Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Demonstrace reinforcement učení v počítačových hrách

2023

Bc. Tomáš Zálešák

Vedoucí práce:
Mgr. Petr Osička, Ph.D.

Studijní program: Aplikovaná informatika,
prezenční forma

Bibliografické údaje

Autor:	Bc. Tomáš Zálešák
Název práce:	Demonstrace reinforcement učení v počítačových hrách
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Petr Osička, Ph.D.
Počet stran:	48
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Bc. Tomáš Zálešák
Title:	Demonstration of reinforcement learning in computer games
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Applied Computer Science, full-time form
Supervisor:	Mgr. Petr Osička, Ph.D.
Page count:	48
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Tato práce se zabývá implementací zpetněvazebného učení neuronové sítě a demonstrace výsledků. Součástí jsou zdrojové soubory potřebné k vlastnímu učení a již naučené váhy sítě.

Synopsis

This thesis deals with implementation of reinforcement learning of neural network and demonstration of results. Part of the thesis are source files required for own learning and already prepared network weights.

Klíčová slova: zpetněvazebné učení; neuronové sítě; algoritmy; arkádové hry; demonstrace;

Keywords: reinforcement learning; neural networks; algorithms; arcade games; demonstration;

Děkuji za vedení a rady vedoucího práce, podpory rodiny, přátel a kolegy, bez kterých by toto nebylo možné.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

1 Úvod **7**

2 Neuronové sítě **8**
2.1 Perceptron . . . 8
2.1.1 Učení perceptronu . . . 9
2.1.2 Omezení samostatného perceptronu . . . 10
2.2 Vícevrstvé sítě s dopředných šířením signálu . . . 12
2.3 Učení sítě . . . 13
2.3.1 Backpropagation . . . 14
2.3.2 Metoda gradientního sestupu . . . 14
2.3.3 Metoda backpropagation . . . 15

3 Reinforcement learning – zpětnovazebné učení **16**
3.1 Markovův rozhodovací proces . . . 17
3.2 Bellmanova rovnice . . . 18
3.3 Q-Learning . . . 18
3.4 Deep Q-Learning . . . 21
3.5 Explorace a exploitace . . . 21

4 Použité technologie a výběr arkádových her **23**
4.1 Použité knihovny jazyka Python . . . 23
4.2 Výběr arkádových her . . . 23

5 Implementační část **23**
5.1 Společné třídy . . . 24
5.2 Použitá neuronová síť . . . 24
5.3 Funkce pro rozhodování výběru další akce . . . 25
5.4 Paměť – Replay memory . . . 26
5.5 Odměna . . . 27
5.6 Agent . . . 27
5.6.1 Konstruktor . . . 27
5.7 Vytvoření neuronové sítě (modelu) . . . 28
5.8 Učení agenta (Q-Trainer) . . . 28
5.9 Přechodová funkce . . . 29
5.10 Implementace Snake . . . 29
5.10.1 Struktura objektů . . . 30
5.10.2 Neuronová síť . . . 31
5.10.2.1 Počítačový vstup . . . 31
5.10.3 Vstup a výstup . . . 31
5.10.4 Odměna . . . 32
5.10.5 Výsledky sítě . . . 32
5.10.6 Shrnutí . . . 33
5.11 Implementace Invaders . . . 34

5.11.1 Struktura objektů . . . 34
5.11.2 Neuronová síť . . . 35
5.11.3 Vstup a výstup . . . 35
5.11.4 Odměna . . . 36
5.11.5 Výsledky sítě . . . 36
5.11.6 Shrnutí . . . 37
5.12 Implementace Asteroid . . . 38
5.12.1 Struktura objektů . . . 38
5.12.2 Neuronová síť . . . 39
5.12.3 Vstup a výstup . . . 39
5.12.4 Odměna . . . 39
5.12.5 Shrnutí . . . 39

6 Uživatelská část **41**
6.1 Předpoklady . . . 41
6.2 Konfigurace neuronové sítě . . . 41
6.3 Spuštění programu . . . 42

Závěr **43**

Conclusions **44**

A Obsah elektronických dat **45**

Literatura **46**

1 Úvod

Strojové učení v dnešní době zažívá boom a mě samotného to neminulo. Zároveň je také často zaměňováno s umělou inteligencí. Ta v základní podobě může vypadat pouze jako jedna dlouhá podmínka.

To, co nás ale zajímá jsou neuronové sítě, a zejména jejich učení pomocí vstupních dat. Základní dělení by se dalo rozlišit na tři kategorie, učení s učitelem, kdy je určen správný výstup, učení bez učitele a zpětnovazebné učení.

My se zaměříme na poslední druh učení, což je tedy zpětnovazebné učení se zaměřením na praktické vyzkoušení, implementaci a demonstraci. Provedeme demonstraci tohoto druhu učení na jednoduchých arkádových hrách: Had, Asteroid a Invaders. Tyto hry si sami vytvoříme a využijeme jejich vnitřní stavy jako základ pro vstupní jednorozměrnou matici.

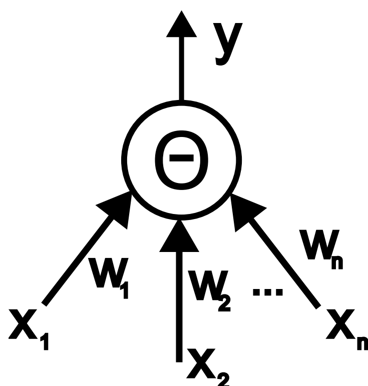
Diplomová práce obsahuje jak teoretickou část, která popisuje princip zpětnovazebného učení, tak implementační část, kde je tato teoretická část použita prakticky. Práce tedy obsahuje i program na učení neuronové sítě a již naučené sítě, v různém stavu učení sítě.

2 Neuronové sítě

Jak už jsme si nastínili, budeme učit umělé neuronové sítě řešit nějaký problém. Tato struktura[1] je matematický model umělých neuronů[2], které jsou mezi sebou propojeny, posílají a přijímají signály. Tento model je založen na reálném fungování neuronů v mozku a jeho cílem je matematické vyjádření tohoto fungování. V této kapitole jsem čerpal zejména ze zápisků z hodin umělé inteligence[2] a z poskytnuté knihy[1].

2.1 Perceptron

Základem neuronové sítě je perceptron, který je jednoduchým modelem biologického neuronu. Je složen ze vstupu (x_i), vah vstupů (w_i), prahu Θ, aktivační funkce a výstupu[3]. Vstupů může být tedy mnoho a výstup pouze jeden. Samotný výpočet[4] probíhá „uvnitř" neuronu, kdy vezme vstupy, vynásobí je vahami, sečte je a podle dané aktivační funkce[5] vydá výstup.



Obrázek 1: Perceptron

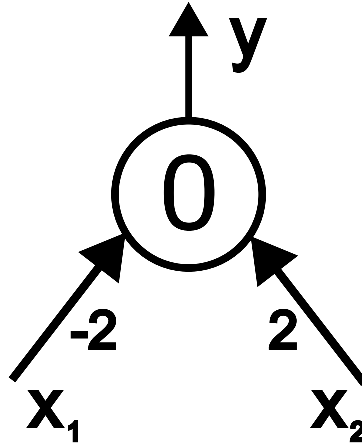
Na obrázku 1 je zobrazena vizualizace libovolného perceptronu. Aktivační funkce perceptronu vypadá tedy následovně:

$$y = 1 \text{ pokud } \sum_{i=1}^{n} x_i w_i \geq \Theta \text{ , jinak } 0$$

Kde je x_i velikost vstupního signálu, w_i váhy spojení, které zesilují nebo zeslabují přicházející signál, Θ práh perceptronu a y, což je tedy výstupní signál, který nabývá hodnoty 1 nebo 0. Můžeme si pomocí perceptronu realizovat i některé logické funkce, například implikaci:

$$y = x_1 \rightarrow x_2$$

[1] Neuronová síť
[2] Perceptronů
[3] y, zde je pouze jeden výstup
[4] aktivace
[5] prahu



Obrázek 2: Perceptron implikace

2.1.1 Učení perceptronu

Obecně řečeno pojmem *učení* se myslí úprava vah a prahů jednotlivých neuronů tak, aby pro daný vstup jsme získali požadovaný výstup. Postupem učení chceme minimalizovat tzv. *chybu neuronové sítě*. Což je nějaká hodnota, která nám udává rozdíl mezi získanými a požadovanými daty. K získání této hodnoty se používá tzv. chybová funkce (např. mean squared error[18]). Učením sítě chceme tedy tuto chybu minimalizovat.

Idea: Máme danou trénovací množinu:

$$T = \{\langle x_p, y_p \rangle |\, p \in P\} \text{ kde } x_p \in \mathbb{R}|\langle x_1, ..., x_n \rangle, y_p \in \{0, 1\}$$

Trénovací množina je konečná množina trénovacích vzorů, která obsahuje dvojici vstup (x_p) a požadovaný výstup (y_p)

A cílem je najít perceptron, který bude realizovat funkci y pro $\forall p \in P$[6] platí $y(x_p) = y_p$. Chceme, aby se perceptron choval tak jak nám předepisuje T [7]. Toto chování je dáno jeho parametry $w_1, ..., w_n$ a Θ.

Odchylku mezi skutečnou a požadovanou odezvou sítě můžeme vyjádřit jako střední kvadratickou chybu, více o tom v kapitole níže 2.3.

Máme dva základní způsoby učení (úpravu parametrů), které jsou oba prováděny iterativně, dokud není splněna dostačující podmínka. Například celá množina T byla naučena (případně skoro celá množina T). *Naučena* tedy znamená, že se nám podařilo naučit síť klasifikovat prvky podle trénovací množiny, nebo minimalizovat chybu neuronové sítě na dostačující hodnotu.

1. **Online learning** – Podívám se na vzor $\langle x_p, y_p \rangle$ a podle něj upravím váhy. Prochází se vzor po vzoru z T a na základě jednotlivého vzoru se provede úprava.

[6] P je množina vzorů
[7] Trénovací množina

2. **Batch learning** – Podívám se na všechny vzory (celá množina T) a poté se upraví váhy.

Základem učení perceptronu je tzv. *delta rule*[19] [8], což je obdoba nalezení lokálního minima ve funkci postupným sestupem gradientu [9].

Je-li vzor $\langle x_p, y_p \rangle$, kde je $\langle x_{p1}, ..., x_{pn} \rangle \in \mathbb{R}^n$ vstup a y_p je požadovaný výstup, kde $y \in \{0, 1\}$, je skutečný výstup perceptronu, uprav Θ a w následovně:

$$\Theta^{(t+1)} := \Theta(t) + \Delta\Theta, \text{kde } \Delta\Theta = -\eta(y_p - y)$$

$$w_i^{(t+1)} := w^{(t)} + \Delta w_i, \text{kde } \Delta w_i = \eta(y_p - y)x_i$$

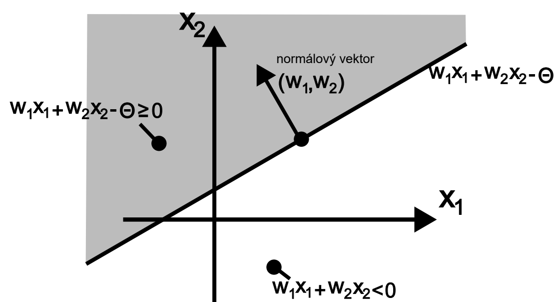
$$\text{Kde } \eta \in \mathbb{R} \text{ je rychlost učení a } t \text{ je čas}$$

2.1.2 Omezení samostatného perceptronu

Samotný perceptron je možné naučit pouze lineárně separabilní množiny. Pokud není množina lineárně separabilní, tak není možné naučit samostatný perceptron, aby se naučil separovat vstupně výstupní množinu korektně. Toto omezení je možné geometricky zobrazit, kde tedy, pokud je možné, tyto množiny rozdělit jedinou přímkou, tak je možné perceptron tento problém naučit. Lze tedy rozděli přímkou rovinu na dva poloprostory tak, aby spolu související množiny byly na stejném poloprostoru a v tom poloprostoru nebyl žádný prvek, který by měl zahrnut do druhé množiny.

$$\sum_{i=1}^{n} w_i x_i \geq \Theta$$

$$\underbrace{\overbrace{w_1 x_1 + ... + w_n x_n}^{\text{nadprostor}} \geq \Theta}_{\text{poloprostor}}$$

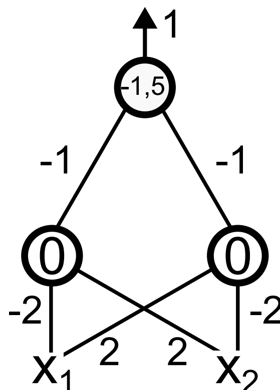


Obrázek 3: Geometrický pohled na lineárně separabilní množiny

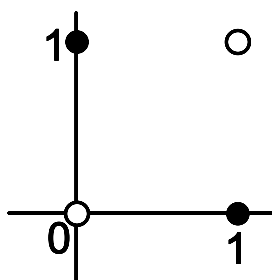
[8] Nebo také Widrow-Hoff rule[20]

[9] Gradient descent learning rule

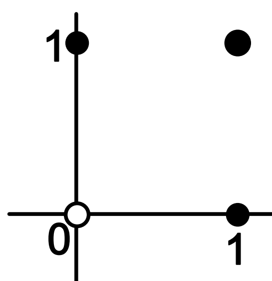
Například logická funkce implikace lineárně separabilní je, ale XOR již ne. Řešením tohoto problému je propojení více perceptronů, tedy vytvoření více vrstvé sítě popisované v kapitole níže.



Obrázek 4: Vícevrstvá síť logické funkce XOR



Obrázek 5: Lineárně neseparabilní XOR

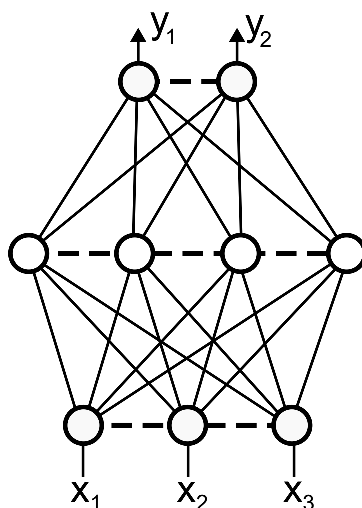


Obrázek 6: Lineárně separabilní AND

2.2 Vícevrstvé sítě s dopředných šířením signálu

Základem vícevrstvé sítě je libovolné množství spojených (spojitých) perceptronů uspořádané ve vrstvách. Vícevrství síť má alespoň tři vrstvy, vstupní, výstupní a skrytou, vnitřní vrstvu. Každý perceptron z vrstvy l je připojen na vstup (výstupem $y_{(l)}$) perceptronů z vrstvy $l+1$ a zároveň jsou na jeho vstup jsou připojeny všechny výstupy perceptronů z vrstvy $l-1$. Každé propojení má samozřejmě i nastavenou váhu. Síť má počet vrstev (l), počet vstupů (n), počet výstupů (m) a počet neuronů ve vrstvě (m_l). Vnitřních vrstev může být libovolný počet. Počet neuronů v jednotlivých vrstvách[10] je také libovolný.

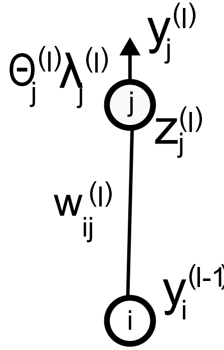
Na obrázku [7] tedy máme jednu vstupní vrstvu se třemi vstupy (x_1, x_2, x_3). Jednu vnitřní vrstvu o šířce 4 ($m_1 = 4$) a jednu výstupní vrstvu, která má dva výstupy (y_1, y_2).



Obrázek 7: Obecná vícevrstvá síť

[10] šířka vrstvy sítě

Na následujícím obrázku [8] máme zobrazenou vrstvu "zblízka". Kde y_i^{l-1} je výstup neuronu i ve vrstvě $l-1$. $w_{ij}^{(l)}$ váha z neuronu i do neuronu j. $z_j^{(l)}$ je potencionál neuronu j. $\lambda_j^{(l)}$ je strmost pro neuron j. $\Theta_j^{(l)}$ práh neuronu. $y_j^{(l)}$ výstup neuronu ve vrstvě l. Obecně z potencionál neuronu, kdy je propojen s více vstupy lze vyjádřit jako $\sum_{i=1}^{n} w_i x_i - \Theta$. Pro výstup y můžeme použít *sigmoidální přenosovou funkci* $y = \frac{1}{1+e^{-\lambda z}}$, ale je možné použít i jiné, například *hyperbolický tangens*. Výhoda tohotu přístupu oproti skokové aktivační funkci[11] je dynamičtější obor hodnot výstupu. Za zmíňku stojí ještě *usměrněná lineární funkce*[12], kde $y = max(0, -\lambda z)$, kterou využíváme v implementaci. Tady máme mnohem větší obor hodnot $[0, \infty)$.



Obrázek 8: Jedna vrstva

Ještě si trochu popíšeme aktivační dynamiku, tzn. dopředného šíření signálu. Taková síť reprezentuje zobrazení $F : \mathbb{R}^n \rightarrow (0,1)^n$. Následovně platí pro všechny vrstvy:

$$y_i^{(0)} := x_i \text{ pro } 1, ..., n$$

$$\text{pro } l = 1, ..., r \text{ definujeme pro } j = 1, ..., m_l$$

$$z_j^{(l)} = \sum_{i=1}^{m_{l-1}} w_{ij}^{(l)} y_i^{(l-1)} - \Theta_j^{(l)}$$

$$y_j^{(l)} = \frac{1}{1 + e^{-\lambda_j^{(l)} z_j^{(l)}}}$$

2.3 Učení sítě

Máme danou trénovací množinu

$$T = \{\langle x^p, o^p \rangle | p \in P, x^p = \langle x_1^p, ..., x_n^p \rangle \in \mathbb{R}^n, o^p = \langle o_1^p, ..., o_m^p \rangle \in (0,1)^m\}$$

Cílem je najít neuronovou síť, to znamená najít topologii sítě, počet vrstev, počet neuronů ve vrstvách a parametry $w_{ij}^{(l)}$, $\Theta_j^{(l)}$, $\lambda_j^{(l)}$ tak, aby $F \approx T$. Kde F je

[11] 0 nebo 1
[12] ReLu

funkce, reprezentovaná sítí, $\approx$ je přibližná rovnost definovaná vhodnou metrikou a T lze chápat jako parciální funkci[13]. Obecný postup učení sítě je nejprve určení architektury expertem a následovně vybraný algoritmus najde parametry $w_{ij}^{(l)}$, $\Theta_j^{(l)}$, $\lambda_j^{(l)}$.

Chceme pomocí učení minimalizovat chyby sítě změnou uvedených parametrů. Tuto míru chybovosti je nutné nějak definovat. Jak jsme se již několikrát zmínili je tato chyba reprezetována tzv. *střední kvadratickou chybou*[14]. Chyba které se sít dopouští vzhledem ke vzoru $p \in P$ je definována jako:

$$E_p = \frac{1}{2}(y(x_p) - o_p)^2$$

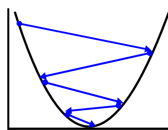
Kde $y(x_p)$ je výstup neuronové sítě pro vstup x_p a o_p je hodnota požadovaného výstupu z trénovací množiny T. Na tuto chybovou funkci navážeme o kapitolu později 2.3.3.

2.3.1 Backpropagation

Tento učící algoritmus využívá zpětného šíření chyby, která vychází z klasické numerické metody gradientního sestupu[15]. Pomocí této metody tedy upravuje parametry neuronové sítě, zejména váhy. Základním principem je rozdělení do epoch, které probíhají iterativně. Klesáním podle gradientu se spočítá parciální derivace chybové funkce. Nejdříve si ukážeme metodu gradientního sestupu a navážeme na ní metodou backpropagation 2.3.3.

2.3.2 Metoda gradientního sestupu

Gradient funkce[21] $F : \mathbb{R}^n \to \mathbb{R}$ v bodě $a \in \mathbb{R}^k$ je vektor $grad\ f(a) := \langle \frac{\delta f}{\delta x_1}(a), ..., \frac{\delta f}{\delta x_n}(a) \rangle$. Je-li f diferencovatelná, je $grad\ f(a)$ směr největšího růstu funkce f v bodě a. Tedy $-grad\ f(a)$ je směr největšího klesání v bodě a.



Obrázek 9: Hledání minima $f = x^2$

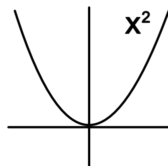
Algoritmus[16] hledání minima funkce f:

1. Začni v libovolném bodě $a(0)$
2. $a^{(t+1)} = a^{(t)} - \eta\ grad\ f(a^{(t)})$

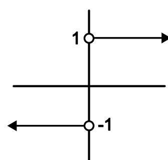
[13] Funkci definovanou na několika bodech. Je to naše trénovací množina.
[14] mean squared error
[15] gradient descent
[16] Idea: Jdeme z bodu do bodu, kde f má menší hodnotu atd.

3. Opakuj předchozí krok, dokud není nalezeno uspokojivé řešení.

Požadavky na funkci jsou, aby byla funkce diferencovatelná a konvexní. Vhodná funkce je například x^2 nebo $3sin(x)$. Nevhodné funkce je například $\frac{x}{|x|}$ nebo $\frac{1}{x}$. Problémem může nastat nalezení pokud hledáme minimum funkce u částečně diferencovatelná, tak je možné, že nalezneme pouze lokální minimum, ale ne globální, které hledáme. Z pohledu učení vícevrstvé sítě jde o minimalizaci chyby sítě změnou parametrů w, Θ, λ.



Obrázek 10: Vhodná funkce $f = x^2$



Obrázek 11: Nevhodná funkce $f = \frac{x}{|x|}$

2.3.3 Metoda backpropagation

Cílem je tedy minimalizace chyby sítě pomocí úprav, zejména, vah, prahů a strmostí aktivační funkce.

Algoritmus backpropagation:

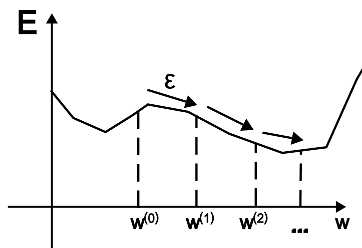
1. $t = 0$
2. Zvol náhodně $w^{(0)}$
3. $w^{(t+1)} = w^{(t)} - \eta \; grad \; E(w^{(t)})$
4. Opakuj předchozí krok, dokud není nalezeno uspokojivé řešení.

E je tedy naše funkce závislá na parametrech (w, Θ, λ). Pro jednoduchost předpokládáme $E = E(..., w_{ij}^{(l)}, ...)$ a máme pevně dané $\Theta_j^{(l)}$ a $\lambda_j^{(l)}$, zjednodušeně tak uvádíme $E(w^{(t)})$.Váhy jsou voleny "malá čísla", například $w_{ij}^{(l)} \in [-1, 1]$. η je naše rychlost učení u kterého platí $0 < \eta < 1$. Důležitý výpočet je $grad \; E(w^{(t)})$, který představuje gradient chybové funkce E. Tento gradient[17] je možné vyjádřit jako $\frac{\delta E}{\delta w}$, který nám udá směr, kterým se posuneme o ε. A dostaneme se

[17] vektor

tedy do nové konfigurace, kde by měla být chybová funkce menší než v přechozí konfiguraci($E(w^{(t)}) \geq E(w^{(t+1)})$).

Pokud definujeme E jako takzvanou střední kvadratickou chybu[18][33], kterou je možné zapsat jako vektor parciálních derivací v určitém bodě. Na tento výpočet jednotlivých parciálních derivací se lze dívat jako na zpětné šíření signálu sítě. Kvůli těmto výpočtům, které lze takto chápat, je odvozen název backpropagation.



Obrázek 12: Gradientí sestup

Ještě si můžeme uvést, pokud bychom na obrázku 12 začali v bodě $w^{(-2)}$ místo $w^{(0)}$, tak bychom pravděpodobně našli lokální minimum místo chtěného globálního.

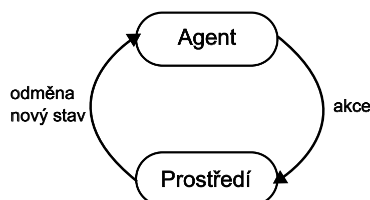
3 Reinforcement learning – zpětnovazebné učení

Reinforcement learning[7], dále jen RL, je typ učení, který nelze jednoznačně zařadit mezi učení s učitelem nebo učení bez učitele[19]. Základní myšlenka tohoto typu učení spočívá v učení pomocí odměňování. Za nějaký úspěšný cíl získá obecný učící algoritmus numerickou odměnu a může získávat i zápornou a nulovou numerickou odměnu. Tento učící algorimus zpětnovazebného učení se tedy snaží maximalizovat odměnu. Algoritmus přistupuje k učení podobně jako lidé. Kdy během učení reagují na změny prostředí na základě jejich akcí. Tyto změny mohou být pozitivní nebo negativní. Například když se dítě učí chodit, tak negativní je když spadne. A pozitivní když se dostane ke svému cíly. Na tomhle principu funguje zpětnovazebné učení.

Co je tedy základ pro zpětnovazebné učení[12]? Základními prvky jsou agent a prostředí. Agenta si lze představit jako hráče, který reaguje na situaci v prostředí prováděním různých akcí. Prostředí si lze představit jako herní plán, ve kterém agent hraje. Agent je tedy si schopen zjistit situaci v aktuálním prostředí, v jakém stavu se nachází a za každou provedenou akci dostane (numerickou) odměnu. Agent umí provést předem definovanou sadu akcí, kterými může měnit svůj stav v prostředí. Jeho cílem je vybírat takové akce, které maximalizují obdržené numerické odměny. Agent provede akci z množiny přístupných akcí, akce

[18]MSE - mean squared error
[19]supervised learning nebo unsupervised learning

se projeví změnou prostředí (například je na novém místě), je v novém stavu a dostane numerickou odměnu (která nemusí být pouze kladná) za nový stav. Agent se ocitne v novém stavu a znovu reaguje na svůj stav výběrem nejvhodnější akce, aby maximalizoval své budoucí odměny.



Obrázek 13: Cyklus učení

3.1 Markovův rozhodovací proces

Nejdříve trochu teorie. Začneme u Markovova rozhodovacího procesu[20][4][5][12], který jednoduše řečeno popisuje výběr akce agenta na základě jeho aktuálního stavu. Formálněji základem MRP je stavový stroj s konečnou množinou stavů S (kde stav je prostředí) a konečnou množinou akcí A a (agent) pomocí akcí se přechází mezi stavy. Přechodová relace MRP vypadá následovně:

$$\Delta \subseteq S \times A \times S_{,}$$

Do tohoto vztahu zahrneme ještě získání odměny z konečné množiny R a poté konkrétní běh stroje je:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots$$

Který přečteme jako ze stavu s_0 se dostaneme pomocí akce a_0 do stavu s_1 a připadne nám odměna r_1, atd.

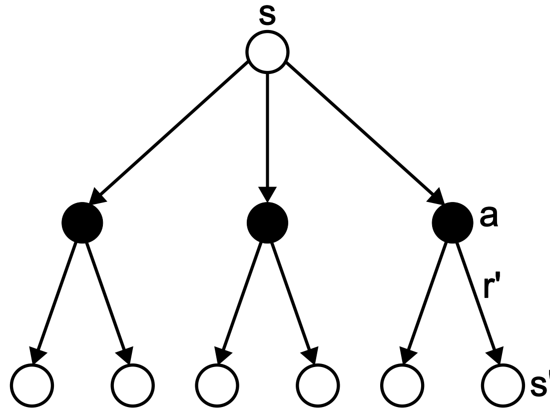
Abychom byly schopni takový MRP vytvořit, musíme ještě připustit určení přechodů pravděpodobnostně, zavedením přechodové funkce T.

$$T : S \times A \times S \rightarrow [0, 1]$$

A teď se konečně dostáváme ke konečné definici. Máme tedy MRP jako n-tici $\langle S, A, T, R \rangle$, kde S je množina stavů, A je množina akcí, T je přechodová funkce a R je míra okamžité odměny.

MRP je také možné vizualizovat jako orientovaný graf[12]. Kde jsou bíla kolečka stavy, černá akce a šipky provedení dané akce. Při výpočtu se tedy díváme o krok dopředu, která akce nám zajistí nový stav s co největší odměnou a danou akci vybereme. Nám prozatím stačí vědět, že něco takové vstupuje do naší problematiky a souvisí s další krátkou teorií, což je Deep Q-Learning, potažmo Q-Table a Bellmanovou rovnicí.

[20] Dále jen MRP



Obrázek 14: MRP jako orientovaný graf

3.2 Bellmanova rovnice

Chceme nalézt strategii pro agenta, kde chceme pro stav S vybrat co nejvhodnější akci A tak, aby byla maximalizovány budoucí získané odměny. Jako řešení lze použít Bellmanovu rovnici[22], kterou budeme používat při úpravách Q-Tabulky[21][23] při učení (iteracích) neuronové sítě. Je to tedy nějaké iterativní hledání optimální politiky. Bellmanova rovnice tedy vyjadřuje vztah mezi očekávanými hodnotami odměn R v jednotlivých stavech S. V rovnici je požadavek na maximalizaci odměn vyjádřen operátorem max. Zde je možné vidět spojitost s Markovovým rozhodovacím procesem[22]. Bellmanova rovnice v deterministickém prostředí může vypadat následovně:

$$V(s) = max_a(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s'))$$

Kde $V(s)$ je očekávaná hodnota aktuálního stavu, $R(s,a)$ je odměna za provedení akce a, γ[23] je diskontní[24] faktor ($0 < \gamma < 1$, kde typicky je γ blízko 1), $V(s')$ je očekávaná hodnota v které zůstane po provedení akce a $P(s,a,s')$ je pravděpodobnost dostání se do koncového stavu s' po provedení akce a.

Máme-li očekávané odměny, je možné vybrat strategii agenta tak, že vybereme akci, která vede do stavu s největším očekávaným ziskem. Můžeme tedy pomocí tohoto maximalizovat v každém kroku odměnu.

3.3 Q-Learning

Jako další si představíme Q-Learning [25][17][11], který patří do skupiny bezmodelových algoritmů. Bude to algoritmus, který využijeme pro učení sítě (agenta)

[21]v podstatě nepoužíváme Q-Table, ale její variaci v neuronové síti
[22]přechody mezi stavy
[23]γ je hyperparametr učení a obvykle je v rozmezí 0.9 až 0.99, kde nižší hodnota upřednostňuje krátkodobé zisky, zatímco vyšší upřednostňuje dlouhodobé.
[24]dampening
[25]Q = quality

v rámci diplomové práce. Q-Learning je metoda řešení Bellmanovy rovnice, kde je v tabulce obsažena aproximace hodnot rovnice $R(s,a)+\gamma\sum_{s'}P(s,a,s')V(s')$. Funkce počítající úspěšnost v daném stavu:

$$Q : S \times A \to \mathbb{R}$$

Tento algoritmus využívá tabulku nazývanou Q-Table, kterou si lze představit jako tabulku stavů, ve které je zapsaná pravděpodobnost, která akce se provede. Nebo se dá také popsat jako vyhledávací tabulka[26], kde je vypočítaná maximální předpokládaná budoucí odměna pro akci pro každý stav. Z této tabulky lze poté vytvořit strategie provedením akce s maximální hodnotou v tabulce. Tuto strategii může poté agent používat k dosažení dlouhodobých cílů[27].

Tato tabulka, nebo můžeme říct matice, je tedy upravována (učena) při iteracích. Tedy po každém provedení akce A, přechodu ze stavu S do stavu S‘, je použita rovnice k aktualizaci příslušené Q hodnoty dané dvojice. V počátečním stavu jsou Q hodnoty nastaveny, obvykle, na 0.

Úpravy jsme tady říkaly, že jsou v nějakém čase(iteracích), které se v tomto případě označují epizody. Jedna epizoda tedy je přechod z jednoho stavu do druhého za pomocí akce.

$$\underbrace{\overbrace{w_1x_1 + ... + w_nx_n}^{\text{nadprostor}} \geq \Theta}_{\text{poloprostor}}$$

$$Q'(s_t,a_t) \leftarrow \overbrace{Q(s_t,a_t) + \alpha\,\underbrace{(r_t + \gamma\, maxQ(s_{t+1},a) - Q(s_t,a_t))}_{\text{metoda řešení Bellmanovi rovnice}}}^{\text{Nová Q hodnota}}$$

- $Q(s_t,a_t)$ – Aktuální Q hodnota
- $Q'(s_t,a_t)$ – Nová (aktualizovaná) Q hodnota
- α – Míra učení
- r_t – Obdržená odměna
- γ – Diskontní faktor
- $maxQ(s_{t+1},a)$ – Maximální možná odměna v novém stavu v čase $t+1$

Po každém provedení akce a ve stavu s v čase t (tohle budeme nazývat epizodou) je pomocí rovnice aktualizovaná hodnota dané dvojice $Q(s,a)$. Provedením akce a se dostane do stavu s v čase $t+1$ kde obdrží odměnu r v čase t.

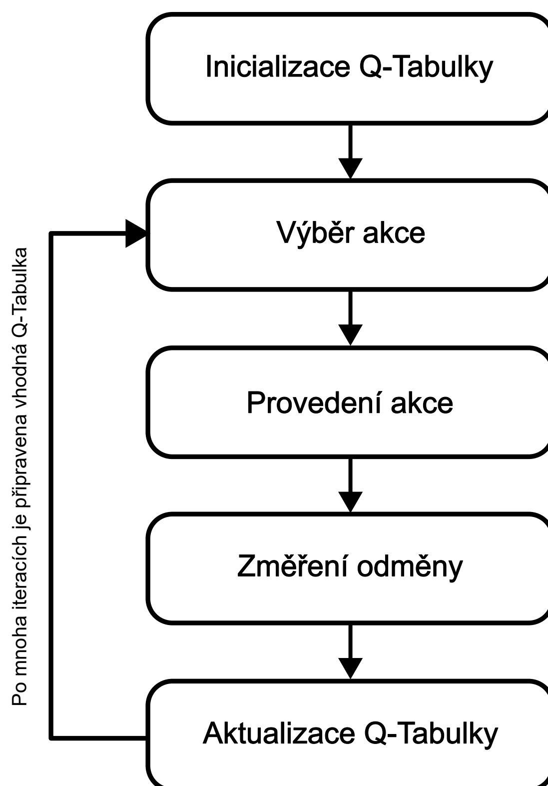
Pomocí funkce $maxQ(s_{t+1},a)$ získá maximální možnou odměnu, kterou může dostat ze stavu s_{t+1}, kdy předpokládá, že se algoritmus řídí optimální strategii a

[26] look-up table
[27] největšího počtu odměn

vybírá pouze nejoptimálnější akce. Trošku laicky, vyzkouší všechny akce a vybere tu s nejlepším výsledkem. A k aktuální hodnotě Q je přičtena odměna s rozdílem Q hodnoty ve stavu s_{t+1} a aktuální hodnoty Q (v čase t).

Nezmínil jsem ještě další parametry[28]. Parametr α[29], který určuje důležitost nového výsledku Q hodnoty oproti staré. Obvykle to je nějaké rozumně malé číslo, například 0.005. Parametr γ[30], který ovlivňuje přínos budoucích stavů. Tato hodnota by měla být v rozmezí $\langle 0.9, 0.99 \rangle$, kde nižší hodnota preferuje krátkodobý zisk a vyšší hodnota naopak dlouhodobý zisk.

Tyto úpravy se provádí za pomocí iterativní vyhodnocení politiky, která je založena na Bellmanově rovnici.



Obrázek 15: Jednoduchý diagram Q-Learningu

V našem případě v tomto učení, ale nepoužíváme Q-Tabulku, ale tedy neuronovou síť, která tuhle tabulku aproximuje v průběhu učení. Dalo by se tedy říci, čím více síť učíme, tím lépe ideální Q-Tabulku aproximuje. Tento přístup se nazývá Deep Q-Learning[6].

[28]hyperparametry
[29]learning rate
[30]discount factor

3.4 Deep Q-Learning

Proč používáme Deep Q-Learning[6] a ne pouze Q-Learning? Odpověď je jednoduchá, Q-Learning pro složitější případy je velmi paměťově náročný. Například u prostředí, které má tisíc stavů a tisíc akcí je milión Q hodnot k uložení. Toto můžeme považovat ještě za jednoduché prostředí. Například hra GO, které je možné naučit, a byly naučeny pomocí Deep Q-learning alogritmu, má 10^{170} stavů, a to už je velmi velké číslo samo o sobě[31].

Pokoušíme se tedy najít aproximaci tohoto prostředí pomocí neuronové sítě, která bude schopna aproximovat Q hodnotu libovolné dvojice. Tento učící algoritmus využívá backpropagation, ale nemá dopředu stanovenou cílovou hodnotu jako u učení s učitelem[32]. Chceme tedy získat[33] cílovou hodnotu Q a její výpočet je definován podobně jako v Q-Learningu. Rovnice pro úpravu Q hodnoty, vypadá tedy následovně:

$$Q(s,a) = r + \gamma(max(Q(s',a'))$$

Kde $Q(s,a)$ je Q hodnota, r je odměna, γ je diskontní faktor a je proveden výběr maximální Q hodnoty, která pro očekávanou Q hodnotu ze stavu provede všechny akce a vybere tu nejlepší.

Dále se využívá tzv. "replay memory"[23], kdy jsou všechny kroky ukládány do bufferu ve formátu $(s, a, s', r[$, volitelně konec epochy$]$[34]). Z tohoto bufferu je poté vybírán náhodný vzorek pomocí kterého je znovu trénována neuronová síť. Tento postup je používán, protože účinnost neuronové sítě je negativně ovlivněna korelací mezi vstupními daty a tento postup vykazuje zlepšené výsledky neuronové sítě.

3.5 Explorace a exploitace

Na začátku nemáme úplný model[11][15], máme pouze nepřesně odhadnuté údaje (například pouze nulové hodnoty) v Q-tabulce[35]. Tyto údaje můžeme chápat například jako dvojice stav S a akce A. Zpočátku je tedy vhodné v daném stavu provést větší množství akcí pomocí neoptimální politiky[36], která může prozkoumat větší množtví prostoru, než při použití optimální politiky. Pokud bychom použili hned ze začátku optimální politiku, je možné, že některé stavy nemusíme vůbec navštívit. Proto se na začátku běhu algoritmu snažíme více prohledávat[37] a až v pozdějších fázích algoritmu se řídit údaji v v Q-tabulce. V pozdějších fázích, kdy je aplikována optimální politika[38], dochází už k výběru nejvhodnější

[31] Prakticky byla implementována jinak[24]
[32] supervised learning
[33] definovat
[34] booleovská hodnota
[35] síti
[36] výběrem náhodných akcí
[37] Výběr náhodných akcí
[38] fáze exploitace

akce z Q-tabulky pro dosažení největší odměny.

Protože dosud nebyl nalezen nejlepší poměr těchto akcí[39], tak můžeme považovat, že jsou všechny politiky vhodné. Pokud je zde zachován nějaký rozumný přechod mezi procesy[35].

U Q-learning-u je nejčastější používaná strategie ε-greedy. Základem je hodnota ε z intervalu $\langle 0, 1 \rangle$. Typicky ze začátku je to číslo blízko 1 a v průběhu algoritmu se tato ε hodnota snižuje. Před výběrem akce je nejdříve vygenerováno náhodné číslo $x \in \langle 0, 1 \rangle$ a pokud je splněna podmínka $\varepsilon > x \in \langle 0, 1 \rangle$, tak se provede náhodná akce nebo v opačném případě ta nejoptimálnější.

Postupem iterací se přesuneme z provádění náhodných kroků ke krokům predikovaných modelem. Implementace tohoto kroku může být i lineární přesun, jak je zobrazeno níže pomocí pseudokódu.

```
1 // epsilon_decay je malé číslo, např.: 1/100
2 epsilon = 1 - (odehrane_iterace * epsilon_decay)
3 if nahodne_cislo z <0,1> < epsilon then:
4  proved_nahodny_krok
5 else:
6  proved_predikovany_krok
```

Zdrojový kód 1: Epsilon Greedy

Samozřejmě jsou i další propracovanější způsoby jako například Gaussovské rozprostření nebo Thomsonovo vzorkování, ale nám dostačuje tento hladový přístup. Prakticky v naší implementaci řeší tento problém přechodová funkce popsaná v kapitole níže. Ve zkratce, ze začátku neuronová síť nic neví a zkouší náhodné kroky a podle těchto akcí si upravuje svůj model. Ke konci učení, obvykle když již má prozkoumáno prostředí, algoritmus využívá již jenom svou predikci.

[39]exploration explotation dilemma

4 Použité technologie a výběr arkádových her

Pro implementaci Reinforcement Learningu jsem si vybral postup nazývaný Deep Q-Learning[12], kde se učí vícevrstvá neuronová síť pomocí Q-Learningu[22]. Tento postup jsem již nastínil v přechozích bodech. Jako programovací jazyk jsem vybral Python, vzhledem k jeho kladnému vztahu k prototypování, který má nepřeberné množství knihoven a je které možné je použít i pro tuto práci. Jako nevýhodu tohoto jazyka bychom mohli považovat jeho rychlost, protože je to interpretovaný jazyk. Pro naše použití je ale tato skutečnost zanedbatelná.

4.1 Použité knihovny jazyka Python

- **Tensorflow**[2] – Hojně používaná knihovna pro strojové učení, poskytuje nepřeberné množství funkcí, například učení sítě na vlastních datech atd.
- **Keras API** – Tato knihovna (framework) se používá společně s **Tensorflow** a je určena pro širokou škálu uživatelů. Umožňuje jednoduché vytváření neurové sítě, tj. vrstev, jejich optimalizace a jejich uložení do formátu .h5py.
- **H5PY**[25] – Balíček obsahující rozhraní k uložení rozsáhlých dat v binárním formátu. My jej používáme pro uložení naučené neuronové sítě. Je součástí balíčku **Keras**.
- **PyGame**[26] – Knihovna, která je určena k tvorbě počítačových her. Obsahuje nástroje pro zobrazení obrázků, vstupy od uživatelů a další rozhraní pro komunikaci s uživatelem. Pomocí této knihovny byly tedy vytvořeny arkádové hry k naučení neuronových sítí.

4.2 Výběr arkádových her

Pro demonstraci jsem si vybral a vytvořil tři arkádové hry, **Snake**[27], **Space Invaders**[28] a **Asteroids**[29]. Jako první jsem implementoval hru Snake. Další byla hra Invaders, která je o něco složitější než Snake, protože do ní vstupuje i „cizí postava“, stejně jako Asteroid, kde je prvek náhody mnohem větší než u hry Snake.

5 Implementační část

S postupným vývojem těchto tří her (a algoritmu učení neuronových tříd) bylo nakonec odvozeno několik základních tříd, které používají stejné funkce a parametry, které si nyní popíšeme. Nejdříve začneme s pro nás nezajímavými, třídami reprezentujícími základní objekty hry a skončíme se třídami, které odpovídají agentům neuronové sítě.

5.1 Společné třídy

Postupnou prací na této diplomové práci se stalo nezbytností vytvořit některé společné třídy a využít dědění. Navíc vzhledem k vybranému programovacímu jazyku jsou některé třídy pouze rozhraní.

- **ObjectBase** – Tato třída je společná třída pro všechny herní objekty a je potomkem třídy `sprite.Sprite` z knihovny `PyGame`. Obsahuje základní parametry a funkce pro zobrazení a pohyb objektu po plátně. Jako například souřadnice, získání rozměrů obrázků, směru objektu (stupně nebo výčet základních 4 směrů), rotaci obrázku a nastavení těchto hodnot.
- **Alive** – Vytvořena jako rozhraní pro objekty, které v průběhu hry mohou zmizet.
- **Direction** – Třída použita jako společný výčet 4 hlavních směrů, které využíváme.
- **PlayerBase** – Základní třída hráče, která obsahuje informaci o tom, zda je v pohybu a dědí třídu `ObjectBase`.
- **UIBase** – Třída, vytvořena jako rozhraní s funkcemi, které obsluhují vykreslování herních objektů na plátno.
- **sharedmain.py – getUserInput, saveWeightsFile** – Soubor obsahuje funkce, které se mohou společně použít v hlavních třídách, které inicializují pomocí vstupu od uživatele z konzole základní parametry běhu a případně uloží výsledné soubory s váhami.
- **RLNBase** – Tato třída je pro nás nejzajímavější, pro všechny tři případy her použita jako základní třída. V konstruktoru si nastaví základní parametry učení a využívá neuronovou síť o 5 vrstvách, 1 vstupní, 3 skryté vnitřní, 1 výstupní vrstvu. Reprezentuje tedy našeho agenta, který má nějakou paměť, lineární funkci pro konvergenci z explorace do exploitace. K tomuto agentovi se ještě vrátíme později.

5.2 Použitá neuronová síť

Pro všechny případy jsem použil neuronovou síť o 5 vrstvách, kdy velikost první a poslední vrstvy závisí na řešeném problému. Počty neuronů vnitřních schovaných vrstev byly ještě pomocí python knihovny optimalizovány. Počet vnitřních vrstev byl vybrán v závislosti na zdrojích, kdy se nejběžněji používají 2 až 3 vrstvy a větší počet nemusí způsobit markantní zlepšení neuronové sítě a kdy i "mělké" neuronové sítě dosahují velmi kompetitivních výsledků.

Pokud bychom vytvořili velmi širokou a hlubokou neuronovou síť, je zde riziko, že si každá vrstva zapamatuje požadovaný výstup a schopna možná generalizovat na nová data (situaci).

Hloubku i šířku neuronové sítě lze nastavit i konzervativními odhady podle počtů vstupů a výstupů, ale i jednu "obecnou" neuronovou síť lze naučit řešit více problémů.

U jedné implementace (Snake) jsme si vyzkoušeli použít i "optimálnější" velikosti vnitřních vrstev podle poučky Jeffa Heatona[34] a trochu jsme si pohráli s nastavením sítě.

Obrázek 16: Snake

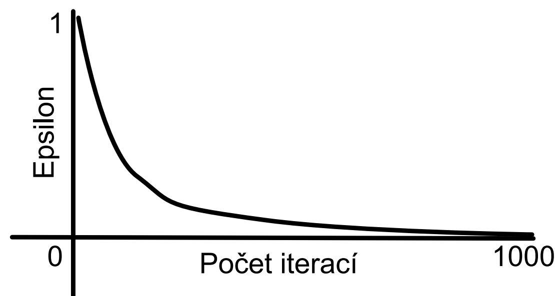
5.3 Funkce pro rozhodování výběru další akce

Jak jsem uvedl dříve, u tohoto typu učení (Deep Q-Learning) se nejdříve provádějí náhodné kroky, které se ukládají do paměti a postupně se přechází na využívání naučených situací. Pro tento přechod jsem si vybral lineární funkci, která je pro všechny naučené neuronové sítě stejná.

```
1 def explorationExploitationFunc(self, playedGames:int, maximumGames:
    int) -> bool:
2   decay = 1 / (maximumGames / 2)
3   epsilon = 1 - (playedGames * decay)
4   return random.uniform(0, 1) < epsilon
```

Zdrojový kód 2: Přechodová funkce

Od určitého počtu iterací, prvek náhody zcela vymizí a rozhodování není již tímto prvekm ovlivněno a provádí akce za pomocí modelu. Prakticky se rozhoduje



Obrázek 17: Vizualizace funkce

tedy model (neuronová síť) ze současného stavu. Stav je jednorozměrná matice, která reprezentuje prostředí a používáme ji jako vstup pro síť.

```
1 prediction = a.model.predict(stav)
```

Zdrojový kód 3: Predikce další akce pomocí knihovny

5.4 Paměť – Replay memory

V naší problematice využíváme paměť pro provedenou akci a její výsledek. Tuto paměť používáme ke znovu přehrání[40] epizod. Paměť v našem případě udržujeme v určité velikosti pomocí fronty, která může mít maximální velikost. Z praktických důvodů, pokud paměť dosáhne určité hranice, je z paměti vybrán náhodný vzorek, který je použit k úpravám modelu. Paměť pro opakované učení se používá pro zrychlení učení sítě (opakované učení k rychlejší konvergenci) a vylepšení výsledků v procesu učení. I tato jednoduchá část procesu lze optimalizovat, například úpravou maximální velikostí paměti[36].

Uložena je do paměti n-tice (starý stav, provedená akce, odměna za přechod, nový stav, ukončení epochy).

```
1 if (len(memory) > self.memory_treshold):
2   batch = random.sample(memory, self.memory_treshold)
3 else:
4   batch = memory
```

Zdrojový kód 4: Výběr vzorku z paměti

[40] replay

5.5 Odměna

Odměna je implementována pomocí metody `setReward` ve sdílené třídě `BaseRLN`. Každá samostatná hra si musí tuto metodu implementovat sama podle potřeby.

Například ve hře Snake je síť odměněna za dosažení jablka (+100) a přiblížení se k jablku (+1). Potrestána za náraz do sebe nebo sebe sama (-100) a za oddálení se od jablka (-1).

5.6 Agent

Ještě se tedy vrátíme k agentovi, který je reprezentován třídou `RLNBase`. Trochu si jej popíšeme a ukážeme jeho souvislosti s přechozí teorií. Nebudeme vytvářet dokumentaci kódu, ale spíše si ukážeme jeho části a případně obecně popíšeme principy.

5.6.1 Konstruktor

Jako první si tedy představíme konstruktor. Zde je tedy možné zadat název souboru s váhami (již naučenou neuronovou sítí) v případě, že chceme jenom spustit "hru neuronové sítě". Tedy něco, co v tuto chvíli není pro nás důležité.

```
1 self.input_dimension = input_dimension
2 self.output_dimension = output_dimension
3 self.setRewardVariable(0)
4 self.setGamma(0.9)
5 self.setMemoryBatchTreshold(1500)
6 self.initVariables()
7 self.setLearningRate(0.0005)
8 self.setWeights(weights_file_path)
9 self.setModel(self.createNetwork())
```

Zdrojový kód 5: Kód konstruktoru

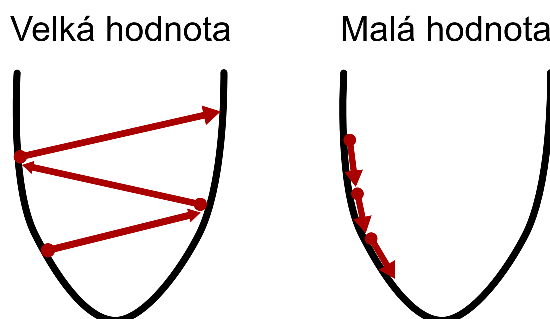
Kvůli obecně vytvořenému agentovi se zde v konstruktoru pomocí parametrů nastavuje velikost vstupní a výstupní jednorozměrné matice (1 a 2 řádek).

Zde provádíme i nastavení γ[41], který ovlivňuje přínos budoucích odměn. V anglických zdrojích je označován jako "damping faktor"nebo "discount factor". U této proměnné platí $0 < \gamma < 1$, typicky je to číslo blízko 1. Použitá hodnota byla vybrána empirickým způsobem a porovnáním dostupných již existujících zdrojů. Můžeme hodnotu nižší γ chápat jako preferenci okamžité odměny oproti vyšší hodnotě, kdy je preference dlouhodobá odměna.

Na řádku 7 provádíme nastavení "learning rate"[42], tedy konvergence k tomu, jak rychle se neuronová síť něco naučí s ohledem na ztráty. Rozdíl mezi "velkým"a

[41] řádek 4
[42] gradient učící funkce

"malým"číslem lze krásně pochopit z obrázku 18. Naše knihovna `Keras` využívá adaptivní learning rate, která překoná statický learning rate.



Obrázek 18: Big a small learning rate

Za zmínku ještě stojí řádek 5, kde se nastavuje velikost vzorku pomocí kterých jsou upravovány váhy. Tohle číslo je možné navýšit, pokud má uživatel větší RAM paměť, jinak hrozí její zaplnění a velké době učení.

Zbylé řádky zajišťují inicializaci agenta, například řádek 9 zajišťuje vytvoření "Deep Neural Network"[43], kterou si popíšeme jako další.

5.7 Vytvoření neuronové sítě (modelu)

Neuronovou síť[44] vytváříme pomocí knihovny `Keras` pomocí třídy `Sequential`, který tedy umožňuje vytvořit více vrstvou síť s jedním vstupním vektorem a jedním výstupním vektorem. Vytváříme pěti vrstvou neuronovou síť, kde vnitřní vrstvy jsou široké po 100 neuronech s náhodnou[45] redukcí neuronů o 15% na každé vrstvě. Takhle vytvořenou síť jsme se alespoň jednou, ze zajímavosti, pokoušeli naučit vybrané hry. Jako aktivační funkci ve vnitřních a vstupní vrstvě používáme tzv. usměrněnou lineární funkci "ReLu"[46] $f(x) = max(0, x)$, protože je jedna z nejpoužívanějších zejména kvůli úspěšnosti při učení. U výstupní vrstvy používáme funkci `softmax` ($f(x) = max(0, x)$), protože výstup lze poté interpretovat jako pravděpodobnost akce.

5.8 Učení agenta (Q-Trainer)

Agent je v našem případě učen ve dvou fázích. V rámci jedné epizody po každém provedení akce dostane agent odměnu a v závislosti na novém stavu (uchováváme si tedy informaci o předchozím stavu, provedené akci, novém stavu a odměně) aktualizuje jeho Q-hodnoty[47]. Zároveň si tyhle informace uloží do "paměti", ze které je po ukončení iterace (například ukončení hry nebo vypršení zvoleného

[43] vícevrstvou neuronovou síť
[44] model
[45] zajišťuje knihovna Keras vlastní metrikou[37]
[46] parametr `relu`
[47] váhy neuronů

```
def createNetwork(self) -> Sequential:
  model = Sequential()

  model.add(Dense(units=100, activation='relu', input_dim=self.
      input_dimension))
  model.add(Dropout(0.15))
  model.add(Dense(units=100, activation='relu'))
  model.add(Dropout(0.15))
  model.add(Dense(units=100, activation='relu'))
  model.add(Dropout(0.15))
  model.add(Dense(units=self.output_dimension,activation='softmax'))
  opt = Adam(self.learning_rate)
  model.compile(loss='mse', optimizer=opt)

  return model
```

Zdrojový kód 6: Vytvoření sítě

limitu), vybrán vzorek a je takto pomocí těchto dat znovu učen[48]. Střídání dvou procesů, výpočtu aktuální politiky a vylepšení aktuální politiky, který se nazývá GPI[49] probíhá v souborech `main.py`. Pro nás si obecně můžeme představit, jak probíhá. Každá implementace tedy obsahuje následující jádro:

```
while games < max_games:
  while running_single_episode:
    trainMemoryWhileLearning(old_state, new_direction, reward,
        new_state, g.crash)
  trainLongMemory(agent.memory)
```

Zdrojový kód 7: Zjednodušený algoritmus učení

5.9 Přechodová funkce

Funkci pro postupný přechod z explorace na exploitaci jsem již nastínil výše. Je to jednoduchá lineární funkce. Tuto funkci lze tedy u konkrétní implementace přepsat a využít nějakou případně vhodnější funkci:

5.10 Implementace Snake

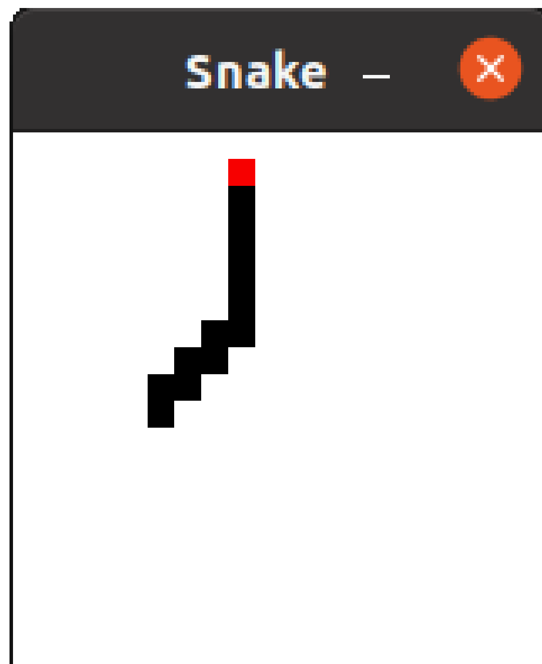
Jako první arkádovou hru jsem implementoval jednoduchou a všem známou hru Snake. Herní plán se skládá z hracího plánu 20x20, tedy 400 polí. V počátečním stavu je uprostřed herního plánu had (černý) a na náhodné pozici je vytvořené

[48]replay
[49]Generalized policy iteration

```python
def explorationExploitationFunc(self, playedGames:int, maximumGames:
    int) -> bool:
  decay = 1 / (maximumGames / 2)
  epsilon = 1 - (playedGames * decay)
  return random.uniform(0, 1) < epsilon
```

Zdrojový kód 8: Zjednodušený algoritmus učení

jablko (červený bod). Model ovládá hada, body získá za snězení jablka a cílem hry je získat co nejvíce bodů. Jako množství her, po kterých bude považována



Obrázek 19: Snake

neuronová síť za naučenou (samozřejmě lze pokračovat), bylo vybráno 200 her, kdy je jedna hra ukončena vyčerpáním kroků, nárazem do zdi nebo do sebe samotného.

5.10.1 Struktura objektů

Strukturu objektů vytvořené hry, stejně jako v následujících případech, si projdeme jenom zběžně, protože to není cílem práce a pokud by to čtenáře zajímalo může si projít přiložené soubory s kódem. Máme hráčskou postavu `Snake`, může se pohybovat ve 4 směrech a postupně mu narůstá tělo, jak získává jablka. Jídlo, `Food`, je tedy to, co hráčská postava následuje a po "snězení" se náhodně zobrazí na prázdném poli na hracím plánu.

Vykreslování, logiku hry, nastavení herního plánu a další související věci zajišťuje třída `UI`. Našeho agenta nám reprezentuje třída `RLN`, kde je tedy nastavení

vstupní dimenze a odměny, obojí popíšeme podrobněji níže.

Nakonec zde máme soubor `main.py`, který je hlavním souborem, co nám spouští a obsluhuje hru. V algoritmu je stejný jako je popsaný výše, jenom obsahuje navíc kód pro vykreslování a podmínky, zda probíhá učení a je zobrazeno uživatelské rozhraní.

5.10.2 Neuronová síť

Vnitřní vrstvy jsou zděděny ze společné třídy `RLNBase`. U této demonstrace jsme vyzkoušeli naučit tři rozdílné topologie sítě, zda bude vidět nějaký markantní rozdíl.

5.10.2.1 Počítačový vstup

Zde máme vstup o velikost 14, který si přiblížíme níže, co to tedy do neuronové sítě vstupuje za reprezentované hodnoty.

- Základní síť má rozměry $100 \times 100 \times 100$
- Minimalistická síť má $14 \times 14 \times$
- Proměnlivá síť má $100 \times 300 \times 100$

5.10.3 Vstup a výstup

Velikost vstupního vektoru je 14 a skládá se z následujících informací: 3× existuje nebezpečí ve směru od hlavy do vzdálenosti 1, 4× jakým směrem se pohybuje, 4× jakým směrem je jídlo, 3× je tělo v některém směru od hlavy.

```
1         state = [self.convertBool(self.dangerFront(game)),
2               self.convertBool(self.dangerLeft(game)),
3               self.convertBool(self.dangerRight(game)),
4               self.convertBool(self.movingLeft()),
5               self.convertBool(self.movingRight()),
6               self.convertBool(self.movingDown()),
7               self.convertBool(self.movingUp()),
8               self.convertBool(self.isFoodLeft(game)),
9               self.convertBool(self.isFoodRight(game)),
10              self.convertBool(self.isFoodDown(game)),
11              self.convertBool(self.isFoodUp(game)),
12              self.convertBool(self.isBodyFront()),
13              self.convertBool(self.isBodyLeft()),
14              self.convertBool(self.isBodyRight())]
```

Zdrojový kód 9: Počítačový vstup

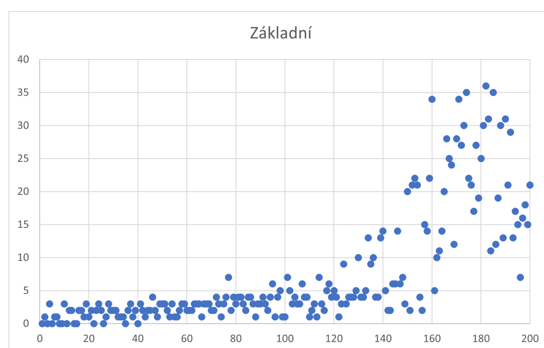
Jako výstup jsou pouze tři možnosti: zda se bude pohybovat ve stejném směru jako doposud, změní směr doleva nebo doprava.

5.10.4 Odměna

Odměna je nastavena následovně: Pokud získá had jídlo, získá 100 bodů. V případě nárazu do zdi nebo do vlastního těla získá záporných 100 bodů. Pokud se přiblíží k jídlu získá 1 bod, jestli se vzdálí získá -1 bod. Odměnu vyčísluje funkce `setReward`.

5.10.5 Výsledky sítě

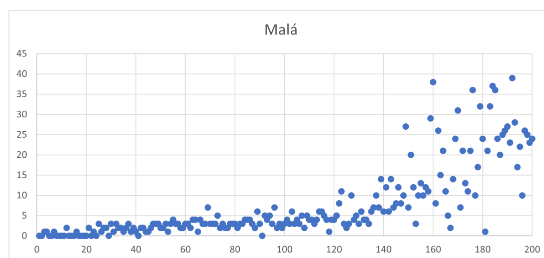
Na následujících třech grafech je zobrazeno dosažené skóre v dané iteraci. Například při iteraci 100 dosáhl agent skóre 10, než prohrál atp. Na vodorovné ose je zobrazeno pořadí iterace a na svislé je zobrazeno skóre. Určitého výraznějšího



Obrázek 20: Graf skóre základní sítě

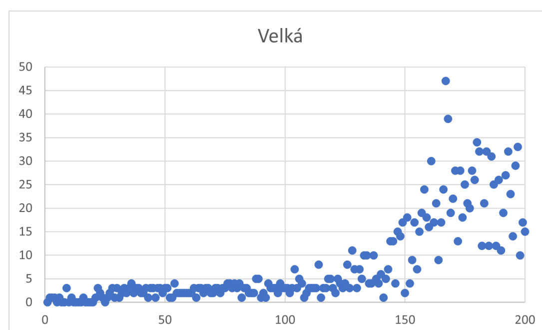
pokroku NS v dosahování skóre zaznamenala kolem 100 iterace, kde začala dosahovat vyššího skóre. To bylo zhruba od bodu, kdy bylo upuštěno od provádění náhodných kroků. Nejvyšší dosažené skóre při učení bylo 36. Doba učení základní sítě byla 8h 42 min 9 sec.

Jako další jsem si připravil vrstvy o vnitřní velikosti 3*14 neuronů. Nejvyšší dosažené skóre při učení byo 43. Doba učení malé sítě byla 8h 45 min 59 sec.



Obrázek 21: Graf skóre malé sítě

Jako poslední jsem vyzkoušel síť o velikosti 100x300x100. Síť se učila delší dobu, ale dosahovala lepších výsledků po kratším množství iterací.



Obrázek 22: Graf skóre velké sítě

Doba učení 8h 52 min 19sec. Při samostatném spuštění již naučená síť dosáhla průměrného skóre 24. Nejvyššího skóre při učení dosáhla 47, které mohlo být spíše nahodilé, než pravidlem.

5.10.6 Shrnutí

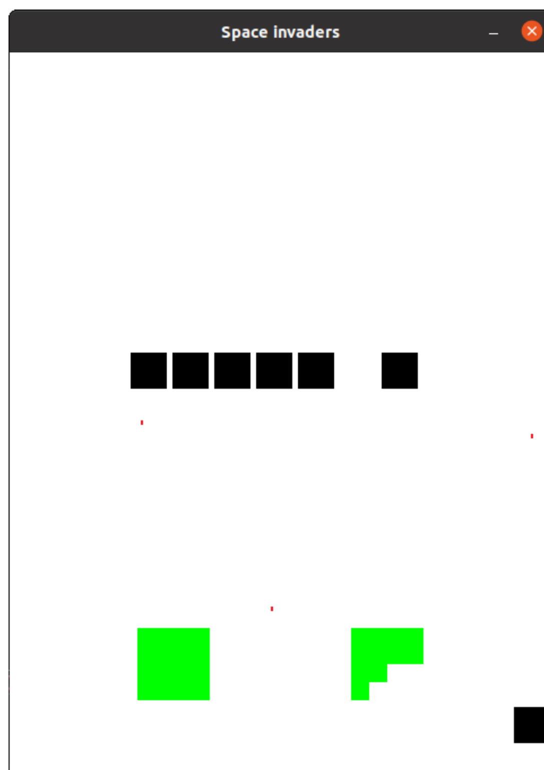
Ukázali jsme si, že lze naučit neuronovou síť hrát hru Snake. V této hře mám jednoduchou metriku, pomocí které můžeme porovnat různé druhy sítí. Obecně vzato se všechny tři sítě učily zhruba stejně dlouhou dobu (největší rozdíl je 10 minut). Průměrné skóre[50] v pořadí základní, malá a velká síť je následující: 24.13, 24.74 a 24.51 a mediánové skóre: 24, 25 a 26. Lze tedy vidět, že rozptyly jsou minimální a že není výrazná změna v dosahovaném skóre. Všechny tyto naučené modely jsou obsaženy v příloze včetně postupných iterací učení modelu[51]. Při kompletním naučení měl had problém se zacyklením sám do sebe, tzn. že se uzavřel do sebe a nemohl uniknout. Lepších výsledků by tedy mohl dosáhnout tím, že bychom mu předali informaci o tom, že při provedení daného kroku se uzavře do sebe.

[50] pěti minutový běh modelu

[51] například při iteraci 20, 40, atd.

5.11 Implementace Invaders

Jako další hru pro demonstraci reinforcement learningu jsem vybral klasiku z roku 1978. Není to přesná kopie, ale dostatečná aproximace původní hry. Hráč-



Obrázek 23: Invaders

ská postava se snaží zničit všechny útočící invadery pomocí střelby. Hráčská postava se může skrýt před nepřátelskými projektily pod bunkry, které může i sám střelbou zničit. Cílem hry je sestřelit všechny útočníky, než přistanou nebo než zničí hráče. Útočníci v náhodných intervalech tak střílí a mohou poškodit hráčskou postavu i bunkry. Pokud jsou všichni útočníci zničeny, začíná se od znova. Hra může skončit taky zničením hráče, který je po třetím zasažením invaderem zničen.

5.11.1 Struktura objektů

Máme tedy hráčskou postavu `Player`, která se může pohybovat v horizontálních směrech, a navíc může vystřelit směrem vzhůru kulku, objekt `Bullet`. Tato hráčská střela může ničit jak bunkry, tak nepřátele. Nad ním jsou tedy tři bunkry (`Bunker`), každý se skládá z bloků (`Block`), který pojme tři střely (`Bullet`), než je zničen. Bunker se tedy skládá z 4*4 objektů `Block`. Nakonec tady máme skupinu nepřátel, objekt `Alien`, kteří se pohybují v horizontálním směru a pokud se některý nepřítel dotkne konce herního plátna udělá vertikální pohyb dolů. Ze

živé skupiny nepřátel v náhodném intervalu nějaký vystřelí (`Bullet`), který může zasáhnout a poškodit bunkr a hráčskou postavu.

5.11.2 Neuronová síť

Vnitřní vrstvy máme ze společné třídy `RLNBase` a používáme počítačový vstup, co se děje na obrazovce.

5.11.3 Vstup a výstup

Vstupní vektor je o velikosti 40 a obsahuje vcelku velké množství informací o prostředí ve kterém se agent nachází. Oproti hře Snake je taky komplikovanější. Informace o vstupu zahrnují:

- Aktuální směr pohybu hráče
- Zda je hráč v pohybu
- Kdy vystřelil (mírná historie v čase) a zda může vystřelit
- Jestli je schován celý za bunkrem nebo částečně
- Jestli a v jakém směru je nad ním Invader
- Směr pohybu skupiny nepřátel
- Zda je nad ním nepřátelská střela
- Vzdálenost skupiny nepřátel (pevně dané vzdálenosti)
- Kde je větší skupina nepřátel s ohledem na hráčkou postavu
- Kde je hráčská postava vzhledem k šířce herního pole rozděleného do 10 stejných částí
- Kde jsou nepřátelé vzhledem k šířce herního pole rozděleného do 10 stejných částí

Výstupní vektor má velikost 6 a může vyvolat následující situace:

- Rozhodne se nedělat nic, tzn může pokračovat v nic nedělání nebo se zastavit
- Pohybovat se ve stejném směru nebo směr změnit
- V každé fázi pohybu i při stání se může pokusit vystřelit

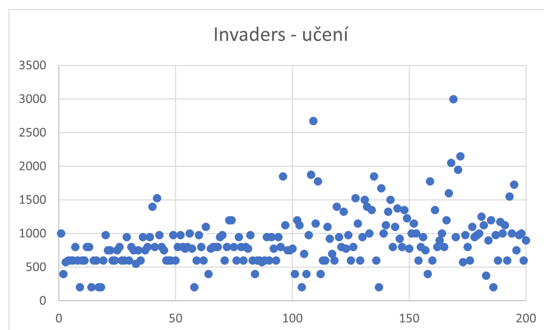
Trochu jednodušeji, může se tedy: Hýbat vlevo nebo vpravo, případně zůstat stát a při tom může vystřelit.

5.11.4 Odměna

Odměnu dostává agent za zničení samostatného Invadera a za zničení všech nepřátel. Zde odměna za zničení samostatného Invadera závisí na jeho vzdálenosti. Čím je blíže, tím dostane větší odměnu. Idea za touto proměnlivou odměnou je, že by pro neuronovou síť mělo být prioritou ničit ty, které jsou nejblíže. Pokud by byla odměna stejná, nemusel by dělat rozdíl mezi tím co je blízko nebo daleko. Úskalí je, ale v tom, že se může naučit je nechat přiblížit, aby měl větší odměnu. Toto není bez rizika, protože poté nemusí všechny Invadery zničit. Vzoreček pro odměnu používáme následující `newScore = (100 * (1 + 0.25 * multiplier) )`, kde `multiplier` je vypočtená mez pevně dané vzdálenosti, 0 je nejdále a 2 je nejblíže. Herní plátno je tak rozděleno do třech úseků, které určují koeficient skóre. Za vyčerpání všech životů nebo přistání invaderů dostane zápornou odměnu. Tedy penalizujeme za prohru.

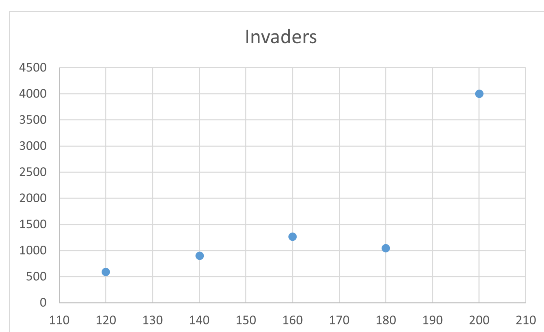
5.11.5 Výsledky sítě

Po zahrání her v prvotním prostředí nebyl schopen nalézt dobrou strategii hry. Problém byl pravděpodobně ve velkém množství informací, které způsobovalo "šum" a bylo by vhodné trénovat delší dobu. Zjednodušili jsme tedy prostředí, kde je menší množství nepřátel a po 200 iteracích byl schopen agent najít strategii, který umožnila vyčistit opakovaně pole od nepřátel. Na grafu máme data z učení, kde na vodorovné ose je pořadí iterace v učení. Jedna hra je ukončena vyčerpáním životů nebo přistáním invaderů. Na svislé ose je dosažené skóre v rámci iterace, které agent získal sestřelením invadera. Z dat v průběhu učení lze



Obrázek 24: Invaders - 200 her

vyčíst, že po zhruba 100 hrách začíná agent dosahovat nějakého vyššího skóre. Ale můžeme se podívat na dosažené skóre i v průběhu učení po 20 hrách, počínaje 120 a končící 200. Při posledních uložených váhách je skokový nárůst skóre, kdy byl tedy agent schopen vyčistit herní pole od nepřátel, aniž by přišel o všechny životy.



Obrázek 25: Invaders - různé fáze učení

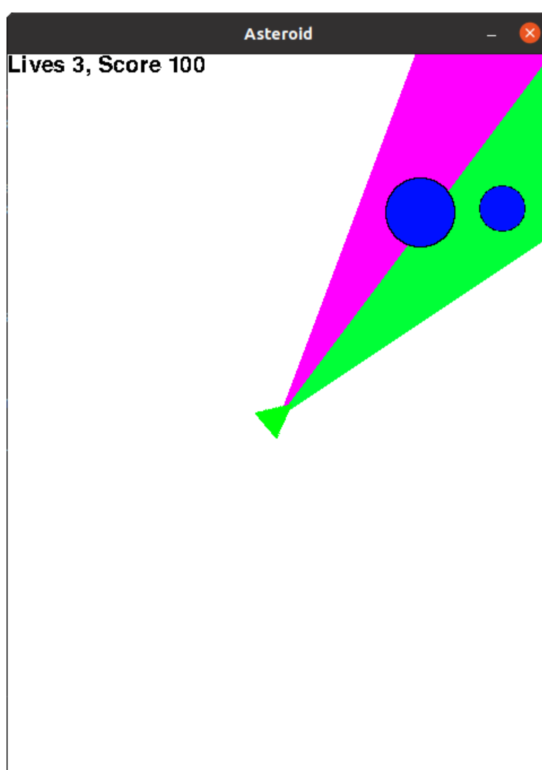
5.11.6 Shrnutí

Agent se naučil po 200 hrách úspěšně čistit pole od nepřátel. Zvolil vcelku nudnou, ale očividně efektivní taktiku, kdy byl v pravém rohu. Z předchozích fází učení lze vidět, že k ní dospěl, když se v předchozích snažil mimikovat pohyb nepřátel. Museli jsme ale zjednodušit prostředí, aby se agent naučil hrát tuto hru v nějaké rozumné míře v rámci našich možností.

5.12 Implementace Asteroid

Jako poslední hru pro demonstraci jsem si vybral hru Asteroid. V této hře model ovládá loď a cílem je sestřelit co nejvíce asteroidů a přežít co nejdéle. Na začátku hry je vytvořen určitý počet asteroidů, které po zasáhnutí střelou se roztříští na menší části, které je zase nutné rozstřelit. Hra je ukončena zničením všech asteroidů, kdy se "začne znovu" nebo po zničení lodi.

Stejně jako v u předchozí implementace, ve hře není pevný konec, ale cílem je získat co nejvíce bodů. Oproti původnímu je i tato hra mírně zjednodušena, zejména z hlediska pohybu. Loď je tedy staticky uprostřed oproti původní akceleraci a deceleraci, zejména proto, aby požadovaná neuronová síť měla o něco jednoduší prostředí, které bude pro ni i tak složité. Takže, aby se hru naučila hrát o něco rychleji.



Obrázek 26: Asteroid

5.12.1 Struktura objektů

Naše hráčská postava, `Player`, která reprezentuje vesmírnou loď, zůstává uprostřed herního plánu a otáčí se kolem své osy. Střílí projektily, `Bullet`, kterými se snaží zničit prolétající asteroidy (`Asteroid`). Tyto asteroidy se po kolizi s projektilem roztříští na více menších asteroidů. Úplně zničit lze pouze již nejmenší možné asteroidy (`Asteroid.size = 0`). Pokud asteroid narazí do vesmírné lodi,

ubere jí jeden život a loď na chvíli získá nesmrtelnost. Asteroidy tedy létají jedním náhodným směrem z 8 možných směru po 45°. Při vyletění z herního plánu je objekt asteroidu přemístěn na odpovídající zrcadlové umístění v plánu.

5.12.2 Neuronová síť

Jako u předchozí demonstrace máme vnitřní vrstvy ze společné třídy `RLNBase` a používáme počítačový vstup, co se děje na obrazovce.

5.12.3 Vstup a výstup

V této hře je tedy hráč statický a poskytujeme mu informace o dvou stavech. Jakým směrem je loď otočena a v kterém směru je asteroid a jestli je blízko nebo daleko[52]. Herní plán tedy máme rozdělen ze středu do 16 výseků, tedy kruh rozdělený po 22.5°. Celková velikost vstupního vektoru je 64.

Výstupní vektor má velikost 6 a může podle něj provést jednu z následujících akcí:

- Může nedělat nic
- Může se otočit vlevo nebo vpravo
- Může provést přechozí akce a zároveň vystřelit

5.12.4 Odměna

Zde je odměna vcelku přímočará, odměnu získá za rozbití asteroidu a zápornou za ztracení životu a konec hry. Vkradla se zde i myšlenka odměnit agenta za „přežití“, tj. dostane každou epizodu malou odměnu za to, že žije, ale v tom případě by mohl schválně prodlužovat hru neničením asteroidů, pokud by měly výhodou trajektorii.

5.12.5 Shrnutí

Nedá se s jistotou říct, že jsme úspěšně naučili neuronovou síť hrát tuhle arkádovou hru. Problémem byl pravděpodobně velký vstupní vektor a velké množství stavů ve kterém se mohla aktuálně hra nacházet. A protože existovalo velké množství stavů, tak některé se kvůli prvku náhody nemusely objevit a některé naopak často opakovat. Je možné, že snížením vstupního vektoru, zjednodušením prostředí a případně selektivním výběrem nějakého prvotního prostředí, na kterém by se postupně učila neuronová síť, dosáhli bychom lepších výsledků. Případně učení za většího množství epizod. Nakonec by šlo vyzkoušet různé možnosti hyperparametrů sítě, ale tohle by nepřineslo drastické zlepšení.

Implementace pomocí zpracování obrazu a použití jej jako vstupu nepřinesla lepší nebo alespoň podobné výsledky než člověk[30][32]. V tomhle učení tedy

[52] 3 vzdálenosti, velká, střední a blízká vzdálenost

proběhlo zhruba 10 miliónu epizod a dosahoval zhruba 7.3% dosaženého skóre lidského hráče. V odkazované implementaci je také možné použít přeskakování obrazu[53], která poskytuje v určitých případech lepších výsledků[31]. Při přeskakování obrazu se tedy nepoužívá každý "snapshot"obrazovky, ale například každý desátý.

[53]frame skipping

6 Uživatelská část

Nyní si ve zkratce popíšeme potřebné technologie, balíčky a využití naprogramovaných aplikací pro učení neuronových sítí. Všechny tři implementace využívají dědění, takže nastavení parametrů neuronové sítě se u všech provádí stejně. Ostatně jako všechna ostatní nastavení či ovládání.

6.1 Předpoklady

- **Python 3.9 (nebo novější)**
- - Nestandardní knihovny pro Python:
  - **Tensorflow**
  - **Keras**[54]
  - **PyGame**

Jako návod na nainstalování předpokladů bych odkázal přímo na stránky knihovny `Tensorflow`, ale můžu varovat, že případná instalace občas nebývá bez problémů z důvodu různé kompatibility CUDA ovladačů grafické karty. Samozřejmě v případě učení pomocí GPU od NVIDIA.

6.2 Konfigurace neuronové sítě

Každá ze tří implementací má svoji třídu, která reprezentuje agenta s názvem souboru `RLN.py`, který je potomkem třídy `RLNBase`. Pokud si tedy uživatel přeje změnit nějaké parametry neuronové sítě, tak v daných souborech. Za zmínku stojí funkce `createNetwork`, která tedy vytváří neuronovou síť, takže pokud by byla chtěna změna vnitřní struktury sítě, tak se provede přepsáním této funkce. Malé upozornění, tato funkce také provádí případné načtení již vytvořených vah (pokud byly předány), takže mít na mysli tuhle funkcionalitu. Výslednou síť je poté nutné uložit funkcí `setModel(Sequential)`. Přechodová funkce `explorationExplotationFunc(int, int)`, zajišťuje lineární přechod mezi náhodnými kroky a predikovanými kroky. Tuto funkci je možné přepsat, aby byl použit jiný výpočet přechodu. Dampening rate (γ) je možné nastavit funkcí `setGamma(float)`. Pomocí funkce `setLearningRate(float)` je možné nastavit learning rate (α) . Odměna, kterou dostane agent, se nastavuje ve funkci `setReward (int, int)`, která je v základní třídě `RLNBase` nedefinovaná a je nutné ji implementovat u každé demonstrace sítě. Tedy v nejzákladnější podobě je nutné nastavit pouze při vytvoření v konstruktoru v děděné třídě velikost vstupního a výstupního vektoru a implementovat metodu `setReward`.

[54]zde by se mohly vyskytnou komplikace při používání GPU, instaloval jsem tyhle dvě knihovny několikrát (jiné stroje, po přeinstalování OS) a nebylo to přímočaré a problémy byly pokaždé jiné, takže bez internetové nápovědy to nepůjde

6.3 Spuštění programu

Pokud máme tedy nainstalované všechny předpoklady a jsme spokojeni s nastavením, tak můžeme spustit veškeré implementace spuštěním souboru `main.py`. Tady v tomto souboru, pokud chceme, můžeme nastavit maximální množství iterací hry pomocí proměnné `max_iter`.

```
1 python3 main.py
```

Zdrojový kód 10: Spuštění programu

Krátká pobídka z konzole nám umožní buď spustit učení nebo načíst váhy a sledovat hru neuronové sítě.

Závěr

Tato práce se zabývá učením umělé neuronové sítě pomocí zpětnovazebného učení. Zaměřuje se částečně na teorii nutnou k pochopení tohoto druhu učení a v praktické části na implementaci vlastních verzí her a praktické použití tohoto učení.

Pokoušeli jsme se tedy prakticky aplikovat tyto postupy a z větší části to můžeme považovat za úspěch. Tato problematika je natolik rozsáhlá, že možná byla chyba snažit se o implementaci ve více hrách namísto jedné. Zejména kvůli následné složitosti v detekování problémů, reprezetaci prostředí a odměny za akce.

Výstupem této práce je tedy několik spustitelných zdrojových kódů, které mohou provést učení sítě nebo zobrazit hru již naučeného modelu. Obsahuje tedy i modely v různých fázích učení.

Dalo by se navíc zamyslet nad výběrem her, zda by nebylo lepší vybrat jiné, například namísto hry Asteroid. Mohly bychom vybrat hru Cartpole, která má násobně jednodušší reprezentaci stavu a odměnu. Ale zase nás to zaneslo směrem do problematiky, že některé přístupy pro vybraný problém nejsou vhodné. Některé přístupy a metody nemusí být obecně vhodné pro řešení problémů. V našem případě naučení jednoho typu sítě všech vybraných her, zejména z hlediska použití vstupu, kdy bychom pravděpodobně dosáhli lepších výsledků za pomocí zpracování obrazu.

Pokud by se v této práci pokračovalo, nejzajímavější by byla implementace a pochopení různých druhů zpětnovazebného učení, i když jeden samostatný typ by byl dostačující na jednu práci.

Conclusions

This thesis deals with learning of artificial neural network using reinforcement learning. We focus partly on theory to understand this kind of learning and in the practical part on the implementation of our own versions of the games and the practical use of this learning.

We have tried to practically apply these procedures and for the most part we can consider it a success. This issue is so vast that it may have been a mistake to try to implement it in multiple games. Mainly because of the resulting complexity in detection, environment representation, and rewards for actions.

The output of this work is several source codes that can learn new model or show the game of an already learned model. It also contains models in different stages of learning.

In addition, one could think about our selecion of games, whether it would not be better to choose another. For example, some other game instead of the game Asteroid. We could choose, for instance, Cartpole game. But again it brought us towards the problem that some approaches are not suitable for the chosen problem. Some approaches and methods do not have to be general solution for solving problems. In our case, teaching one type of network to play all selected games. Mainly because of input type, where we propably would reach better results with processed image as input.

If this work were to continue, the most interesting thing would be to implement and understand different kinds of reinforcement learning, although one type alone would be sufficient for one work.

A Obsah elektronických dat

Na samotném konci textu práce je uveden stručný popis obsahu elektronických dat odevzdaných v systému katedry informatiky spolu s textem. Tato data jsou nedílnou součástí práce a tvoří (datovou) přílohu textu práce. Povinné položky struktury dat jsou:

`docs/`
: Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech (textových) příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu.

`src/`
: Zdrojové kódy aplikace.

`data/`
: Data z průběhu učení a předtrénované modely neuronových sítí.

`readme.txt`
: Pokyny ke spuštění aplikace a požadavky pro aplikaci.

Literatura

[1] SUTTON, Richard S; BARTO, Andrew G. *Reinforcement learning: An introduction.* Second edition. 2018. ISBN 978-0-262-19398-6.

[2] BĚLOHLÁVEK, Radim. *Umělá inteligence.* [2019].

[3] *Module TensorFlow Keras.* [online]. [cit. 2023-07-12]. Dostupné z: https://www.tensorflow.org/api_docs/python/tf/keras

[4] SINGH, Ayush. *Reinforcement Learning : Markov-Decision Process (Part 1)* [2019-07-18]. Dostupné z: https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da

[5] LEE, Dan. *Reinforcement Learning, Part 3: The Markov Decision Process* [2019-10-30]. Dostupné z: https://medium.com/ai%C2%B3-theory-practice-business/reinforcement-learning-part-3-the-markov-decision-process-9f5066e073a2

[6] CHOUDHARY, Ankit. *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python* [2019-04-18]. Dostupné z: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[7] WIKIPEDIE. *Reinforcement learning* [2023-07-10]. Dostupné z: https://en.wikipedia.org/wiki/Reinforcement_learning

[8] WIKIPEDIE. *Markov decision process* [2023-07-10]. Dostupné z: https://en.wikipedia.org/wiki/Markov_decision_process

[9] COMI, Mauro. *How to teach AI to play Games: Deep Reinforcement Learning* [2018-11-15]. Dostupné z: https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a

[10] SALLOUM, Ziad. *Exploration in Reinforcement Learning* [2019-04-24]. Dostupné z: https://towardsdatascience.com/exploration-in-reinforcement-learning-e59ec7eeaa75

[11] VIOLANTE, Andre. *Simple Reinforcement Learning: Q-learning* [2019-03-18]. Dostupné z: https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56

[12] KEON, Kim. *Deep Q-Learning with Keras and Gym* [2017-02-06]. Dostupné z: https://keon.github.io/deep-q-learning/

[13] KRAJčA, Petr;OSIČKA, Petr. *Magazín katedry informatiky, číslo 15* Zpětnovazebné učení I. [2021-09-01].

[14] KRAJčA, Petr;OSIČKA, Petr. *Magazín katedry informatiky, číslo 16* Zpětnovazebné učení II. [2022-02-01].

[15] KRAJčA, Petr;OSIČKA, Petr. *Magazín katedry informatiky, číslo 17* Zpětnovazebné učení III. [2022-09-01].

[16] MAYANK, Mohit. *Reinforcement Learning with Q tables* [2018-05-02]. Dostupné z: https://itnext.io/reinforcement-learning-with-q-tables-5f11168862c8

[17] WIKIPEDIE. *Mean squared error* [2023-07-11]. Dostupné z: https://en.wikipedia.org/wiki/Mean_squared_error

[18] WIKIPEDIE. *Delta Rule* [2023-07-11]. Dostupné z: https://en.wikipedia.org/wiki/Delta_rule

[19] NAYAK, Priyanka. *Learning rule* [2013-05-28]. Dostupné z: https://blog.oureducation.in/learning-rule/

[20] VOLNÁ, Eva. *NEURONOVÉ SÍTĚ 1* [2008-01-01]. Dostupné z: https://web.osu.cz/~Volna/Neuronove_site_skripta.pdf

[21] AKRUYHAU. *Bellman Equation* [2021-09-17]. Dostupné z: https://www.geeksforgeeks.org/bellman-equation/

[22] ADL. *An introduction to Q-Learning: reinforcement learning* [2018-09-03]. Dostupné z: https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/

[23] BAKKER, Bram. *Reinforcement Learning with Long Short-Term Memory* [2001-01-01]. Dostupné z: https://proceedings.neurips.cc/paper_files/paper/2001/file/a38b16173474ba8b1a95bcbc30d3b8a5-Paper.pdf

[24] SOEMERS, Dennis. *Why AlphaGo didn't use Deep Q-Learning?* [2020-04-29]. Dostupné z: https://ai.stackexchange.com/questions/20585/why-alphago-didnt-use-deep-q-learning/

[25] COLLETE, Andrew. *HDF5 for Python* [2023-07-13]. Dostupné z: https://docs.h5py.org/en/stable/

[26] PYGAME. *PyGame* [2023-07-14]. Dostupné z: https://www.pygame.org/news

[27] WIKIPEDIE. *Snake (video game genre)* [2023-07-10]. Dostupné z: https://en.wikipedia.org/wiki/Snake_(video_game_genre)

[28] WIKIPEDIE. *Space Invaders* [2023-07-10]. Dostupné z: https://en.wikipedia.org/wiki/Space_Invaders

[29] WIKIPEDIE. *Asteroids (video game)* [2023-07-10]. Dostupné z: https://en.wikipedia.org/wiki/Asteroids_(video_game)

[30] GOOGLE DEEPMIND, 5 New Street Square, London EC4A 3TW, UK. *Human-level control through deep reinforcement learning* [2015-02-26]. Dostupné z: https://www.cs.swarthmore.edu/~meeden/cs63/s15/nature15b.pdf

[31] BRAYLAN Alex, HOLLENBECK Mark, MEYERSON Elliot a MIIKKULAINEN Risto. *Frame Skip Is a Powerful Parameter for Learning to Play Atari* [2015-01-01]. Dostupné z: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d893434d78a81f7c7a7c9aee44894b53c7b1359d

[32] GOOGLE DEEPMIND, 5 New Street Square, London EC4A 3TW, UK. *Massively Parallel Methods for Deep Reinforcement Learning* [2016-07-17]. Dostupné z: https://arxiv.org/pdf/1507.04296.pdf

[33] WIKIPEDIE. *Mean squared error* [2023-07-18]. Dostupné z: https://en.wikipedia.org/wiki/Mean_squared_error

[34] HEATHON Jeff, 5 New Street Square, London EC4A 3TW, UK. *The Number of Hidden Layers* [2017-06-01]. Dostupné z: https://www.heatonresearch.com/2017/06/01/hidden-layers.html

[35] BERGER-TAL, Oded; NATHAN, Jonathan; MERON, Ehud; SALTZ, David. *The Exploration-Exploitation Dilemma: A Multidisciplinary Framework* [2014-04-22]. Dostupné z: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0095693

[36] ZOU, James. *The Effects of Memory Replay in Reinforcement Learning* [2018-11-5]. Dostupné z: https://proceedings.allerton.csl.illinois.edu/2018/media/files/0091.pdf

[37] BROWNLEE, Jason. *Dropout Regularization in Deep Learning Models with Keras* [2022-07-6]. Dostupné z: https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/