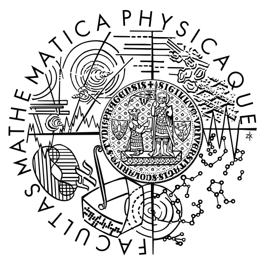


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

DIPLOMOVÁ PRÁCE

Bc. Jan Klůj

Obecná umělá inteligence pro hraní her

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce:	Mgr. Martin Pilát, Ph.D.
Studijní program:	Informatika
Studijní obor:	Umělá inteligence

Praha 2017

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 15. 7. 2017 Podpis autora

Název práce: Obecná umělá inteligence pro hraní her

Autor: Bc. Jan Klůj

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Martin Pilát, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Hraní her je v současné době poměrně zajímavý problém na poli umělé inteligence. V diplomové práci se zabýváme tvorbou obecné umělé inteligence, která je schopna hrát vybrané jednoduché počítačové hry na základě informací, které jsou dostupné i lidskému hráči. Našimi vybranými hrami jsou 2048, Mario, závodní simulátor TORCS a Alhambra. Všechny informace, které umělá inteligence získává, jsou poskytovány hrami pomocí rozhraní a žádný model tak nevyužívá obrazový vizuální vstup. Využíváme evolučních přístupů jako jsou evoluční algoritmy, evoluční strategie CMA a diferenciální evoluce, aplikované na různé typy neuronových sítí. Dále se zabýváme hlubokým zpětnovazebním učením. Tyto přístupy testujeme a jejich výsledky porovnáváme.

Klíčová slova: umělá inteligence, strojové učení, hraní her

Title: General Artificial Intelligence for Game Playing

Author: Bc. Jan Klůj

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Game playing is a relatively interesting task in the field of artificial intelligence in these days. The master thesis deals with general artificial intelligence which is capable of playing selected simple games based on information that is also available to the human player. Our selected games are 2048, Mario, racing simulator TORCS and Alhambra. All the information acquired by artificial intelligence is provided by games through an interface, therefore none of the models uses visual input. We use evolutionary approaches such as evolutionary algorithms, evolutionary strategy CMA and differential evolution applied to different types of neural networks. We are also dealing with deep reinforcement learning. We test these approaches and compare their results.

Keywords: artificial intelligence, machine learning, game playing

Rád bych na tomto místě poděkoval především Mgr. Martinu Pilátovi, Ph.D. za všechny cenné rady a pomoc při vytváření této práce. Dále bych rád poděkoval své rodině za neustálou podporu během celého studia.

Obsah

Úvod 3

1 Vybrané hry 5
1.1 Hra 2048 5
1.2 Mario 5
1.3 Závodní hra TORCS 6
1.4 Alhambra 7

2 Umělé neuronové sítě 8
2.1 Úvod do neuronových sítí 8
2.2 Proces učení 9
2.2.1 Učení bez učitele 9
2.2.2 Učení s učitelem 9
2.3 Echo-state síť 11

3 Zpětnovazební učení 13
3.1 Princip 13
3.2 Q-učení 14
3.2.1 Explorace versus exploatace 15
3.3 Aplikace neuronových sítí ve zpětnovazebním učení 16
3.3.1 Hluboké zpětnovazební učení 16
3.4 Spojité akce agentů 18
3.4.1 Algoritmus DDPG 18

4 Evoluce 21
4.1 Jednoduchý evoluční algoritmus 21
4.1.1 Inicializace 21
4.1.2 Selekce 21
4.1.3 Křížení 22
4.1.4 Mutace 22
4.2 Evoluční strategie 22
4.2.1 CMA-ES 23
4.3 Diferenciální evoluce 24

5 Algoritmy pro zvolené hry 26

6 Aplikace modelů a experimenty 28
6.1 Obecnost 28
6.2 Reprezentace her 29
6.2.1 Hra 2048 29
6.2.2 Mario 29
6.2.3 TORCS 30
6.2.4 Alhambra 31
6.3 Zvolené modely 33
6.3.1 Evoluční přístupy 33
6.3.2 Přístupy zpětnovazebního učení 35

6.4 Zmínka o implementaci . . . 35
6.5 Experimenty . . . 36
6.5.1 Parametry . . . 36
6.5.2 Testování . . . 37
6.5.3 Hra 2048 . . . 37
6.5.4 Mario . . . 40
6.5.5 TORCS . . . 41
6.5.6 Alhambra . . . 44
6.5.7 Vyhodnocení a poznámky . . . 46

Závěr **47**

Seznam použité literatury **48**

Seznam obrázků **52**

Seznam tabulek **53**

Seznam použitých zkratek **54**

A Detailní přehled experimentů **56**

B Implementace a dokumentace **64**
B.1 Alhambra a původní umělá inteligence . . . 64
B.2 Implementace . . . 64
B.3 Technické informace . . . 65

C Obsah přiloženého DVD **68**

Úvod

V poslední době se rozvíjí způsoby a algoritmy, které jsou schopné se samy naučit hrát jednoduché počítačové hry. V takových úlohách často nevíme, jaké akce by měl vlastně počítačový hráč (agent) v dané situaci udělat, proto je velmi obtížné takového agenta programovat přímo. Existují však techniky strojového učení, kterým stačí poskytovat informace o prostředí a ohodnocovat je, na základě čehož se dané modely snaží zlepšovat. V této práci se podíváme na některé algoritmy a přístupy, jak naučit modely strojového učení tyto hry hrát. Dalším cílem práce bude tyto modely porovnat, a to jak mezi sebou, tak s existujícími implementacemi. Hlavní požadavek na naše modely bude to, že nebudeme využívat specifických znalostí jednotlivých her, ale pokusíme se je aplikovat obecně pro více her.

Mnoho problémů z různých odvětví je řešeno pomocí umělých neuronových sítí, ať už se jedná o odvětví strojírenství, medicíny nebo ekonomie. Právě neuronové sítě v poslední době dosahují poměrně dobrých výsledků, například ve zpracování přirozeného jazyka [1], nebo klasifikaci obrázků [2]. Dalším z takových odvětví je však právě hraní her, kde významných výsledků bylo dosaženo například ve hraní Atari her [3, 4], a také ve hře Go [5].

Rozhodli jsme se tedy také využívat některé typy neuronových sítí. Vhodnými kandidáty, jak učit neuronové sítě, jsou evoluční techniky. Jedná se o algoritmy a postupy, které jsou inspirovány přírodou, evoluční teorií. Zaměříme se na různé typy algoritmů z výpočetní evoluce, jako jsou evoluční algoritmy a diferenciální evoluce. Také vyzkoušíme učit neuronové sítě pomocí evoluční strategie, konkrétně strategií CMA, která je v současné době považována za jednu z nejvýznamnějších ve výpočetní evoluci.

Další vhodný způsob v naší situaci je hluboké zpětnovazební učení. Jedná se o techniku založenou na předkládání informací o prostředí a následnému ohodnocování agenta, tedy poskytování zpětné vazby, která udává jistou míru toho, jak dobře se agent v daném okamžiku chová. Tento přístup je často přirovnáván ke skutečnému životu, kdy se agent snaží přizpůsobit vnějšímu prostředí, aby dosáhl svých cílů.

Tyto modely jsme si vybrali především proto, protože je považujeme za zajímavé a stále aktuální v současném světě.

Naše modely budeme testovat na několika jednoduchých hrách, které se však od sebe významně liší. Budeme pracovat s diskrétními i spojitými světy. Podíváme se na 2048, což je hra s posuvnými bloky na mřížce velikosti 4×4. Tato hra je poměrně nová, a stala se oblíbenou především díky její mobilní verzi. Dále použijeme hru Mario, která je velmi známou plošinovou arkádovou videohrou. The Open Racing Car Simulator (TORCS) je závodní simulátor, který byl vyvinut především pro vývoj umělé inteligence a je často používán jako benchmark kvality daných agentů. Budeme jej používat i my a je hlavním reprezentantem hry se spojitým světem. Jako poslední jsme zvolili karetní hru Alhambra.

Struktura práce

Práce je členěna do následujících kapitol. V kapitole 1 detailněji rozebereme jednotlivé hry a to, jaký mají cíl. Dále se v kapitole 2 podíváme na způsob fungování

umělých neuronových sítí. Kapitola 3 definuje pojmy zpětnovazebního učení, popisuje jak funguje a jak jej můžeme propojit s neuronovými sítěmi. Podobně jako předchozí kapitoly, bude i kapitola 4 definovat pojmy, tentokrát týkající se evolučních přístupů. V kapitole 5 zmíníme pro naše zvolené hry jiné autory a jiné implementace agentů, kteří byli vytvořeni již dříve. Následuje kapitola 6, ve které uvedeme hlavní část práce, a to jak jsme aplikovali dříve popsané modely a přístupy na náš problém. Všechny vykonané experimenty jsou k dispozici v příloze A. Technické části práce jako je implementace a dokumentace se věnujeme v příloze B. Na závěr je uvedena příloha C, popisující přiložené DVD.

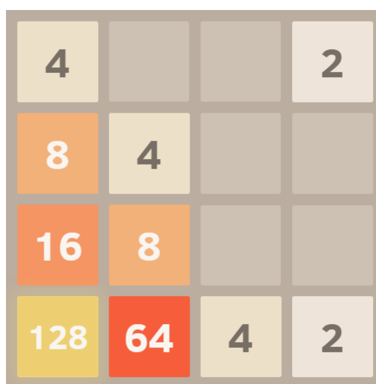
1. Vybrané hry

Hry, které jsme si zvolili jsou reprezentanty různých žánrů a obtížností. Zároveň jsme zvolili hry s diskrétními (tahovými) světy i hry se světem spojitým. Jedním z našich požadavků na výběr je to, aby dané hry byly značně rozdílné.

1.1 Hra 2048

První ze zvolených her je hra 2048. Jedná se o jednoduchou hru s posuvnými dlaždicemi na mřížce 4×4, kterou vymyslel a poprvé naprogramoval Gabriele Cirulli [6] v roce 2014.

Cílem této hry je posouvat dlaždice po herní ploše, kombinovat je a vytvářet tak dlaždice nové (větší). Dlaždice je možné posunovat ve čtyřech směrech a při každém posunu se vždy pohybují všechny dlaždice, pokud jim to herní plocha dovolí. Dlaždice mají hodnoty mocnin dvojky a při posunech se mohou spojovat, ale pouze dlaždice se stejnými hodnotami. Pokud k takovému spojení dojde, vznikne dlaždice s hodnotou dvakrát větší než původní. Na začátku hry jsou náhodně rozmístěny dvě dlaždice a při každém tahu se vygeneruje dlaždice nová. Nově vzniklé dlaždice se generují s hodnotou dva s pravděpodobností 90 % a s hodnotou čtyři s pravděpodobností 10 %. Hráč vyhrává pokud, vytvoří dlaždici 2048. Hra trvá, dokud se nezaplní celé hrací pole. Hra může tedy pokračovat i po dosažení cílové dlaždice 2048.

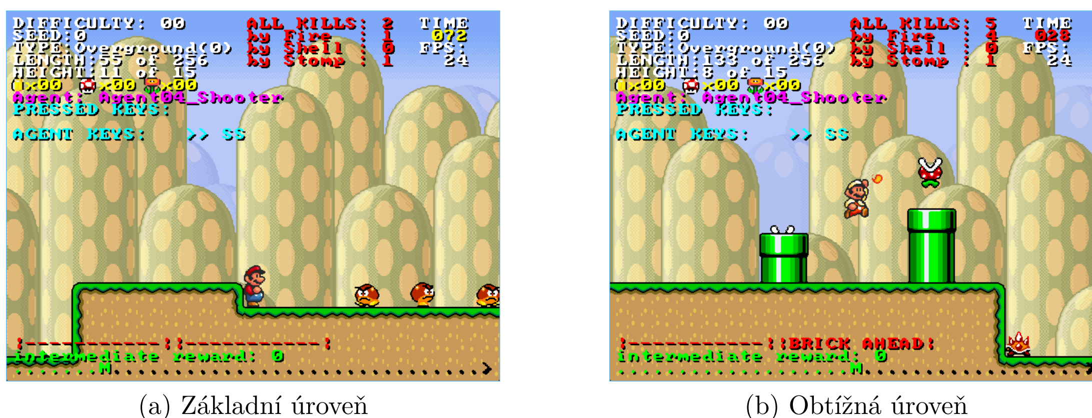


Obrázek 1.1: Hra 2048

1.2 Mario

Mario je velmi známá arkádová plošinová hra od společnosti Nintendo [7]. Hráč ovládá postavu Maria a snaží se dostat na konec dané úrovně (levelu). Dle nastavení obtížnosti úrovně Mario cestou potkává různé objekty a nepřátele, které může přeskočit nebo zničit. Mario je také schopen střílet a má tři životy. Při kolizi s nepřátelskými jednotkami Mario ztrácí jeden život, a pokud ztratí všechny, hra končí prohrou. Budeme využívat dvou různých konfigurací obtížnosti hry a každá jednotlivá hra bude náhodně vygenerována (různý terén, dané objekty v různých místech apod.).

První nastavení, které považujeme za jednoduchou úroveň, obsahuje hornatý terén a základní nepřátele, které nazýváme Gomba. Tyto jednotky lze zničit buď vystřelením, nebo dopadnutím po výskoku Maria. Druhým nastavením je složitější úroveň, která navíc obsahuje nepřátele, které nelze zničit ani jedním ze dvou zmiňovaných způsobů a označujeme je jako Spike. Dále tato úroveň obsahuje navíc zničitelné jednotky Flower. Na obrázku 1.2 můžeme vidět obě dvě nastavení hry.



(a) Základní úroveň

(b) Obtížná úroveň

Obrázek 1.2: Hra Mario

Mario může udělat několik akcí, jako jsou běh doleva, běh doprava, skok a střelba.

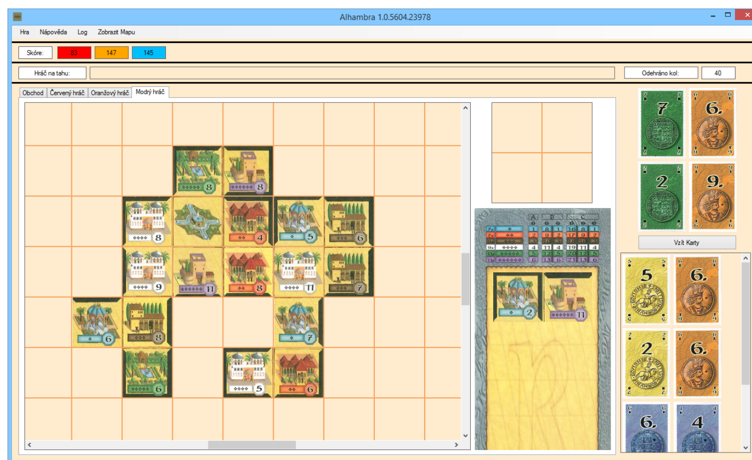
1.3 Závodní hra TORCS

TORCS [8] je závodní simulátor určený především pro vývoj umělé inteligence a slouží ve většině případů i jako dobrý benchmark, a právě proto jsme se jej rozhodli použít. Hra jako taková nabízí spoustu herních modů, jako je například kvalifikace, tréning a závod samotný. Pro naše účely jsme si zvolili mód tréning, kdy je na zvolené trati pouze jedno vozidlo a má za cíl objet jedno (nebo více) kol. Hra rovněž podporuje poškození auta a palivo, které auto během jízdy spotřebovává, tyto vlastnosti lze však vypnout a v našem případě využíváme pouze poškození vozidla.

Obrázek 1.3: Závodní simulátor TORCS

1.4 Alhambra

Alhambra je desková karetní hra, jejíž autorem je Dirk Henn [9]. Hra obsahuje dva typy karet — karty peněz a budov. Každý hráč postupně odebírá karty, za které může následně kupovat dlaždice budov a ty umisťuje ve svém herním poli. Hra obsahuje čtyři typy (barvy) peněžních karet a šest typů (barev) budov. Hry se účastní dva až šest hráčů a cílem je postavit co možná nejlepší Alhambru. Kvalita postavené Alhambry se hodnotí podle počtu postavených budov dané barvy a body se také udělují za nejdelší postavenou stěnu kolem Alhambry — některé budovy mají stěny, což můžeme vidět na obrázku 1.4.



Obrázek 1.4: Karetní hra Alhambra

Body se hráčům udělují v tzv. bodovacích kolech. Během hry se udělují body třikrát. Dvakrát po vytažení speciální bodovací karty a jednou na konci hry. Hru tak můžeme rozdělit na tři fáze, mezi jednotlivá sčítací kola.

Tato hra se liší od ostatních také tím, že implementace, kterou používáme, byla napsána jako součást vlastní bakalářské práce [10]. Z toho plyne několik omezení a úprav, které rozebereme v kapitole aplikace modelů. Pravidla zde nebudeme podrobněji vypisovat, pro detailnější popis doporučíme online pravidla[1].

[1] `http://deskovehry.cz/index.php/Alhambra`, Cit. 2017-06-18.

2. Umělé neuronové sítě

V posledních letech slýcháváme o neuronových sítích stále častěji a častěji. Umělé neuronové sítě jsou matematickým modelem, který byl inspirován lidským mozkem a biologickým nervovým systémem.

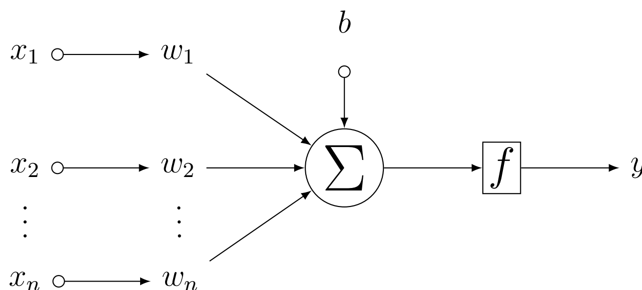
Tato kapitola poskytuje úvod do problematiky neuronových sítí a také popis typů sítí, které budeme později používat v našich modelech.

2.1 Úvod do neuronových sítí

Neuronová síť se skládá z neuronů a ty jsou propojeny váženými spojeními. Nejzákladnější typ neuronové sítě je perceptron [11]. Nechť n je dimenze našeho prostoru. Perceptron se skládá z vah $w = (w_1,w_2,\ldots,w_n)$, prahu b a aktivační funkce f. Máme-li vstup $x = (x_1,x_2,\ldots,x_n)$, pak výstup perceptronu y spočítáme jako

$$y = f\left(\sum_{i=1}^{n} x_i w_i + b\right).$$

Perceptron je znázorněn na obrázku 2.1:



Obrázek 2.1: Perceptron

Aktivační (také nazývaná přenosová) funkce f může být různých typů. Mezi nejpoužívanější patří:

a) skoková

$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{jinak} \end{cases}$$

b) sigmoidální

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

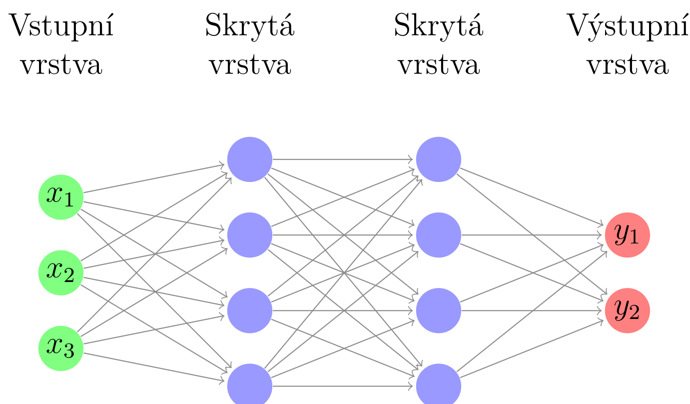
c) ReLU (Rectified Linear Unit)

$$f(x) = \begin{cases} x & x > 0 \\ 0 & \text{jinak} \end{cases}$$

d) hyperbolický tangens

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Perceptron sám o sobě má velmi omezené schopnosti. To můžeme ale zlepšit tím, že jednotlivé perceptrony budeme skládat do vrstev — jedna vstupní, m skrytých a jedna výstupní. Každý neuron je propojen s každým neuronem v další vrstvě. Takové síti budeme říkat MLP[1]. Ilustraci MLP sítě můžeme vidět na obrázku 2.2.



Obrázek 2.2: MLP síť

Průchod z i-té do $(i + 1)$-ní vrstvy spočítáme jako $\mathbf{y}_i = W_i \cdot \mathbf{y}_{i-1} + \mathbf{b}_i$, kde W_i je váhová matice (vah mezi danými vrstvami) a $\mathbf{b}_i$ prahový vektor. Tomuto průchodu sítí se také říká dopředný průchod.

2.2 Proces učení

Abychom dostali požadovaný výstup z MLP sítě, je potřeba, aby váhové matice (tj. parametry sítě) byly správně nastaveny. Existují dva hlavní přístupy, jak učit neuronové sítě, a to učení s učitelem a učení bez učitele.

2.2.1 Učení bez učitele

Učení bez učitele[2] pro svůj běh nepotřebuje předem znát správné výstupy sítě a učení probíhá pouze na základě vstupních vzorů. Jedná se například o problémy klastrování, zaměříme se však spíše na učení s učitelem, hned v následující sekci.

2.2.2 Učení s učitelem

Učení s učitelem[3] předpokládá, že máme pro dané vstupy i očekávané výstupy. Jedním z nejznámějších způsobů učení s učitelem pro učení MLP sítě je algoritmus zpětného šíření chyby Backpropagation (BP) [12]. Nechť $\{(x_1,d_1),\dots,(x_N,d_N)\}$ je trénovací množina, kde $x_i \in \mathbb{R}^n$ jsou vstupy a $d_i \in \mathbb{R}^n$ k nim požadované výstupy. Na počátku algoritmus nastaví parametry (váhy) sítě náhodně a postupně se jej snaží opravovat. Algoritmus se snaží minimalizovat chybu sítě na trénovacích

[1] Z anglického multi-layer perceptron.
[2] Z anglického unsupervized learning.
[3] Z anglického supervized learning.

datech. Tato chyba může mít různé definice, například rozdíl čtverců MSE (Mean Squared Error):

$$E = \frac{1}{2}\sum_{i}^{N}\sum_{j}^{n}(y_{i,j} - d_{i,j})^2,$$

kde N je počet vzorů, n dimenze prostoru, $\mathbf{y}$ výstup ze sítě pro vstup $\mathbf{x}$ a $\mathbf{d}$ k němu požadovaný výstup.

Označme $\theta_i(t)$ jako i-tou váhu sítě v čase t a α parametr učení (learning rate). Aktualizace parametrů sítě (tj. změna vah θ_i) probíhá na základě gradientu chybové funkce vzhledem ke specifickým vahám:

$$\theta_i(t+1) = \theta_i(t) - \alpha\frac{\partial E}{\partial \theta_i}.$$

Odečítáme tedy gradient od současných parametrů. Pro aktivační funkce neuronů předpokládáme, že splňují podmínky diferencovatelnosti a spojitosti, abychom mohli daný gradient spočítat.

Pro další informace a podrobnější technickou analýzu algoritmu Backpropagation doporučujeme online interaktivní knihu Michaela Nielsena [13].

Možná vylepšení

Základní verze algoritmu každou iteraci počítá gradient přes celou trénovací množinu. Kvůli rychlosti se využívá aproximace tohoto gradientu pouze na základně několika vzorů, které tvoří dávku (batch). Tato dávka se vybírá náhodně, proto se tomuto vylepšení říká stochastická gradientní metoda, neboli SGD[4] [14].

Další možností je zavedení momentu [15], který upraví aktualizační pravidlo tak, že kromě gradientu v rovnici aktualizace přidá rozdíl současných a minulých vah:

$$\theta_i(t+1) = \theta_i(t) - \alpha\frac{\partial E}{\partial \theta_i} + \alpha_m(\theta_i(t) - \theta_i(t-1))$$

To má za cíl „držet směr“ v prostoru chybové funkce a zamezit oscilacím. Obecně můžeme říci, že čím déle „jdeme“ jedním směrem, tím větší kroky algoritmus provádí. α_m je parametr, určující důležitost momentu.

Dále můžeme upravovat parametr učení, a to několika způsoby. Místo globálního parametru učení jej můžeme zavést pro každou váhu sítě zvlášť a navíc nebude statický, ale bude exponenciálně klesat. Takto vylepšenými algoritmy jsou Adagrad a RMSProp [16, 17]. Můžeme však udělat ještě jeden krok navíc — kromě adaptivního parametru učení budeme upravovat gradient pro aktualizaci. Nechť β_1 a β_2 jsou parametry a β_1^t, β_2^t značí jejich mocniny. Nechť g_t je vektor gradientů pro jednotlivé váhy (budeme používat vektorový zápis kvůli přehlednosti) a g_t^2 jejich druhé mocniny. Definujme

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)g_t^2,$$

kde m_t je nazýván *první moment* a v_t *druhý moment*. Na počátku volíme $m_0 = 0$ a $v_0 = 0$, což však během prvních iterací způsobuje zkreslení, proto provádíme navíc korekci:

$$\widehat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

[4] Z anglického Stochastic Gradient Descent.

$$\widehat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Výsledné aktualizační pravidlo pak je

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\widehat{v}_t} + \epsilon}\widehat{m}_t.$$

Výše popsaný algoritmus je znám pod jménem Adam [18]. Jeho autoři za výchozí vhodné hodnoty parametrů považují $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. Pro úplnost uvedeme pseudokód algoritmu, kde $f(\theta)$ značí obecnou objektivní funkci[5] s parametry θ:

Algoritmus 1 Adam (Adaptive Moment Estimation)

1: $m_0 \leftarrow 0$
2: $v_0 \leftarrow 0$
3: $t \leftarrow 0$
4: **while** θ_t nezkonvergovalo **do**
5: $\quad t \leftarrow t+1$
6: $\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
7: $\quad m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$
8: $\quad v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$
9: $\quad \widehat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
10: $\quad \widehat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
11: $\quad \theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t + \epsilon}}$
12: **end while**
13: **return** θ_t

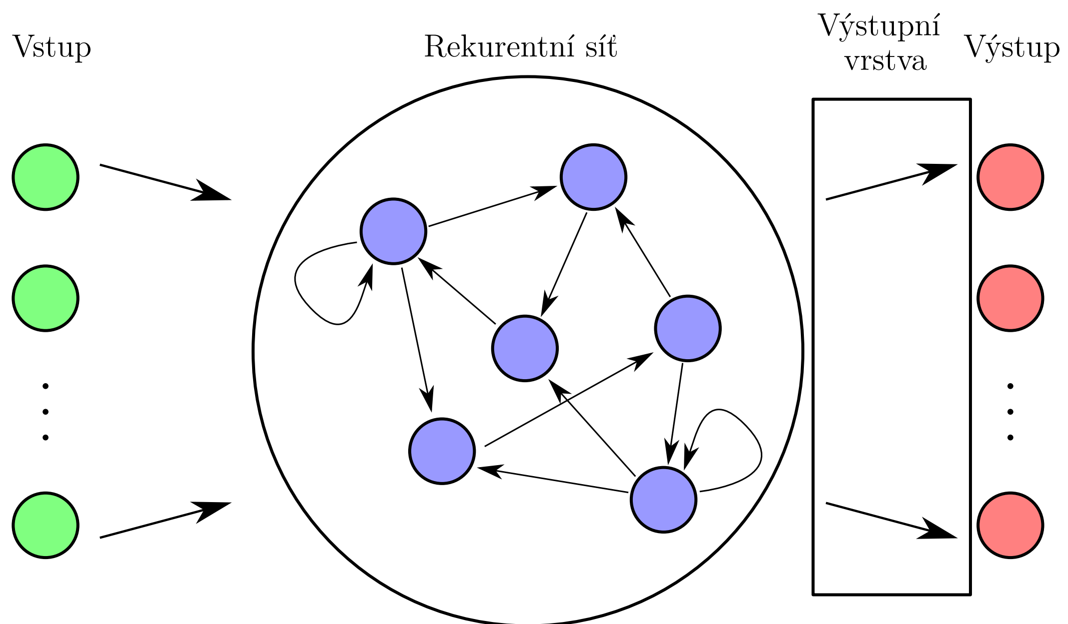
Podrobnější přehled vylepšení gradientních metod (i s visualizacemi) je možné najít v článku [14].

2.3 Echo-state síť

Abychom si mohli zadefinovat Echo-state síť, musíme si nejprve povědět něco o rekurentních neuronových sítích [19]. Rekurentní síť se od standardní MLP sítě liší tím, že obsahuje rekurentní spoje, které jsou spojením jakýchkoliv neuronů v síti, tedy spojení neuronů ve stejné vrstvě, spojení do protější vrstvy, nebo spojení jednoho neuronu se sebou. Jinými slovy, graf neuronové sítě může být cyklický a jedná se tak o zobecnění MLP sítě.

Rekurentní neuronové sítě však může být těžké trénovat, chtěli bychom tedy síť, která bude mít vlastnosti rekurentních neuronových sítí, ale bude jednoduchá jako MLP. Takovým řešením jsou echo-state sítě (ESN) [20], které mají následující princip: vygenerujeme náhodnou rekurentní síť předem specifikované velikosti a následně vybereme náhodné neurony z rekurentní části a připojíme standardní plně propojenou[6] vrstvu MLP sítě (nebo více vrstev). Obrázek 2.3 znázorňuje takovou síť.

[5] Objective function.
[6] V anglické terminologii se jedná o fully-connected layer.



Obrázek 2.3: Echo-state síť

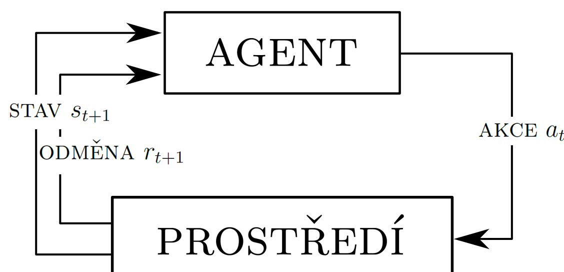
Rekurentní část ESN je fixní a po celou dobu učení ji nijak neupravujeme a neměníme. Naopak výstupní vrstva (znázorněna na obrázku 2.3) je to místo, které se snažíme opravit. Samotné učení výstupní vrstvy může probíhat různými způsoby, například gradientním algoritmem, nebo pomocí evolučního přístupu, kterému se budeme věnovat v kapitole Aplikace modelů a experimenty.

3. Zpětnovazební učení

V této kapitole se podíváme na zpětnovazební učení[1] [21]. Problém zpětnovazebního učení můžeme vnímat jako obecný problém, který lze řešit různými přístupy. Mezi takové přístupy můžeme zařadit adaptivní dynamické programování, Q-učení, hluboké zpětnovazební učení, nebo dokonce i evoluční přístup. O samotném zpětnovazebním učení neříkáme, zda se jedná o učení s učitelem nebo bez učitele. Takovou kategorizaci můžeme používat spíše až u konkrétního způsobu řešení.

3.1 Princip

Základními složkami zpětnovazebného učení je prostředí a agent, který s tímto prostředím interaguje. Agent má vždy k dispozici stav, který se po dobu běhu mění. Označme $s_t \in S$ stav agenta A v čase t, kde S je konečná množina všech možných stavů, ve kterých se může agent nacházet. Počáteční stav agenta budeme značit s_0. Agent na základě svého stavu zvolí akci $a \in A(s_t)$, kterou vykoná. Pomocí $A(s)$ značíme konečnou množinu možných akcí, které může agent vykonat ve stavu s. Tím přejde do stavu s_{t+1} a od prostředí získá odměnu r, která představuje zpětnou vazbu pro agenta (jeho ocenění). Odměňující funkce je funkce $R : S \to \mathbb{R}$. Celý proces se následně opakuje, dokud agent nepřejde do nějakého z cílových stavů, nebo výpočet neukončí prostředí.



Obrázek 3.1: Schéma zpětnovazebního učení

Agent se řídí vlastním algoritmem, kterému říkáme strategie (policy). Strategie je funkce π, která mapuje stavy $s \in S$ na akce $a \in A(s)$. Cílem je najít takovou strategii π, která získá co největší kumulativní odměnu

$$R = r_0 + \sum_{i=1}^{\infty} \gamma^i r_i,$$

kde $0 \leq \gamma \leq 1$ je konstanta zvaná *discount factor*. S volbou konstanty musíme být opatrní, zejména pokud nemáme zaručeno, že se agent dostane do cílového stavu. Aby byl součet dobře definován, musíme v tomto případě volit $\gamma < 1$. Tato konstanta představuje, jak moc se agent bude „ohlížet na budoucnost“ — při hodnotách blízkých nule agent preferuje takové akce, které mají velkou okamžitou odměnu, naopak při hodnotách blízkých jedné se agent ohlíží i na odměny, které může přinést budoucnost.

[1] Také známe pod pojmem reinforcement learning.

Nechť máme agenta ve stavu s_t, do kterého se dostal přes stavy $s_0, \dots, s_{t-1}$ pomocí akcí $a_0, \dots, a_{t-1}$. Označme

$$\mathcal{P}[s_{t+1} = s | s_t, a_t, \dots, s_0, a_0]$$

pravděpodobnost toho, že agent ze stavu s_t přejde akcí a_t do stavu s. Ve zpětnovazebním učení budeme dále předpokládat, že je splněna Markovská podmínka[2], tedy že platí

$$\mathcal{P}[s_{t+1} = s | s_t, a_t, \dots, s_0, a_0] = \mathcal{P}[s_{t+1} = s | s_t, a_t].$$

Stav tedy záleží pouze na předchozím stavu a provedené akci a je nezávislý na historii. Pravděpodobnostem $\mathcal{P}[s'|s,a]$ budeme říkat přechodový model.

Zpětnovazební učení můžeme rozdělit do dvou skupin, a to na pasivní a aktivní učení. V pasivním učení je agentova strategie π předem dána, agent tedy ví, jakou akci má udělat ve stavu s. Nezná však přechodový model ani odměňující funkci. Cílem je v tomto případě vyhodnotit kvalitu dané strategie vyhodnocením očekávaného užitku. Agent se tak učí funkci užitku U^π:

$$U^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right].$$

V aktivním učení neznáme ani konkrétní strategii π, agent se musí sám rozhodovat, jaké akce provede, a strategii se učit.

Jako další se podívejme na samotné stavy. Ty jsou na sobě závislé, konkrétněji, jejich vztah se řídí Bellmanovou rovnicí:

$$U^\pi = R(s) + \gamma \sum_{s' \in S} \mathcal{P}[s'|s, \pi(s)] U^\pi(s').$$

Máme několik způsobů jak řešit hledání strategie π a užitkové funkce U^π. Pro pasivní učení se jedná například o adaptivní dynamické programování nebo temporální diferenci. Pro aktivní učení můžeme využít Q-učení, na které se podíváme podrobněji v další sekci. Obecně můžeme říci, že se tyto metody snaží postupně aktualizovat užitkovou funkci (případně strategii), dokud se jejich hodnoty neustálí.

Přehled algoritmů pro zpětnovazební učení je možné nalézt v knize Reinforcement Learning: An Introduction [22]. Pro obecný přehled o umělé inteligenci (včetně zpětnovazebního učení) doporučujeme také knihu Artificial Intelligence: A Modern Approach [21].

3.2 Q-učení

Q-učení [21] je aktivní učení, agent se tedy sám učí strategii, tedy akce, které provede. Definujme funkci $Q^\pi(s, a)$ jako ohodnocení za provedení akce a ve stavu s, kde π je strategie agenta, zvolená jako:

$$\pi(s) = \arg\max_{a \in A(s)} Q(s, a).$$

[2] Markovská podmínka, která platí pro Markovské rozhodovací procesy.

Q funkce nám poskytuje alternativní způsob zápisu užitkové funkce, neboť platí

$$U^\pi(s) = \max_{a \in A(s)} Q(s, a).$$

Pro Q funkci také platí Bellmanova rovnice (neboť je to jen jiný zápis původní užitkové funkce):

$$Q(s, a) = R(s) + \gamma \sum_{s' \in S} \left(\mathcal{P}[s'|s, a] \cdot \max_{a' \in A(s')} Q(s', a') \right).$$

Nechť máme agenta, který ve stavu s udělal akci a, dostal se do stavu s' a získal odměnu $R(s)$. Agent v tuto chvíli upraví svou Q funkci podle následujícího aktualizačního pravidla:

$$Q^\pi(s, a) \leftarrow Q^\pi(s, a) + \alpha \left[R(s) + \gamma \max_{a' \in A(s')} Q^\pi(s', a') - Q^\pi(s, a) \right],$$

kde α a γ jsou předem zvolené parametry.

Algoritmus 2 Q-učení

1: $Q(s, a) \leftarrow 0 \ (\forall s, a)$
2: $s \leftarrow$ Získání počátečního stavu
3: **repeat**
4: Proveď akci $\text{argmax}_{a \in A(s)} Q(s, a)$
5: $s', r \leftarrow$ Nový stav, Odměna
6: $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a' \in A(s')} Q(s', a') - Q(s, a) \right]$
7: $s \leftarrow s'$
8: **until** není splněno vhodné kritérium konce

3.2.1 Explorace versus exploatace

Doposud jsme ve zpětnovazebním učení vybírali jen ty nejlepší akce, které máme v dané chvíli k dispozici. To ovšem nemusí být to nejefektivnější řešení. Pokud se v průběhu výpočtu dostaneme do nějakého lokálního optima, může být obtížné se z něj dostat ven. Nabízí se otázka, zda bychom v takové situaci neměli vybírat i jiné akce, ne jen ty nejlepší. Pokud budeme vybírat i jiné akce, budeme se pohybovat po větších částech prostoru. Takovému způsobu chování říkáme explorace. Naopak, exploatace znamená, pokud stále vylepšujeme jedno určité řešení. Tomu, jak nastavit hranici mezi explorací a exploatací se říká problém explorace a exploatace.

Ve zpětnovazebním učení můžeme zavést ϵ-hladovou strategii[3], která s pravděpodobností ϵ provede náhodnou akci a s pravděpodobností $(1 - \epsilon)$ provede akci nejlepší. Pravděpodobnost explorace se také může během učení měnit, například můžeme začínat s vysokou pravděpodobností, která bude postupně klesat.

Ještě bychom chtěli zmínit, že problém explorace a exploatace není jen záležitostí zpětnovazebního učení, ale týká se obecně jakéhokoliv prohledávání prostoru.

[3]Známo pod pojmem ϵ-greedy policy.

3.3 Aplikace neuronových sítí ve zpětnovazebním učení

Zpětnovazební učení a způsoby řešení, které jsme si uvedli, mají však své trhliny. Podívejme se na Q-učení. Během něj si agent neustále aktualizuje svou tabulku, Q-funkci. Tato funkce je však pro všechny stavy a pro všechny možné akce v daném stavu. S narůstajícím počtem stavů a počtem možných akcí, bude mít tato tabulka velmi velké rozměry. Připomeňme, že Q-funkce nám pro daný stav a akci říká očekávaný užitek a agent si následně zvolí tu akci, která má tuto hodnotu maximální. To, jak se bude počítat Q-funkci, však můžeme udělat i jinými způsoby, například ji můžeme předpovídat pomocí neuronové sítě.

3.3.1 Hluboké zpětnovazební učení

Hluboké zpětnovazební učení [4] aproximuje Q-funkci pomocí neuronové sítě, které říkáme Q-síť. Zadefinujme si následující značení:

- $Q(s, a)$ je Q-funkce ze standardního Q-učení (to jsme si popsali výše),
- $Q(s, a|\theta)$ je výstup neuronové sítě, kde θ jsou její parametry.

Po neuronové síti chceme, aby dobře aproximovala Q-funkci, tedy aby

$$Q(s, a|\theta) \approx Q(s, a).$$

Na učení takové sítě budeme používat učení s učitelem, proto potřebujeme znát požadované výstupy, což je zároveň klíčová věc pro definování chybové funkce. Pro výpočet požadovaných výstupů[4] využijeme část aktualizačního pravidla Q-učení:

$$R(s) + \gamma \max_{a' \in A(s')} Q(s', a'|\theta).$$

Chybovou funkci pak budeme definovat následovně:

$$L = \left(\underbrace{R(s) + \gamma \max_{a' \in A(s')} Q(s', a'|\theta)}_{\text{požadovaný výstup}} - Q(s, a|\theta) \right)^2 .$$

Nyní můžeme tuto ztrátovou funkci minimalizovat pomocí standardních gradientních metod. Tomuto algoritmu také říkáme DQN[5].

Zpětnovazební učení může být při použití nelineární aproximace pro reprezentaci Q-funkce nestabilní [4, 23]. Jedním z takových případů jsou právě neuronové sítě, tedy Q-sítě. Hlavními důvody jsou korelace mezi vstupními daty a také způsob aktualizace požadovaných výstupů.

[4] V anglické terminologii target.

[5] Deep Q-network.

Korelace mezi vstupními daty

Pro učení neuronové sítě je vhodné mít vstupní data nezávislá, což u zpětnovazebního učení nemáme, neboť stav, který agent dostane, je závislý na stavu předchozím a provedené akci. Tento problém je možné řešit pomocí ukládání vstupů do bufferu[6] [4]. Tento buffer funguje jako fronta a ukládají se do něj čtveřice (s, a, s', r), kde

- s je stav,
- a je akce, která byla provedena ve stavu s,
- s' je stav, do kterého se agent dostal po vykonání akce a,
- r je odměna, kterou agent dostal (po vykonání akce a).

Učení sítě pak probíhá tak, že se náhodně vybírají data z bufferu, která tvoří dávku (batch). Tato dávka je pak předkládána síti.

Aktualizace požadovaných výstupů

Dalším problémem spojeným s nestabilitou učení může být způsob, jak aktualizujeme požadované výstupy. Výpočet požadovaných výstupů probíhá na základě aktuální Q-sítě podle

$$R(s) + \gamma \max_{a' \in A(s')} Q(s', a'|\theta).$$

Jakmile aktualizujeme parametry Q-sítě θ, změní se i požadované výstupy. Pro zlepšení budeme chtít, aby výstupy zůstaly nějakou dobu stejné. Nechť $\overline{\theta}$ jsou parametry sítě, kterou použijeme pouze pro výpočet požadovaných výstupů. Budeme je aktualizovat pouze každých C kroků a chybová funkce pak vypadá následovně (batch vybereme uniformně náhodně z bufferu):

$$L = \sum_{(s,a,s',r) \in \text{batch}} \left(R(s) + \gamma \max_{a' \in A(s')} Q(s', a'|\overline{\theta}) - Q(s, a|\theta) \right)^2 .$$

Regularizace

Regularizace [13] je jakýkoliv způsob, jak zlepšit generalizaci sítě. Ukážeme si dva způsoby regularizace, které jsou také popsány v online knize Neural Networks and Deep Learning [13].

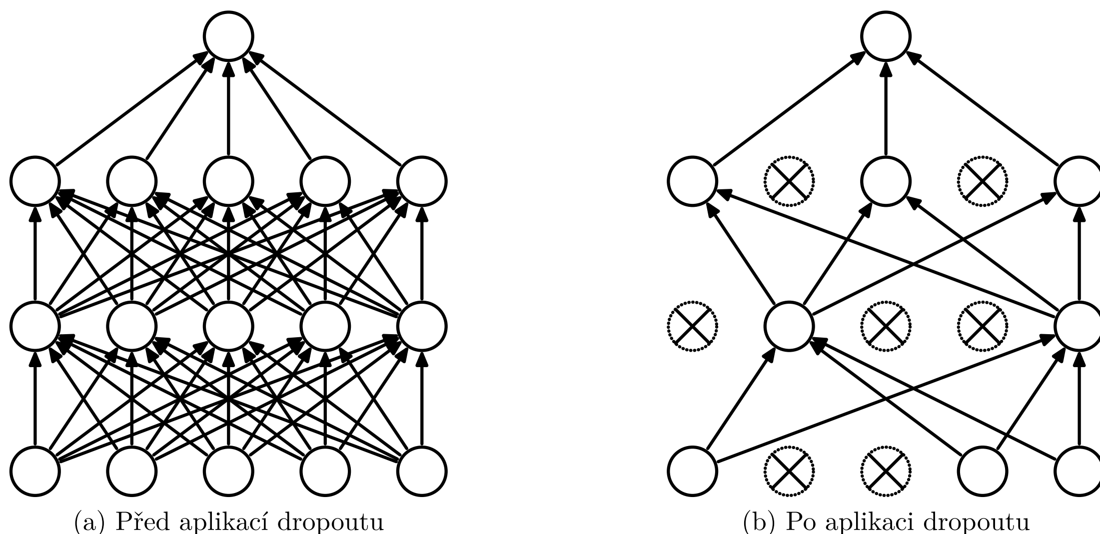
Jedním ze způsobů je zavést do ztrátové funkce regularizační člen $\lambda \sum_w w^2$, kde w značí váhy sítě a λ je parametr. Nová ztrátová funkce je následující:

$$L = \sum_{(s,a,s',r) \in \text{batch}} \left(R(s) + \gamma \max_{a' \in A(s')} Q(s', a'|\overline{\theta}) - Q(s, a|\theta) \right)^2 + \lambda \sum_w w^2.$$

Druhým (a poněkud odlišným) způsobem je dropout [24]. Dropout je metoda vypuštění některých neuronů ze sítě během učení. Zároveň jsou zapomenuty i všechny váhy, které interagují s odstraněnými neurony. V každé vrstvě ponecháme

[6] Známo pod pojmy replay buffer nebo také experience replay.

neurony s pravděpodobností, která je předem dána jako parametr. Výsledná síť pak vypadá jako na obrázku 3.2.



(a) Před aplikací dropoutu (b) Po aplikaci dropoutu

Obrázek 3.2: Ilustrace techniky dropout. Obrázky převzaty ze zdroje [24].

3.4 Spojité akce agentů

Doposud jsme se bavili o zpětnovazebním učení (a následném využití neuronových sítí) pouze pro agenty s diskrétními akcemi. Pokud budeme chtít zpětnovazebním učením učit agenta, jehož akce bude například nastavení hodnoty nějakého efektoru (množství přidání plynu v závodní hře), jehož validní hodnoty jsou z intervalu reálných čísel, nelze použít Q-síť a Q-učení v takové podobě, jako jsme si uvedli výše. To především proto, že hodnoty, které předpovídá Q-učení, jsou kumulativní očekávané užitky, nikoli hodnoty, jaké by měl agent použít pro své akce.

3.4.1 Algoritmus DDPG

Problém se spojitými akcemi řeší algoritmus DDPG[7] [25], který využívá tzv. actor-critic přístup. Tento přístup obecně obsahuje dvě různé funkce. V DDPG algoritmu používáme jejich aproximace pomocí neuronových sítí:

- Actor síť — $\mu(s|\theta^{\mu})$, kde θ^{μ} jsou parametry sítě. Actor síť má na starosti mapování stavů na akce. Výstup této sítě nám říká přímo hodnoty, které nastavíme jednotlivým efektorům agenta,
- Critic síť — $Q(s, a|\theta^{Q})$, kde θ^{Q} jsou parametry sítě. Critic síť má za cíl odhadovat kumulativní očekávané užitky (pomocí Bellmanovy rovnice) tak, jak je tomu u DQN.

[7]Z anglického deep deterministic policy gradient.

Critic síť budeme učit stejným způsobem jako Q-síť, tedy pomocí Bellmanovy rovnice. Minimalizujeme následující ztrátovou funkci:

$$L = \frac{1}{N} \sum_{i}^{N} \left(y_i - Q(s_i, a_i | \theta^Q) \right)^2 ,$$

kde

$$y_i = r_i + \gamma Q(s_{i+1}, \mu(s_{i+1} | \theta^\mu) | \theta^Q).$$

Nechť J značí očekávanou odměnu z počáteční distribuce (z daného počátečního stavu hry). Aktualizace actor sítě probíhá pomocí aplikace řetízkového pravidla[8] na J vzhledem k parametrům actor sítě θ^μ:

$$\begin{aligned} \nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t} \left[\nabla_{\theta^\mu} Q(s, a | \theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)} \right] \overset{\text{chain rule}}{=} \\ &= \mathbb{E}_{s_t} \left[\nabla_a Q(s, a | \theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s | \theta^\mu) \right] {}^9. \end{aligned}$$

Q-sítě měly problém s aktualizací požadovaných výstupů, který řešili neaktualizováním parametrů sítě ihned, ale vždy až po C krocích. Algoritmus DDPG řeší tento problém s nestabilitou pomocí „pomalé“ aktualizace, a to tak, že jsou vytvořeny kopie actor $\mu'(s|\theta^{\mu'})$ a critic $Q'(s|\theta^{Q'})$ sítí, které jsou použity pro výpočet požadovaných výstupů. Tyto kopie jsou aktualizovány na základě parametru $\tau \ll 1$:

$$\begin{aligned} \theta^{\mu'} &\leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}, \\ \theta^{Q'} &\leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}. \end{aligned}$$

Algoritmus DDPG také řeší prohledávání prostoru akcí, tedy exploraci, pomocí přidání náhodného šumu k výstupům actor sítě:

$$\mu'(s) = \mu(s | \theta^\mu) + \mathcal{N},$$

kde $\mathcal{N}$ může být zvoleno různě. Autoři DDPG algoritmu doporučují používat Ornstein-Uhlenbeckův proces [26], který generuje specifický šum. Pro úplnost uvedeme i pseudokód celého algoritmu.

[8] Chain rule.

[9] V reálných implementacích je $\mathbb{E}_{s_t}$ ignorováno a používá se průměr přes daný batch.

Algoritmus 3 DDPG

1: Náhodná inicializace actor sítě $\mu(s|\theta^\mu)$ a critic sítě $Q(s, a|\theta^Q)$
2: Inicializace sítí Q' a μ' pro výpočet výstupů, $\theta^{\mu'} \leftarrow \theta^\mu, \theta^{Q'} \leftarrow \theta^Q$
3: $R \leftarrow \emptyset$ inicializace bufferu
4: **for** episoda $= 1, \ldots, M$ **do**
5: Inicializace náhodného procesu $\mathcal{N}$ pro exploraci prostoru akcí
6: $s_1 \leftarrow$ počáteční stav
7: **for** $t = 1, \ldots, T$ **do**
8: Akce $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ vybraná podle actor sítě (se zašuměním)
9: $s_{t+1}, r_t \leftarrow$ nový stav a odměna po vykonání akce a_t
10: Uložení čtveřice (s_t, a_t, s_{t+1}, r_t) do bufferu R
11: Vyber náhodný batch velikosti N z bufferu R
12: Spočítej požadovaný výstup pro $i = 1, \ldots, N$:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

13: Aktualizuj critic síť, minimalizací chybové funkce:

$$L = \frac{1}{N} \sum_{i}^{N} \Big(y_i - Q(s_i, a_i|\theta^Q) \Big)^2$$

14: Aktualizuj actor síť pomocí:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

15: Aktualizuj parametry pro výpočet výstupů:

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau)\theta^{\mu'}$$

16: **end for**
17: **end for**

4. Evoluce

Genetické a evoluční algoritmy jsou jedním z mnoha způsobů učení modelů ve strojovém učení. Jedním z významných autorů v této oblasti je Holland [27]. Jejich základním principem je aplikovat principy a procesy známé z evoluční biologie, inspirované Darwinovou evoluční teorií. V této kapitole rozebereme algoritmy a varianty evolučních přístupů, které jsme se rozhodli využít.

Evoluční algoritmy (EA) jsou optimalizační metoda, ve které vyvíjíme skupinu jedinců, kterou nazýváme populace. Každý jedinec reprezentuje bod v n-dimenzionálním prostoru fitness funkce, kterou se snažíme optimalizovat. Jedinec je obvykle řetězec čísel, neboli genů. Pokud budeme pracovat s binárními jedinci, říkáme o algoritmu, že je genetický. Algoritmus pracuje v iteracích, ve kterých je postupně přetvářena populace pomocí selekce, křížení a mutace tak, abychom dosáhli co možná nejlepších hodnot fitness. Tato funkce nám říká míru „kvality" jedince.

4.1 Jednoduchý evoluční algoritmus

Jednoduchý evoluční algoritmus (Simple Evolutionary Algorithm) je jednoduchou variantou evolučního algoritmu:

Algoritmus 4 Jednoduchý evoluční algoritmus

```
1: P ← Init
2: Evaluate(P)                          ▷ Vyhodnocení fitness
3: repeat
4:     C ← Select(P)                    ▷ Selekce
5:     C ← Crossover(C)                 ▷ Křížení
6:     C ← Mutate(C)                    ▷ Mutace
7:     P ← Evaluate(C)
8: until done
```

4.1.1 Inicializace

Inicializace populace jedinců je často náhodná, ať už se jedná o binární jedince, nebo jedince reálných čísel. Pro reálná čísla můžeme uvažovat nějaká další omezení a vlastnosti, například generujeme jedince s danou střední hodnotou a rozptylem, tedy generujeme z nějakého rozdělení.

4.1.2 Selekce

Selekce je proces výběru jedinců do další generace. Můžeme ji rozdělit na enviromentální selekci (výběr přeživších jedinců) a na selekci pro křížení (výběr jedinců, kteří projdou křížením). Součástí selekce může být elitismus, což je výběr těch nejlepších jedinců, kteří již nebudou procházet křížením a mutací a rovnou přejdou do další generace.

Kromě účelu dané selekce jedinců můžeme selekce rozdělit do několika typů, a to podle jejího způsobu chování. Základní typy selekcí jsou:

- uniformní selekce — náhodný výběr mezi jedinci,
- turnajová selekce — turnaj n náhodně vybraných jedinců, přičemž vítěz je vybrán jako výsledek selekce,
- ruletová selekce — každý jedinec má šanci být vybrán, pravděpodobnost tohoto výběru p_i je dána velikostí jeho fitness f_i:

$$p_i = \frac{f_i}{\sum_{j \in |P|} f_j}$$

- selekce nejlepších - výběr n jedinců s nejvyšší hodnotou fitness f_i.

4.1.3 Křížení

Křížení je proces kombinace dvou nebo více jedinců do jedinců nových a aplikujeme ho s pravděpodobností p_c. Způsob křížení jedinců se opět může lišit a mezi základní patří:

- uniformní křížení — pro každý „bit“ si zvolíme, ze kterého jedince jej vybereme. Vybíráme většinou se stejnou pravděpodobností,
- bodové křížení — určíme si jeden nebo více bodů přechodu a rozdělíme tak řetězce. Komplementární řetězce spojíme do nových jedinců.

4.1.4 Mutace

Mutace je modifikace jedince, která je aplikována s pravděpodobností p_m. Jedná se o drobnou změnu, například prohození „bitu“ jedince nebo je náhodně změněno číslo v řetězci jedince.

4.2 Evoluční strategie

Optimalizovat reálné funkce však můžeme i jinými algoritmy než je jednoduchý evoluční algoritmus. Jedním z možných řešení jsou evoluční strategie [28, 29]. Evoluční strategie jsou specifické tím, že obsahují jistou vlastní adaptaci[1]. Výpočet je opět iterativní, kdy na populaci aplikujeme selekci a mutaci. Křížení v evolučních strategiích není. Selekce je v případě evolučních strategií deterministická a navíc zde rozhoduje pořadí jedinců a ne skutečná hodnota fitness funkce. Pro selekci v evoluční strategii je zavedena zvláštní notace:

- $(\mu + \lambda)$-ES — μ nejlepších jedinců do nové populace je vybráno z $\mu + \lambda$ jedinců (starých i nových),
- (μ, λ)-ES — μ nejlepších jedinců do nové populace je vybráno z λ nových jedinců,

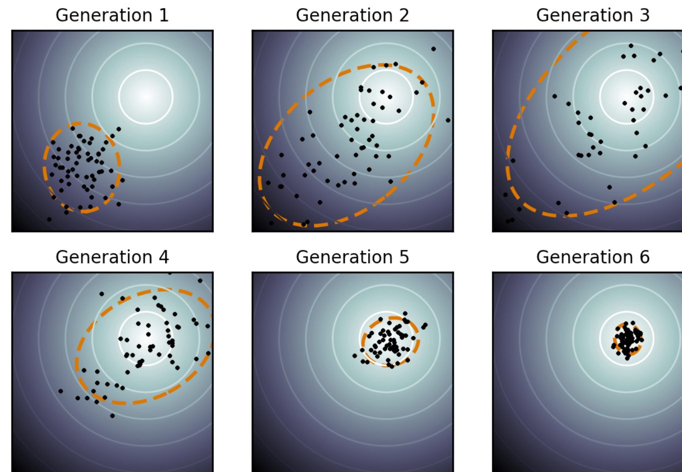
kde λ je velikost populace a μ je počet nově vznikajících potomků.

[1] Self-adaptation.

4.2.1 CMA-ES

CMA-ES [2] je evoluční strategie s adaptivní kovarianční maticí [30]. Tato evoluční strategie se během výpočtů snaží optimalizovat kovarianční matici mezi jednotlivými složkami vektoru jedince.

Každý nový jedinec vznikne tak, že je vygenerován pomocí normálního rozdělení, jehož parametry, tj. střední hodnota a kovarianční matice, jsou adaptovány během učení. Na obrázku 4.1 můžeme vidět příklad běhu CMA-ES na jednoduchém 2-dimenzionálním problému.



Obrázek 4.1: Exemplární běh CMA-ES. Černé tečky značí populaci, oranžová čárkovaná čára představuje distribuci populace a to, jak se vyvíjí v průběhu generací. Zdroj obrázku: Wikipedia[3].

Nyní si zadefinujme pojmy a značení:

- $\sim$ značí rovnost distribucí,
- $\mathcal{N}(0, C^{(g)})$ vícerozměrné normální rozdělení se střední hodnotou nula a kovarianční maticí $C^{(g)}$,
- $x_k^{(g+1)} \in \mathbb{R}^n$ k-tý potomek z generace $g+1$,
- $m^{(g)} \in \mathbb{R}^n$ střední hodnota distribuce v generaci g,
- $\sigma^{(g)} \in \mathbb{R}^+$ směrodatná odchylka v generaci g,
- $C^{(g)} \in \mathbb{R}^{n \times n}$ kovarianční matice v generaci g,
- $\lambda \geq 2$ velikost populace (počet potomků).

Jednotliví jedinci (body v prostoru) jsou pak generováni jako

$$x_k^{(g+1)} \sim m^{(g)} + \sigma^{(g)} \mathcal{N}(0, C^{(g)}).$$

[2] Z anglického covariance matrix adaptation evolution strategy.
[3] `https://en.wikipedia.org/wiki/CMA-ES`

Nová střední hodnota se spočítá pomocí váženého průměru μ nejlepších jedinců z původních $x_1^{(g+1)}, \dots, x_\lambda^{(g+1)}$:

$$m^{(g+1)} = \sum_{i=1}^{\mu} w_i x_{i:\lambda}^{(g+1)}, \tag{4.1}$$

$$\sum_{i=1}^{\mu} w_i = 1 \ \wedge \ w_1 \geq w_2 \geq \dots \geq w_\mu > 0,$$

kde $x_{i:\lambda}^{(g+1)}$ je i-tý nejlepší jedinec v populaci $(g+1)$-ní generace a $w_i \in \mathbb{R}^+$ jsou váhy pro rekombinaci.

Aktualizace kovarianční matice je složitější. Uvedeme si princip pouze na evoluční strategii typu $(1+\lambda)$, tedy $\mu = 1$. Nechť

$$y_{g+1} = \frac{x_{1:\lambda}^{(g+1)} - m^{(g)}}{\sigma^{(g)}}, \tag{4.2}$$

pak aktualizace kovarianční matice, nazývaná *rank-one-update* je následující:

$$C^{(g+1)} = (1-c_1)C^{(g)} + c_1(y_{g+1})(y_{g+1})^T, \tag{4.3}$$

kde c_1 je konstanta. Toto aktualizační pravidlo můžeme zobecnit pomocí tzv. evolučních cest. Evoluční cestu v generaci g značíme $p_c^{(g)} \in \mathbb{R}^n$ a platí pro ni

$$p_c^{(g+1)} = (1-c_c)p_c^{(g)} + \sqrt{c_c(2-c_c)}\frac{m^{(g+1)} - m^{(g)}}{\sigma^{(g)}}. \tag{4.4}$$

$c_c \leq 1$ je konstanta evoluční cesty. Rank-one-update pro kovarianční matici $C^{(g)}$ pomocí evolučních cest je potom

$$C^{(g+1)} = (1-c_1)C^{(g)} + c_1(p_c^{(g+1)})(p_c^{(g+1)})^T. \tag{4.5}$$

Na začátku volíme $p_c^{(0)} = 0$. Pokud $c_c = 1$ (a také $\mu = 1$) platí, že rovnice 4.5 a 4.3 jsou ekvivalentní.

Zdrojem pro tuto sekci byl tutoriál samotného autora evoluční strategie CMA Nikolause Hansena [31]. Uvádíme zde pouze přehledový popis. Pokud čtenáře bude zajímat podrobnější popis CMA-ES, doporučíme již zmíněné zdroje [30, 31].

4.3 Diferenciální evoluce

Dalším z podobných přístupů, jako je evoluční algoritmus, je diferenciální evoluce (DE) [32], která se liší především ve způsobu kombinace jedinců. Nechť P je populace jedinců, na počátku náhodně vygenerovaná. Pak pro každého jedince x náhodně vybereme tři jiné jedince a, b, c a nový jedinec je spočítán jako

$$x' = a + F \cdot (b - c)$$

s pravděpodobností CR, kde F představuje parametr vyjadřující váhu rozdílu a CR pravděpodobnost křížení. Pokud má jedinec x' větší fitness než x, zaujme jeho místo v další generaci. Vždy je alespoň jeden člen jedince x spočítán pomocí ostatních jedinců a, b, c. Obvykle $F \in [0{,}2]$.

Algoritmus 5 Diferenciální evoluce

```
1:  P ⊂ ℝ^n ← náhodná inicializace
2:  repeat
3:      for x ∈ P do
4:          a, b, c ∈ P; a ≠ x ∧ b ≠ x ∧ c ≠ x
5:          j ~ randint({1, 2, ..., n})
6:          for i ∈ 1, 2, ..., n do
7:              p ~ rand(0,1)                  ▷ Uniformě náhodně zvolené číslo p
8:              if (p < CR) or (i == j) then
9:                  x'_i ← a_i + F · (b_i − c_i)
10:             else
11:                 x'_i ← x_i
12:             end if
13:         end for
14:         if f(x') > f(x) then               ▷ Porovnání fitness
15:             x ← x'
16:         end if
17:     end for
18: until dostatečně dobrý výsledek
```

5. Algoritmy pro zvolené hry

Hry, které jsme si zvolili, již nějakou dobu existují a je samozřejmé, že se o vytvoření umělé inteligence pro tyto hry již někdo v minulosti pokusil. Základní linií je však náhodný hráč.

Hra 2048

Jedním z přístupů, jak řešit tuto úlohu, je řešení pomocí konvolučních sítí. Tento způsob byl prezentován v článku Gui a kol. [33]. Autoři dosahují úspěšnosti (winrate) zhruba 7 %. O poznání lepších výsledků dosahují Szubert a Jaskowski [34], a to více než 97 %. Toto řešení využívá tzv. n-tuple networks reprezentující určité vzory na herní ploše.

Dalším řešením je využití technik, které nespoléhají na neuronové sítě (a učící se algoritmy), ale využívají „klasické“ prohledávání. Jedním z takových způsobů je expectimax optimalizace, kterou používá Robert Xiao [35] a ve které je rovněž dosahováno úspěšnosti vyšší než 97 %.

Náhodný hráč pro tuto hru dosáhne zhruba skóre 1100, kde nejčastějšími dlaždicemi jsou hodnoty 64 a 128.

Mario

Hra Mario má mnoho podobných verzí, všechny ale mají stejný princip a chovají se podobně. Hluboké zpětnovazební učení s využitím konvolučních sítí použil Alexander Jung [36], konkrétně na verzi Super Mario World. Toto řešení je schopné dokončovat rozumně obtížné úrovně.

Dále se také touto problematikou zabývali Karakovskiy a Togelius [37, 38]. Tito autoři uvádí implementaci vhodnou pro programování dalších AI a také ji využívali pro několik kompetitivních soutěží v předchozích letech. Tato implementace je založena na verzi Super Mario Bros., kterou využíváme i my.

Ještě další možností je využití neuroevolučního algoritmu NEAT [39], který také poskytuje řešení se schopností dokončovat úrovně rozumné obtížnosti[1].

Náhodný hráč v případě Maria pochopitelně nedokáže dokončit žádnou úroveň. Veškerou herní dobu stráví na začátku úrovně a náhodný pohyb jej nedokáže dostat směrem doprava.

TORCS

Závodní simulátor TORCS byl přímo vytvořen především pro závody AI agentů. Hra nabízí několik různých módů, například závodní mód, ve kterém se v předchozích letech konala soutěž, pořádaná autory hry. Nás zajímá především tréningový mód, který budeme používat pro vlastní AI. Stejným způsobem přistoupil k TORCS Lau [40], který docílil rovnoměrné jízdy bez nehod. Navíc však využívá toho, že během učení (a explorační fáze) používá brzdu jen v 10 % času.

[1]K tomuto projektu se nepodařilo dohledat žádnou patřičnou publikaci. Výsledek je pouze uveden na YouTube kanálu SethBling `https://www.youtube.com/watch?v=qv6UVOQ0F44` [cit. 2017-07-14] a uvádíme jej pouze jako zajímavý projekt, zapadající do kontextu této práce.

Náhodný agent řídící vozidlo je (stejně jako v případě Maria) pochopitelně velmi špatný a téměř se nedokáže ani rozjet, neboť se neustále náhodně střídá přidání plynu a brzdy.

Alhambra

Karetní hru Alhambra budeme srovnávat s předchozí implementací, která byla uvedena v práci [10]. Tato umělá inteligence je založena na pravidlovém systému, jehož pravidla se používají pro rozhodování, jaký tah zvolit. Klíčové jsou váhy pro tato pravidla, které se učily pomocí jednoduchého evolučního algoritmu. Tato umělá inteligence je schopna hrát zhruba na úrovni lidského začátečníka. V současné práci budeme používat stejnou implementaci (a používat stejný pravidlový systém), ale dané váhy budeme učit jinak.

Pod náhodným hráčem pro Alhambru v tomto případě rozumíme náhodně vygenerované váhy, což není ekvivalentní s tím, že by agent udělal skutečně náhodný tah. Nicméně platí, že takový „náhodný“ agent je o poznání horší, než agent s naučenými váhami.

6. Aplikace modelů a experimenty

V této části práce rozebereme to, jak jsme aplikovali výše vysvětlené modely na vybrané hry. Popíšeme, jak se snažíme zachovat obecnost umělé inteligence a také to, co nás v tom omezuje.

6.1 Obecnost

Snažíme se vytvořit umělou inteligenci, která bude hrát vybrané hry. Náš požadavek je, aby tato umělá inteligence dosáhla jisté obecnosti, tedy byla aplikovatelná na hry, které nemusí být vůbec podobné.

Naše umělá inteligence (dále jen model) je oddělená od samotných her. Každá hra však musí předem definovat, jaký očekává počet vstupů a výstupů modelu. Každý model očekává vstupy a výstupy jako reálná čísla. Počet vstupů modelu je velikost stavu dané hry a výstupy modelu definují akci, kterou má hra udělat. Zde připomeňme, že model nemá nijak dáno, jak má interpretovat dané vstupy či výstupy a (téměř) jediné, na co se může spolehnout je, že se jedná o reálné vektory čísel. Ještě jednu z věcí je však třeba rozhodnout dopředu, a to zda se jedná o hru s diskrétními akcemi, nebo se spojitými, a podle toho zvolíme typ modelu.

Příklad: hra 2048 očekává od modelu vektor čtyř čísel, ze kterého si vybere argument maxima a tu akci provede. Akce ve hře 2048 jsou diskrétní, tzn. zajímá nás pouze index akce, kterou máme udělat. Pro takový případ můžeme použít model, který se bude snažit odpovědět hodnotami ve „správném“ pořadí, kde nezáleží na konkrétních hodnotách, které odpoví. Naopak závodní simulátor TORCS očekává jako výstupy spojité akce, nevybírá tedy maximum z výstupu modelu, ale použije jeho hodnoty přímo na nastavení například hodnoty plynu nebo brzdy. V takovém případě musíme použít model, který je pro takové chování přizpůsoben.

S takovými minimálními předpoklady na vstup modelu nemůžeme používat typy modelů, které již v základu předpokládají jistou vazbu mezi položkami vstupu. Může se jednat například o předpoklad vícedimenzionálních dat a následné použití konvolučních sítí [33].

Škálování samotných hodnot, které předávají hry modelům jako vstup, probíhá na straně her. Typicky chceme, abychom měli hodnoty například z intervalu $[0, 1]$, což nelze zaručit vždy. Každá hra se o „rozumné“ škálování stará sama. Pokud hodnoty mohou teoreticky nabývat neomezených hodnot, používáme škálování pomocí logaritmu.

Některé hry mohou být rozděleny na několik fází, ve kterých by bylo vhodné přistupovat k aktuálnímu stavu hry různě. Proto zavedeme další zobecnění naší umělé inteligence, a to tak, že daný model pro řešení bude ve skutečnosti více modelů, každý pro jinou fázi hry. Každý model pak může mít různý počet výstupů (protože každá fáze může nabízet různý počet akcí). Tento počet fází a jednotlivé

počty očekávaných výstupů musí být definovány dopředu.

Skóre a odměna je počítána na straně her, to znamená, že jejich definice a způsob výpočtu je nezávislý na modelu.

6.2 Reprezentace her

V této sekci se podíváme jak každou hru reprezentujeme (jak vypadají stavy) a jaké výstupy očekávají od modelu. Také rozebereme, jak jsme definovali odměny pro zpětnovazební přístupy.

6.2.1 Hra 2048

Hra 2048 je z námi vybraných her nejjednodušší v ohledu reprezentace stavu a akcí. Hru tvoří čtvercové pole o rozměrech 4×4 a akce je jeden ze čtyř směrů pohybu. Způsob reprezentace stavu hry, který se přímo nabízí, je přímé kódování, tj. stav hrací plochy linearizujeme do vektoru po řádcích. V tomto případě jsou hodnoty rovny hodnotám hrací plochy. Jiným kódováním může být vytvoření 16-tice pro každou z 16 pozic na hrací ploše (a následné linearizování do vektoru). Každá z těchto 16-tic bude tvořena pomocí nul, kde na pozici i je jednička právě tehdy, když je na odpovídající pozici na hrací ploše hodnota 2^i. Toto kódování je inspirováno tím, které použili Gui a kol.[33], je však náročnější na výpočet.

Akce budeme kódovat do čtyřsložkového vektoru, jak by se dalo předpokládat. Akce této hry jsou diskrétní, zajímá nás pouze index akce, která má největší hodnotu, nikoliv její hodnota.

Skóre hry je v tomto případě triviální a jedná se o skutečnou hodnotu skóre definovanou samotnou hrou 2048. Odměnu za daný tah definujeme jako přírůstek skóre, tedy součet všech bloků, které se spojily. Tato hra může mít neplatné tahy, neboť je možné, že se po daném tahu nezmění hrací plocha. Pro takový tah je také potřeba definovat odměnu, v tomto případě se nabízí dvě možnosti -1, nebo 0. Takovou situaci je potřeba řešit i technicky, neboť u některých modelů by se mohlo stát, že výpočet bude v takovém případě cyklický — můžeme například udělat jiný náhodný tah, který je k dispozici, abychom se dostali z takového cyklu. Ve zpětnovazebním učením je třeba však mít na paměti, že máme replay-buffer, do kterého ukládáme čtveřice (stav, akce, nový stav, odměna) a pokud tedy provedeme jinou akci než byla odpověď modelu, je třeba správně uložit danou čtveřici (tj. stav a nový stav je stejný; akce je ta, kterou chtěl odpovědět model; odměna adekvátní k tomu, že se hra nezměnila).

6.2.2 Mario

Mario obsahuje mnoho informací o hře a je třeba rozhodnout, které chceme použít jako informace o současném stavu. Stav hry Mario reprezentujeme vektorem dlouhým 384 položek, ve kterém je obsaženo:

- Jedna hodnota pro každou pozici na viditelné části prostředí (obrazovky). Toto prostředí je rozděleno na 19×19 pozic.
- Hodnoty popisující samotnou postavu Maria: pozice vzhledem k mřížce hry nebo absolutní pozice vzhledem k obrazovce hry (přesná pozice postavy

Maria v pixelech), status (počet životů), rychlost, výška, kolik se vyskytuje objektů pod postavou Maria.

Akcí, které může provádět Mario, je celkem pět. Jedná se o běh doleva, běh doprava, skok, střelba a sprint (rychlejší běh). První variantou, jak může odpovídat model hře, je vektor délky pět, kdy následně bude vybrána akce s nejvyšší hodnotou (neboť akce jsou diskrétní, stejně jako ve hře 2048). Druhou variantou, o něco komplikovanější, je vytvoření podmnožin daných pěti tahů. Díky tomu můžeme v jednu chvíli použít například akce běhu doprava a skoku. Je třeba však vytvořit podmnožiny, které budou smysluplné (podmnožina obsahující běh doleva a zároveň běh doprava nebude mít žádný efekt). Výstup modelu pak bude vektor stejné délky. Tato varianta je však komplikovanější a rozhodli jsme se ji nepoužívat. Důvodem pro toto rozhodnutí je přístup Occamovy břitvy a nechceme model zbytečně komplikovat.

Skóre hry Mário definujeme jako poměr překonané vzdálenosti dané úrovně vůči její délce (1 pokud byla úroveň celá dokončena, 0 pokud se hráč vůbec nepohnul). Odměna je v tomto případně složitější. Nechť ΔP je kladná změna pozice hráče směrem doprava, ΔK je změna počtu zničení nepřátel a ΔS je změna počtu životů hráče (tj. nula pokud hráč neztratil žádný život). Pak odměnu definujeme jako

$$r = \alpha \cdot \Delta P + \beta \cdot \Delta K - \gamma \cdot \Delta S,$$

kde α, β, γ jsou konstanty. Vyzkoušíme dva typy odměny (různé konstanty), označíme je A a B:

$$A : \alpha = 1 \wedge \beta = 10 \wedge \gamma = 100$$

$$B : \alpha = 1 \wedge \beta = 10 \wedge \gamma = 1$$

6.2.3 TORCS

Závodní simulátor TORCS obsahuje spoustu senzorů, ze kterých můžeme dostat data. Některé z nich jsou pro nás nerelevantní, protože se chceme soustředit především na prostou jízdu po trati s jediným vozidlem. Přehled všech senzorů a jejich popis je k dispozici v manuálu k samotnému simulátoru [41]. Vybereme si pouze takové senzory, které nám budou více užitečné, neboť je zbytečné uvažovat informace o ostatních vozidlech, pokud žádná jiná na trati nejsou. Stejným způsobem přistoupil k tomuto simulátoru i Ben Lau [40], který rovněž zvolil pouze užitečné senzory. Popis jednotlivých senzorů je v tabulce 6.1. Celkem se jedná o 29 hodnot, které dáme jako vstup modelu.

Jméno	Popis
`angle`	Úhel mezi směrem vozidla a osou tratě.
`track`	Vektor 19 senzorů, které udávají vzdálenost mezi okrajem tratě a vozidlem (do maximální vzdálenosti 200 metrů).
`trackPos`	Vzdálenost mezi vozidlem a osou tratě.
`wheelSpinVel`	Vektor udávající rychlost otáčení kol vozidla.
`rpm`	Množství otáček motoru vozidla za minutu.
`speedX`	Rychlost vozidla vzhledem k podélné ose vozidla.
`speedY`	Rychlost vozidla vzhledem k příčné ose vozidla.
`speedZ`	Rychlost vozidla vzhledem k vertikální ose vozidla.

Tabulka 6.1: Využívané senzory v simulátoru TORCS

Simulátor TORCS očekává pouze tři akce od našeho modelu. Jedná se o směr jízdy, přidání plynu a brzdy. Směr jízdy je určován pomocí hodnoty z intervalu $[-1, 1]$. Plyn a brzda jsou hodnoty z intervalu $[0, 1]$. Tato hra má spojité akce, tedy vždy nastavujeme hodnoty efektorů přímo podle toho, co vrátí model. Nelze tedy použít takový model, který předpokládá výběr pouze akce s maximální hodnotou.

Skóre hráče můžeme vyhodnocovat několika způsoby. Jedním z nich je definování skóre jako vzdálenost, kterou vozidlo ujede (bez poškození). Druhá varianta je tuto vzdálenost normovat časem, za kterou je tato vzdálenost ujeta. Tento způsob (který vlastně odpovídá průměrné rychlosti) není však zcela vypovídající, protože hráč, který na začátku pojede co nejrychleji dokud nenarazí, může mít průměrnou rychlost tohoto úseku větší, než hráč, který pomalu obkrouží celou trať. U první varianty zase těžko posoudíme, který z agentů je lepší, pokud urazí stejnou vzdálenost. Přesto se budeme orientovat na první variantu, neboť nás spíše zajímá dokončení tratě.

Odměna v případě TORCS bude reflektovat základní myšlenku jízdy: držet se uprostřed trati a zároveň jet co nejrychleji. Lau [40] uvádí, že vhodná definice odměny je v této situaci

$$A : r = v\cos(\varphi) - v\sin(\varphi) - v|\texttt{trackPos}|,$$

kde v je rychlost vozidla, φ úhel mezi osou vozidla a tratě a `trackPos` je vzdálenost vozidla od středu tratě. Tuto odměnu si označíme jako A. Dále můžeme zkoušet jisté parametrizované varianty, kde například dáme důraz na první složku (označíme jako B):

$$B : r = v\cos(\varphi) - \frac{1}{2}v\sin(\varphi) - \frac{1}{2}v|\texttt{trackPos}|.$$

6.2.4 Alhambra

Alhambra je karetní hra, která se však poměrně liší od ostatních svým způsobem zpracování. Implementace hry, kterou využíváme, byla vytvořena jako součást vlastní bakalářské práce [10]. Předtím než popíšeme, jaké bude mít model vstupy a výstupy, musíme vysvětlit, jak funguje logika rozhodování v této hře. Tato sekce tedy bude z určité části pouze rekapitulací zdroje [10].

Původní umělá inteligence, která byla pro tuto hru vytvořena, obsahuje strom rozhodování, jak vytvořit herní tah. Takový herní tah se skládá z mnoha částí a

sub-rozhodování (lokální rozhodování). Diagram rozhodování uvádíme v příloze B.1. Pro každé lokální rozhodování existují možnosti, mezi kterými je potřeba vybírat. Uvážíme všechny takové možnosti. Pro každé z lokálních rozhodování máme kritéria, což jsou jevy, u kterých určujeme, zda pro danou možnost v lokálním rozhodování nastaly či nikoliv. Každé kritérium má svou váhu — typicky hodnota z intervalu $(0, 1)$. Výběr možnosti určujeme pomocí kvality, což je součet vah všech kritérií lokálního rozhodování, které pro danou možnost nastaly. Vybíráme tu možnost, která má nejvyšší hodnotu kvality. Původní řešení vytvořilo z vah pro jednotlivá kritéria vektor, který byl následně použit jako jedinec pro jednoduchý evoluční algoritmus. Tímto způsobem byla vytvořena jednoduchá umělá inteligence pro tuto hru.

Pro náš obecný přístup používáme původní implementaci, tedy rozhodování, které probíhá ve hře, je stejné jako v původním zpracování. To, čím se budeme nyní lišit, bude způsob učení jednotlivých vah pro lokální rozhodování. Pro každé lokální rozhodování máme zvláštní model, který pro něj dává odpověď. Takových modelů zde máme dvanáct.

Vstup pro model musí být popis stavu hry, tak jak je tomu i u ostatních her. Tento stav budeme kódovat do vektoru dimenze 708, kde jsou obsaženy tyto položky:

- počet hráčů dané hry,
- pro každou unikátní kartu ve hře máme jejich počet pro daného hráče,
- pro každou budovu ve hře máme příznak, zda ji daný hráč vlastní nebo ne,
- pozice všech postavených budov daného hráče,
- jednotlivé karty a budovy, které jsou v současné chvíli na trhu.

Každá karta je popsána typem a hodnotou. Každá budova je popsána typem, hodnotou a několika příznaky (jednotlivé stěny budovy a tím, zda je na odkládací ploše). Všechny tyto hodnoty jsou škálovány do intervalu $[0, 1]$, neboť se jedná o omezené příznaky. Všimněme si, že popis stavu je tvořen z pohledu karet (hry) a ne z pohledu konkrétního hráče — to potřebujeme proto, abychom měli pokaždé stejný počet vstupů. Máme tedy popis stavu hry, který dostane příslušný model.

Akce jsou v každé fázi hry různé, proto mají jednotlivé modely i různé velikosti. Akcí v případě Alhambry rozumíme něco jiného než u ostatní her, a to konkrétní nastavení vah pro kritéria lokálních rozhodování. Od modelu tedy chceme, aby nám vrátil každý tah seznam těchto vah, které budou následně použity pro výběr tahu. Z pohledu obecného modelu se tak jedná o spojité akce, podobně jako je tomu u simulátoru TORCS.

Alhambra je jedinou hrou, u které využíváme většího množství fází. Pro evoluční přístupy používáme pro každou fázi jeden model, což znamená, že máme n neuronových sítí pro n fází. Evoluce je v tomto případě jednoduchá, neboť jedinec obsahuje parametry všech těchto modelů. Nevýhodou je v tomto případě velmi velké množství parametrů a velikost takového jedince. Proto je žádoucí používat sítě menší velikosti. Co se týče zpětnovazebního učení pro Alhambru, i tam je třeba vyřešit fakt většího množství fází. Kromě velikosti je zde však problém se způsobem učení takové sítě — máme-li n neuronových sítí, potřebujeme učit každou zvlášť (protože je učíme gradientním algoritmem). Nelze zaručit, že se agent

bude nacházet ve všech fázích hry rovnoměrně, to znamená, že by se mohlo stát, že některou ze sítí přeučíme a jinou nebudeme učit vůbec. Tomuto problému se můžeme vyhnout tak, že budeme mít jednu síť, která bude mít počet výstupů roven součtu výstupů pro všechny fáze. Pro danou fázi si pak vybereme adekvátní část výstupu.

Alhambra je hra více hráčů (dva až šest), je potřeba tedy mít i nějaké protihráče, se kterými se náš model utká. Jako vhodná možnost se přímo nabízí (již zmíněná) původní AI vytvořená pro tuto hru. Zvolíme celkem tři hráče pro každý zápas — model bude hrát proti dvěma původním AI. Skóre hry volíme jako skóre, kterého bylo dosaženo naším modelem. Nechť ΔC je změna počtu karet hráče a ΔB je změna počtu postavených budov. Odměnu definujeme jako

$$r = \alpha \cdot \Delta C + \beta \cdot \Delta B,$$

kde α a β jsou opět konstanty. Rovněž si zde označíme odměny pro zvolené konstanty:

$$A : \alpha = 1 \wedge \beta = 10$$
$$B : \alpha = 1 \wedge \beta = 100.$$

Navíc vyzkoušíme odměnu třetí — nechť C_{new} značí počet nově odebraných karet (tj. buď ΔC nebo nula) a ΔW nechť značí změnu délky nejdelší stěny okolo Alhambry. Definujme odměnu jako

$$r = C_{new} + 10\Delta B + 10\Delta W.$$

Tuto odměnu budeme označovat C.

6.3 Zvolené modely

Pojďme se nyní podívat na jednotlivé modely, které používáme pro učení zvolených her. Tyto modely můžeme rozdělit do dvou skupin podle jejich způsobu řešení, a to na evoluční přístupy a přístupy zpětnovazebného učení. Pro všechny modely platí, že máme n vstupů a očekáváme m výstupů. Každý typ modelu si vlastní výpočetní jádro řeší svým vlastním způsobem. Vždy, když je potřeba odehrát tah, hra získá od daného modelu odpověď (předložení stavu hry modelu a spočítání výstupu).

6.3.1 Evoluční přístupy

Pokud chceme využít evolučních přístupů, je třeba napřed určit, co bude tvořit jedince a populaci, která se bude vyvíjet. Rozhodli jsme se využít neuronových sítí různých typů:

- model s MLP sítí,
- model s Echo-state sítí.

Parametry těchto sítí následně použijeme jako jedince, který bude vyvíjen evoluční technikou. Všechny evoluční techniky, které zde uvedeme, mají jako jedince seznam vah neuronové sítě. Všimněme si, že při použití evoluce nezáleží na tom, zda hra má diskrétní či spojité akce. Odpovědí modelu po předložení stavu je v tomto případě dopředný průchod danou neuronovou sítí. Každou z těchto sítí zkusíme vyvíjet různými algoritmy:

Jednoduchý evoluční algoritmus

Tuto základní variantu evolučního přístupu používáme tak, jak jsme ji popsali v kapitole 4. Na počátku náhodně vygenerujeme populaci. Každý jedinec reprezentující parametry dané neuronové sítě. Fitness jedinců počítáme v každé generaci znovu; výpočet fitness je průměrné skóre několika her. Tabulka 6.2 definuje všechny parametry, které zde používáme.

Jméno	Popis
`pop_size`	Velikost populace.
`ngen`	Počet generací, které se spočítají.
`mut`	Typ mutace a dané pravděpodobnosti.
`cross`	Pravděpodobnosti křížení; jedna hodnota pro křížení (ano či ne) a druhá hodnota udávající pravděpodobnost křížení jednotlivých „bitů“ jedince.
`game_batch`	Počet her, který je vždy odehrán při výpočtu fitness jednoho jedince.
`elite`	Počet nejlepších jedinců, kteří automaticky přežijí do další generace. V každé generaci jej vybíráme znovu. Pokud chceme ukládat ukládat jedince napříč generacemi, použijeme parametr `hof_size` (pouze jiná varianta elitismu).
`selection`	Typ selekce.

Tabulka 6.2: Parametry využívané jednoduchým evolučním algoritmem

Evoluční strategie CMA

Evoluční strategie CMA aplikovaná na výpočet parametrů neuronové sítě má nevýhodu, kterou je složitost problému. Kovarianční matice v této evoluční strategii jsou příliš velké a pro rozumně rychlý běh musí být neuronová síť poměrně malá, abychom měli malé množství parametrů. V tomto případě se nabízí jako výhodnější použít Echo-state síť před klasickou MLP sítí. Obecně však můžeme říci, že evoluční strategie by mohly být vhodnou alternativou pro učení neuronových sítí. To v současné době potvrdili Salimans a kol. [42], kteří evoluční strategii srovnávali se zpětnovazebním učením.

Základní parametry využívané evoluční strategií CMA jsou stejné jako v předchozím případě (počet generací, velikost populace, ...). Nový parametr `sigma` určuje výchozí směrodatnou odchylku distribuce.

Diferenciální evoluce

Diferenciální evoluce má základní parametry rovněž stejné, navíc máme parametry `CR` a `F`. Jejich funkčnost jsme popsali v sekci 4.3, kde jsme popsali i pseudokód algoritmu. Diferenciální evoluci jsme také zvolili proto, protože v porovnání s CMA evoluční strategií není tak náročná na výpočet a umožňuje tak vyvíjet větší sítě.

6.3.2 Přístupy zpětnovazebního učení

Kromě evolučních algoritmů používáme také klasičtější přístupy zpětnovazebného učení jako je Q-učení, resp. hluboké zpětnovazební učení.

Hluboké zpětnovazební učení a Q-síť

Aplikujeme DQN tak, jak jsme jej popsali v sekci 3.3.1. Tento model je aplikovatelný pouze na hry, které předpokládají diskrétní akce (tedy výběr jedné akce z n možných). Také zde používáme adaptivní (klesající) ϵ-hladovou exploraci. Přehled parametrů pro hluboké zpětnovazební učení je následující:

Jméno	Popis
`batch_size`	Velikost dávky využívané v učení z bufferu akcí.
`init_exp`	Počáteční hodnota explorace.
`final_exp`	Koncová hodnota explorace.
`anneal_steps`	Počet kroků učení, po které bude hodnota explorace klesat od `init_exp` po `final_exp`.
`replay_buffer_size`	Velikost bufferu pro ukládání čtveřic (stav, akce, následující akce, odměna).
`store_every_replay`	Udává, jak často se budou ukládat hodnoty do bufferu (jedna znamená všechny).
`discount_factor`	Hodnota γ z Q-učení.
`target_update_freq`	Udává, jak často se bude aktualizovat síť pro výpočet výstupů (target network).
`reg_param`	Konstanta pro velikost regularizace.
`dropout`	Hodnota pro dropout (pravděpodobnost ponechání neuronu).

Tabulka 6.3: Parametry využívané algoritmem DQN

DDPG

Na hry se spojitými akcemi (n akcí a ke každé potřebujeme získat nějakou hodnotu, jak danou akci nastavit) aplikujeme algoritmus DDPG, který jsme popsali v sekci 3.4.1. Parametry tohoto algoritmu jsou podobné jako u DQN, navíc zde máme další neuronovou síť.

6.4 Zmínka o implementaci

Popsané algoritmy a modely byly naimplementovány v jazyce Python s využitím mnoha knihoven. Pro výpočty spojené s evolučními technikami je použita knihovna DEAP [43]. Výpočty, které jsou spojeny se zpětnovazebním učením, jsou implementovány pomocí TensorFlow [44]. Propojení AI a her je nezávislé na jejich implementaci, mohou tedy využívat jiných programovacích jazyků, pouze musí být splněny podmínky obecného rozhraní pro komunikaci. Například, hra Mario je v programovacím jazyce Java, Alhambra v C#. Technické podrobnosti ohledně implementace a dalších knihoven jsou k dispozici v příloze B.

Pro výpočty jsme využívali současných počítačů s i7-6700 @ 3.40GHz, na kterých byly pouštěny především evoluční výpočty. Na různým instancích jsme pouštěli různá nastavení evoluce pro urychlení. Pro učení neuronových sítí v hlubokém zpětnovazebním učení v TensorFlow jsme využívali pro urychlení grafickou kartu NVIDIA GTX 1070.

6.5 Experimenty

Na vybraných hrách jsme provedli mnoho experimentů a zde si uvedeme jejich výsledky. Ke každé hře uvádíme přehledové nejlepší výsledky. Kompletní seznam všech výsledků se všemi nastaveními parametrů je možné najít v příloze A. Každý běh má uloženy informace o nastavení a vygenerovaný graf průběhu učení. Tyto soubory jsou k dispozici na přiloženém DVD (příloha C). Každý experiment (například evoluční běh) byl prováděn kvůli velké časové náročnosti vždy jen jedenkrát. Je ale žádoucí, abychom předložili více reprezentativní výsledky, a proto ty nejlepší experimenty několikrát zopakujeme. Pro každou hru uvedeme tyto výsledky.

Dále budeme chtít, aby experimenty byly schopné doběhnout „v rozumném čase“, což pro nás bude zhruba jeden den. Vzhledem k faktu, že experimenty byly prováděny na různých počítačích, s různými systémy, je doba běhu spíše orientační. Některým experimentům jsme také zkoušeli poskytnout delší dobu na výpočet, abychom zjistili, zda existuje prostor pro zlepšení, nebo zda se po určité době přestane například evoluce zlepšovat. Právě z důvodu dlouho trvajících běhů experimentů jsme se rozhodli nedělat například grid-search nad parametry, neboť množství kombinací pro parametry je příliš velké.

6.5.1 Parametry

Nastavování konkrétních parametrů pro jakékoliv experimenty je poměrně obtížné, neboť je daných parametrů velké množství. Proto jsme se snažili pro každou hodnotu parametru volit na začátku vhodnou výchozí hodnotu, kterou jsme následně experimentálně měnili. Taková vhodná výchozí hodnota může být doporučována autory algoritmu apod. Příkladem je parametr `F` v diferenciální evoluci, který se obvykle nastavuje na hodnotu z intervalu $[0, 2]$. Experimentování s parametry jsme prováděli na začátku pomocí krátkých běhů, které zde neuvádíme. Tyto běhy sloužily pouze na zlepšení odhadu parametrů, abychom měli nějaké výchozí hodnoty, od kterých bychom se mohli odrazit. Všechny ostatní (dlouhodobé experimenty) zde samozřejmě uvádíme (příloha A). Tento postup vylepšování a zjišťování hodnot parametrů způsobuje však to, že parametry pro různé hry nejsou zcela totožné a také se ukazuje, že pro každou hru je vhodné jiné nastavení nebo i jiný model.

Některé parametry nastavujeme často na stejnou hodnotu pro většinu experimentů, například velikost populace a velikost elity. Tyto parametry mají spíše menší vliv na výsledek než parametry jiné. Ve všech experimentech, které spadají pod evoluční algoritmy, využíváme uniformní křížení i mutaci (skládající se ze dvou pravděpodobností, jedna pro rozhodnutí o daném jedinci, druhá o rozhodnutí pro jednotlivé bity). Hlavním parametrem, který zásadně ovlivňuje výsledek,

je velikost neuronové sítě, kterou učíme. To je také parametr, s kterým nejvíce experimentujeme a zkoušíme různé počty neuronů i počty vrstev.

Zpětnovazební přístupy jsou v tomto směru obdobné. I tam máme parametry, které mají spíše menší vliv na výsledek, například velikost *replay-bufferu*. To, co je zde zásadní, tak jako v předchozím případě, je velikost použité sítě a také typ odměny (její definice) pro danou hru zároveň se zpětnovazebním parametrem *discount factor*.

6.5.2 Testování

Testování naučených modelů probíhá většinou odehráním určitého počtu her, kde naměříme průměrné skóre (například pro hru 2048 pustíme 1000 náhodných her). V případě TORCS za test považujeme ujetou vzdálenost na testovací trati (která je odlišná od trénovací).

Během evolučních experimentů si ukládáme jedince z elity z poslední generace a také z generace, ve které bylo dosaženo nejvyšší průměrné fitness. V případě hlubokého zpětnovazebního učení (ať už DQN nebo DDPG) je třeba také ukládat dosavadně nejlepší jedince. Protože učení probíhá po jednotlivých hrách (nemáme generace, ve kterých bychom mohli určit kvalitu jedinců, ale máme vždy jen jeden výsledek právě dokončené hry) je třeba po několika odehraných epizodách udělat test dosavadního modelu. Opět si zde ukládáme ten nejlepší model i aktuální poslední.

6.5.3 Hra 2048

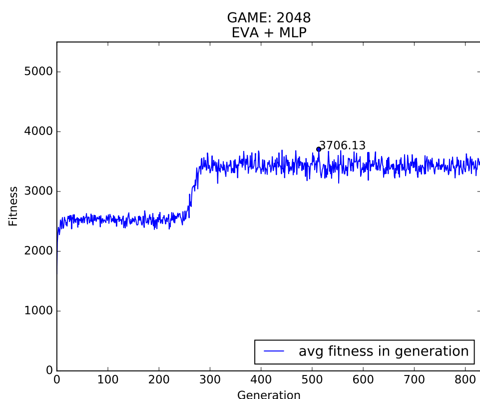
Tato hra je poměrně malá co se týče velikosti stavu a množství informací, které tato hra obsahuje. Rovněž má malý větvící faktor. Proto bychom očekávali, že bude jednoduché naučit nějaký z našich modelů tuto hru hrát. Ukázalo se však, to tak jednoduché není, pokud skutečně nemáme žádnou dodatečnou informaci (náš model je obecný).

Ukazuje se, že hluboké zpětnovazební učení se nedokáže přizpůsobit této hře a překvapivě nedává tak dobré výsledky jako bychom čekali. Nejlepšího výsledku zde bylo dosaženo pomocí evoluční strategie CMA aplikované na MLP síť. V tomto případě musí být síť poměrně malá, neboť CMA je velmi náročná na výpočet, pokud má jedinec velkou dimenzi (což odpovídá počtu parametrů sítě, kterou učíme). Tabulka 6.4 poskytuje přehled výsledků (testů) pro různé modely.

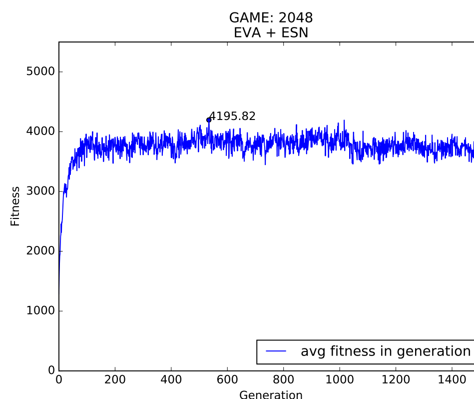
Hra 2048	
`model`	`výsledek`
EA + MLP	3536.15
EA + ESN	3949.04
ES + MLP	**4393.43**
ES + ESN	3980.03
DE + MLP	3718.93
DE + ESN	3721.52
DQN	2928.69

Tabulka 6.4: Hra 2048 – přehled výsledků (průměrné skóre 1000 her)

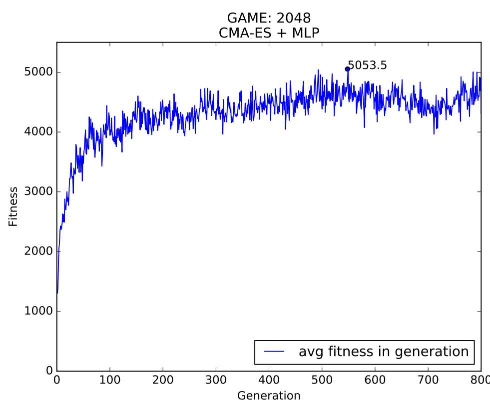
Nyní si ukažme jednotlivé grafy učení těchto modelů (všechny experimenty běžely netriviální a podobnou dobu, počet vykonaných generací a episod tedy závisí na velikosti dané sítě):



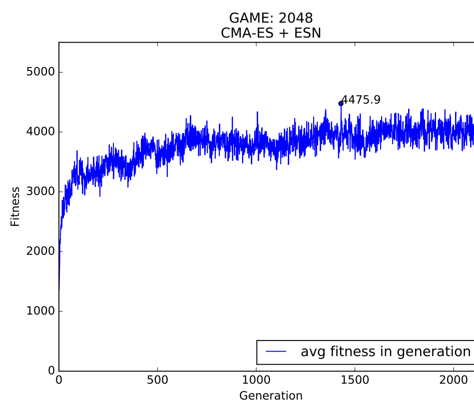
(a) EA + MLP



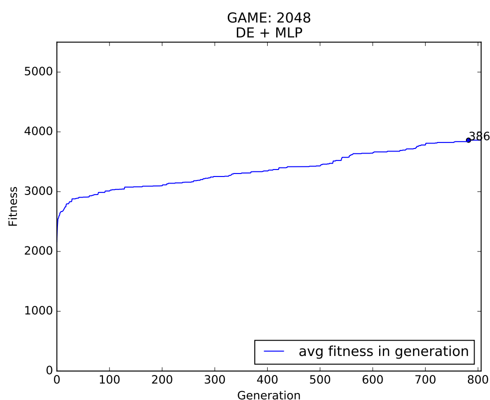
(b) EA + ESN



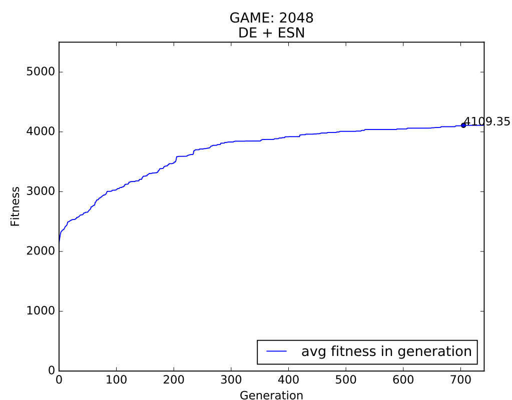
(c) ES + MLP



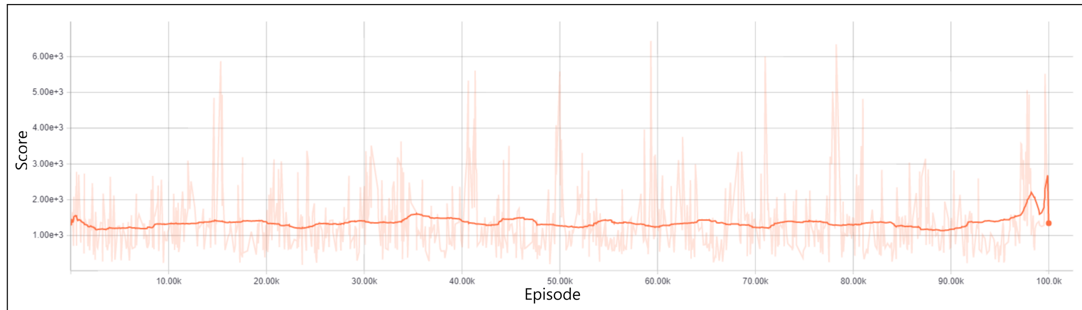
(d) ES + ESN



(e) DE + MLP



(f) DE + ESN



(g) DQN (TensorBoard smoothing 0.7)

Obrázek 6.1: Hra 2048 – průběh učení pro jednotlivé modely

Z grafů můžeme dále vidět, že diferenciální evoluce konverguje pomaleji než ostatní evoluční přístupy, ale zase poměrně stabilně. Oproti tomu evoluční strategie se na začátku rychle zlepší a pak zpomalí. Jak jsme již zmínili, nejlepšího výsledku (ne jen z učení, ale i testovacího) dosáhla v tomto případě kombinace ES a MLP (v grafu 6.1 část C). Tento výpočet jsme tedy několikrát zopakovali a uvádíme minimum, průměr a maximum z těchto opakování:

Hra 2048 – opakování			
`model`	`min`	`avg`	`max`
ES + MLP	4132.31	4400.20	4530.27

Tabulka 6.5: Hra 2048 – opakování nejlepšího výsledku (5x)

Opakování výpočtu pro daný model jsme prováděli pouze pro nejlepší výsledek, a to kvůli časové náročnosti na výpočet. Můžeme vidět, že i nejhorší běh z opakování je lepší než všechny ostatní modely. Ostatní modely také běžely vícekrát, když uvážíme rozdílná nastavení parametrů, a i tyto výpočty dopadly hůře než vybraný nejlepší model (což je také důvod, proč jsme jej vybrali jako nejlepší). Doporučujeme na tomto místě nahlédnout do již zmiňované přílohy A, kde jsou všechny výsledky.

Podle uvedených výsledků se ukazuje, že hra 2048 je nejspíše velice závislá na informaci o okolí dané pozice a následná linearizace do vektoru je tedy nejspíše silné zobecnění. Největší dlaždice, které bylo dosaženo, je tak pouze 1024. Navíc, tato hra má poměrně malý větvící faktor a dává smysl zde používat i zcela jiné techniky, jako jsou například Monte Carlo metody a prohledávání prostoru.

Poznámka

Základní jednoduchou techniku inspirovanou obecným přístupem Monte Carlo jsme si také pro zajímavost vyzkoušeli. Princip je následující: pokud chceme odehrát tah, pro všechny čtyři tahy zkusíme odehrát určitý počet náhodných her (začne se daným tahem, a pak vždy náhodně do konce). Který z výsledků bude mít nejvyšší průměrné skóre je vítězný tah a ten se skutečně provede. Celý proces se následně opakuje.

Jak můžeme z popisu vidět, tento způsob řešení vždy probíhá stejně rychle, jedná se totiž o zcela odlišný přístup, kde nemáme fázi učení, ale doba hraní každé

hry je velmi dlouhá. Čím větší počet her počítáme v jednom tahu, tím je pochopitelně doba výpočtu celé hry delší, což se negativně projeví v porovnání s našimi modely (učení modelu probíhá také déle, ale použití již naučeného modelu na výpočet hry vyžaduje značně menší čas).

Vyzkoušeli jsme tuto metodu s různými počty náhodně dohraných her v každém tahu (výsledek testu je průměrné skóre 1000 her). Pro 10 náhodně dohraných her v každém tahu dostáváme průměrné skóre 13018.75, v některých případech dosáhneme dlaždice 2048, celkový čas tohoto testu je však cca 100 minut, oproti několika málo minutám dopředného průchodu naučeným modelem. Pro 100 her je to skóre dokonce 26707.34 a v několika (málo) případech bylo dosaženo i dlaždice 4096, nicméně doba běhu tohoto testu byla více než 29 hodin.

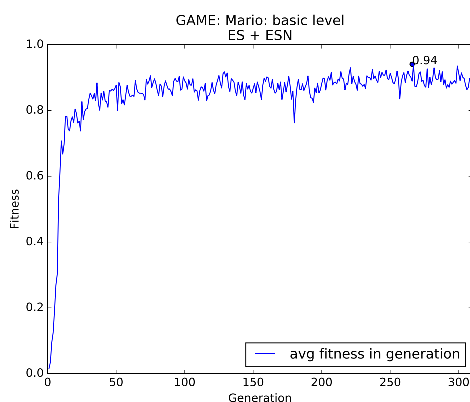
6.5.4 Mario

Hru Mario jsme zkoušeli ve dvou různých obtížnostech. Pro základní úroveň se ukázala jako nejlepší kombinace diferenciální evoluce a ESN. Pro obtížnou úroveň dosáhla nejlepších výsledků také ESN, ale v kombinaci s evoluční strategií.

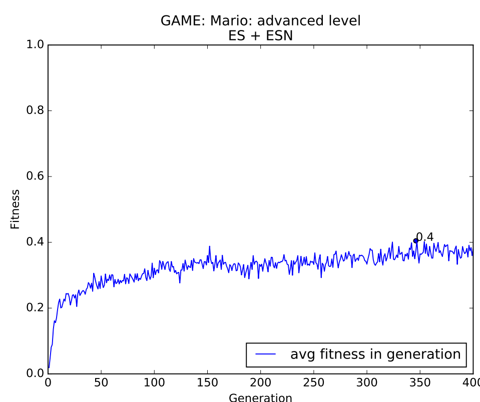
Mario		
model	výsledek	
	základní	obtížný
EA + MLP	58.9 %	6.0 %
EA + ESN	84.8 %	31.2 %
ES + MLP	80.9 %	31.8 %
ES + ESN	82.5 %	**32.1 %**
DE + MLP	82.1 %	27.3 %
DE + ESN	**93.8 %**	28.1 %
DQN	36.0 %	13.6 %

Tabulka 6.6: Mario – přehled výsledků (průměrné skóre 100 her)

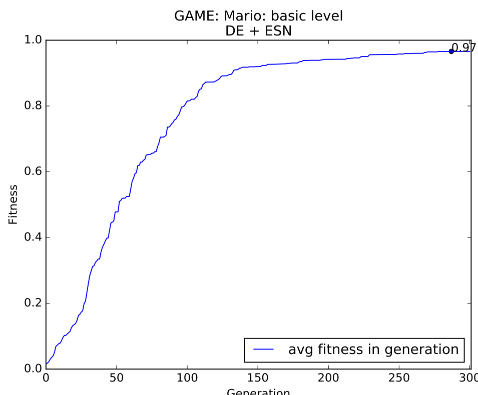
Také zde si uveďme některé grafy s průběhem učení. Všechny grafy zde neuvádíme z důvodu jejich množství, vybereme tak několik nejlepších. Pro každý experiment je graf vygenerován a uveden v příloze C.



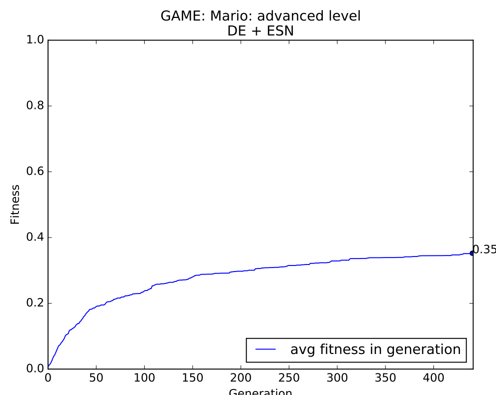
(a) ES + ESN (základní)



(b) ES + ESN (obtížný)



(c) DE + ESN (základní)



(d) DE + ESN (obtížný)

Obrázek 6.2: Mario – průběh učení pro různé úrovně hry a modely

Opět přikládáme opakování nejlepších výsledků:

Mario – opakování				
`model`	`úroveň`	`min`	`avg`	`max`
DE + ESN	základní	84.32 %	88.63 %	93.83 %
ES + ESN	obtížná	29.40 %	31.62 %	34.03 %

Tabulka 6.7: Mario – opakování nejlepšího výsledku (5x)

Opakování výpočtu jsme i zde prováděli jen pro nejlepší model (pro každou z obtížností), ze stejného důvodu jako v předchozím případě. Nejhorší běh pro základní úroveň se zde dostává na srovnatelnou úroveň s modelem EA + ESN, nicméně platí, že i tak je lepší než modely zbývající. Opakování výsledku pro variantu s těžší úrovní již neposkytuje tak výrazný rozdíl a dostává se pod úroveň některých zbývajících modelů, nicméně faktický rozdíl ve hratelnosti není žádný, neboť výsledky těchto modelů jsou téměř stejné. Opět zde odkažme na přílohu A obsahující všechny výpočty.

Hra Mario potvrzuje, že se evoluční strategie na počátku výpočtu dokáží dostat rychle do optimální části prostoru, oproti diferenciální evoluci, která je pomalejší. Zpětnovazební učení závisí na definici odměny. Pro experimenty s definicí odměny A tj. „trest“ pro agenta v podobě příliš velkého záporného skóre (za kolizi s nepřítelem) ukazuje, že se agent spíše naučí zůstat na začátku a nepohybovat se doprava. Proto jsme se rozhodli vyzkoušet odměnu B. Výsledky však ukazují, že ani takto se nedokáže model přiblížit evolučním přístupům. Typy těchto odměn (respektive nastavení konstant) jsme si popsali v sekci 6.2.2.

6.5.5 TORCS

Simulátor TORCS je velmi zajímavou částí této práce. Náš nejlepší model je zde zpětnovazební učení, které se dokáže naučit jízdu okolo trati. Stejně tak jako píše Lau [40], i zde je problém se zabrzděním vozidla, což se projevuje jako problém pro velmi ostré zatáčky, které jsou za dlouhým rovným úsekem, nebo také „rychlé esíčko“. Příklad takové zatáčky můžeme vidět na obrázku 6.3.

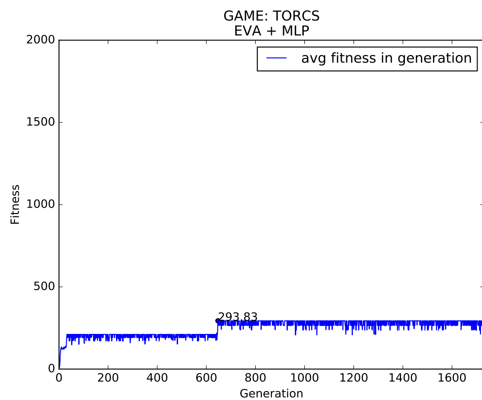
Obrázek 6.3: TORCS – ostrá zatáčka

Výsledek zpětnovazebního učení (algoritmu DDPG) je nejlepší ze všech použitých technik. Jakýkoliv evoluční přístup v kombinaci s MLP sítí je v případě TORCS velmi náchylný na lokální optima. Jak již bylo uvedeno, skóre agenta (tedy fitness) je celková ujetá vzdálenost. Vhodný lokální extrém, do kterého evoluce spadne, je pak co možná největší přidání plynu s přímou jízdou a následný konec vyjetím z tratě. Vhodnější pro tento případ je použití ESN sítě, která si díky rekurentní části může poradit s některými zatáčkami, což potvrzují i experimenty. Stručné srovnání výsledků pro TORCS uvádíme v tabulce 6.8.

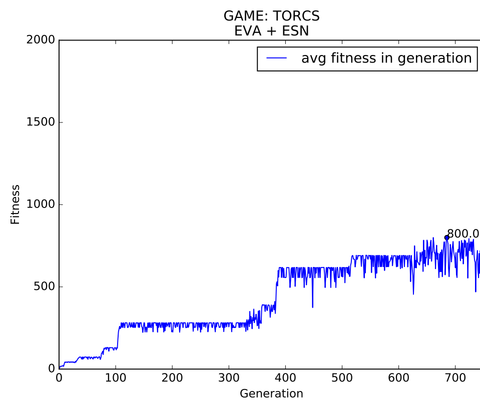
TORCS	
`model`	`výsledek`
EA + MLP	81m
EA + ESN	1030m
ES + MLP	101m
ES + ESN	25m
DE + MLP	902m
DE + ESN	2180m
DDPG	**2692m**

Tabulka 6.8: TORCS – přehled výsledků (testovací trať)

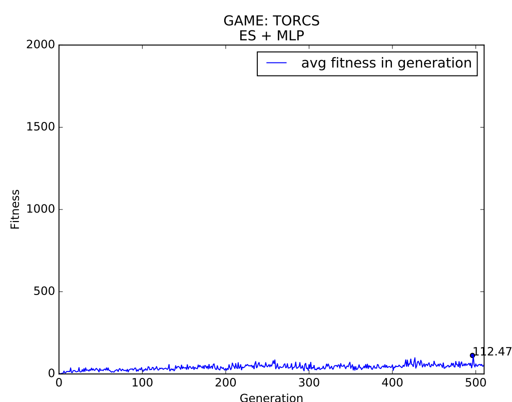
Jak můžeme vidět, rozdíl mezi ESN a MLP sítí je v případě TORCS poměrně zásadní. Opět uveďme grafy učení pro jednotlivé modely:



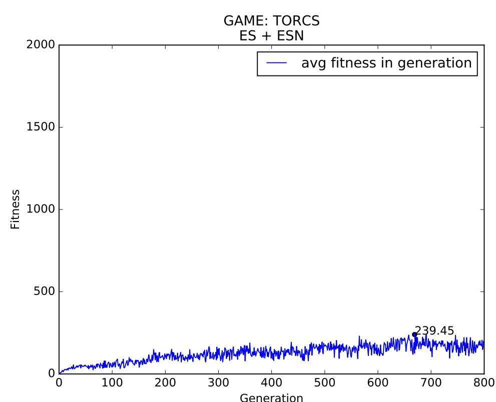
(a) EA + MLP



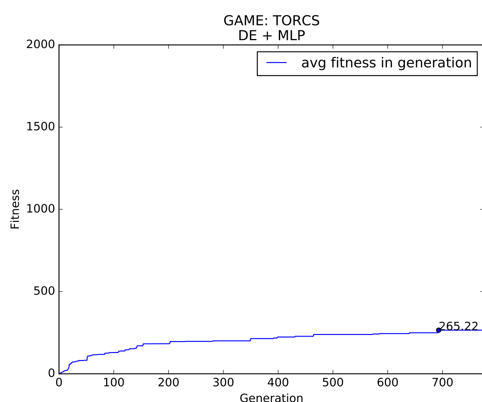
(b) EA + ESN



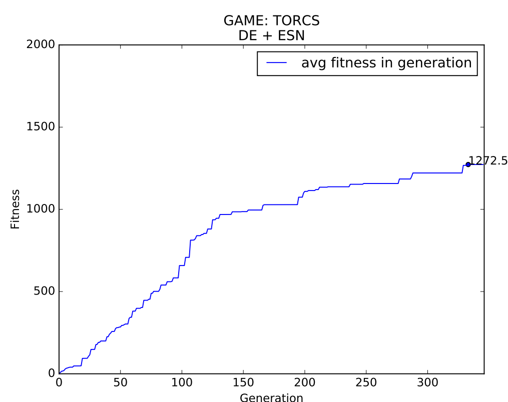
(c) ES + MLP



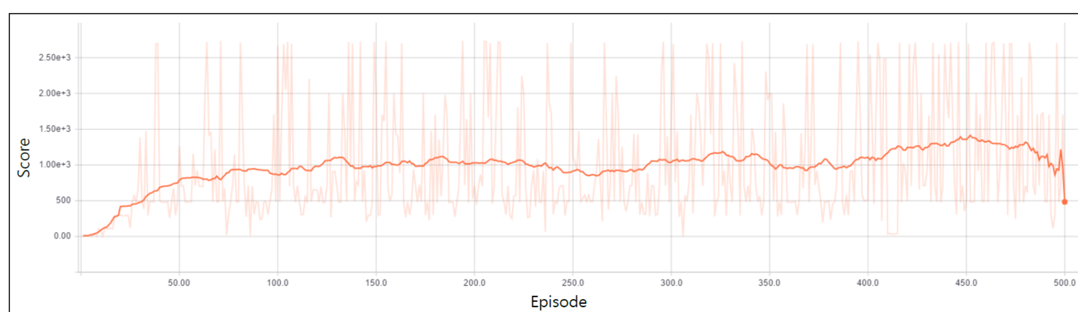
(d) ES + ESN



(e) DE + MLP



(f) DE + ESN



(g) DDPG (TensorBoard smoothing 0.7)

Obrázek 6.4: TORCS – průběh učení pro jednotlivé modely

Nesmí chybět výsledek pro opakování nejlepšího modelu, kterým je v tomto případě DDPG. Po prozkoumání nejlepšího výsledku pro TORCS však vidíme, že rozptyl je poměrně velký (ve srovnání s opakováním nejlepších modelů u ostatních her). Může za to především „nepodařený“ běh s ujetou vzdáleností pouze 439.78 m. Rozhodli jsme se tedy v tomto případě opakovat i druhý nejlepší model, a to diferenciální evoluci v kombinaci s ESN:

TORCS – opakování			
model	min	avg	max
DDPG	439.78m	1763.10m	2499.69m
DE + ESN	1929.95m	2084.89m	2337.71m

Tabulka 6.9: TORCS – opakování nejlepších výsledků (5x)

Z výsledků vidíme, že rozptyl pro diferenciální evoluci a ESN není tak velký, jako u algoritmu DDPG. Oba tyto způsoby dokážou řídit vozidlo, nicméně pokud si spustíme hru s vizuálním prostředím a podíváme se, jak dané vozidlo jede, zjistíme, že algoritmus DDPG je mírně „trhavý“. To se projevuje tak, že při odchýlení vozidla od středu trati je ihned provedena akce pro zatočení směrem k ose, což je nejspíše důsledek definice odměny. Pokud se podíváme na ESN a DE, zjistíme, že jedinec jede velmi pomalu, nicméně dokáže překonat značnou vzdálenost oproti evoluci obyčejné MLP sítě.

6.5.6 Alhambra

Hru srovnáváme s již dříve hotovou jednoduchou umělou inteligencí, které evolučním algoritmem učila váhy pro pravidla. V našem případě učíme neuronovou síť, která udává váhy pro tatáž pravidla. Ukazuje se, že takto předpovězené váhy pro pravidla dosahují lepších výsledků než původní AI, a to jak z pohledu průměrně dosaženého skóre, tak ze samotného počtu vítězství. Nejlepším modelem je zde MLP síť trénovaná evolučním algoritmem.

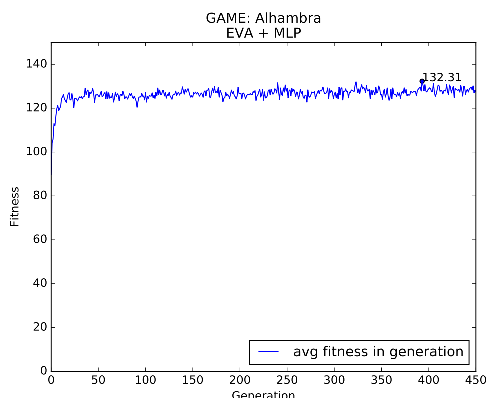
Vzhledem k většímu počtu fází v této hře není kombinace evoluční strategie CMA na MLP síť aplikovatelná kvůli příliš velké dimenzi prostoru. Lze však použít ESN, a to poměrně bez problému.

Alhambra	
model	výsledek
EA + MLP	**129.80** (116.02; 114.08)
EA + ESN	122.90 (120.61; 113.39)
ES + ESN	122.74 (114.73; 116.97)
DE + MLP	122.70 (117.76; 119.76)
DE + ESN	122.78 (117.05; 116.31)
DDPG	108.64 (124.81; 124.79)

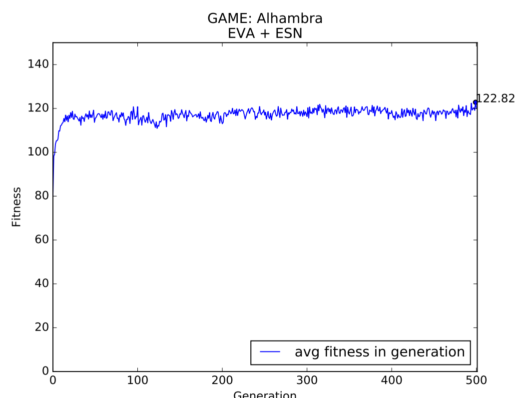
Tabulka 6.10: Alhambra – přehled výsledků (průměrné skóre 100 her). Výsledek je ve formátu `<1> (<2>; <3>)`, kde `<1>` značí výsledek daného modelu a `<2>`, `<3>` značí skóre protihráčů původní umělé inteligence.

I zde vybereme několik grafů učení (6.5) a tabulku s výsledky (6.11) opakovaného modelu s nejlepšími výsledky.

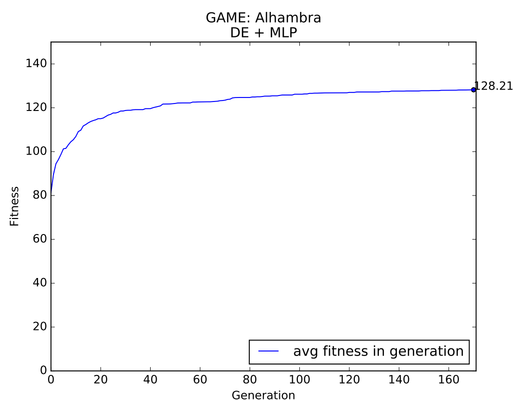
V grafu 6.5, části (a-b), můžeme vidět, že evoluce dosáhne určité fitness a následně se zlepšuje velmi málo a pomalu. To může být dáno také tím, že ve hře Alhambra existují protihráči, kteří již dokáží nějak netriviálně hrát (protože výsledek protihráčů přímo ovlivňuje výsledek našeho modelu). Náš model dokáže porážet původní umělou inteligenci (proti které se učil hrát) jak z pohledu průměrného skóre, tak z pohledu počtu vítězství. Procentuální úspěšnost hráčů ve hře je 55 % – 20 % – 25 % (pořadí stejné jako v tabulce 6.10).



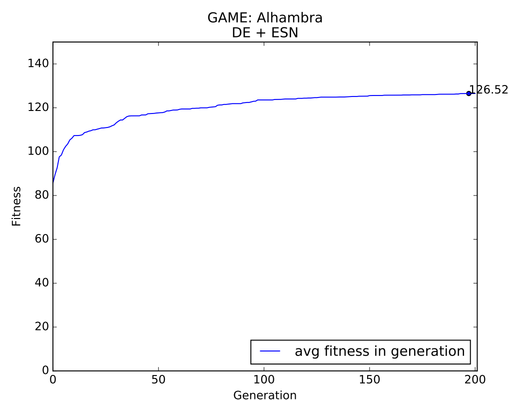
(a) EA + MLP



(b) EA + ESN



(c) DE + MLP



(d) DE + ESN

Obrázek 6.5: Alhambra – průběh učení pro jednotlivé modely

Alhambra – opakování			
`model`	`min`	`avg`	`max`
EA + MLP	128.40	130.05	133.65

Tabulka 6.11: Alhambra – opakování nejlepšího výsledku (5x)

Výsledky opakování nejlepšího modelu pro hru Alhambra dopadly ve všech případech lépe než ostatní modely a především tento model dokáže porazit původní umělou inteligenci pro tuto hru.

6.5.7 Vyhodnocení a poznámky

Po prozkoumání výsledků vidíme, že vhodný model velice závisí na nastavení a na hře, pro kterou je trénován. Najít jedny parametry, které by fungovaly pro všechny hry daného modelu, je velice obtížné.

Pro zpětnovazební učení DQN a DDPG je klíčová odměna, kterou agent dostává během učení. To, jak tuto odměnu budeme definovat, je zásadní pro úspěšnost modelu. Rovněž ve výsledcích můžeme vidět, že je parametr *discount factor* zpětnovazebního učení velice klíčový. Ukazuje se, že je vhodné tento parametr nastavovat na hodnotu blízkou hodnotě jedna.

Evoluční techniky a jejich výsledky ukazují, že tyto přístupy mohou porážet hluboké zpětnovazební učení. Je také rozdíl, který typ neuronové sítě trénujeme. Ukazuje se, že použití ESN má smysl, a to zejména v případech, kde si evoluce MLP sítě neporadí. Hezkým příkladem je v této situaci TORCS, kde je rozdíl mezi ESN a MLP zásadní. Další již zmiňovanou výhodou ESN je fakt, že počet parametrů (čili dimenze prostoru) je daleko menší než při použití MLP sítě. Dále se také ukazuje, že u některých her obecně platí, že evoluční strategie nebo evoluční algoritmus ze začátku konverguje rychleji než diferenciální evoluce.

Je žádoucí, aby experimenty byly opakovatelné. V našem případě tomu tak je, neboť stačí mít pouze zdrojový kód (a dané hry). Data, která používáme, se generují přímo z her a není tedy třeba mít přístup k datasetům, které by mohly být například privátní. Všechny výsledky experimentů v detailní formě jsou uvedeny v příloze A.

Protože jsme pro experimenty s hlubokým zpětnovazebním učením využívali grafickou kartu, nebyl problém používat i větší sítě, než které jsme používali při evolučním učení. Ukázalo se však, že lepších výsledků dosahují sítě spíše menší.

Závěr

Diplomová práce poskytuje zpracování mnoha modelů, které jsme aplikovali na několik vybraných her. Bylo provedeno poměrně velké množství experimentů, kde každý experiment měl poměrně netriviální dobu běhu.

Ukazuje se, že jedno nastavení umělé inteligence (parametrů) nebo i výběr jednoho modelu není zcela jednoduché, a pro různé hry dávají modely různé výsledky. Přesto je každý model možné trénovat na každé z použitých her a je v tomto ohledu obecný, protože nepředpokládá žádné specifické vlastnosti o dané hře (až na velikost stavu). Také jsme zjistili, že mezi různými modely mohou být i zásadní rozdíly ve výkonnosti v některých hrách, jako je tomu například u ESN a MLP sítí ve hře TORCS.

Závěrem bychom chtěli dát doporučení, jaký model vlastně používat. Vždy však záleží na úloze, nicméně podle našich výsledků můžeme říci, že kombinace evolučního přístupu a ESN sítě je solidní volbou. Pro hry typu TORCS dále stojí za vyzkoušení zpětnovazební přístup.

Z výsledků je také patrné, že ne vždy je vhodné používat složité techniky, jako jsou neuronové sítě. Za úvahu vždy stojí nejprve vyzkoušet jednoduchý model, který může dávat poměrně dobrá řešení, a až později se uchýlit k modelům komplikovanějším. I když pro konkrétní hry mohou existovat lepší specifické modely, myslíme, že práce poskytuje velmi zajímavé srovnání různých technik a přístupů, aplikovaných na více her zároveň, což bylo hlavním cílem této práce.

Budoucnost

Na projektu by se v budoucnu samozřejmě dalo dále pracovat. Především množina kombinací parametrů pro nastavení modelů a her je skoro nevyčerpatelná a je tak možné provádět desítky dalších experimentů. Také je možné přidávat další typy modelů, například použití i jiné evoluční strategie vedle CMA-ES.

Co se týče implementace, je samozřejmostí, že i na ní by se dalo v budoucnu pracovat a zlepšovat ji. Příklad takového zlepšení by mohla být implementace dalších parametrů na straně AI tak, aby se nemusela odměna nebo nastavení kódování pro hru 2048 provádět uvnitř dané hry.

Seznam použité literatury

[1] Xiong, W., Droppo, J., Huang, X., Seide, F., Seltzer, M., Stolcke, A., Yu, D. a Zweig, G. (2016). The microsoft 2016 conversational speech recognition system. arXiv preprint arXiv:1609.03528v2.

[2] Krizhevsky, A., Sutskever, I. a Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. a Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602v1.

[4] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. a Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, **518**(7540), 529–533. doi: 10.1038/nature14236.

[5] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. a Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, **529**(7587), 484–489. doi: 10.1038/nature16961.

[6] Cirulli, G. (2014). 2048. [online]. URL `https://github.com/gabrielecirulli/2048`. [cit. 2017-03-13].

[7] Nintendo (1985). Super mario bros.

[8] Wymann, B., Espié, E., Guionneau, C., Dimitrakakis, C., Coulom, R. a Sumner, A. (2014). TORCS, The Open Racing Car Simulator. [online]. URL `http://www.torcs.org`. [cit. 2017-03-13].

[9] Henn, D. (2003). *Alhambra*. Corfix. ISBN 0-9740913-7-5.

[10] Klůj, J. (2015). Alhambra. Bakalářská práce, Univerzita Karlova v Praze. Matematicko-fyzikální fakulta.

[11] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, **65**(6), 386–408.

[12] Rumelhart, D. E., Hinton, G. E. a Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, **323**(6088), 533–536. doi: 10.1038/323533a0.

[13] Nielsen, M. A. (2015). *Neural Networks and Deep Learning.* Determination Press. [online]. URL `http://neuralnetworksanddeeplearning.com`. [cit. 2017-03-20].

[14] Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747v1.

[15] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks*, **12**(1), 145–151. doi: 10.1016/s0893-6080(98)00116-6.

[16] Duchi, J. C., Hazan, E. a Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, **12**, 2121–2159.

[17] Tijmen, T., Hinton, G., Srivastava, N. a Swersky, K. (2012). RMSProp: Divide the gradient by a running average of its recent magnitude. Coursera: Lecture 6e. [online]. URL `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. [cit. 2017-03-21].

[18] Kingma, D. P. a Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980v9.

[19] Medsker, L. R. a Jain, L. C. (1999). *Recurrent Neural Networks: Design and Applications.* International Series on Computational Intelligence. CRC Press. ISBN 978-0849371813.

[20] Jaeger, H. (2001). The "echo state" approach to analysing and training recurrent neural networks. *GMD Report 148, German National Research Center for Information Technology.*

[21] Russell, S. a Norvig, P. (2010). *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3rd edition. ISBN 978-0-13-604259-4.

[22] Sutton, R. a Barto, A. (1998). *Reinforcement Learning: An Introduction.* MIT Press. ISBN 978-026-2193-986.

[23] Tsitsiklis, J. a Roy, B. V. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Trans. on Automatic Control*, **42**(5), 674–690.

[24] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I. a Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, **15**(1), 1929–1958.

[25] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. a Wierstra, D. (2016). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971v5.

[26] Uhlenbeck, G. E. a Ornstein, L. S. (1930). On the theory of the brownian motion. *Physical Review*, **36**(5), 823–841. doi: 10.1103/physrev.36.823.

[27] Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems*. MIT University Press Group Ltd. ISBN 9780262581110.

[28] Rechenberg, I. (1971). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Dizertační práce, Technical University of Berlin.

[29] Schwefel, H.-P. (1974). *Numerische Optimierung von Computer-Modellen*. Dizertační práce, Technical University of Berlin.

[30] Hansen, N. a Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, **9**(2), 159–195. doi: 10.1162/106365601750190398.

[31] Hansen, N. (2016). The cma evolution strategy: A tutorial. arXiv preprint arXiv:1604.00772v1.

[32] Storn, R. a Price, K. (1995). Differential evolution—a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical report, International Computer Science Institute, Berkeley. CA, 1995, Tech. Rep. TR-95–012.

[33] Gui, H., Wei, T., Huang, C.-B. a Wu, I.-C. (2015). An early attempt at applying deep reinforcement learning to the game 2048. [online]. URL `http://docplayer.net/20912615-An-early-attempt-at-applying-deep-reinforcement-learning-to-the-game-2048.html`. [cit. 2017-04-15].

[34] Szubert, M. a Jaskowski, W. (2014). Temporal difference learning of n-tuple networks for the game 2048. In *2014 IEEE Conference on Computational Intelligence and Games*. Institute of Electrical and Electronics Engineers (IEEE). doi: 10.1109/cig.2014.6932907.

[35] Xiao, R. (2014). Expectimax optimization for 2048 game. [online]. URL `https://github.com/nneonneo/2048-ai`. [cit. 2017-04-26].

[36] Jung, A. (2016). Deep reinforcement learning for Super Mario World. [online]. URL `https://github.com/aleju/mario-ai`. [cit. 2017-04-27].

[37] Karakovskiy, S. a Togelius, J. (2009). Mario AI benchmark. [online]. URL `https://code.google.com/archive/p/marioai/`. [cit. 2017-04-27].

[38] Karakovskiy, S. a Togelius, J. (2012). The Mario AI Benchmark and Competitions. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAG)*, **4**, 55–67.

[39] Stanley, K. O. a Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, **10**(2), 99–127.

[40] Lau, B. (2016). Using keras and deep deterministic policy gradient to play TORCS. [online]. URL `https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html`. [cit. 2017-04-17].

[41] Loiacono, D., Cardamone, L. a Lanzi, P. L. (2013). Simulated car racing championship: Competition software manual. arXiv preprint arXiv:1304.1672v2.

[42] Salimans, T., Ho, J., Chen, X. a Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. arXiv preprint arXiv:1703.03864v1.

[43] Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A., Parizeau, M. a Gagné, C. (2012). DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, **13**, 2171–2175.

[44] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. a Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. [online]. URL `http://tensorflow.org/`. [cit. 2017-04-30].

Seznam obrázků

1.1 Hra 2048 5
1.2 Hra Mario 6
1.3 Závodní simulátor TORCS 6
1.4 Karetní hra Alhambra 7

2.1 Perceptron 8
2.2 MLP síť 9
2.3 Echo-state síť 12

3.1 Schéma zpětnovazebního učení 13
3.2 Ilustrace techniky dropout 18

4.1 Evoluční strategie CMA 23

6.1 Hra 2048 – průběh učení pro jednotlivé modely 39
6.2 Mario – průběh učení pro různé úrovně hry a modely 41
6.3 TORCS – ostrá zatáčka 42
6.4 TORCS – průběh učení pro jednotlivé modely 43
6.5 Alhambra – průběh učení pro jednotlivé modely 45

B.1 Diagram rozhodování původní umělé inteligence pro hru Alhambra 64

Seznam tabulek

6.1 Využívané senzory v simulátoru TORCS 31
6.2 Parametry využívané jednoduchým evolučním algoritmem 34
6.3 Parametry využívané algoritmem DQN 35
6.4 Hra 2048 – přehled výsledků 37
6.5 Hra 2048 – opakování nejlepšího výsledku 39
6.6 Mario – přehled výsledků 40
6.7 Mario – opakování nejlepšího výsledku 41
6.8 TORCS – přehled výsledků 42
6.9 TORCS – opakování nejlepších výsledku 44
6.10 Alhambra – přehled výsledků 44
6.11 Alhambra – opakování nejlepšího výsledku 45

A.1 Experiment: hra 2048 – evoluční algoritmus 58
A.2 Experiment: hra 2048 – CMA-ES 58
A.3 Experiment: hra 2048 – diferenciální evoluce 58
A.4 Experiment: hra 2048 – hluboké zpětnovazební učení (DQN) . . . 59
A.5 Experiment: Mario – evoluční algoritmus 59
A.6 Experiment: Mario – CMA-ES 60
A.7 Experiment: Mario – diferenciální evoluce 60
A.8 Experiment: Mario – hluboké zpětnovazební učení (DQN) 60
A.9 Experiment: TORCS – evoluční algoritmus 61
A.10 Experiment: TORCS – CMA-ES 61
A.11 Experiment: TORCS – diferenciální evoluce 61
A.12 Experiment: TORCS – hluboké zpětnovazební učení (DDPG) . . 61
A.13 Experiment: Alhambra – evoluční algoritmus 62
A.14 Experiment: Alhambra – CMA-ES 62
A.15 Experiment: Alhambra – diferenciální evoluce 62
A.16 Experiment: Alhambra – hluboké zpětnovazební učení (DDPG) . 63

Seznam použitých zkratek

CMA Covariance matrix adaptation.

TORCS The open racing car simulator.

ReLU Rectified linear unit.

MLP Multi-layer perceptron.

BP Backpropagation.

MSE Mean squared error.

SGD Stochastic gradient descent.

RMSProp Root mean square propagation.

Adam Adaptive moment estimation.

ESN Echo-state network.

DQN Deep Q-network.

DDPG Deep deterministic policy gradient.

EA Evoluční algoritmus.

ES Evoluční strategie.

DE Diferenciální evoluce.

EVA Evoluční algoritmus.

AI Artificial intelligence.

Přílohy

A. Detailní přehled experimentů

V této části přílohy nalezneme tabulky s kompletními výpočty a jejich nastavením pro daný běh.

Většinu značení v následujících tabulkách jsme popsali v kapitole 6, nicméně několik parametrů a zjednodušení stále zbývá vysvětlit (využíváme zkratek pro některé parametry kvůli velikostem tabulek):

- `gb` značí `game_batch`,
- `ngen` počet uběhlých generací,
- `result` značí adekvátní výsledek pro daný model, tedy nejlepší průměrnou fitness v generaci, které bylo dosaženo pro evoluční techniky. Pro různé hry jde o různé hodnoty:
  - 2048 — skóre hry,
  - Mario — překonaná vzdálenost úrovně,
  - TORCS — ujetá vzdálenost,
  - Alhambra — skóre hráče.
- `test` pak říká výsledek naměřený při testovacím běhu a jedná se o průměrné skóre naměřené v určitém počtu her (specifické pro každou hru, např. v případě TORCS používáme jinou testovací trať).
- `time` značí přibližnou dobu běhu,
- MLP sítě značíme MLP: $l_1, \dots, l_n$, kde l_i je počet neuronů ve vrstvě,
- ESN sítě značíme výrazem ESN: R/O$[l_1, \dots, l_n]$, kde R je počet neuronů v rekurentní části, O počet neuronů z nich vybraných a případné l_i další vrstvy,
- `rbs` značí `replay_buffer_size`,
- `sre` značí `store_replay_every`,
- `gamma` značí `discount_factor`,
- `update_freq` značí `target_update_freq`,
- `level` pro hru Mario značí obtížnost (základní, obtížná úroveň).
- `reward` popisuje typ odměny pro zpětnovazební učení, pro některé hry zkoušíme více typů odměn:
  - 2048 — Odměna A značí typ odměny, kde nevalidní tahy mají odměnu -1. B značí typ odměny, kde tyto tahy mají odměnu 0.
  - Mario — Odměny A a B se také liší pouze v konstantách, popsáno v sekci 6.2.2.

- TORCS — A značí standardní typ odměny, B značí odměnu s důrazem na první člen (konkrétní vzorce jsou k nalezení v části 6.2.3).
- Alhambra — Odměny A, B, C, jejichž význam jsme opsali v sekci 6.2.4.

- `exp` seskupuje hodnoty `init_exp` a `final_exp`.

Všechny experimenty zpětnovazebního učení probíhaly s `batch_size` nastaveným na 100.

Hra 2048 – Evoluční algoritmus												
pop_size	cross	mut	selection	gb	elite	net	activation	encoding	ngen	time	result	test
50	0.55/0.5	unif/0.01/0.5	tourn 2	10	5	MLP: 256, 256	ReLU	standard	1200	24h	3420.26	3137.33
50	0.75/0.2	unif/0.1/0.1	tourn 3	10	5	MLP: 500, 500	ReLU	standard	800	24h	3706.13	3536.15
50	0.25/0.1	unif/0.01/0.5	tourn 3	10	5	MLP: 300, 600	ReLU	standard	800	22h	3095.11	2929.13
50	0.75/0.1	unif/0.1/0.1	tourn 3	10	5	ESN: 256/32	ReLU	standard	2300	23h	3965.32	3648.09
50	0.75/0.2	unif/0.1/0.1	tourn 3	10	5	ESN: 1024/256	ReLU	standard	1500	23h	4195.82	3949.04
25	0.75/0.2	unif/0.1/0.1	tourn 3	10	5	ESN: 1024/256	ReLU	advanced	1500	20h	4078.43	3636.00
50	0.75/0.2	unif/0.1/0.1	tourn 3	10	5	ESN: 1000/200 [100]	ReLU	standard	2000	21h	4332.16	3922.53

Tabulka A.1: Experiment: hra 2048 – evoluční algoritmus

Hra 2048 – CMA-ES										
pop_size	sigma	gb	elite	net	activation	encoding	ngen	time	result	test
25	1.0	10	5	MLP: 16	ReLU	standard	5000	16h	4786.75	4199.80
25	1.0	10	5	MLP: 16, 16	ReLU	standard	5000	17h	4433.78	3920.61
25	1.0	10	5	MLP: 32, 32	ReLU	standard	5000	18h	5016.22	4222.55
25	1.0	10	5	MLP: 32, 32	ReLU	advanced	800	26h	5053.50	4393.43
25	1.0	10	5	MLP: 100, 100	ReLU	standard	360	26h	3831.87	3869.46
25	1.0	10	5	ESN: 1024/256	ReLU	standard	2150	20h	4475.90	3980.09

Tabulka A.2: Experiment: hra 2048 – CMA-ES

Hra 2048 – Diferenciální evoluce											
pop_size	CR	F	gb	hof_size	net	activation	encoding	ngen	time	result	test
25	0.25	1.0	50	5	MLP: 256, 256	ReLU	standard	600	20h	3386.75	3438.47
25	0.5	2.0	50	5	MLP: 200, 200	ReLU	standard	800	23h	3860.99	3718.93
25	0.25	1.0	50	5	ESN: 1000/200	ReLU	standard	740	26h	4109.35	3721.52

Tabulka A.3: Experiment: hra 2048 – diferenciální evoluce

Hra 2048 – Hluboké zpětnovazební učení (DQN)							
init_exp final_exp	anneal_steps update_freq	rbs sre	dropout reg_param	discount_factor reward & encoding	net activation & learning rate	time	test
0.5 0.01	500 000 10 000	100 000 5	0.9 0.01	0.99 A; standard	MLP: 900, 500, 300, 300, 300, 300, 300, 300 ReLU, 0.0005	22h	2828.06
0.9 0.01	1 000 000 10 000	100 000 10	0.5 0.01	0.99 A; standard	MLP: 300, 200, 200, 200 tanh, 0.0005	22h	1390.30
0.9 0.01	1 000 000 10 000	100 000 5	0.8 0.1	0.9 A; standard	MLP: 100, 100 ReLU, 0.001	17h	2928.69
0.9 0.01	1 000 000 10 000	100 000 5	0.8 0.01	0.9 A; standard	MLP: 4000, 2000, 1000, 1000 ReLU, 0.001	33h	1270.89
0.9 0.01	1 000 000 1000	100 000 1	0.8 1	0.9 A; standard	MLP: 100, 100 ReLU, 0.0001	25h	2390.92
0.5 0.1	1 000 000 10 000	100 000 1	0.7 0.01	0.5 B; advanced	MLP: 1000, 1000, 1000 ReLU, 0.001	39h	2618.53

Tabulka A.4: Experiment: hra 2048 – hluboké zpětnovazební učení (DQN)

Mario – Evoluční algoritmus												
pop_size	cross	mut	selection	gb	elite	net	activation	level	ngen	time	result	test
25	0.2/0.1	unif/0.05/0.5	tourn 3	10	5	MLP: 100, 100	ReLU	basic	500	26h	12.9 %	12.5 %
25	0.75/0.2	unif/0.1/0.1	tourn 3	10	5	MLP: 100, 100	ReLU	basic	500	36h	60.1 %	58.9 %
25	0.75/0.2	unif/0.1/0.1	tourn 3	5	5	MLP: 100, 100, 100	ReLU	basic	500	18h	9.0 %	7.0 %
25	0.5/0.1	unif/0.1/0.05	tourn 2	5	5	MLP: 200, 200	ReLU	basic	500	22h	23.8 %	20.6 %
25	0.5/0.1	unif/0.2/0.1	selbest	5	5	MLP: 100, 100	ReLU	basic	500	18h	58.7 %	58.1 %
25	0.75/0.2	unif/0.1/0.1	tourn 3	5	5	MLP: 100, 100	ReLU	advanced	500	15h	7.5 %	6.0 %
25	0.75/0.2	unif/0.1/0.1	tourn 3	5	5	ESN: 1000/200	ReLU	basic	500	28h	92.2 %	82.1 %
25	0.75/0.2	unif/0.1/0.1	tourn 3	5	5	ESN: 1000/200 [200]	ReLU	basic	500	32h	90.0 %	76.2 %
25	0.5/0.1	unif/0.1/0.05	tourn 3	5	5	ESN: 1000/200	ReLU	basic	500	32h	94.1 %	84.8 %
25	0.75/0.2	unif/0.1/0.1	tourn 3	5	5	ESN: 1000/200 [300, 200]	ReLU	basic	500	35h	78.2 %	72.3 %
25	0.5/0.1	unif/0.1/0.05	tourn 3	5	5	ESN: 2000/500	ReLU	basic	260	33h	82.2 %	78.6 %
25	0.5/0.1	unif/0.1/0.05	tourn 3	5	5	ESN: 1000/200	ReLU	advanced	500	18h	34.2 %	31.2 %

Tabulka A.5: Experiment: Mario – evoluční algoritmus

Mario – CMA-ES										
pop_size	sigma	gb	elite	net	activation	level	ngen	time	result	test
25	1.0	5	5	MLP: 30, 30, 30	ReLU	basic	300	32h	91.4 %	80.9 %
25	1.0	5	5	MLP: 30, 30, 30	ReLU	advanced	250	23h	38.0 %	31.8 %
25	1.0	5	5	ESN: 1000/200	ReLU	basic	310	17h	94.0 %	82.5 %
25	1.0	5	5	ESN: 1000/200	ReLU	advanced	400	14h	40.0 %	32.1 %

Tabulka A.6: Experiment: Mario – CMA-ES

Mario – Diferenciální evoluce											
pop_size	CR	F	gb	hof_size	net	activation	level	ngen	time	result	test
25	0.25	1.0	15	5	MLP: 200, 200	ReLU	basic	380	41h	45.5 %	82.1 %
25	0.25	1.0	15	5	MLP: 200, 200	ReLU	advanced	460	40h	24.8 %	27.3 %
25	0.25	1.0	15	5	ESN: 1000/200	ReLU	basic	300	41h	96.5 %	93.8 %
25	0.25	1.0	15	5	ESN: 1000/400	ReLU	advanced	440	41h	35.2 %	28.1 %

Tabulka A.7: Experiment: Mario – diferenciální evoluce

Mario – Hluboké zpětnovazební učení (DQN)							
init_exp final_exp	anneal_steps update_freq	rbs sre	dropout reg_param	discount_factor level & reward	net activation & learning rate	time	test
0.5 0.01	100 000 100	10 000 1	None 0.01	0.9 basic, A	MLP: 500, 500 ReLU, 0.01	19h	36.0 %
0.9 0.01	1 000 000 10 000	100 000 10	0.9 0.01	0.99 basic, A	MLP: 200, 200, 200, 200 ReLU, 0.0005	20h	7.0 %
0.9 0.01	1 000 000 10 000	100 000 10	0.9 0.01	0.99 basic, A	MLP: 200, 200 ReLU, 0.001	20h	23.5 %
0.9 0.01	100 000 10 000	100 000 10	0.9 0.01	0.99 basic, B	MLP: 200, 200 ReLU, 0.001	14h	7.0 %
0.5 0.01	1 000 000 1000	100 000 1	0.9 0.01	0.99 advanced, A	MLP: 500, 500 ReLU, 0.01	22h	13.6 %

Tabulka A.8: Experiment: Mario – hluboké zpětnovazební učení (DQN)

TORCS – Evoluční algoritmus											
pop_size	cross	mut	selection	gb	elite	net	activation	ngen	time	result	test
10	0.75/0.2	unif/0.1/0.1	tourn 3	1	5	MLP: 200, 200	ReLU	5000	14h	296m	77m
10	0.75/0.2	unif/0.1/0.1	tourn 3	1	5	MLP: 100, 100, 100, 100	ReLU	1730	26h	293m	81m
10	0.75/0.25	unif/0.1/0.1	tourn 3	1	5	ESN: 1000/200	ReLU	750	24h	800m	1030m

Tabulka A.9: Experiment: TORCS – evoluční algoritmus

TORCS – CMA-ES									
pop_size	sigma	gb	elite	net	activation	ngen	time	result	test
10	1.0	1	3	MLP: 50, 50, 50	ReLU	510	24h	112m	101m
10	1.0	1	3	ESN: 1000/200	ReLU	800	20h	239m	25m

Tabulka A.10: Experiment: TORCS – CMA-ES

TORCS – Diferenciální evoluce										
pop_size	CR	F	gb	hof_size	net	activation	ngen	time	result	test
10	0.25	1.0	1	5	MLP: 200, 200	ReLU	775	22h	265m	902m
10	0.25	1.0	1	5	ESN: 1000/200	ReLU	350	21h	1272m	2180m

Tabulka A.11: Experiment: TORCS – diferenciální evoluce

TORCS – Hluboké zpětnovazební učení (DDPG)				
rbs	discount_factor reward	actor: net, activation, learn. rate critic: net, activation, learn. rate	time	test
100 000	0.99 A	MLP: 400, 300; ReLU; 0.0001 MLP: 400, 300; ReLU; 0.001	17h	2692m
100 000	0.85 A	MLP: 400, 300; ReLU; 0.0001 MLP: 400, 300; ReLU; 0.001	26h	10m
100 000	0.99 B	MLP: 400, 300; ReLU; 0.0001 MLP: 400, 300; ReLU; 0.001	15h	482m

Tabulka A.12: Experiment: TORCS – hluboké zpětnovazební učení (DDPG)

Alhambra – Evoluční algoritmus											
pop_size	cross	mut	selection	gb	elite	net	activation	ngen	time	result	test
25	0.75/0.25	unif/0.1/0.1	tourn 3	5	5	MLP: 100, 100	ReLU	500	31h	130.81	128.94 (116.18; 113.01)
25	0.75/0.25	unif/0.1/0.1	tourn 3	5	5	MLP: 200, 200	ReLU	500	33h	132.31	129.8 (116.02; 114.08)
25	0.75/0.25	unif/0.1/0.1	tourn 3	5	5	MLP: 100, 100, 100	ReLU	320	23h	131.65	127.75 (112.99; 118.71)
25	0.75/0.2	unif/0.1/0.1	tourn 3	5	5	ESN: 1000/200	ReLU	500	26h	122.82	122.9 (120.61; 113.39)

Tabulka A.13: Experiment: Alhambra – evoluční algoritmus

Alhambra – CMA-ES									
pop_size	sigma	gb	elite	net	activation	ngen	time	result	test
25	1.0	5	5	ESN: 200/100	ReLU	300	25h	123.56	117.24 (121.82; 120.16)
25	1.0	5	5	ESN: 300/50	ReLU	450	25h	123.37	122.74 (114.73; 116.97)

Tabulka A.14: Experiment: Alhambra – CMA-ES

Alhambra – Diferenciální evoluce										
pop_size	CR	F	gb	hof_size	net	activation	ngen	time	result	test
25	0.25	1.0	15	5	MLP: 100,100	ReLU	200	33h	126.52	122.7 (117.76; 119.76)
25	0.25	1.0	15	5	ESN: 200/100	ReLU	200	33h	126.52	122.78 (117.05; 116.31)

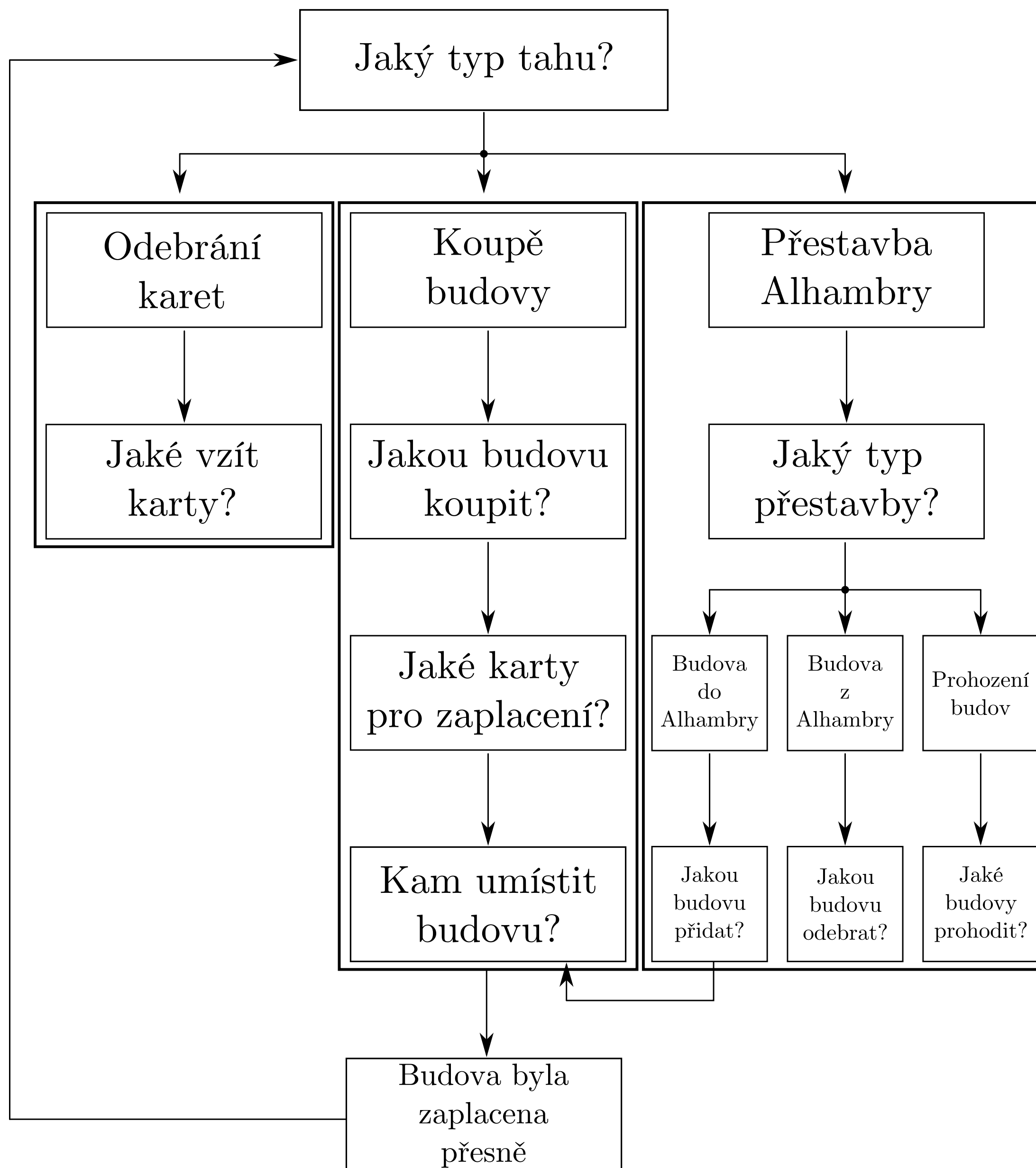
Tabulka A.15: Experiment: Alhambra – diferenciální evoluce

Alhambra – Hluboké zpětnovazební učení (DDPG)				
rbs	discount_factor reward	actor: net, activation, learn. rate critic: net, activation, learn. rate	time	test
100 000	0.99 A	MLP: 400, 300; ReLU; 0.0001 MLP: 400, 300; ReLU; 0.001	19h	108.64 (124.81; 124.79)
100 000	0.7 A	MLP: 400, 300; ReLU; 0.0001 MLP: 400, 300; ReLU; 0.001	22h	93.05 (129.65; 130.23)
100 000	0.99 B	MLP: 400, 300; ReLU; 0.0001 MLP: 400, 300; ReLU; 0.001	23h	89.56 (131.74; 132.48)
100 000	0.99 C	MLP: 400, 300; ReLU; 0.0001 MLP: 400, 300; ReLU; 0.001	18h	105.51 (127.17; 126.64)

Tabulka A.16: Experiment: Alhambra – hluboké zpětnovazební učení (DDPG)

B. Implementace a dokumentace

B.1 Alhambra a původní umělá inteligence



Obrázek B.1: Diagram rozhodování původní umělé inteligence pro hru Alhambra. Diagram převzat z [10].

B.2 Implementace

Algoritmy (modely) jsou implementovány v jazyce Python a implementace her jsou na nich nezávislé a mnohdy v jiném jazyce: 2048 (Python), Mario (Java), TORCS (client, Java), Alhambra (C#).

Interface na straně her je v adresáři `general-ai/Game-interfaces`. Zde jsou zároveň uloženy konfigurační soubory, které říkají, jaký počet vstupů a výstupů je očekáván od AI (případně počet fází hry).

Komunikace mezi AI a hrami

Komunikace mezi hrami a umělou inteligencí je zprostředkována pomocí podprocesů (vyžaduje-li si to situace; ve většině případů) a následně pomocí standardních vstupů a výstupů. Tato komunikace je standardizována pomocí formátu JSON. Příklad průběhu komunikace: Kontrolor spustí hru a získá stav hry na jejím standardním výstupu, následně předá stav zvolenému modelu, ten odpoví určitou akcí, které je následně zakódována do formátu JSON a předána hře na standardní vstup. Proces se následně opakuje.

B.3 Technické informace

Projekt obsahuje tři hlavní složky: `Controller` obsahuje veškeré implementace algoritmů a toho, co se týká AI, `Experiments` obsahuje záznamy experimentů, které byly provedeny a `Game-interfaces` obsahuje implementace rozhraní na straně her (u hry Mario je toto rozhraní obsaženo ve zvláštní složce na úrovni `general-ai`, jmenovitě `MarioAI`). Požadavky na spuštění:

- Python 3.5
- NumPy
- SciPy
- Matplotlib
- Sklearn
- Deap
- Gym
- TensorFlow (použitá verze byla 0.12.1)
- CUDA
- NVIDIA cuDNN
- Java 8
- .NET Framework 4.5

Další využité knihovny[1]:

- SimpleESN [https://github.com/sylvchev/simple_esn/blob/master/simple_esn.py]
- TensorFlow-Reinforce [https://github.com/yukezhu/tensorflow-reinforce]

[1] Cit. 2017-06-24.

- DDPG [https://github.com/songrotek/DDPG]
- 2048 [https://github.com/tjwei/2048-NN/blob/master/c2048.py]
- Mario [https://github.com/kefik/MarioAI]
- Gson [https://github.com/google/gson]
- Json.NET [http://www.newtonsoft.com/json]

Závěrečné poznámky

- Soubor `controller.py` je hlavním spouštěcím kódem, ze kterého se spouští učení.
- Soubor `constants.py` obsahuje veškeré konstanty týkající se adresářů.
- Soubor `visualizations.py` obsahuje užitečné metody pro spouštění testů a generování grafů. Tento soubor je k dispozici v adresáři

  `Controller/utils`,

  stejně tak jako tímto souborem vygenerované grafy a výsledky.
- Typ odměny je definován na straně hry, tedy v patřičném interface. Stejně tomu tak je při výběru úrovně pro hru Mario.
- Kód hry 2048 je obsažen přímo v `Game-interfaces/Game2048/` a hru není potřeba instalovat. Kódování hry se nastavuje přímo ve zdrojovém kódu hry. Soubor `game_2048.py`.
- Hra Mario je obsažena v adresáři na úrovni `general-ai`, protože je to separátní git repozitář.
- Implementace hry Alhambra je v patřičném adresáři

  `general-ai/Game-interfaces/Alhambra/`

  jako `.dll` soubor.
- Soubor `install_directory.txt` v `Game-interfaces/TORCS` obsahuje lokaci, kde je nainstalována hra TORCS.
- Hra TORCS je složitější na instalaci (je třeba použít správnou verzi, patch). Dále se během výpočtu daný klient (model, který hraje hru) připojuje k lokálnímu TORCS serveru. Dané porty jsou vždy specifikovány v patřičném XML s nastavením. Pro více informací doporučujeme manuál [41].
- Nastavení trati a informací o závodě se provádí přes konfigurační XML soubor. Tyto soubory jsou k nalezení v adresáři

  `general-ai/Game-interfaces/TORCS/`.

- Vizuální spuštění TORCS je možné pomocí funkcí, které jsou uloženy v souboru `visualizations.py`. Je však ale potřeba také mít konkrétní nastavení (XML konfigurační soubor) v místě instalace hry

  `<install_dir>/torcs/config/raceman/`

  Takový konfigurační soubor (`race_config.xml`) přikládáme na DVD do složky TORCS. Toto nastavení je možné provést i „ručně“ pomocí herního GUI.

- Pro konkrétní programovací jazyky se rovněž předpokládá přítomnost běžných knihoven (NumPy apod.). Výpočty týkající se zpětnovazebního učení probíhaly na Windows s GPU GTX 1070. Evoluční výpočty převážně na Linuxu. Hru Alhambra je také možné pustit na Linuxových systémech, pomocí Projektu Mono. Hry Mario a 2048 lze počítat na Linuxu bez větších problémů. Výpočty pro TORCS jsme prováděli pouze na Windows.

- Pro ladění vnitřku her je možné využít standardních chybových výstupů, které jsou propagovány do modelu a zobrazeny na konzoli.

C. Obsah přiloženého DVD

- `general-ai` — hlavní projekt s následující strukturou:
  - `Controller` — obsahuje kód modelů, `controller.py` je hlavní vstupní bod programu, `utils/visualizations.py` obsahuje funkce pro vykreslování grafů, testování a spouštění modelů (inference).
  - `Experiments` — experimenty rozdělené do podsložek podle her a modelů. Každý experiment obsahuje adekvátní informace: nastavení, vygenerovaný graf (pokud se jedná o TensorFlow, tak je obsažen patřičný events log), doba běhu apod. Také je zde uložen výsledný model (obvykle `best/best_0.json`).
  - `Game-interfaces` — obsahuje interface na straně her. Pro Alhambru se jedná o Visual Studio projekt (C#, kód hry Alhambra je přidán jako `.dll`), hra 2048 je obyčejný Python soubor, Mario zde má pouze konfigurační soubor, interface hry je na stejné úrovni s celým projektem `general-ai`, protože byl po celou dobu používán jako separátní GIT repozitář a hra TORCS zde obsahuje Java projekt s daným interface a konfiguračními `.XML` soubory.
- `MarioAI` — projekt pro hru Mario, rozhraní pro umělou inteligence je v adresáři `MarioAI4J-Playground/src/mario`.
- `GeneralAI.pdf` — tato práce v elektronické podobě.