^2}{\sigma\theta^{'}}$
25: **end for**
26: Perform asynchronous update of θ using $d\theta$
27: **if** T mod $I_{\text{target}} == 0$ **then**
28: $\theta^{-} \leftarrow \theta$
29: **end if**
30: **until** $T > T_{\max}$

Metoda Q(σ) – jednotící algoritmus Vícekroková metoda Q(σ) sjednocuje již existující vícekrokové kontrolní metody založené na temporální diferenci a zavádí nové vzorkovací kritérium σ, které je aplikovatelné na učení *on-policy* i *off-policy*. Kritérium algoritmu umožňuje, aby míra vzorkování byla v každém kroku souvisle měněna (na jednom konci přitom stojí Sarsa umožňující plné vzorkování, na druhém Expected Sarsa – čisté očekávání). Nastavení kritéria na hodnotu mezi těmito dvěma extrémy (tzn. kombinace existujících algoritmů) vede k nejlepším výsledkům a kombinaci lze dynamicky měnit tak, aby bylo dosaženo lepšího výkonu. [19]

1.3.8 SARSA

SARSA znamená sekvenci stav-akce-odměna-stav-akce (*State-Action-Reward-State-Action*). Agent nacházející se ve stavu s, zvolí akci a a obdrží odměnu r. Přesune

se do stavu s', zvolí akci a' a obdrží za ni odměnu. Následně se vrátí a aktualizuje hodnotu akce a, kterou zvolil ve stavu s. Aktualizace je tedy provedena až po další akci dle vztahu 1.13:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r(s) + \gamma Q(s',a') - Q(s,a)). \tag{1.13}$$

V Q-učení je první krok totožný, ale po obdržení odměny 1 agent zjišťuje, jaká je maximální možná odměna za možné akce ve stavu 2, tu zvolí a následně tuto hodnotu použije pro aktualizaci hodnoty při výběru akce 1 ve stavu 1. [32] Agent vždy zvolí nejvýnosnější akci, která se ve stavu 2 nabízí. Pseudokód metody Sarsa viz 5:

Algorithm 5 Sarsa – pseudocode

```
1:  Initialize Q(s,a) arbitrarily
2:  Repeat (for each episode):
3:      Initialize s
4:      Choose a from s using policy derived from Q
5:      Repeat (for each step of episode):
6:          Take action a, observe r, s'
7:          Choose a' from s' using policy derived from Q
8:          Q(s,a) ← Q(s,a) + α [r + γQ(s',a') − Q(s,a)]
9:          s ← s'; a ← a'
10:     until s is terminal
```

Jde o On-Policy algoritmus pro učení prostřednictvím temporální diference. V porovnání s Q-učením u metody SARSA maximální odměna za další stav není nezbytně použita pro aktualizaci Q-hodnot, ale místo toho je nový stav a odměna zvolena s využitím stejné strategie, která určila původní akci. [33]

2 VÝSLEDKY STUDENTSKÉ PRÁCE

2.1 Testovací hra Had

2.1.1 Princip hry a herní prostředí

Zjednodušeně řečeno je cílem agenta pouhou interakcí s herním prostředím maximalizovat očekávanou odměnu v každém stavu, do něhož se dostane určením optimální strategie. K tomu je nejprve nutné obraz prostředí tak, jak ho vnímá člověk, zpracovat do podoby, v níž mu počítačový agent bude schopen rozumět. [34] Každý herní snímek představuje stav, k němuž náleží určitý počet akcí, lišící se pro každou hru. V případě zvolené hry Had jde o čtyři možné akce – pohyb nahoru, doprava, doleva a dolů. Agent se má přitom učit sám, bez vnějšího zasahování do paměti s uloženými herními zážitky.

Automatizovanou hru Had lze implementovat mnoha různými způsoby. Kromě algoritmů umělé inteligence (například uspořádané prohledávání (*best first search*) na prohledávání stavového prostoru nebo algoritmus A* (*A star*) s hladovým principem pro nalezení optimální cesty) lze pro automatizaci hry použít i další algoritmy (např. brute force), které se navzájem liší svou výkonností. [35] Hra Had nemá tak složitý algoritmus, který by nešel naprogramovat jiným způsobem, než pomocí neuronových sítí, navíc jsou známy i všechny kombinace vstupů a výstupů, které v ní mohou nastat. Cílem vytvořeného programu však bylo naučit agenta hrát hru pouhým jejím sledováním, se zpětnou vazbou, na základě které se učí zlepšovat své kroky při prozkoumávání okolního prostředí.

Herní prostředí tvoří plocha o rozměrech 200 x 200 pixelů. Cílem hry je, aby had zůstal naživu co nejdéle a nasbíral co nejvíce jídla, tzn. aby byla jeho délka co největší (agent může upřednostnit buď rychlost (najít co nejvíce jídla a dosáhnout nejvyššího skóre) nebo spolehlivost (zůstat naživu co nejdéle)). Na počátku učení se had o rozměrech 60x20 pixelů nachází na herní ploše (stav), na které je náhodně umístěno jídlo (odměna) o rozměrech 20x20 pixelů. Jeho úkolem je nalézt odměnu co nejrychleji (tedy zvolit akci, která k odměně povede co nejkratší cestou). Může se přitom pohybovat pouze ortogonálně (doprava, doleva, nahoru a dolů), diagonální pohyby nejsou povoleny. Pokud se mu to podaří, obdrží kladnou odměnu +4 bodů a jeho délka vzroste o 20 pixelů. V tu chvíli je náhodně generováno nové jídlo a hra pokračuje. Naopak v okamžiku, kdy had narazí do okrajů herní plochy, případně se dotkne sám sebe, obdrží negativní odměnu −4 body, herní epizoda je ukončena, je ukázáno výsledné skóre – ve výchozím stavu je však zobrazování skóre deaktivováno (počet nalezených jídel, respektive délka, o kterou had vzrostl) a hra restartována. Pravidla hry, která nastavují mantinely, v jejichž rámci může agent hrát, jsou shrnuta

v následujících bodech:

- had se může pohybovat jen v pravých úhlech,
- nemůže přeskočit do opačného směru (tzn. pohybuje-li se doprava, nemůže doleva a naopak, při pohybu nahoru nemůže dolů a naopak),
- pohybuje se bez zastavení,
- zahyne, pokud se dotkne hranic herního prostředí,
- zahyne, pokud se dotkne sám sebe,
- pokud najde jídlo, jeho délka vzroste o 20 pixelů (jeho pohyb herním prostředím se tak postupně stále více znesnadňuje),
- každý segment těla hada vždy následuje jeho první článek („hlavu“), který udává směr a detekuje změny v prostředí,
- na herní ploše je v daný okamžik jedno jídlo, které se nachází na neměnné pozici až do chvíle, než ho had nalezne; v tom okamžiku je náhodně generováno umístění nového jídla a hra pokračuje.

Byla tedy zachována původní koncepce hry s neomezeným časovým limitem, během kterého může had jídlo nasbírat. Herní plocha o rozměrech $n \times n$ je reprezentována plochou $G = (V,E)$, kde V je soubor vertexů v_i, kdy každý vertex koresponduje s jedním čtvercem herní plochy. Jedním z hlavních problémů herních algoritmů RL je velikost stavového prostoru. Pokud bychom uvažovali herní plochu o rozměru $n \times n$, každý čtverec může reprezentovat jeden ze čtyř stavů – obsahuje jídlo, obsahuje hlavu hada, obsahuje tělo hada, je prázdný. Pak by byl stavový prostor velký n^8, což by u větších herních ploch prakticky znemožňovalo agentovo učení. [33] Právě extrémně velký počet možných stavů znesnadňuje zdánlivě jednoduchou hru pro efektivní aplikaci zpětnovazebního učení. Učící proces lze urychlit například tím, že bude brána v úvahu ne absolutní, nýbrž pouze relativní pozici hada a jídla, tzn. místo určování souřadnic od okrajů herní plochy se určuje posunutí z pozice, kde by se had normálně nacházel. Každý stav v takto redukovaném stavovém prostoru je pak indikován dle toho, zda se v levém, pravém nebo přímém směru pohybu hada nachází okraj herní plochy a dále relativní pozicí jídla a konce těla hada s ohledem na jeho hlavu. Tím se velikost stavového prostoru zmenší na 128, a učící proces se tak značně usnadní.

2.1.2 Návrh a implementace programu

Nastavení pravidel odměny Následující odstavec bude věnován úvaze o správném nastavení odměn v implementované hře, které je u zpětnovazebního učení jedním z klíčových problémů, protože jen tak bude agent motivován dosahovat co nejvyššího skóre. Nastavení odměn pro RL agenta může být (jak bylo zmíněno v úvodu) záludné a je nutné správně vyvážit jednotlivé možné typy odměn. Testovaná hra je

specifická tím, že jejím cílem je herní epizodu ukončit co nejpozději, protože konec pro agenta v tomto případě neznamená obdržení kladné odměny (jako v jiných hrách). Stěžejním úkolem pro agenta je zůstat naživu, jinými slovy nenarazit do okrajů herní plochy nebo sám do sebe. Logicky by tak měl být kladně odměněn za každý herní krok, po němž zůstal naživu a negativně v opačném případě. Při trénování se však ukázalo, že při takovém nastavení odměn by agent nebyl motivován hledat jídlo, protože by kladnou odměnu získával již za pouhý pohyb prostředím. Na druhé straně se agent může zaměřit na dosažení krátkodobých výher – tzn. nalezení co největšího množství jídla v co nejkratším časovém horizontu (možnost použití časového limitu pro nalezení jídla však není v této hře využita). Zvyšování skóre je přitom hlavní cíl hry, takže odměna za nalezení jídla musí být dostatečně vysoká, aby byl agent motivován jídlo hledat. Agent přitom může zpočátku obtížně identifikovat odměnu spojenou s nalezením jídla, nejprve totiž jídlo nachází náhodně a musí si získání odměny vůbec uvědomit. Existuje také riziko, že u agenta převáží touha po minimalizaci rizika spojená s nárazem do zdi nebo do sebe sama a bude se pohybovat dokola v nekonečné smyčce, která mu zajistí jistotu přežití. Je tedy nutné ho přimět k systematickému průzkumu prostředí. To je řešeno prostřednictvím strategie epsilon-greedy, která je přiblížena v následující podkapitole.

Ve hře jsou stanoveny tři druhy odměn: kladnou odměnu +4 agent obdrží při nalezení jídla, zápornou −4 při nárazu do okrajů herní plochy či do sebe sama. Třetí typ odměny je definován v proměnné odmena_jinak a byl původně nastaven na nulovou hodnotu. Jde o odměnu za pohyb prostředím, kdy nenastane žádný z předchozích stavů (tzn. had nenajde jídlo, nenarazí do okrajů herního prostředí ani do sebe sama). Původní nastavení proměnné odmena_jinak na nulu bylo upraveno na hodnotu −0,001, protože původně se had i při delších dobách tréninku pohyboval prostředím spíše náhodně a nestaral se o to, jak dlouho mu trvá nalezení odměny.

Agent Princip činnosti agenta je implementován v třídě Agent. Agent při svém učení uplatňuje strategii epsilon-greedy, kdy vyšší hodnota epsilon znamená vyšší pravděpodobnost volby náhodné akce a podporu explorace na úkor exploatace – viz funkce zvol_akci:

```
def zvol_akci(self, stav):
    pohyb = np.zeros([4])
    if (random.random() <= self.epsilon):
        print("Zvolena nahodna akce.")
        akce = random.choice(nahodny_smer)
        pohyb[akce] = 1
        #print(pohyb)
    else:
```

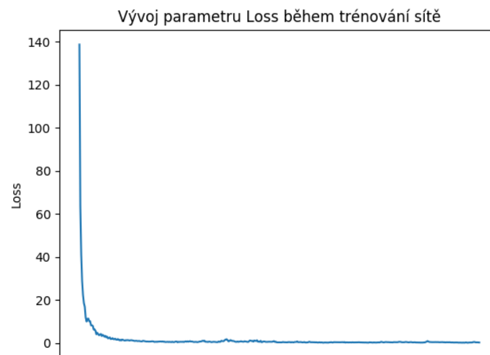
```
print("Zvolena akce s maximalni Q—hodnotou")
Q_hodnoty = nn.predict(stav)
maxQ = np.argmax(Q_hodnoty)
akce = maxQ
pohyb[akce] = 1
#print(pohyb)
```

Pohyb je detekován prostřednictvím numpy pole, které je na začátku vyplněno nulami. Zpočátku jsou akce agenta náhodné (náhodný výběr z listu [0=nahoru, 1=doprava, 2=doleva, 3=dolu]), než zjistí, jaký je cíl hry a prozkoumá možné stavy. V opačném případě zvolí akci, které je přiřazena maximální Q-hodnota. Hodnota epsilon je obvykle nastavována jako kladná konstanta v rozmezí od nuly do jedné (v tomto případě bude počáteční hodnota epsilon nastavena na 0,9999). Čím vyšší je její hodnota, tím pravděpodobněji agent zvolí v daném stavu náhodnou akci a upřednostní exploraci na úkor exploatace. Protože se dá očekávat, že v průběhu tréninku bude agent nabývat stále více vědomostí o okolním prostředí, hodnota explorace se bude snižovat a agent se zaměří spíše na zužitkování svých znalostí – exploataci.

Aby vytvořený program fungoval v souladu s tímto předpokladem, bude hodnota epsilon snižována ve funkci trenovani() z počáteční hodnoty 0,9999 o 0,001 každé tréninkové kolo až na případnou minimální hodnotu 0.0001, která pak již zůstane po zbytek tréninku konstantní. Tímto způsobem bude řešen problém vyvážení explorace a exploatace. Epsilon-greedy strategie je tedy implementována následujícím způsobem: pokud je hodnota epsilon větší než náhodná hodnota, bude vybrána náhodná akce z možných akcí v daném stavu. V opačném případě bude ze všech možných akcí zvolena akce, která přinese agentovi maximální užitek. Diskontní faktor γ je nastaven na 0,8; při nastavení parametru na hodnotu blízkou nule se bude agent snažit dosáhnout spíše okamžité odměny. Na druhé straně při nastavení na hodnotu blízkou jedné se zaměří na dosažení budoucí odměny.

Trénování konvoluční neuronové sítě Pro trénování neuronové sítě byla zvolena metoda hlubokého zpětnovazebního učení (*Deep Reinforcement Learning*), která je založena na následujícím přístupu: do sítě jsou dodávány herní obrázky a síť následně na výstupu předloží čtyři Q-hodnoty, jednu pro každou možnou akci. Takto jsou v rámci jednoho dopředného průchodu získány všechny Q-hodnoty pro daný stav. [36] K trénování hry byla použita DQN – vícevrstvá konvoluční neuronová síť (CNN) trénovaná pomocí Q-učení. Po vyzkoušení různých architektur byla zvolena nestandardní architektura neuronové sítě, v níž chybí poolingové vrstvy, a to z toho důvodu, že tyto vrstvy slouží k podvzorkování a okolní pixely seskupují do jedné hodnoty; v případě zvolené testovací hry by to však bylo kontraproduktivní, protože

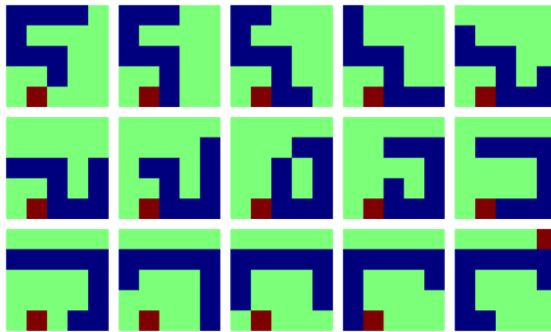
zachování informací o přesné pozici hada a jídla v prostoru je klíčové. Důsledek vypuštění poolingových vrstev je diskutabilní; na jednu stranu může neúměrně zvýšit dobu trénování vlivem zvýšení dimenzionality dat, na opačné straně se toto zjednodušení architektury nemusí ve výsledku vůbec projevit ztrátou přesnosti odhadů sítě a naopak může mít pozitivní efekt snížení výpočetní náročnosti. Jako řešení je navrhováno vrstvy max pooling nahradit další konvoluční vrstvou s vyšší hodnotou parametru stride. [37] Toto alternativní řešení je aplikováno v použité architektuře neuronové sítě. Zvolená architektura ukazovala při trénování dobré výsledky, jak lze vidět z grafu vývoje parametru Loss při evaluaci modelu pomocí metody evaluate 2.1.



Obr. 2.1: Hodnoty parametru Loss při trénování sítě

Dále byla využita opatření pro redukci rizika přeučení sítě a naopak zvýšení její schopnosti generalizace. U zpětnovazebního učení spočívá riziko především v přeučení agenta u raných vzorků získaných z prostředí (kdy jeho znalosti o okolním prostředí nebyly ještě komplexní), které by mohly znesnadnit učení v pozdějších fázích. [38] Toto riziko bude omezeno zavedením regulátoru dropout, jehož výchozí hodnota bude nastavena na 0,15 (tzn. zahození 15 % tréninkových dat), což by mělo zároveň vést i ke zrychlení tréninkového procesu a bude tím zajištěno, že síť se v pozdějších fázích učení nebude spoléhat na nejstarší vzorky a data poskytnutá k učení budou aktuální. Jako vstupy do neuronové sítě jsou použity dva po sobě následující obrázky ze hry, redukované na velikost 70×70 pixelů a transformované do hodnot šedotónové stupnice. Počet vstupních obrázků může být jedním z parametrů s velkým vlivem na výkonnost sítě; více obrázků na jednu stranu znamená, že síť má k dispozici více informací, ale zároveň také více parametrů, pomocí kterých může modelovat problém. To vede k různým nepředvídatelným situacím, které znemožňují predikovat chování modelu; např. had upřednostní cíl zůstat naživu a sbírání jídla je pro něj druhořadé, tedy ve snaze neuvíznout a nenarazit sám do sebe se snaží co

nejvíce rozprostřít na herní ploše, jak je vidět na obrázku 2.2:



Obr. 2.2: Had volí cestu přežití na úkor sběru jídla [39]

Po ukončení tréninku se do složky se hrou uloží model sítě, který lze následně načíst pomocí metody load_model se jménem souboru ve formě textového řetězce. V programu je implementována také možnost načíst model z příkazového řádku, a to zadáním nepovinného parametru -l (load) s následným uvedením názvu souboru opět ve formě textového řetězce. Tento parametr lze kombinovat např. s parametrem -trenink, a pokračovat tak v trénování načteného modelu.
Model architektury neuronové sítě na obrázku 2.3 byl získán prostřednictvím modulu knihovny Keras keras.utils.vis_utils [40]:

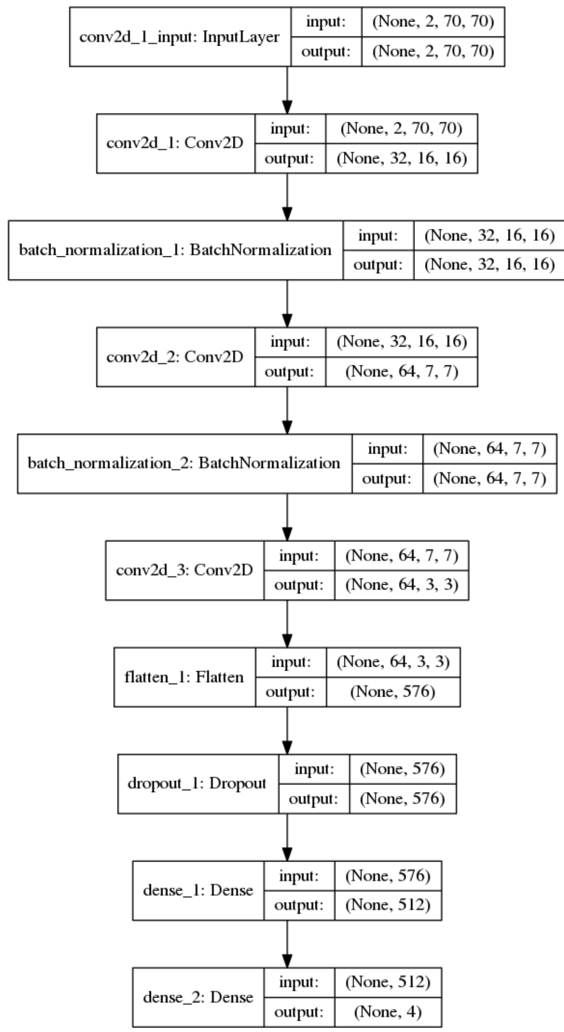
2.1.3 Popis programu

Program byl vytvořen ve vývojovém prostředí PyCharm Community Edition (verze 2016.3) – interpreter Python 3.5 (TensorFlow Backend) [41], v operačním systému Ubuntu 16.04. Pro spuštění programu je potřeba mít nainstalovány:

- Python 3.5
- Keras 2
- PyGame

Tréninkový mód lze spustit z příkazového řádku zadáním příkazů: 'cd had' a 'python3 had.py -m "Trenink", mód hraní pak pomocí příkazů: 'cd had' a 'python3 had.py -m "Hra". Proces trénování je vidět ze zpráv, které se zobrazují v jeho průběhu – viz obrázek 2.4.

Program sestává ze souborů had.py, neuronova_sit.py a souborů, které se ukládají do složky se hrou v průběhu tréninku – modelu architektury neuronové sítě a jejích vah, screenshotu hry a datasetů s uloženými hodnotami. Při testování různých architektur neuronové sítě nabízí Keras také možnost využít načíst váhy sítě z HDF souboru prostřednictvím metody model.load_weights(filepath, by_name=False), stačí jen parametr by_name nastavit na hodnotu True; pak se váhy sítě načtou pouze u



Obr. 2.3: Struktura použité konvoluční neuronové sítě

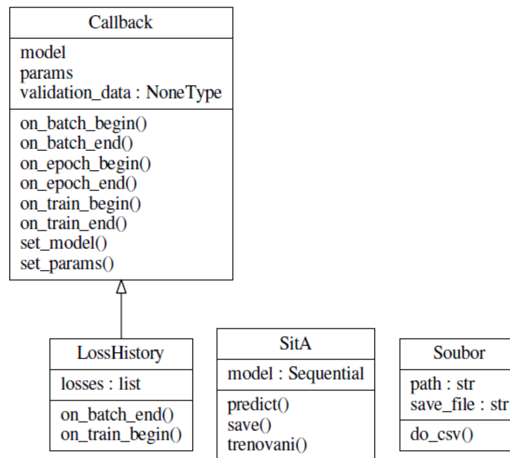
```
dict_keys(['val_loss', 'val_acc', 'acc', 'loss'])
Skore epizody: 0 Epizod celkem: 1 Delka epizody: 196 Odmena za epizodu: -4.195
Skore epizody: 0 Epizod celkem: 2 Delka epizody: 106 Odmena za epizodu: -4.105
Skore epizody: 0 Epizod celkem: 3 Delka epizody: 183 Odmena za epizodu: -4.182
Skore epizody: 0 Epizod celkem: 4 Delka epizody: 194 Odmena za epizodu: -4.193
Skore epizody: 0 Epizod celkem: 5 Delka epizody: 115 Odmena za epizodu: -4.114
Skore epizody: 0 Epizod celkem: 6 Delka epizody: 60 Odmena za epizodu: -4.059
Skore epizody: 0 Epizod celkem: 7 Delka epizody: 137 Odmena za epizodu: -4.136
Skore epizody: 0 Epizod celkem: 8 Delka epizody: 41 Odmena za epizodu: -4.040
Skore epizody: 0 Epizod celkem: 9 Delka epizody: 34 Odmena za epizodu: -4.033
Skore epizody: 0 Epizod celkem: 10 Delka epizody: 50 Odmena za epizodu: -4.049
Skore epizody: 0 Epizod celkem: 11 Delka epizody: 76 Odmena za epizodu: -4.075
Skore epizody: 0 Epizod celkem: 12 Delka epizody: 14 Odmena za epizodu: -4.013
Skore epizody: 0 Epizod celkem: 13 Delka epizody: 63 Odmena za epizodu: -4.062
Skore epizody: 0 Epizod celkem: 14 Delka epizody: 60 Odmena za epizodu: -4.059
Skore epizody: 0 Epizod celkem: 15 Delka epizody: 46 Odmena za epizodu: -4.045
Skore epizody: 0 Epizod celkem: 16 Delka epizody: 85 Odmena za epizodu: -4.084
Skore epizody: 0 Epizod celkem: 17 Delka epizody: 78 Odmena za epizodu: -4.077
Skore epizody: 0 Epizod celkem: 18 Delka epizody: 64 Odmena za epizodu: -4.063
Skore epizody: 0 Epizod celkem: 19 Delka epizody: 173 Odmena za epizodu: -4.172
Skore epizody: 0 Epizod celkem: 20 Delka epizody: 39 Odmena za epizodu: -4.038
Skore epizody: 0 Epizod celkem: 21 Delka epizody: 105 Odmena za epizodu: -4.104
Skore epizody: 0 Epizod celkem: 22 Delka epizody: 10 Odmena za epizodu: -4.009
Skore epizody: 0 Epizod celkem: 23 Delka epizody: 11 Odmena za epizodu: -4.010
Skore epizody: 0 Epizod celkem: 24 Delka epizody: 245 Odmena za epizodu: -4.244
Skore epizody: 0 Epizod celkem: 25 Delka epizody: 54 Odmena za epizodu: -4.053
Skore epizody: 0 Epizod celkem: 26 Delka epizody: 108 Odmena za epizodu: -4.107
Skore epizody: 0 Epizod celkem: 27 Delka epizody: 66 Odmena za epizodu: -4.065
Skore epizody: 0 Epizod celkem: 28 Delka epizody: 48 Odmena za epizodu: -4.047
Skore epizody: 0 Epizod celkem: 29 Delka epizody: 25 Odmena za epizodu: -4.024
Skore epizody: 0 Epizod celkem: 30 Delka epizody: 86 Odmena za epizodu: -4.085
Skore epizody: 0 Epizod celkem: 31 Delka epizody: 86 Odmena za epizodu: -4.085
Skore epizody: 0 Epizod celkem: 32 Delka epizody: 19 Odmena za epizodu: -4.018
Skore epizody: 0 Epizod celkem: 33 Delka epizody: 45 Odmena za epizodu: -4.044
Skore epizody: 0 Epizod celkem: 34 Delka epizody: 82 Odmena za epizodu: -4.081
Skore epizody: 0 Epizod celkem: 35 Delka epizody: 759 Odmena za epizodu: -4.758
Skore epizody: 0 Epizod celkem: 36 Delka epizody: 218 Odmena za epizodu: -4.217
Skore epizody: 0 Epizod celkem: 37 Delka epizody: 65 Odmena za epizodu: -4.064
Skore epizody: 1 Epizod celkem: 38 Delka epizody: 625 Odmena za epizodu: -0.623
Pocet ulozenych vzorku: 625 Skore epizody: 1 Delka epizody: 625
Celkem epizod: 38
Pocet treninku:  4  Epsilon: 0.95
```

Had

Skóre : 1

Obr. 2.4: Průběh tréninku při spuštění z příkazového řádku

vrstev, které mají oba modely společné. Zdrojový kód souboru neuronova_sit.py ve formě UML diagramu je na obrázku 2.5 [42].

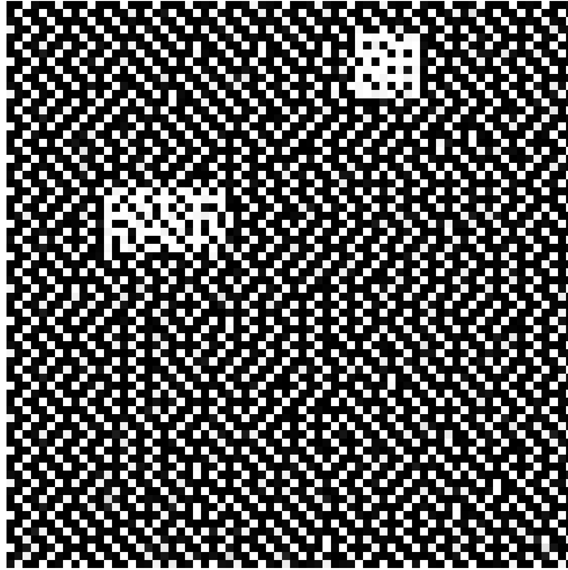


Obr. 2.5: Zdrojový kód souboru neuronova_sit.py ve formě UML diagramu

GUI hry bylo naprogramováno pomocí PyGame. Z metod popsaných v teoretické části byla zvolena metoda hlubokého učení s vícevrstvou konvoluční neuronovou sítí. Jako vstupní obrázky byly použity screenshoty ze hry, které byly upraveny do vhodné formy [43] – předzpracovány ve funkci zaznamenej_stavy(). Forma předzpracovaného obrázku je na obrázku 2.6. Byly vyzkoušeny dva různé možné způsoby, jak screenshoty zpracovat – prostřednictvím knihovny PIL (*Python Imaging Library*), druhá možnost využívá knihovnu scikit-image. Princip fungování lze vidět na následující ukázce:

```
obrazek1 = skimage.color.rgb2gray(obrazek1)
obrazek1 = skimage.transform.resize(obrazek1, (70, 70))
obrazek1 = skimage.exposure.rescale_intensity(obrazek1, out_range=(0, 255))
obrazek1 = obrazek1/255.0
stav_trenink = np.stack((obrazek1, obrazek1, obrazek1, obrazek1), axis=2)
stav_trenink = stav_trenink.reshape(1, stav_trenink.shape[0],
stav_trenink.shape[1], stav_trenink.shape[2])
```

Nejprve je zachycený barevný obrázek herní plochy převeden do hodnot šedotónové stupnice, následně jsou jeho rozměry nastaveny na 70x70 a upraveny hodnoty jeho intenzity. Vydělením hodnotou 255.0 může být použitý rozsah (0, 255) vyjádřen rozsahem 0.0-1.0, kde 0.0 znamená 0 a 1.0 vyjadřuje 255. Pomocí metody stack jsou seskupeny čtyři herní obrázky (pro lepší zachycení rychlosti hada) a pomocí metody reshape přizpůsobeny pro neuronovou síť.



Obr. 2.6: Předzpracování herního screenshotu pro vstup do neuronové sítě

Model neuronové sítě Třída SitA implementuje model konvoluční neuronové sítě použitý pro trénování hry. Po vyzkoušení různých architektur byla zvolena následující struktura CNN: první konvoluční vrstva obsahuje 32 konvolučních filtrů (množství filtrů konvoluční vrstvy udává tzv. hloubku sítě), s velikostí jádra $8x8$, parametrem stride=4 a usměrňovací aktivační funkcí ReLU (ta přináší výhodu menší výpočetní náročnosti v porovnání se sigmoidální aktivační funkcí i tanh – hyperbolický tangens). Provádí konvoluci filtrů a pixelů vstupních screenshotů. Trénování neuronové sítě by mělo být zrychleno zavedením vrstvy batch normalization, která provádí normalizaci dat v tzv. minibatch, namísto jednorázové normalizace na začátku trénování, která umožňuje využít vyšší rychlosti učení. Batch normalization by měla vést i ke zvýšení přesnosti odhadů neuronové sítě; učení sítě bude pokračovat až do momentu, kdy se přesnost jejích odhadů již nebude zlepšovat. Vrstva Batch Normalization též umožňuje vrstvě učit se s větší nezávislostí na ostatních vrstvách. Hlavním důvodem její aplikace je ale omezení rizika přeučení (a to díky jejímu regularizačnímu efektu, který se dále znásobí, pokud tuto vrstvu použijeme společně s vrstvou Dropout; v některých případech je účinek Batch Normalization jako regularizátoru tak silný, že dokonce eliminuje potřebu vrstvy Dropout [44]).

Následuje konvoluční vrstva s 64 filtry, velikostí jádra $4x4$ a parametrem stride=2; jako aktivační funkce je opět použita ReLU. Následná konvoluční vrstva má také 64 konvolučních filtrů, velikost jádra je v tomto případě $2x2$, parametr stride=2 a aktivační funkce ReLU. Vrstvy Dense potřebují pro svou funkci klasifikátorů vektorové objekty – konverzi výstupu konvoluční části sítě do 1D vektoru zajišťuje vrstva Flatten prostřednictvím tzv. zplošťování (*flattening*). Úkolem vrstvy Dropout je zlepšit

generalizaci zamezením silné korelace při aktivaci neuronů, které jsou eliminovány – vyřazeny vynulováním aktivace pro daný neuron. Zároveň tím zabraňuje přetrénování (to hrozí při stálém nárůstu trénovací množiny, kdy se schopnost generalizace zhoršuje a síť se místo toho zaměří na specifickou oblast danou charakterem trénovacích dat). [45] Hodnota dropout byla nastavena na 15 %. Poslední skrytá vrstva je plně propojená, skládající se z 512 usměrňovacích jednotek (*rectifier units*). Výstupní vrstva je plně propojená s jedním výstupem (neuronem) pro každou možnou akci. Výstupem sítě je hodnotová funkce odhadující výši budoucích odměn při provedení akce v daném stavu. Aktivační funkce v poslední vrstvě je lineární (v Kerasu to znamená aktivační funkci u dané vrstvy nespecifikovat a tedy žádnou nepoužít) $a(x) = x$, protože síť musí na výstupu dávat velké hodnoty proměnlivého znaku. [46] ReLu ani hyperbolický tangens v tomto případě nelze využít, protože mají nelineární charakter a zvyšují nelinearitu sítě. Bez přítomnosti aktivační funkce se z vrstvy v podstatě stává lineární klasifikátor. Dodejme, že konvoluční vrstvy jsou zde využity pro detekci objektů na vloženém vstupním screenshotu, plně propojené vrstvy pak slouží k extrakci rysů a určení nejlepší akce.

Na hru Had lze aplikovat výše zmíněný Markovův rozhodovací proces, kdy si agent musí pamatovat pouze informaci o posledním stavu ve formě stav, akce, odměna a následující stav. Stavy jsou v programu reprezentovány dvěma po sobě jdoucími snímky ze hry, což je podobný přístup použitý při aplikaci zpětnovazebního učení na hry Atari 2600. [47] Jedním z hlavních problémů, které je nutné při hraní her zohlednit, je prodleva mezi provedenou akcí a výslednou odměnou a korelace po sobě jdoucích akcí. [39] Tento problém lze řešit více způsoby a v této práci jsou použity dva z nich – tzv. minibatch trénink a *experience replay*. Druhá zmíněná technika bude použita pro vytvoření herní paměti, pomocí níž byla trénována neuronová síť, kdy je každý zážitek průběžně ukládán jako tuple (stav, akce, odměna, následující stav, ukončení). Z obsahu této paměti jsou pak pro trénink vybírány malé dávky dat – minibatche, takže síť trénuje na širokém spektru zážitků, nejen na těch nedávných. Strategie se postupem času mění, a lze předpokládat, že herní obrázky z úvodních fází učení budou mít odlišnou distribuci akcí pro daný stav než ty pozdější. Síť funguje na principu odhadu hodnot pro následující stav a pro trénink využívá zážitky získané jak prostřednictvím původní, tak nedávné strategie.

Průběh tréninku Trénování agenta lze zahájit buď od začátku (příkazem python3 had.py -m "Trenink") nebo načíst předtrénovaný model model.h5, který se uloží do herní složky vždy po skončení tréninku (jeho obsah lze zobrazit např. pomocí aplikace ViTables). Ve výchozím stavu se na herní ploše neobjevuje informace o výši skóre, nicméně zaznamenávání a zobrazování skóre je v programu implementováno ve funkci skore1. Výsledek trénování sítě je prezentován ve formě videa

uloženého ve složce s programem, kde je vidět vytrénovaný agent hrající hru. Nejvyšší dosahované skóre v rámci jednotlivých epizod nepřesahuje 19 bodů. Pro další zlepšení výsledků by bylo potřeba síti poskytnout větší množství tréninků, aby byla schopna maximalizovat budoucí očekávanou odměnu a agentovi udílet správné pokyny pro rozhodování. Byly vyzkoušeny různé způsoby realizace tréninku sítě, jeden z nich ukazuje následující ukázka:

```
# minibatch, na kterem bude probihat trenink
minibatch = random.sample(zkusenosti_list, batch_size)
# pro trenovani je z listu zkusenosti vybran
# vzorek o velikosti batch_size
stav_t, akce_t, odmena_t, stav_t1, ukonceni = zip(*minibatch)
# experience replay
stav_t = np.concatenate(stav_t)
stav_t1 = np.concatenate(stav_t1)
targets = model.predict(stav_t)
Q_vzorek = model.predict(stav_t1)
targets[range(batch_size), akce_t] = odmena_t + diskontni_faktor *
np.max(Q_vzorek, axis=1) * np.invert(ukonceni)
model.train_on_batch(stav_t, targets) #trenovani modelu
```

Přímým výstupem programu jsou také datasety (ve formátu csv) rozdělené na tréninková a testovací (validační data tvoří 10 % posledních získaných vzorků – jsou nastavena argumentem validation.split ve funkci model.fit). Na základě dat získaných z datasetů lze vytvářet grafy závislostí jednotlivých parametrů. Prezentace výsledků praktické části předkládané bakalářské práce je uvedena v přílohách ve formě zdrojových kódů ke hře (soubory had.py a neuronova_sit.py).

3 ZÁVĚR

Bakalářská práce popsala teoretické základy zpětnovazebního učení a následně se zaměřila na popis a porovnání jeho jednotlivých metod, se zvláštním důrazem na metody aktivního učení. V praktické části se zabývala aplikací jedné z metod zpětnovazebního učení – hlubokého zpětnovazebního učení – na hru Had. Výsledky praktické části jsou prezentovány ve formě programu, který sestává ze tří částí: herního prostředí vytvořeného v PyGame, herního agenta a vícevrstvé konvoluční neuronové sítě zkonstruované v knihovně Keras. Jedním z výstupů programu jsou datasety, přičemž data v nich obsažená lze využít jako podklad pro další zpracování a také pochopení, jak program funguje „uvnitř“ – vývoj sledovaných parametrů při jejich různém nastavení.

Hlavním výstupem práce je program upravený do podoby, která umožňuje testovací hru Had automatizovat, kdy s využitím umělé inteligence program vyhodnocuje informace ze hry a agentovi uděluje pokyny, jaké akce zvolit, aby maximalizoval svůj užitek ze hry. Bylo ukázáno, že agent učící se prostřednictvím hluboké neuronové sítě, která jako vstupy používá obrazové pixely, byl schopen, při poskytnutí dostatečně dlouhého tréninku, ovládnout pravidla hry a využít je ve svůj prospěch. Chování agenta a volba jeho strategií však mohou být faktory mnohdy nepředvídatelné i v relativně jednoduché hře jako Had. V současnosti zůstává nadále nutností manuální ladění hyperparametrů a jejich opakované testování, stejně jako sledování dalších faktorů, které mohou mít dopad na učící proces (např. velikost herní plochy apod.). Pochopit, proč se agent chová určitým způsobem a odhadnout, jak se bude rozhodovat v různých stavech, zůstává výzvou, podobně jako vytvoření univerzálního algoritmu zpětnovazebního učení, který by byl aplikovatelný na všechna prostředí. V neposlední řadě zůstává jedním z úkolů i navržení algoritmů, které zajistí bezpečnost systémů využívajících umělou inteligenci, aby byly schopny jasně specifikovat své cíle tak, aby jim agenti porozuměli a vyhnuli se jejich nesprávné interpretaci.

LITERATURA

[1] MIIKKULAINEN, Risto, Bobby D. BRYANT, Ryan CORNELIUS, Igor V. KARPOV, Kenneth O. STANLEY a Chern Han YONG. *Computational Intelligence in Games* [online]. 2006 [cit. 22. 5. 2018]. Dostupné z URL: <`ftp://ftp.cs.utexas.edu/pub/neural-nets/papers/miikkulainen.wcci06.pdf`>

[2] LOCKHART, Christopher. *Application of temporal difference learning to the game of Snake* [online]. květen 2010 [cit. 24. 5. 2018]. Dostupné z URL: <`https://pdfs.semanticscholar.org/ae7e/e40204e374719a9b2babcc534dee4f0d5956.pdf`>

[3] CHVOJ, Martin. *Pokročilá teorie her ve světě kolem nás*. Praha: Grada, 2013. ISBN 978-80-247-4620-3.

[4] CLARK, Jack a Dario AMODEI. *Faulty Reward Functions in the Wild* [online]. 21. 12. 2016 [cit. 26. 5. 2018]. Dostupné z URL: <`https://blog.openai.com/faulty-reward-functions/`>

[5] *Snake* [online]. [cit. 28. 5. 2018]. Dostupné z URL: <`http://datagenetics.com/blog/april42013/index.html`>

[6] JULIANI, Arthur. *Simple Reinforcement Learning with Tensorflow Part 5: Visualizing an Agent's Thoughts and Actions* [online]. 10. 9. 2016 [cit. 26. 5. 2018]. Dostupné z URL: <`https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-5-visualizing-an-agents-t`

[7] NILSSON, Nils J. *Introduction to Machine Learning* [online]. Stanford University, 3. 11. 1998 [cit. 30. 9. 2017]. Dostupné z URL: <`http://robotics.stanford.edu/~nilsson/MLBOOK.pdf`>

[8] SUTTON, Richard S. a Andrew G. BARTO. *Reinforcement Learning: An Introduction.* 1. vyd. Cambridge: Bradford Book, 1998. 322 s. ISBN 0-262-19398-1.

[9] SUITS, David. *A Simplified Drive-Reinforcement Model for Unsupervised Learning in Artificial Neural Networks* [online]. 2004 [cit. 26. 9. 2017]. Dostupné z URL: <`https://people.rit.edu/dbsgsh/sdrmodel.pdf`>

[10] BERKA, Petr. *Neuronové sítě* [online]. [cit. 27. 9. 2017]. Dostupné z URL: <`http://sorry.vse.cz/~berka/docs/izi456/kap_5.4.pdf`>

[11] MATIISEN, Tambet. *Demystifying Deep Reinforcement Learning* [online]. Computational Neuroscience Lab, 19. 12. 2015

[cit. 17. 10. 2017]. Dostupné z URL: <http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/>

[12] *Adaptivní agenti a evoluční algoritmy* [online]. [cit. 26. 9. 2017]. Dostupné z URL: <http://wiki.matfyz.cz/index.php?title=Adaptivn%C3%AD_agenti_a_evolucn%C3%AD_algoritmy#Inteligentn.C3.AD_agent>

[13] KALYANAKRISHNAN, Shivaram a Peter STONE. *Characterizing Reinforcement Learning Methods Through Parameterized Learning Problems* [online]. Mach Learn, 3. 6. 2011 [cit. 17. 10. 2017]. Dostupné z URL: <https://link.springer.com/content/pdf/10.1007%2Fs10994-011-5251-x.pdf>

[14] DETTMERS, Tim. *Deep Learning in a Nutshell: Core Concepts* [online]. Parallel Forall, 3. 11. 2015 [cit. 26. 9. 2017]. Dostupné z URL: <https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/>

[15] BENGIO, Yoshua. *Learning Deep Architectures for AI* [online]. Foundations and Trends in Machine Learning 2009 [cit. 28. 10. 2017]. Dostupné z URL: <https://www.iro.umontreal.ca/~bengioy/papers/ftml.pdf>

[16] BROWNLEE, Jason. *Clever Algorithms: Nature-Inspired Programming Recipes.* Morrisville: Lulu Press, 2012. ISBN 978-1446785065.

[17] RASCHKA, Sebastian. *Python Machine Learning.* Birmingham: Packt Publishing Limited, 2015. ISBN 978-1-78355-513-0.

[18] MATERNA, Jiří. *Deep Learning: budoucnost strojového učení?* [online]. 9. 1. 2013 [cit. 28. 10. 2017]. Dostupné z URL: <https://vyhledavani.sblog.cz/2013/01/09/deep-learning-budoucnost-strojoveho-uceni/>

[19] DE ASIS, Kristopher, Fernando J. HERNANDEZ-GARCIA, G. Zacharias HOLLAND a Richard S. SUTTON. *Multi-Step Reinforcement Learning: A Unifying Algorithm* [online]. 3. 3. 2017 [cit. 20. 10. 2017]. Dostupné z URL: <https://arxiv.org/pdf/1703.01327>

[20] *Attacking Machine Learning with Adversarial Examples* [online]. 24. 2. 2017 [cit. 17. 10. 2017]. Dostupné z URL: <https://blog.openai.com/adversarial-example-research/>

[21] *Úvod do stochastických optimalizačních metod (metaheuristik)* [online]. 17. 10. 2012 [cit. 17. 10. 2017]. Dostupné z URL: <http://mech.fsv.cvut.cz/~leps/teaching/mmo/prednasky/prednaska04_intro_SA_TA.pdf>

[22] SZEPESVÁRI, Csaba. *Algorithms for Reinforcement Learning* [online]. 9. 6. 2009 [cit. 28. 10. 2017]. Dostupné z URL: <https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf>

[23] *Reinforcement Learning* [online]. [cit. 21. 10. 2017]. Dostupné z URL: <www.cse.unsw.edu.au/~cs9417ml/RL1/introduction.html>

[24] BELLMAN, Richard. *The Theory of Dynamic Programming* [online]. Mach Learn, 30. 7. 1954 [cit. 17. 10. 2017]. Dostupné z URL: <https://www.rand.org/content/dam/rand/pubs/papers/2008/P550.pdf>

[25] DETTMERS, Tim. *Deep Learning in a Nutshell: Reinforcement Learning* [online]. Parallel Forall, 8. 9. 2016 [cit. 17. 10. 2017]. Dostupné z URL: <https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-reinforcement-learning/>

[26] MNIH, Volodymyr, Koray KAVUKCUOGLU a David SILVER. *Human-Level Control Through Deep Reinforcement Learning* [online]. [cit. 30. 9. 2017]. Dostupné z URL: <https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf>

[27] VARSHAVSKAYA, Paulina, Leslie P. KAELBLING a Daniela RUS. *Efficient Distributed Reinforcement Learning Through Agreement* [online]. [cit. 28. 10. 2017]. Dostupné z URL: <http://people.csail.mit.edu/lpk/papers/dars08.pdf>

[28] JULIANI, Arthur. *Simple Reinforcement Learning with Tensorflow Part 4: Deep Q-Networks and Beyond* [online]. 2. 9. 2016 [cit. 28. 10. 2017]. Dostupné z URL: <https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-bey

[29] MNIH, Volodymyr, Asdrià Puigdomènech BADIA, Mehdi MIRZA, Alex GRAVES, Tim HARLEY, Timothy P. LILLICRAP, David SILVER a Koray KAVUKCUOGLU. *Asynchronous Methods for Deep Reinforcement Learning* [online]. Stanford University, 16. 6. 2016 [cit. 20. 10. 2017]. Dostupné z URL: <https://arxiv.org/pdf/1602.01783>

[30] BARRON, Jonathan T., Dave S. GOLLAND a Nicholas J. HAY. *Parallelizing Reinforcement Learning* [online]. [cit. 28. 10. 2017]. Dostupné z URL: <https://jonbarron.info/Barron-ParallelizingRL.pdf>

[31] BOBRENKO, Dmitry. *Asynchronous Deep Reinforcement Learning from Pixels* [online]. 3. 11. 2016 [cit. 29. 10. 2017]. Dostupné z URL: <https://dbobrenko.github.io/2016/11/03/async-deeprl.html>

[32] DEWOLF, Travis. *Reinforcement Learning Part 2: SARSA vs Q-learning* [online]. 1. 7. 2017 [cit. 29. 10. 2017]. Dostupné z URL: <https://studywolf.wordpress.com/2013/07/01/reinforcement-learning-sarsa-vs-q-learning/>

[33] MA, Bowei, Meng TANG a Jun ZHANG. *Exploration of Reinforcement Learning to SNAKE* [online]. [cit. 16. 5. 2018]. Dostupné z URL: <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=2ahUKEwii3pG124nbAhVKzqQKHSXsDtEQFjAAegQIARAo&url=http%3A%2F%2Fcs229.stanford.edu%2Fproj2016spr%2Freport%2F060.pdf&usg=AOvVaw0he7-PHYHNFI3lNcOJk2rp>

[34] ONG, Hao Yi, Kevin CHAVEZ a Augustus HONG. *Distributed Deep Q-Learning* [online]. 15. 10. 2015 [cit. 22. 5. 2018]. Dostupné z URL: <https://arxiv.org/pdf/1508.04186.pdf>

[35] KONG, Shu a Joan Aguilar MAYANS. *Automated Snake Game Solvers via AI Search Algorithms* [online]. [cit. 26. 11. 2017]. Dostupné z URL: <http://sites.uci.edu/joana1/files/2016/12/AutomatedSnakeGameSolvers.pdf>

[36] ANANTO, Azizul Haque. *Self-Learning Game Bot using Deep Reinforcement Learning* [online]. 13. 12. 2017 [cit. 24. 5. 2018]. Dostupné z URL: <http://dspace.bracu.ac.bd/xmlui/bitstream/handle/10361/9509/14301050_CSE.pdf?sequence=1&isAllowed=y>

[37] SPRINGENBERG, Jost Tobias, Alexey DOSOVITSKIY, Thomas BROX a Martin RIEDMILLER. *Striving for Simplicity: The All Convolutional Net* [online]. 13. 4. 2015 [cit. 24. 5. 2018]. Dostupné z URL: <https://arxiv.org/pdf/1412.6806.pdf>

[38] ZOCCA, Valentino, Gianmario SPACAGNA, Daniel SLATER a Peter ROELANTS. *Deep Learning for Computer Games*. In: ZOCCA, Valentino, Gianmario SPACAGNA, Daniel SLATER a Peter ROELANTS. *Python Deep Learning*. Birmingham-Mumbai: Packt Publishing, 2017, s. 251-292. ISBN 978-1-78646-445-3.

[39] MARTIN, Louis a Pierre STOCK. *Deep reinforcement learning for Snake* [online]. 2016 [cit. 25. 5. 2018]. Dostupné z URL: <https://github.com/pierrestock/deep-snake/blob/master/report/MARTIN_STOCK_Project.pdf>

[40] CHOLLET, François. *Keras* [online]. 2015 [cit. 2. 12. 2017]. Dostupné z URL: <https://keras.io/>

[41] *The Python Standard Library* [online]. [cit. 2. 12. 2017]. Dostupné z URL: <https://docs.python.org/3.5/library/index.html>

[42] BUCKLAND, Mat. *Programming Game AI By Example*. Burlington: Jones & Bartlett Learning, 2014. ISBN 978-1556220784.

[43] *CS231n Convolutional Neural Networks for Visual Recognition* [online]. [cit. 2. 12. 2017]. Dostupné z URL: <http://cs231n.github.io/convolutional-networks/>

[44] IOFFE, Sergey a Christian SZEGEDY. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* [online]. 2. 3. 2015 [cit. 2. 12. 2017]. Dostupné z URL: <https://arxiv.org/pdf/1502.03167>

[45] TOMPSON, Jonathan, Ross GOROSHIN, Arjun JAIN, Yann LECUN a Christoph BREGLER. *Efficient Object Localization Using Convolutional Networks* [online]. 9. 6. 2015 [cit. 22. 5. 2018]. Dostupné z URL: <https://arxiv.org/pdf/1411.4280.pdf>

[46] GRATTAROLA, Daniele. *Deep Q-Learning to play Snake* [online]. 1. 8. 2016 [cit. 16. 5. 2018]. Dostupné z URL: <https://danielegrattarola.github.io/files/projects/2016_grattarola_snake.pdf>

[47] MNIH, Volodymyr, Koray KAVUKCUOGLU, David SILVER, Alex Graves, Ioannis ANTONOGLOU, Daan WIERSTRA a Martin RIEDMILLER. *Playing atari with deep reinforcement learning* [online]. [cit. 16. 5. 2018]. Dostupné z URL: <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

AI artificial intelligence – umělá inteligence

CNN convolutional neural network – konvoluční neuronová síť

CPU central processing unit – centrální procesorová jednotka

DQN deep q-network – hluboká q-síť

GPU graphic processing unit – grafický procesor

MDP Markov decision process – Markovův rozhodovací proces

RL reinforcement learning – zpětnovazební učení

TD temporal difference – temporální diference

SEZNAM PŘÍLOH

A Obsah přiloženého CD **48**

A OBSAH PŘILOŽENÉHO CD

- elektronická verze bakalářské práce,
- 185967.zip – zdrojové kódy vytvořeného programu, společně se všemi výstupy programu – datasety ve formátu csv, uloženým modelem neuronové sítě a souborem s vizualizací architektury neuronové sítě.