Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Učení virtuálního robota metodami umělé inteligence

2019

Bc. Ondřej Procházka

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

Studijní obor: Aplikovaná informatika, prezenční forma

Bibliografické údaje

Autor:	Bc. Ondřej Procházka
Název práce:	Učení virtuálního robota metodami umělé inteligence
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2019
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Petr Osička, Ph.D.
Počet stran:	48
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Bc. Ondřej Procházka
Title:	Using machine learning to teach a virtual robot
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2019
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Petr Osička, Ph.D.
Page count:	48
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Pro učení umělé inteligence se využívá metody reinforcement learningu. V diplomové práci se věnuji této metodě. Pro demonstraci reinforcement learningu jsem implementoval virtuální prostředí robota. Ve virtuálním prostředí se robot učí plnit úkoly. Metoda reinforcement learningu za pomocí umělých neuronových sítí je pak využita při trénování robota. Tato metoda má aplikaci ve spoustě složitých případů. Reinforcement learning lze například využít u autonomního řízení aut.

Synopsis

In artificial intelligence exists methods for creating trainable agent. In master thesis is demonstrated this technique. For explanation and example I implemented virtual enviroment for robot. Robot is learning to solve problems in this enviroment. Reinforcement learning with neural networks are used for training the robot. This method can be used in many tough cases. One of the cases can be autonomous driving car.

Klíčová slova: reinforcement learning; umělá inteligence; neuronová síť; agent

Keywords: artificial intelligence; trainable agent; reinforcement learning; neural network

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

1 Úvod **7**

2 Základy reinforcement learningu **9**
2.1 Základní pojmy reinforcement learningu 9
2.2 Funkce Q pomocí tabulky se zpožděnou odměnou 10
2.2.1 Temporal difference 11
2.3 Strategie výběru akcí 11
2.4 Trénování agenta s omezeným počtem možných stavů 11
2.4.1 Prostředí FrozenLake 11
2.4.2 Ukázka učení se ve FrozenLake prostředí 12

3 Umělé neuronové sítě **15**
3.1 Perceptron 15
3.2 Vícevrstvé neuronové sítě 16
3.3 Dopředný chod 16
3.4 Chybová funkce 17
3.4.1 Regression problém 17
3.4.2 Klasifikační problém 17
3.5 Zpětný chod 18
3.6 Dataset, batch a dropout 19
3.7 Konvoluce 19
3.8 Rekurentní neuronové sítě 21
3.8.1 LSTM neuronové sítě 22

4 Reinforcement learning s neuronovými sítěmi **24**
4.1 Predikování odměn 24
4.2 Vstup 24
4.3 Tvorba množiny dat a trénování 25
4.3.1 Učení s učitelem 25
4.4 Ukázka struktury neuronové sítě agenta 25

5 Virtuální prostředí robota **27**
5.1 Infrastruktura aplikace 27
5.2 Objekty v prostředí 27
5.3 Vstupy a výstupy robota 28
5.4 Módy a odměny 30

6 Umělá inteligence robota **31**
6.1 Třída Hráč AI 31
6.2 Třída Mód reinforcement AI 31
6.2.1 Neuronová síť 32
6.2.2 Pohyb robota pomocí umělé inteligence 33
6.2.3 Učení neuronové sítě 33

6.3 Třída Paměť 34

7 Uživatelské rozhraní aplikace **36**
7.1 Nastavení parametrů odměn a parametrů paměti 36
7.2 Tvorba vlastní sítě 37
7.3 Trénování robota, testování robota a příklady natrénovaných sítí . 38

8 Experimenty **41**
8.1 Experimentování s učením bez učitele 41
8.1.1 Mód Elevator 41
8.1.2 Mód Weapons 42
8.2 Experimentování s učením s učitelem 42
8.2.1 Mód Elevator 43
8.2.2 Mód Weapons 43

Závěr **44**

Conclusions **45**

A Obsah přiloženého CD/DVD **46**

Literatura **47**

Seznam obrázků

1 Proces reinforcement learningu zdroj: [3] 10
2 Perceptron 15
3 Ukázka neuronové sítě 16
4 Proces učení inspirováno zdrojem: [7] 18
5 Ukázka konvoluce zdroj: [8] 20
6 Ukázka rekurentní neuronová sít 21
7 Ukázka dopředného chodu v rekurentní neuronové sítě 22
8 LSTM buňka inspirováno zdrojem: [9] 23
9 Ukázka sekvence buněk v LSTM 23
10 Ukázka neuronové sítě agenta, inspirováno zdrojem: [6] 26
11 Architektura aplikace 27
12 Herní objekty 28
13 Ukázka prostředí 29
14 Ukázka vstupu 29
15 Ukázka architektury kolem reinforcement learningu 31
16 Okno pro nastavení odměn a parametrů paměti 37
17 Okno pro tvorbu nové umělé inteligence 38
18 Okno výběru prostředí hry 39
19 Okno výběru prostředí hry 40
20 Graf učení bez učitele v módu *Elevator*. 42
21 Graf učení bez učitele v módu *Weapons*. 42
22 Graf učení s učitelem v módu *Elevator*. 43
23 Graf učení s učitelem v módu *Weapons*. 43

Seznam tabulek

Seznam vět

Seznam zdrojových kódů

1 Vytvoření neuronové sítě 32
2 Vytvoření chybové funkce 33
3 Volba akce 33
4 Vytvoření neuronové sítě 34

1 Úvod

Učení robota za pomocí odměn je v dnešní době poměrně probírané téma. Pro učení agenta za pomocí odměn se využívá tzv. *reinforecement learning* metody. Cílem reinforcement learningu je vytrénování *agenta* pro úspěšné plnění úkolů. Agent dostává vstup ze svých senzorů. Vstupem může být například 2D obraz z kamery. Na základě svého vstupu se snaží odhadnout, jaká další jeho akce bude nejvýhodnější. Výhodnost akce se určuje dle odměny, kterou za ni obdrží. Musí se také zohlednit budoucí odměna, to je ovšem netriviální problém.

Problémy řešené reinforcement learningem lze rozdělit do dvou skupin. U problému v první skupině se agent může nacházet ve velmi omezeném počtu stavů. Příkladem mohou být piškvorky na polích 3×3. V tomto případě se agent nachází v méně než 50 stavech. Z toho důvodu dokážeme efektivně zjistit všechny odměny pro každý stav. Do druhé skupiny se řadí problém, kde agent může nabývat počtu stavů blížící se nekonečnu, což se podobá reálnému světu. Snažíme se pouze přiblížit optimálním hodnotám.

Neuronové sítě jsou nástroj pro jednoduché uchopení algoritmicky složitého problému. Neuronové sítě se dokáží postupně učit řešit problém z dat. K tomu, aby byly efektivní, je třeba splnit několik ne vždy triviálních požadavků, např. najít ideální strukturu. Mnohdy se totiž může stát, že neuronová síť se bude učit špatně nebo vůbec. Často záleží na drobnostech a velmi těžko se odhaduje, jaká struktura bude pro daný problém vhodná.

V této práci jsem vytvořil virtuální prostředí pro robota. Toto prostředí je navrhnuto pro trénování agenta. Ve virtuálním prostředí může být více módů. Módy jsou dobré pro experimentování s reinforcement learningem. V tomto prostředí lze generovat množinu dat pro neuronovou síť. Dále pak lze v aplikaci vygenerovat různé modely neuronových sítí. Také pomocí aplikace lze měnit velikost odměn.

V aplikaci pro práci s neuronovými sítěmi používám *Torch framework* [1]. Torch v základu poskytuje abstrakci nad matematickými operacemi, především s *tenzory*. Dále pak má v sobě moduly pro tvorbu neuronových sítí. Nad tímto frameworkem je také vytvořen *rnn* modu [2], který se stará o tvorbu *rekurentních neuronových sítí*. Rekurentní neuronové sítě jsou druhem sítí, které si dokáží uchovávat stav z předchozích výpočtů. Tento typ sítí využíváme, protože v prostředí záleží na předchozích stavech.

První kapitolu věnuji základům reinforcement learningu. Cílem této kapitoly je uvedení do problematiky reinforcement learningu v jeho nejjednodušší podobě. Druhá kapitola představuje umělé neuronové sítě. Zde jsou popsány základy umělé neuronové sítě, dopředná a zpětná propagace. Dále jsou zde rozebrány rekurentní neuronové sítě. Třetí kapitola vysvětluje spojení reinforcement learningu s umělými neuronovými sítěmi. Zbývající kapitoly popisují aplikaci, kterou jsem sám vytvořil. V těchto kapitolách představuji vytvořené virtuální prostředí, učení robota v tomto prostředí pomocí reinforcement learningu a umělých neuronových sítích. Dále pak je představeno uživatelské rozhraní aplikace.

Poslední kapitola je věnována experimentům provedenými v aplikaci s virtuálním prostředím robota.

2 Základy reinforcement learningu

Reinforcement learning metoda je založena na principu učení agenta za pomocí odměn. Díky tomuto přístupu učení je zapotřebí pouze určit odměny za akce provedené agentem. Agent pak iterativně zkouší akce a z nich zjišťuje získanou odměnu. Tento přístup lze najít v přírodě a psychologii. Pro aplikování v umělé inteligenci je důležité implementovat způsob, jakým se bude agent učit určovat odměny. Zde je klíčové předvídání budoucích odměn.

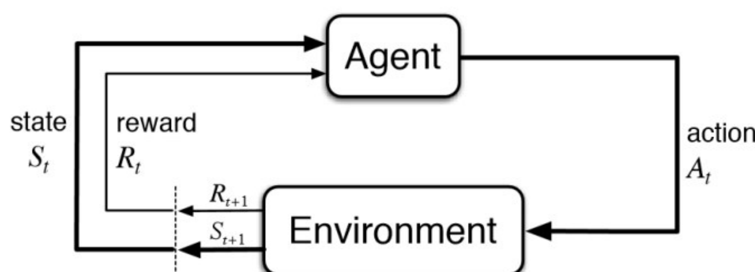
Tento přístup je zajímavý z pohledu problémů, u kterých je vyžadována přítomnost člověka, protože jsou pro člověka snadné. Spousta těchto problémů by v ideálním případě byla řešitelná, protože učení člověka má podobnost s učením agenta pomocí reinforcement learningu. Podobnost je v tom, že člověk se snaží zdokonalovat na základě předchozích úspěchů či neúspěchů. Využití reinforcement learningu můžeme najít například u autonomních aut nebo chatbotů. Zajímavou aplikací je použití této techniky jako umělé inteligence ve hrách.

2.1 Základní pojmy reinforcement learningu

Reinforcement learning má definovaných několik základních pojmů. Nyní zavedeme značení, která budeme využívat dále:

- Agent: Vyvolává akci a přijímá stav.
- A: je množina všech možných akcí, které agent může provádět. Například může mít definované akce jako pohyb doprava, doleva a rovně.
- Environment: je prostředí, kde se agent pohybuje. Prostředí přijímá akci agenta vybranou z A jako vstup a prostředí agentovi vrací odměnu a jeho následující stav. Například agent může provést akci rovně, přičemž stoupne na past. Na základě toho prostředí udělí agentovi zápornou odměnu a předá mu další stav (nacházíš se v pasti).
- S: je množina stavů, do kterých se agent může dostat.
- r: je okamžitá odměna obdržená po akci a, r symbolizuje zpětnou vazbu agentovi. Vyjadřuje, do jaké míry byla jeho akce úspěšná. Kupříkladu agent zvedl bonusový objekt a na základě této akce mu prostředí udělilo odměnu jedna.
- γ: *slevový faktor (discount factor)* je reálné číslo z intervalu $\langle 0, 1 \rangle$. Toto číslo se násobí s budoucí maximální odměnou. Používá se jako vyjádření důležitosti budoucí odměny. Je to z důvodu sekvence akcí vedoucí k odměně. Když ji nastavíme k hodnotě blízké nule, bude budoucí odměna zanedbatelná. Opačně na ní bude kladen velký důraz.
- π: je *strategie výběru akce*. To znamená, že agent vybere akci dle této strategie. Strategie se využívá k sofistikovanému zkoumání prostředí agentem. Díky tomu je agent schopný získat přehled o odměnách.

- s: současný stav agenta.
- a: zvolená akce z množiny A.
- $s^{'}$: další stav, do kterého agent přešel po zvolení akce a.
- $Q(s, a)$: je funkce, která přiřazuje akcím maximální odměnu, jakou mohou získat. Velikost odměny se počítá pomocí okamžité odměny a budoucích odměn, nad kterými je aplikován slevový faktor. Tato funkce může být reprezentovaná tabulkou.
- $\max_{a \in A}(Q(s, a))$: je funkce, která hledá ze stavu s maximální hodnotu, ke které může dospět s ohledem na množinu akcí A.
- α: reprezentuje učící konstantu (learning rate).



Obrázek 1: Proces reinforcement learningu zdroj: [3]

Na grafu 1 je demonstrován proces vyhodnocení. Agent se nachází ve stavu S_t. Ze stavu S_t provede akci A_t, kterou obdrží prostředí Environment. Prostředí vyhodnotí akci agenta a následně agentovi pošle odměnu $R_t + 1$ a stav $S_t + 1$. Tento proces se poté neustále opakuje.

2.2 Funkce Q pomocí tabulky se zpožděnou odměnou

V odměně se musí zohledňovat i možná budoucí odměna. Z toho důvodu musíme vytvořit takovou funkci $Q(s, a)$, která predikuje odměnu s ohledem na budoucnost. K tomu nám poslouží *Bellmanova rovnice*. Rovnice je vyjádřena:

$$Q(s, a) = r + \gamma * \max_{a \in A}(Q(s^{'}, a))$$

Z rovnice tedy plyne, že pro výpočet musíme zohlednit všechny budoucí stavy, ke kterým agent může dospět z dané akce, abychom dostali správnou hodnotu. Ve většině problémů ale není možné projít tolik stavů. Této problematice se bude věnovat kapitola reinforcement learning s neuronovými sítěmi.

2.2.1 Temporal difference

Funkci $Q(s, a)$ učíme z interakce agenta s prostředím, aby měla přesnější hodnoty. Pro zdokonalování této funkce budeme používat rovnici, která se nazývá *Temporal difference*. Funkce po aktualizaci bude značena takto $Q^{'}(s, a)$. Rovnice je vyjádřena:

$$Q^{'}(s, a) = Q(s, a) + \alpha * (r + \gamma * \max_{a \in A}(Q(s^{'}, a)) - Q(s, a))$$

Existuje více přístupů tabulkového učení. Například *Dynamické programování* a *Monte Carlo learning*. Těmito přístupy se zabývat nebudeme. Pro vysvětlení rozdílu mezi metodami zavedu pojem *Epizoda*. Epizoda je sekvence akcí ukončena splněním či nesplněním úkolu. Jako příklad může být dohraní partie šachů. Temporal difference je výhodnější, protože u Monte Carlo learning se musí pro aktualizování $Q(s, a)$ odehrát celá epizoda až do konce. U dynamického programování se pro aktualizaci $Q(s, a)$ musí projít všechny možné následující stavy. Tato metoda je v podstatě brute force, tedy složitost je exponenciální vzhledem k počtu stavů, do kterých agent přešel. U Temporal difference není třeba mít ukončenou epizodu. Zároveň se nezkouší všechny následující stavy.

2.3 Strategie výběru akcí

Důležitá funkce je také π, tedy strategie výběru akcí. Úkolem strategie je prozkoumávat prostředí pro zjištění možných odměn. Jedna z účinných strategií je *Epsilon greedy policy*. Je to jednoduchá funkce, která je definována následovně:

$$\varepsilon = 1/k$$

k je proměnná, která určuje počet epizod, kterými se agent trénoval.

$$\pi = \begin{cases} \text{argmax}_{a \in A}(Q(s, a)) & \text{s pravděpodobností } 1 - \varepsilon \\ \text{náhodná akce} & \text{s pravděpodobností } \varepsilon \end{cases}$$

2.4 Trénování agenta s omezeným počtem možných stavů

K reinforcement learningu se dá přistoupit pomocí tabulkové metody. Ta je založená na tabulce, do které se ke každému stavu přiřazuje odměna. K tomu, abych tuto metodu mohl demonstrovat, použiji jednoduchý příklad z *Open ai gym*. Open ai gym je webová stránka, kde je několik aplikací pro testování umělé inteligence. Vybraná aplikace z open ai gym se jmenuje *FrozenLake* [4]. Tato aplikace nabízí jednoduché prostředí pro učení se s reinforcement learningem.

2.4.1 Prostředí FrozenLake

Kapitola je inspirovaná zdrojem [5]. *FrozenLake* prostředí se skládá z mřížky o velikosti 4×4. Agent se v předem vygenerovaném prostředí snaží ze startovní

pozice dostat do cíle. Agent bude znát jen svou pozici. Buňky z mřížky mohou být následujících typů:

- S: startovací pozice
- F: ledový povrch - zde je agent v bezpečí
- H: symbolizuje díru, do které agent může spadnout a tím prohrát
- G: cíl, kam se má agent dostat

Prostředí hry vypadá například:

$$\begin{matrix} S & F & F & F \\ F & H & F & H \\ F & F & F & H \\ H & F & F & G \end{matrix}$$

Agent má čtyři možné akce a snaží se pomocí nich dostat do cíle G bez toho, aniž by spadl do díry H:. Akce agenta jsou následující:

- doprava
- doleva
- nahoru
- dolů

Množina stavů, do kterých se agent bude moci dostat, je $16 * 4$. Je to velikost pole vynásobená počtem akcí. Aby to agent neměl tak jednoduché, je v prostředí aplikace implementován vítr. Vítr může s určitou náhodou agentovi změnit směr pohybu.

Odměny R jsou nastavené:

- za úspěšné dosažení cíle: 1
- jinak: 0

2.4.2 Ukázka učení se ve FrozenLake prostředí

Nyní představím pseudokód 1 učícího se algoritmu. V pseudokódu si jako první inicializujeme pole o velikosti mřížky z prostředí krát počet akcí agenta. Proměnná epsilon slouží agentovi pro náhodné volení akce. Je to proto, aby se $Q(s, a)$ funkce mohla konvergovat ke správným hodnotám. Každou iterací by měla být $Q(s, a)$ funkce přesnější. Epsilon pak v každé iteraci snižuje počet epizod. Toto snižování má za následek, že se více volí akce, které nám odhaduje $Q(s, a)$ funkce. Díky přesnějším hodnotám z $Q(s, a)$ bude síť hlouběji konvergovat k maximální odměně. Na 23. řádku je rovnice Temporal difference, která nám upravuje $Q(s, a)$ funkci.

Algorithm 1 Pseudokód reinforcement learningu pomocí tabulky

```
1:  A ← množina akcí agenta
2:  B ← pole reprezentující mapu prostředí
3:  Q ← pole o velikosti |B| x |A| s inicializací prvků na 0
4:
5:  learningRate ← hodnota z rozsahu ⟨0, 1⟩
6:  gamma ← hodnota z rozsahu ⟨0, 1⟩
7:  isFinished ← false
8:  epsilon ← 1
9:
10: procedure Max(s)
11:     return z pole: Q[s] najdi maximalně možnou akci
12:
13: for k ← 1, počet restartovaných prostředí do
14:     s ← Reset prostředí
15:     epsilon ← epsilon/k
16:     while isFinished = false do
17:         random ← náhoda z ⟨0, 1⟩
18:         if random < epsilon then
19:             a ← zvol náhodně akci z A
20:         else
21:             a ← Max(Q[s])
22:         s1, r, isFinished ← krok v prostředí s akcí a
23:         Q[s, a] ← Q[s, a] + learningRate ∗ (r + gamma ∗ max(Q[s1]) − Q[s, a])
```

V tomto prostředí je vhodné proměnnou gamma definovat na hodnotu 0.95 a learningRate na hodnotu 0.8.

Na tomto algoritmu je tedy vidět, jak se agent iterativně učí. Nejdříve $Q(s,a)$ funkce odhaduje špatné hodnoty. Z první iterace jen nuly. Postupně v dalších iteracích se $Q(s,a)$ funkce zlepšuje a přibližuje ke správným odhadům.

3 Umělé neuronové sítě

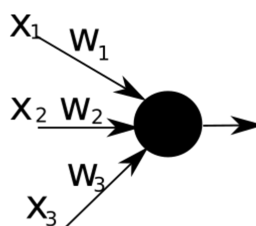
Umělá neuronová síť, dále již jen neuronová síť, se využívá v oblasti umělé inteligence. Neuronová síť se dokáže učit predikovat výsledky na základě vstupů. Aplikuje se často tam, kde řešení pomocí algoritmů by bylo neúnosné, jak z pohledu složitosti kódu, tak z pohledu náročnosti výpočtu. Neuronová sít se používá například na rozpoznávání objektů na fotce.

3.1 Perceptron

Nejdříve si představíme *perceptron*. Vycházím ze zdroje [6]. Perceptron má vstupy $x_1, x_2, ..., x_n$ a perceptron produkuje jeden výstup. Pro spočítání výstupu potřebujeme *váhy*. Váhy nabývají reálných hodnot a vyjadřují důležitost vstupu v závislosti k výsledku. Pro každý vstup máme jednu váhu. Dále si určíme *threshold*. Threshold je reálné číslo. Výsledek výstupu musí být větší než treshold, aby výstup nabýval hodnoty jedna. Pro spočítání výstupu tedy použijeme následující formuli.

$$výstup = \begin{cases} 0 & \text{jestliže } \sum_n w_n x_n \leq threshold \\ 1 & \text{jinak} \end{cases}$$

Na obrázku 2 je ukázka perceptronu, který má 3 vstupy.



Obrázek 2: Perceptron

Z důvodu terminologie si zavedeme pojem *neuron* a *bias*. Neuron je perceptron, který nepoužívá threshold a nemusí mít nutně binární výstup a ani stejnou funkci pro výpočet výstupu jako perceptron. Neuron má definovaný bias b. Bias je reálné číslo a má stejnou funkci jako threshold. Pro zjednodušení definice použijeme následující operaci $w*x$ definovanou jako $\sum_n w_n x_n$. Neuron může mít více typů. Abych mohl demonstrovat typy neuronů, zavedu pojem aktivační funkce. Aktivační funkce zobrazuje výsledek z $w*x$ na nějaké reálné číslo. *Binární neuron* je stejný jako perceptron díky své binární aktivační funkci, liší se pouze znaménkem biasu oproti tresholdu v perceptronu. Vzorec pro výpočet výstupu binárního neuronu je definován následovně:

$$výstup = \begin{cases} 0 & \text{jestliže } w * x + b \leq 0 \\ 1 & \text{jinak} \end{cases}$$

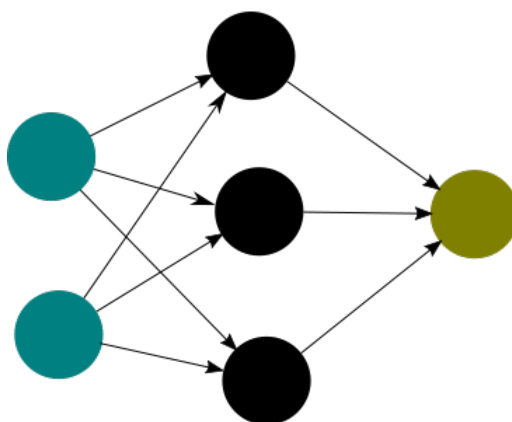
Další typ neuronu je *sigmoid neuron*. Sigmoid neuron používá jako aktivační funkci sigmoid. Díky sigmoidu lze lépe optimalizovat váhy neuronu, gradientními metodami. Sigmoid neuron vypočítá výstup následovně:

$$výstup = \frac{1}{1 + e^{-w*x-b}}.$$

3.2 Vícevrstvé neuronové sítě

Ve *vícevrstvé neuronové síti* jsou neurony rozděleny do *vrstev*. Na obrázku 3 je příklad *dopředných neuronových sítí*. Existují 3 základní typy vrstev. Na obrázku 3 jsou oddělené barvami:

- Vstupní vrstva nejvíce vlevo znázorněna modrou barvou.
- Skrytá vrstva uprostřed znázorněna černou barvou. Skrytých vrstev může být více.
- Výstupní vrstva nejvíce vpravo znázorněna zelenou barvou.



Obrázek 3: Ukázka neuronové sítě

Pro dopředné neuronové sítě platí, že vrstvy jsou propojené tak, že výstup vrstvy l_n jde na vstup další vrstvě l_{n+1}. Výstup každého neuronu z vrstvy l_n je vstupem všem neuronům ve vrstvě l_{n+1}. Pro dopředné neuronové sítě platí, že mezi vrstvami neexistují cykly. U rekurentních neuronových sítí to neplatí.

3.3 Dopředný chod

Dopředný chod je algoritmus pro spočítání výstupu neuronů ze vstupů. Při dopředném chodu se aktivují postupně vrstvy $l_1, l_2, ...l_n$, kde l_n je výstupní vrstva. Při dopředném chodu ve vrstvě l_k se vypočtou aktivace všech neuronů, které patří do vrstvy l_k. Výsledky z neuronů se pak použijí jako vstupy v další vrstvě.

3.4 Chybová funkce

Chybová funkce počítá chybu mezi predikovaným výstupem a správným výstupem. Na základě této chyby se síť učí například *gradientní metodou.* Představím dva typy chybových funkcí. První se používá pro *regression problém* a druhý pro *klasifikační problém.* Existuje více typů chybových funkcí.

3.4.1 Regression problém

Regression problém je takový, kde výstupem je číslo. Příkladem regression problému je problém predikce výsledku sčítání vstupů x_1 a x_2. Máme několik typů chybových funkcí pro regression problém. Například mean square error, která je definovaná takto:

$$\frac{1}{n}\sum_{i=0}^{n}(y_i - y_i^{'})^2.$$

V definici je:

- y: je vektor obsahující predikovaný výstup
- $y^{'}$: je vektor obsahující očekávaný výstup
- n: je počet výstupních neuronů

3.4.2 Klasifikační problém

U klasifikačního problému každá složka výstupu reprezentuje třídu, na kterou se má mapovat vstup. Výstupní hodnota pak ukazuje, s jakou pravděpodobností se má vázat konkrétní třída ke vstupu. Můžeme pak zvolit výstup s nejvyšší pravděpodobností. Typický klasifikační problém je například třídění obrázků do tříd, kdy chceme určit, zda–li je na obrázku člověk, nebo zvíře. *Binary cross entropy* je vhodná pro klasifikační problémy, je definovaná takto:

$$-\frac{1}{n}\sum_{i=0}^{n}(y_i^{'}\log{(y_i)} + (1 - y_i^{'})\log{(1 - y_i)}),$$

kde

- y: je vektor obsahující predikovaný výstup
- $y^{'}$: je vektor obsahující očekávaný výstup
- n: je počet výstupních neuronů

3.5 Zpětný chod

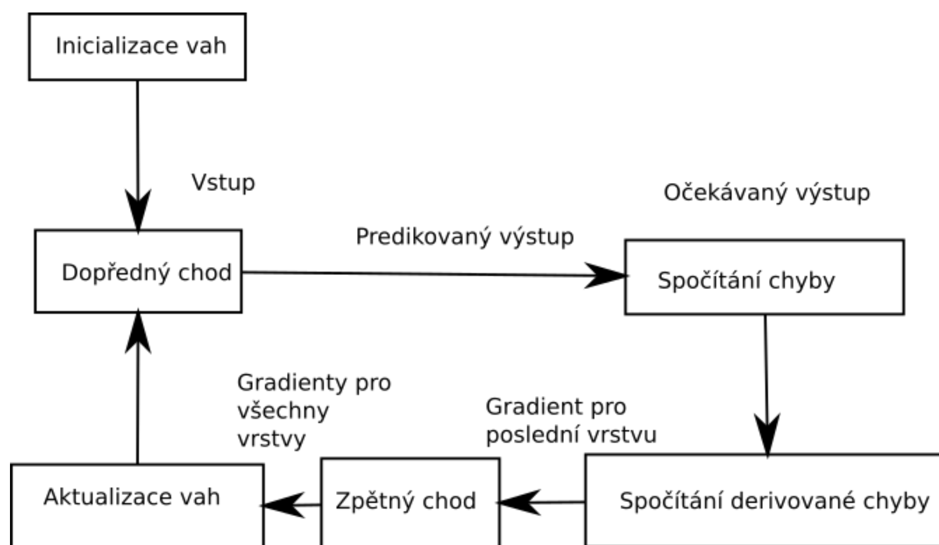
Zpětný chod je algoritmus sloužící k úpravě vah neuronů, tak aby neuronová síť konvergovala ke správnějším výsledkům. Zpětný chod vyhodnocuje postupně vrstvy v pořadí $l_n, l_{n-1}, ...l_1$, kde l_n je výstupní vrstva a l_1 je vstupní vrstva. Ve vrstvách postupně upravuje váhy neuronů. Neurony se upravují vzhledem k chybě, která vznikla na výstupu. U zpětného chodu se uplatňuje parciální derivace. Pomocí parciální derivace postupně derivujeme podle váhy $w_1, w_2..., w_n$ ve vrstvě l_k.

Zpětný chod se snaží v ideálním případě najít globální minimum chybové funkce. Ovšem globální minimum je ve většině případů nemožné najít, takže většinou neuronová sít skončí v lokálním minimu. V některých případech se nám může stát, že neuronová sít uvízne v nevyhovujícím lokálním minimu.

Výpočet zpětného chodu se skládá z několika základních kroků.

- Dopředný chod pro získání výstupních hodnot.
- Spočítání parciální derivace z chybové funkce. Pomocí toho je získán vektor o velikosti počtu výstupů.
- Vektor po derivaci chybové funkce se zpropaguje do další vrstvy. V této vrstvě se opět spočítají gradienty. A tento proces se opakuje do první vrstvy.
- Aktualizování vah pomocí *optimalizační funkce*.

Pro lepší názornost poslouží obrázek 4. Na obrázku je vyobrazeno, jak se neuronová síť učí v iteracích.



Obrázek 4: Proces učení inspirováno zdrojem: [7]

Optimalizační funkce aktualizují váhy po výpočtu zpětného chodu na základě gradientů. Nejznámější optimalizační funkce je *stochastic gradient descent.* Tato

funkce je zadána následovně:

$$w_n = w_s - \sigma(n_s) * \alpha$$

V této funkci w_n představuje vypočtenou novou váhu, w_s je stávající váha a $\sigma(n_s)$ je derivace pro konkrétní váhu. Dále α značí učící konstantu. Tento vzorec se aplikuje pro všechny váhy v neuronové síti.

Učící konstanta symbolizuje jemnost, s jakou se neuronová síť bude přibližovat k lokálnímu minimu. Pokud bude nastavená na velkou hodnotu, lokální minimum přeskočí. Když bude nastavená na malou hodnotu, bude dlouho trvat, než se dostane do lokálního minima.

3.6 Dataset, batch a dropout

Pro učení neuronové sítě je potřeba mít *dataset*. Dataset je množina dat. Data obsahují vstup do neuronové sítě a správný výsledek. Dataset se skládá ze dvou množin:

- Trénovací data.
- Testovací data.

Množina trénovacích dat s testovacími by neměla mít společný průnik. Trénovací data se používají při tréninku neuronové sítě. Je důležité mít množiny testovacích dat pokrývající co nejvíce různorodých případů. Toto pravidlo je důležité z důvodu zobecnění problému. Testovací množina se používá pro testování neuronové sítě. Pomocí ní můžeme zjistit, jestli si neuronová síť správně zobecnila problém.

Neuronovou síť lze trénovat v takzvaných *batchích*. Batch si lze představit jako skupinu vstupů. Velikost batche je pak počet vstupů v batchi. Neuronová síť pomocí dopředného chodu zpracuje batch s trénovacími vstupy. Pro každý vstup z batche neuronová síť vrátí výstupy. Nad těmito výstupy se pomocí chybové funkce spočítají gradienty. S těmito gradienty je pak proveden zpětný chod. Pomocí batchí lze efektivněji trénovat síť.

Dropout je funkce, pomocí níž se ignorují zvolené neurony. Dropout se používá výhradně v trénovacím módu. Tato funkce s určitou náhodou nuluje výstupy neuronů ve zvolené vrstvě. Pomocí nulování výstupu neuronu se neuron dále ignoruje. Pravděpodobnost nulování neuronu je určená parametrem ve funkci. Tato metoda se využívá kvůli závislosti mezi neurony, která během tréninku způsobuje horší zobecňovací schopnost neuronové sítě.

3.7 Konvoluce

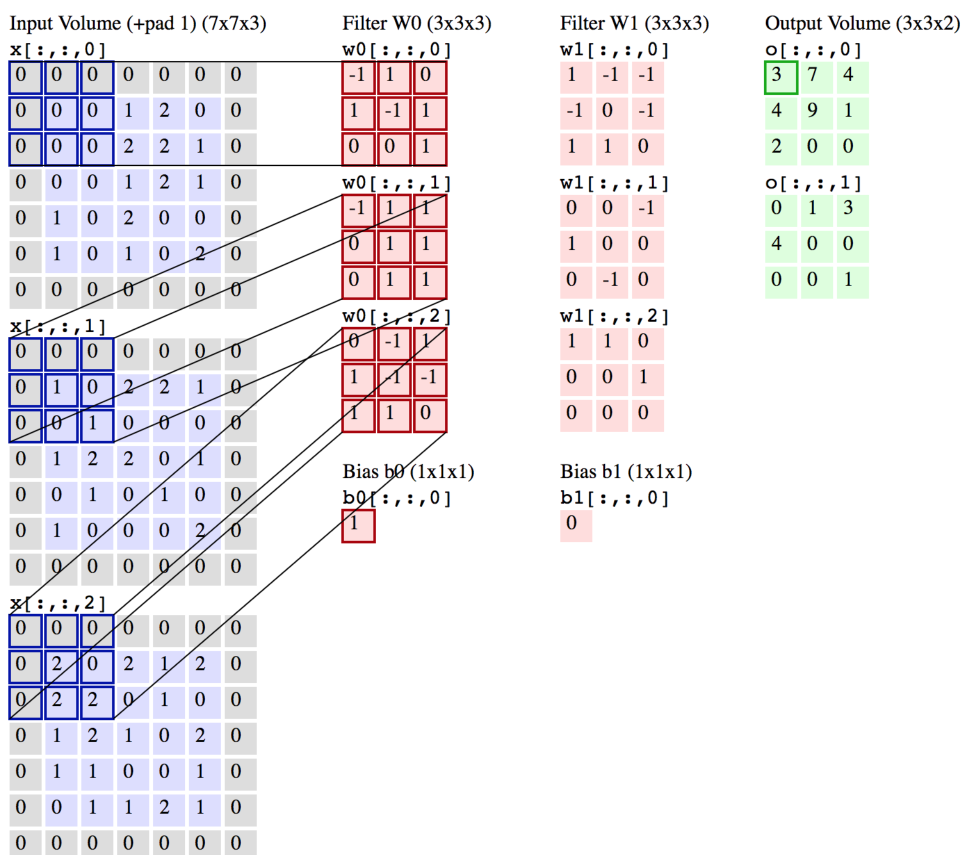
Konvoluční neuronové sítě se používají pro vstup, kde existuje metrika jako například u obrázku. Konvoluční sítě nám umožňují zobecňovat tento vstup.

Konvoluce muže být n–dimenzionální. Typická konvoluční síť pro rozpoznávání 2D obrazů je třídimenzionální:

- hloubka
- šířka
- výška

Výpočet konvoluce probíhá pomocí konvolučního jádra. Konvoluční jádro může být n–dimenzionální. Pro jednoduchost ale budeme uvažovat ve dvoudimenzionálním. Dvoudimenzionální konvoluční jádro je definováno šířkou a výškou. Při konvoluci se nastavuje několik parametrů.

- Krok jádra v horizontálním a vertikálním směru, pokud se pohybujeme ve 2D (*stride*).
- Velikost okrajů obrazu (*padding*).



Obrázek 5: Ukázka konvoluce zdroj: [8]

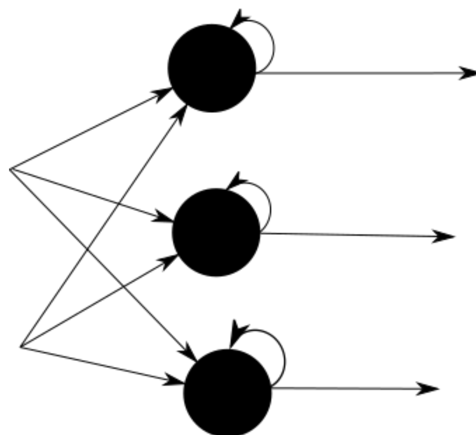
Na obrázku 5 je ukázán dopředný chod pro neuronovou konvoluční síť. Vstup má 3 kanály s nastavenou velikostí okrajů na 1. Dále jsou zde dva filtry pro výstupy ve dvou kanálech. Jádro má velikost 3×3. Toto jádro se posouvá o dvě

buňky. Výpočet probíhá tak, že jádro se posunuje po vstupu. Na prvky v jádru se aplikuje filtr s tím, že se vynásobí buňky v jádře se stejnou pozicí ve filtru. Takto se vypočtou všechny složky v jádře a následně se sečtou. Stejný postup se provede nad dalšími kanály. Výsledky ze všech jader v kanálu se sečtou a přičte se bias. Vypočtená hodnota je pak první složkou výstupu. Poté se posune jádro, v našem případě o 2, a opakuje se postup. Takto se vypočte první výstupní kanál a následně se stejný postup aplikuje na vypočtení druhého kanálu s druhým filtrem.

Pro demonstraci si uvedeme příklad. Vytvoříme si neuronovou síť s konvolucí. Vytvořená síť bude rozpoznávat číslice. Díky konvoluci by měla natrénovaná neuronová síť být schopna rozpoznat číslice v různé rotaci a velikosti. Můžeme do obrázků s číslicemi přidat i nějaký šum. I přesto by číslice měla být neuronovou sítí s konvolucí rozpoznatelná. Jsou tam samozřejmě limity v rozpoznávání. Tyto limity jsou i u lidského oka.

3.8 Rekurentní neuronové sítě

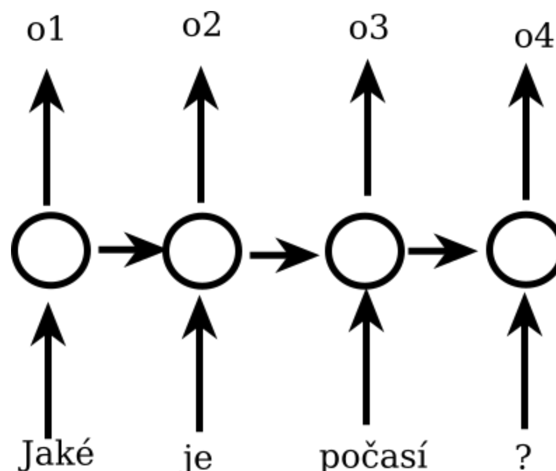
Rekurentní neuronové sítě obsahují cykly. Tyto sítě umožňují reagovat na předešlé vstupy. To znamená, že nová predikovaná hodnota je závislá na předchozích vstupech. Díky této vlastnosti je rekurentní neuronová síť schopná orientovat se v posloupnostech. Například rekurentní neuronová síť nachází využití v generování slov.



Obrázek 6: Ukázka rekurentní neuronová sít

Na obrázku 6 je ukázka rekurentní neuronové sítě. Je na ní vidět, že výstup neuronu přechází do nové vrstvy, ale zároveň jde i zpátky na vstup neuronu.

Při dopředném chodu je vstupem do rekurentní sítě sekvence. Na obrázku 7 je ukázán dopředný chod nad rekurentní neuronovou sítí. Vidíme na něm sekvenci slov. Při postupném přidávání slov do rekurentní neuronové sítě ztrácí staré vstupy důležitost. Základní rekurentní neuronové sítě mají jen krátkodobou paměť. Z této sekvence by dokázala rekurentní neuronová síť odhadnout, že se ptáme na počasí.



Obrázek 7: Ukázka dopředného chodu v rekurentní neuronové sítě

Zpětný chod probíhá v rekurentních sítích odlišně než u dopředných sítí. U rekurentních neuronových sítí ve zpětném chodu záleží na velikosti sekvence. Čím menší časová sekvence pro zpětný chod, tím kratším sekvencím bude neuronová síť rozumět. S délkou sekvence je ale výpočet zpětného chodu náročnější.

3.8.1 LSTM neuronové sítě

V této podkapitole budu vycházet ze zdroje [9]. *LSTM* jsou rekurentní neuronové sítě s dlouhodobou pamětí. Dokáží zahazovat nepotřebné informace z paměti a důležité si uchovávat. Jsou vhodné pro delší sekvence, kde záleží i na starých vstupech. LSTM síť se skládá ze tří bran:

- brána pro vstup
- brána pro zapomínání
- brána pro výstup

Na obrázku 8 je ukázána LSTM buňka. Pro tento obrázek je následné značení:

- σ: je sigmoid funkce
- tanh: je tangent funkce
- x: je vstup
- h: je výstup
- C: je stav buňky

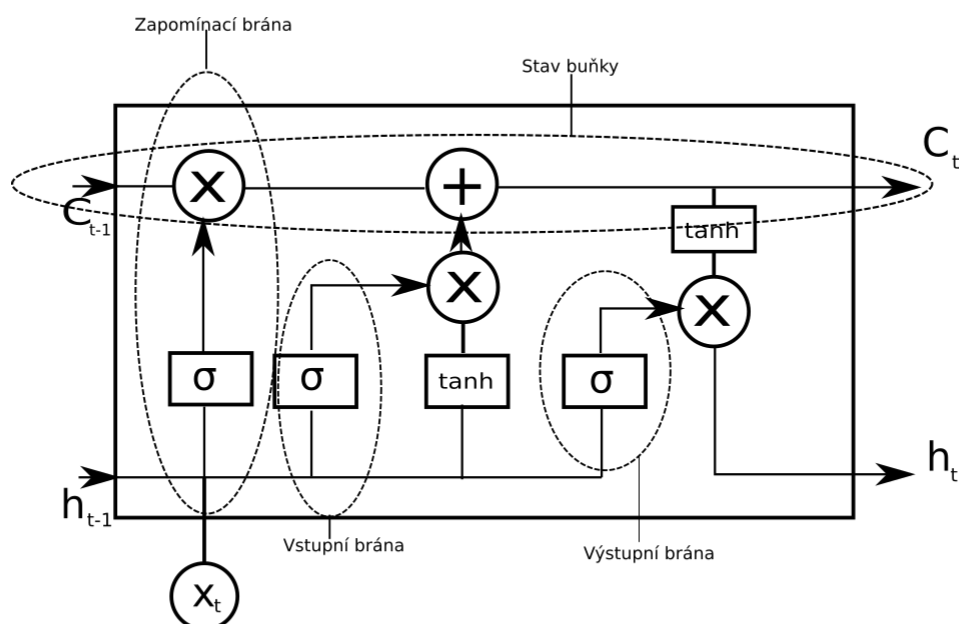
Brána pro zapomnění funguje tak, že rozhoduje, jaká informace má být udržena nebo zahozena. Do této brány vstupuje součet výstupů z předchozí buňky

a nynější vstup. Čím je hodnota po aplikování sigmoid funkce nižší, tím je informace méně důležitá pro síť.

Vstupní brána slouží k aktualizování stavu buňky. U vstupní brány jsou předány do sigmoid funkce součty předchozího výstupu a nynější vstup. Dále do tangent funkce je předán stejný součet jako do sigmoid funkce. Výsledky těchto funkcí jsou poté vynásobeny. Sigmoid funkce zde určuje důležitost vstupu a tangent funkce vstupní informace převede do intervalu $(-1; 1)$.

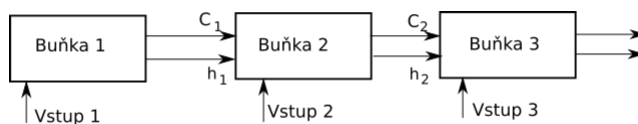
Z výpočtů brány zapomnění a vstupní brány se vypočítá konečný stav buňky. Tento stav pak nese relevantní informace podle LSTM sítě.

Brána výstupu aplikuje opět sigmoid funkci nad součtem předchozího výstupu a nynějšího vstupu. Dále se pak na stav buňky aplikuje tangent funkce. Výsledky funkcí se vynásobí a tím vznikne nový výstup.



Obrázek 8: LSTM buňka inspirováno zdrojem: [9]

Na obrázku 9 je ukázáno, jak jsou LSTM buňky za sebou zapojeny.



Obrázek 9: Ukázka sekvence buněk v LSTM

4 Reinforcement learning s neuronovými sítěmi

Reinforcement learning s využitím neuronových sítí umožňuje učit agenta v reálném prostředí. Nebudeme tak potřebovat tabulku na reprezentování Q funkce, a tudíž už nebudeme omezeni velikostí tabulky. Q funkci lze implementovat pomocí neuronových sítí. Díky tomu jsme schopni postupně predikovat výhodné odměny v prostředí.

4.1 Predikování odměn

Pro predikování odměn budeme vycházet z Bellmanovy rovnice.

$$Q(s,a) = r + \gamma * \max_{a \in A}(Q(s^{'}, a))$$

Cílem pak bude najít takovou $Q_\theta(s,a)$, která se blíží $Q(s,a)$. K tomu, abychom ji našli, využijeme neuronové sítě. Aby $Q_\theta(s,a)$ mohla konvergovat k $Q(s,a)$, musíme definovat spočítání chyby. Pro spočítání chyby vycházíme z toho, že známe jen r odměnu v Bellmanově rovnici. Vše ostatní z Bellmanovy rovnice musí jen konvergovat ke správným hodnotám.

Pro spočítání chyby můžeme použít funkci mean square error, kterou jsme definovali v kapitole Chybová funkce. Definice bude vypadat následovně:

$$y_t = r + \gamma * \max_{a \in A}(Q(s^{'}, a))$$

$$L_t(\theta) = (y_t - Q_\theta(s,a))^2$$

V našem případě je $n = 1$, proto zde chybí $\frac{1}{n}\sum_{i=0}^{n}$. Je to z toho důvodu, že se chyba počítá jen pro jeden výstup. Výstup, který se počítá, je ten, který zastupuje zvolenou akci.

Obecně tedy v chybové funkci zadáváme jako predikovanou hodnotu $Q_\theta(s,a)$ a jako správnou hodnotu y_t

4.2 Vstup

Vstup agenta bude vstupem do neuronové sítě. Pro správné fungování neuronové sítě musí být vstup zobecnitelný. Budeme mít dva druhy vstupů. První vstup bude pro orientaci v prostředí. Pro orientaci v prostředí je vhodné zvolit metrický vstup, například můžeme vzít 2D obraz ze hry. Druhý vstup bude reprezentovat proměnné agenta. Například agent může mít za proměnné stav životů nebo energie.

Pro efektivnost sítě je vhodné mít vstup co nejmenší. Vstup ale musí zachovat podstatu informace. Proto, když použijeme 2D obraz, je vhodné jej transformovat do odstínů šedi. Další možností je 2D obrázek škálovat dolů. Díky těmto technikám si vstup zachová podstatnou informaci, ale zároveň se vydatně zmenší. Nakonec pro metrický vstup je vhodné použít konvoluční neuronovou síť. Díky

konvoluční neuronové síti jsme schopni dosáhnout zobecnění. Neuronová síť potom pochopí různě rotovaný vstup.

Vstup s proměnnými agenta přidáme do vrstvy za konvoluční síť. Díky tomu nebudou smíchány dva nesourodé vstupy.

4.3 Tvorba množiny dat a trénování

Pro trénování neuronových sítí potřebujeme mít množinu dat. Z trénovací množiny dat pak vybíráme náhodné vzorky o námi zadané velikosti sekvence. Náhodné jsou z toho důvodu, aby síť postupně nezdegenerovala. Velikost sekvence je potřebná kvůli používání rekurentních neuronových sítích. Agenta učíme pomocí sekvencí. Bez sekvence by rekurentní neuronová síť nepochopila návaznosti na konkrétní stav.

Dále je dobré neuronovou síť trénovat po batchích. Zaprvé je to výpočetně rychlejší. Druhou výhodou je lepší optimalizace problému. Také záleží na velikosti batche, jaká je zvolena.

Pro trénování neuronové sítě je důležité použít vhodnou optimalizační funkci. Vhodná optimalizační funkce neuvízne jen tak v lokálním minimu. Díky tomu se lokální minimum bude přibližovat globálnímu minimu.

Tvorba množiny dat spočívá v ukládání struktury $\langle s_t, a, r, s_{t+1} \rangle$. Díky této struktuře je možno simulovat pohyb agenta v reálném prostředí.

Dále je dobré nevykonávat akci agenta každý frame hry. Když se přeskočí hodně framů, bude se agent chovat zpomaleně nebo nebude přesný. Například když agent bude chtít zamířit na malý cíl. Po přeskočení k framů se může stát, že mířidla agenta cíl přeskočí. Stejně to funguje i u člověka, když hraje hru a hra má malý počet FPS.

Množina dat pro testování není třeba. Je to proto, že stačí spuštění agenta v prostředí. Zde se agent testuje automaticky.

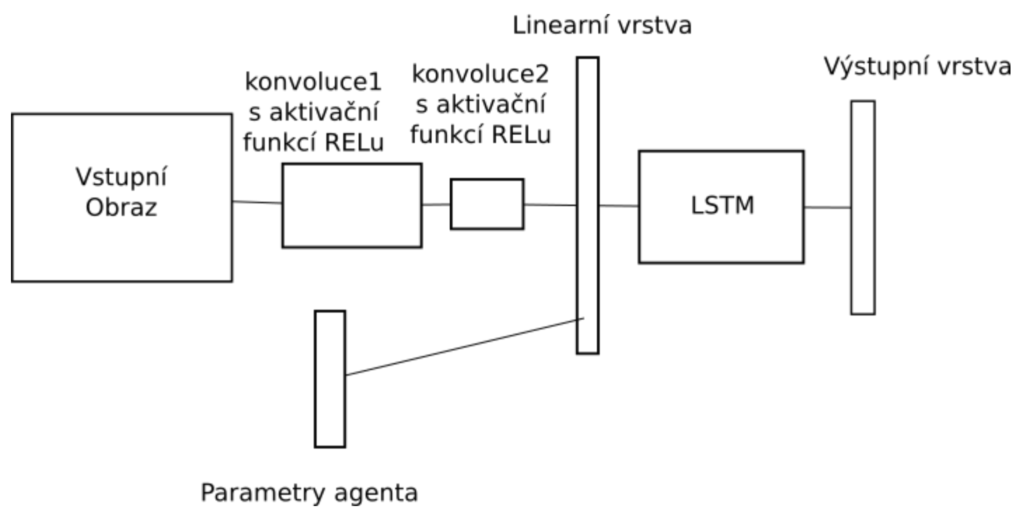
4.3.1 Učení s učitelem

Učení s učitelem probíhá tak, že člověk hraje místo umělé inteligence. Člověk tedy nahrazuje strategii výběru akce. Tím může ukázat agentovi velikosti odměn, jakých může dosáhnout. K těmto odměnám se agent jen za pomocí strategie výběru akce nemusí nikdy dostat. Tímto způsobem pak neuronová sít rychleji konverguje k maximální hodnotě. Můžeme dokonce i kombinovat učení s učitelem a pomocí strategie výběru akce. Pomocí těchto dvou technik pak můžeme vytvořit trénovací množinu dat.

4.4 Ukázka struktury neuronové sítě agenta

Nyní představím základní návrh neuronové sítě agenta zastupující Q funkci.

Neuronová síť bude mít dva vstupy. První je 2D obraz toho, co agent vidí. Druhý vstup jsou jeho parametry. Pro jednoduchost je ze začátku lepší vynechat parametry agenta. Agent by se měl dokázat naučit pohybovat v prostředí i bez



Obrázek 10: Ukázka neuronové sítě agenta, inspirováno zdrojem: [6]

svých parametrů na základě toho, co vidí. Tím se dá ověřit správnost metrického vstupu.

Po konvoluci vždy následuje aktivační funkce *ReLU*. ReLU je definována následovně:

$$ReLU(x) = max(0, x)$$

To znamená, že záporné hodnoty se zobrazí na 0 a kladné na sebe sama.

Po konvolucích následuje lineární vrstva. Lineární vrstva sjednocuje výstup konvoluce a parametry agenta. Výstup lineární vrstvy jde do rekurentní neuronové sítě LSTM. Z LSTM jde výstup opět do výstupní lineární vrstvy. Výstupní lineární vrstva je výstupem Q pro všechny akce.

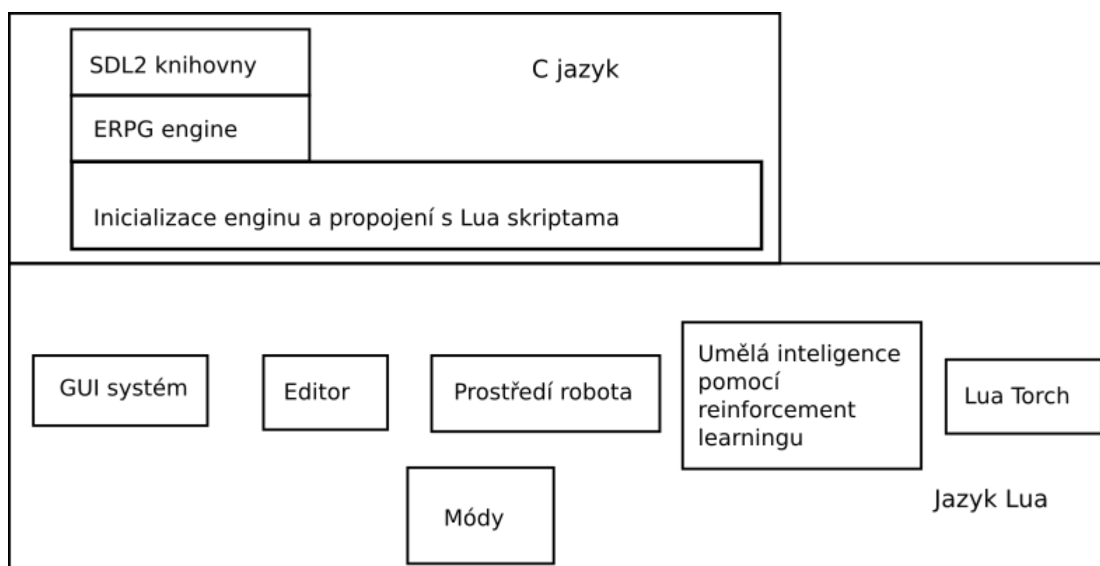
5 Virtuální prostředí robota

Pro demonstraci a experimentování s reinforcement learningem jsem vytvořil speciální aplikaci. V této aplikaci je vytvořeno virtuální prostředí robota. Robot se v prostředí pohybuje a má za úkol plnit úkoly. Za plnění určitých úkolů dostává odměnu. Díky tomuto prostředí bude možno sledovat chování robota.

5.1 Infrastruktura aplikace

Infrastruktura aplikace je rozdělená do několika částí. Tyto části jsou zobrazeny na obrázku 11. Aplikace je založená na *ERPG enginu*, který byl vytvořen v mé bakalářské práci [10]. Engine prošel několika úpravami pro tuto aplikaci . Hlavní úpravou enginu byla implementace scény s vyhledáváním objektů pomocí *Quad tree* algoritmu. Jádro aplikace je tedy napsáno v jazyce C.

Nad jádrem aplikace jsem v této aplikaci vytvořil *GUI systém*. Dále pak editor pro vlastní účely. Editor je v distribuované aplikaci vypnutý. Poté jsem udělal prostředí robota a umělou inteligenci za použití *frameworku Lua Torch.* Nakonec jsem vytvořil módy s různými prostředími hry.

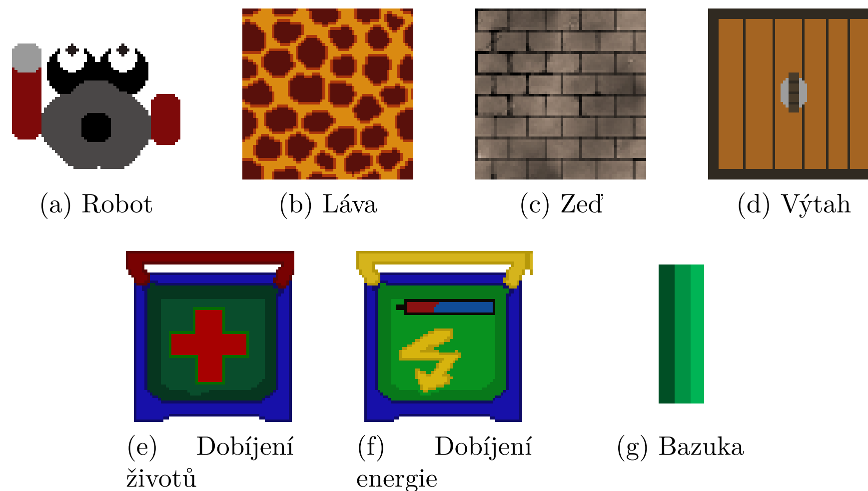


Obrázek 11: Architektura aplikace

5.2 Objekty v prostředí

V prostředí je několik typů objektů. Všechny objekty jsou vyobrazené na obrázku 12.

- *Robot* se může pohybovat po prostředí a interagovat s ním. Robota bude ovládat agent. Robot má životy a energii. Robot přestane fungovat po



(a) Robot (b) Láva (c) Zeď (d) Výtah

(e) Dobíjení životů (f) Dobíjení energie (g) Bazuka

Obrázek 12: Herní objekty

vyprchání životů nebo energie. Potom se znovu objeví na nějakém místě, které lze zvolit v editoru map.

- *Láva* při intersekci s robotem ubírá postupně životy robotovi.
- *Zeď* je neprostupná. Tedy nikdo přes ní nemůže projít.
- *Výtah* je určen pro cíl, kam má robot dojít v módu *elevator*.
- *Dobíjení životů* začne dobíjet robotovi životy při intersekci. Má omezenou kapacitu životů. Určitou dobu pak trvá, než se zde zase robot může dobíjet.
- *Dobíjení energie* začne dobíjet robotovi energii při intersekci. Funkcionalita s omezenou kapacitou energie je stejná jako u dobíjení životů.
- *Bazuka* je zbraň, kterou může robot zvednout.

Každý objekt v prostředí má speciální identifikační číslo z intervalu $\langle 0, 1 \rangle$. Toto číslo se bude používat pro vstup. Prostředí tedy může vypadat jako na obrázku 13.

5.3 Vstupy a výstupy robota

Agent dostává na vstup vizuální vstup pro orientaci v prostředí. Může dostat na vstup také parametry životů a energie. Vizuální vstup je vytvořen pomocí n paprsků, které vysílá robot. Počet paprsků je v základu nastaven na 31. Vizuální vstup má velikost $2n$. Proč $2n$? To vysvětlím v dalším odstavci. Tento vstup jsem zvolil z důvodu rychlosti. Výsek kusu prostředí jako 2D obraz v pixelech by obsahoval větší vstup. Tím pádem by trvalo delší dobu neuronovou síť naučit a zároveň by bylo zapotřebí většího výpočetního výkonu.

Obrázek 13: Ukázka prostředí

Paprsky jsou relativní vůči robotovi. Pokud se robot otočí,tak i paprsek dostane směr podle natočení robota. Každý paprsek má určitý index. Paprsek může intersektovat s nějakým objektem z prostředí. Po intersekci paprsku se uloží do pole pod index paprsku dvě hodnoty. První hodnota je identifikační číslo objektu. Druhá hodnota je vzdálenost intersekce od ohniska vypuštění paprsku. Ohnisko vypuštění paprsku je uprostřed robota. Pokud nezasáhl paprsek žádný objekt, obě čísla jsou 0.

Na obrázku 14 vidíme názornou ukázku. Tento vstup koreluje s obrázkem 13. Na obrázku 13 jsou vidět také body, které se promítly do vstupu na obrázku 14. Tyto body jsou zaznačeny bílými tečkami.

Obrázek 14: Ukázka vstupu

Robot může dělat několik akcí. V módech jsou většinou tyto akce omezené z důvodu rychlejší konvergence sítě. V základu jsou tedy tyto akce:

- dopředu
- dozadu

- rotace doleva
- rotace doprava
- zvednutí zbraně
- položení zbraně
- výstřel

5.4 Módy a odměny

Módy modifikují základní prostředí hry. Pomocí módů lze změnit počet akcí. Dále v módu můžeme změnit třídu starající se o umělou inteligenci.

Virtuální prostředí v základu obsahuje dva módy. První mód, ten jednodušší, se nazývá *Elevator*. V Elevator módu se má agent s robotem dostat k výtahu. Při hledání výtahu může robotovi vyprchat energie nebo ho může zabít láva. Robot by se měl snažit dostat k výtahu s co největší energií a největším počtem životů.

Druhý mód *Weapons* je zaměřen na sbírání bonusových objektů. Bonusovým objektem v tomto módu je bazuka. Když robot zvedne zbraň, tak zbraň automaticky zmizí. Agent poté obdrží odměnu. Cíl je naučit robota zvednout co nejvíce zbraní. Robot se opět může vybít nebo umřít v lávě.

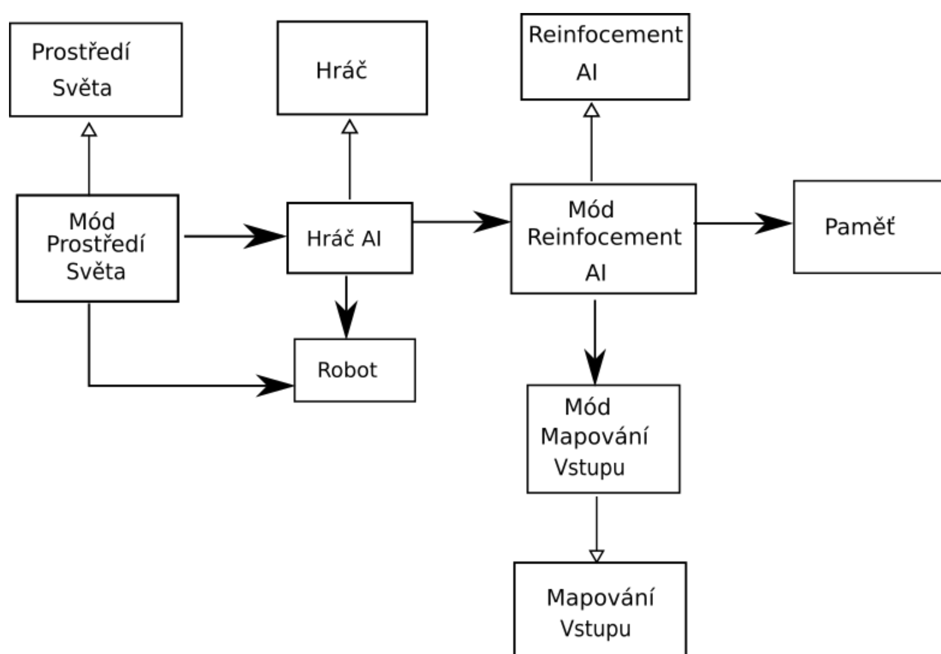
Agent dostává odměny za tyto stavy:

- Robot si dobíjí životy
- Robot si dobíjí energii
- Robot dostává zranění
- Robot vstoupil na výtah
- Robot sebral zbraň

Odměny agenta můžeme měnit z uživatelského rozhraní. Nastavování odměn bude vysvětleno v další kapitole.

6 Umělá inteligence robota

Pro lepší orientaci v aplikaci představím na obrázku 15 základní architekturu kolem umělé inteligence. Černá šipka značí komunikaci mezi třídami. Nevyplněná šipka ukazuje dědičnost. Třídy se ve zdrojovém kódu takto nejmenují. Jména jsou zvolena jen pro demonstrační účely.



Obrázek 15: Ukázka architektury kolem reinforcement learningu

Třídy z obrázku 15 budou vysvětleny v následujících podkapitolách.

6.1 Třída Hráč AI

Třída *Hráč AI* dědí z třídy *Hráč*. Tato dědičnost je z toho důvodu, že se myslí i na lidského hráče. V třídě *Hráč AI* je metoda volající se v každém snímku hry. V této metodě je nastaveno, že každou třetí iterací se zavolá metoda pro obsloužení umělé inteligence.

Před dalším krokem umělé inteligence se získají odměny hráče. Tyto odměny jsou pak předány jako argument do metody pro další krok umělé inteligence.

6.2 Třída Mód reinforcement AI

Třída *Mód reinforcement AI* dědí z *Reinforcement AI*. Dědičnost je zde kvůli módům. Každý mód může mít jinak implementovanou třídu pro umělou inteligenci.

6.2.1 Neuronová síť

Nyní představím základní zdrojový kód tvořící neuronovou síť používající se v této třídě. Síť má za úkol naučit se predikovat odměny dle zvolené akce.

```
1  -- Počet neuronů v rekurentní síti
2  local neuronCount = 512
3  -- Počet výstupních akcí
4  local actionCount = 3
5  -- Velikost historie v modulu LSTM
6  local historySize = 100
7
8  -- Vytvoření konvoluční neuronové sítě
9  local convolution = nn.Sequential()
10   :add(nn.ParallelTable()
11     :add(nn.Sequential()
12       :add(nn.SpatialConvolution(2, 16, 4, 1, 1, (4-1)/2, 0))
13       :add(nn.ReLU())
14       :add(nn.SpatialConvolution(16, 32, 2, 1, 1, (4-1)/2, 0))
15       :add(nn.ReLU())
16       :add(nn.View(-1, 27))
17       :add(nn.View(-1, 32*27))))
18   :add(nn.JoinTable(2))
19   :add(nn.Linear(32*27, neuronCount))
20   :add(nn.Dropout(0.4))
21
22 -- Vytvoření rekurentní neuronové sítě
23 local rnnModule = nn.Sequential()
24 rnn:add(nn.LSTM(neuronCount, neuronCount, historySize))
25 rnn:add(nn.Dropout(0.4))
26
27 -- Spojení konvoluční neuronové sítě s rekurentní neuronovou sítí
     plus přidání výstupní vrstvy
28 local network = nn.Sequential()
29   :add(convolution)
30   :add(rnnModule)
31   :add(nn.Linear(neuronCount, actionCount))
32
33 -- Vložení celé neuronové sítě do rekurentního kontejneru sequnceru
34 local sequenceNetwork = nn.Sequencer(network)
35
36 -- Inicialize vah podle specifických algoritmů v Torchi
37 sequenceNetwork:reset()
```

Zdrojový kód 1: Vytvoření neuronové sítě

Pomocí tohoto kódu je vytvořena obdobná síť jako na obrázku. 10. V tomto kódu chybí vrstva pro přidání proměnných robota. Proměnné robota v základu nepotřebujeme, proto je prozatím vynechám.

Na řádku číslo 10 je vytvoření konvoluční vrstvy. Lze zde přidávat různé moduly pro práci s konvolučními sítěmi. Vytvořená konvoluční vrstva přijímá

dva kanály. První kanál se používá pro identifikaci objektu a druhý kanál je pro vzdálenost objektu.

Vytvořená neuronová síť bude přijímat tensor o rozměrech $1 \times h \times n \times 2 \times 1 \times 31$. Proměnná n značí velikost batche a h velikost sekvence. První číslo 1 z rozměrů znamená, že je vstupem pouze jeden vstup, a to vizuální. Kdybychom chtěli přidat parametry robota, tak budou dva vstupy.

Pro určení chyby také potřebujeme definovat chybovou funkci. Chybová funkce bude v základu používat mean square error funkci. K tomu poslouží kód 2.

```
1 local criterion = nn.MSECriterion()
2 sequenceCriterion = nn.SequencerCriterion(criterion)
```

Zdrojový kód 2: Vytvoření chybové funkce

6.2.2 Pohyb robota pomocí umělé inteligence

Vytvořená neuronová síť má několik výstupů. Každý výstup je napojen na nějakou akci. Ve zdrojovém kódu 3 je ukázka volení akce s ϵ–greedy strategií výběru. S určitou náhodou se vybere náhodná akce. Jinak se zvolí akce navázaná na největší predikovanou odměnu.

```
1  sequenceNetwork:evaluate()
2  local output = sequenceNetwork:forward({{input}})
3  local tmp, out = torch.max(output[1][1], 1)
4
5  if math.random(0,100) < random then
6    -- Znak hash znamená velikost mapy
7    local randAction = math.random(1, #actionsHashMap)
8    out = {randAction}
9  end
10 -- Mapování výstupu neuronové sítě na akci
11 local action = actionsHashMap[out[1]]
```

Zdrojový kód 3: Volba akce

6.2.3 Učení neuronové sítě

Učení neuronové sítě je představeno na zdrojovém kódu 4. Jako optimalizační funkce je zvolena *adagrad*. *Učící se konstanta* je nastavena na poměrně malou hodnotu 0.0002. Díky této malé hodnotě bude síť pomalu konvergovat, což je v našem případě vhodné, protože záleží na velmi malých číslech na výstupu. Jako vstup do neuronové sítě se vkládá *input tensor*, což je vstupní tensor, který je vytvořen z paměti. Tento vstup má v základu nastavenou velikost batche na 32.

Velikost sekvence je nastavená na 10. *RewardBatch* proměnná si udržuje odměny k jednotlivým stavům. Tyto odměny se pak používají při výpočtu target tensoru.

Pro určení *target tensoru*, tedy očekávaných výstupních hodnot neuronové sítě, slouží funkce *getEstimationRewards*. Funkce je implementovaná dle kapitoly *Predikování odměn*. V experimentech je používána ještě jedna implementace, která je vhodnější pro učení s učitelem. Funkce předpokládá, že učitel volil akce s maximálními hodnotami. Díky tomuto předpokladu nemusíme odhadovat maximální odměnu, tato hodnota lze spočítat přesně.

```
1  -- Nastavení trénovacího módu
2  sequenceNetwork:training()
3  local parameters, gradParameters = sequenceNetwork:getParameters()
4  -- Použití optimalizační funkce rmsprop
5  optim.adagrad(function(params)
6      sequenceNetwork:gradParamClip(5)
7      -- Predikování odměn
8      local predictedReward = sequenceNetwork:forward(inputTensor)
9      -- Vytvoření target tensoru
10     local targetReward = getEstimationRewards(rewardBatch,
           predictedReward, inputTensor)
11         -- Spočítání chyby
12     local err = sequenceCriterion:forward(predictedReward,
           targetReward)
13     sequenceNetwork:zeroGradParameters()
14     -- Hloubka backpropagation vzhledem k rekurentním sítím
15     sequenceNetwork:maxBPTTstep(5)
16     -- Vytvoření gradientů pro zpětný chod
17     local gradOutputs = sequenceCriterion:backward(predictedReward,
           targetReward)
18     -- Zpětný chod nad neuronovou sít'í
19     local gradInputs = sequenceNetwork:backward(inputTensor,
           gradOutputs)
20
21     return err, gradParameters
22   end,
23   parameters,
24   -- Nastavení learning ratu.
25   {learningRate = 0.0002})
26 -- Zapomenutí sekvencí kvůli rekurentním sítím
27 sequenceNetwork:forget()
```

Zdrojový kód 4: Vytvoření neuronové sítě

6.3 Třída Paměť

Pro učení neuronové sítě je zapotřebí množina dat. Aby se neuronová síť mohla zlepšovat, je důležité mít rozsáhlou množinu dat. Při spuštění robota ve virtuálním prostředí se automaticky ukládá do paměti struktura obsahující:

- vstupy agentovi s
- odměna r
- zvolená akce a
- jestli je konec hry

Díky posloupnosti těchto struktur lze reprodukovat průchod robota prostředím. Tato třída serializuje sekvenci těchto struktur na disk. Z uživatelského rozhraní lze následně nastavit délku uložených her, tedy velikost trénovací množiny. S malou trénovací množinou neuronová síť nedokáže zobecnit vstupy.

7 Uživatelské rozhraní aplikace

Aplikace s virtuálním prostředím robota obsahuje uživatelské rozhraní. Po spuštění aplikace se zobrazí menu obsahující tyto položky:

- *Examples*: zde jsou ukázány příklady natrénovaných neuronových sítí.
- *New Game*: slouží pro spuštění trénování robota a testování jeho úspěšnosti.
- *Create AI*: slouží pro vytvoření nové umělé inteligence s různými parametry pro neuronovou síť.
- *Reinforcement settings*: slouží pro nastavování parametrů odměn a velikosti paměti.
- *Quit*: slouží pro ukončení aplikace.

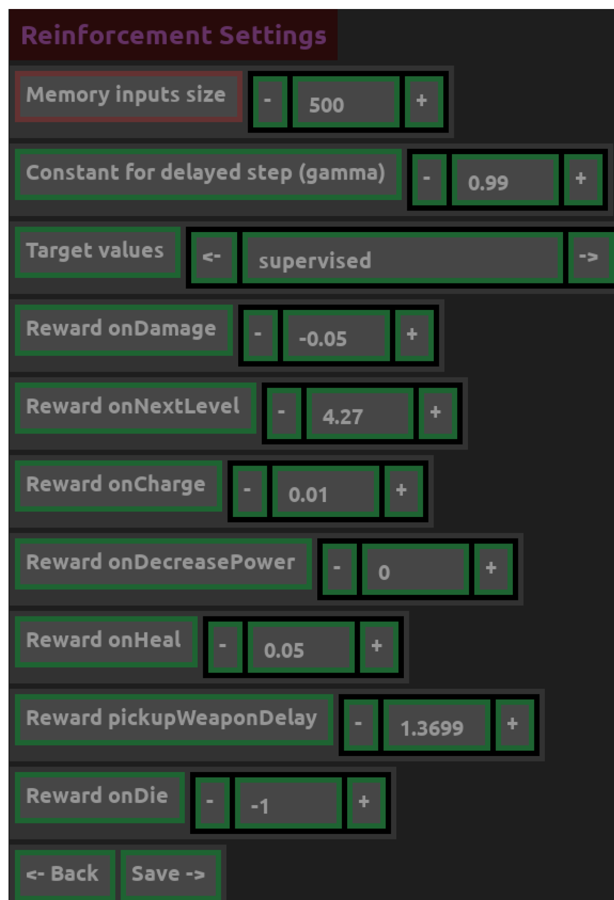
7.1 Nastavení parametrů odměn a parametrů paměti

Po kliknutí na tlačítko *Reinforcement settings* se otevře okno jako na obrázku 16.

Jednotlivé položky z obrázku 16 mají následující význam:

- *Memory input size:* určuje počet her, který se ukládá a slouží jako dataset pro neuronovou síť.
- *Constant for delayed step (gamma):* nastavuje γ pro výpočet zpožděné odměny.
- *Target values*: nastavuje funkci určující očekávané výstupní hodnoty neuronové sítě. S těmito hodnotami se určuje chyba. V nastavení jsou dvě metody *supervised* a *unsupervised*. Obě metody jsou vysvětlené v kapitole *Učení neuronové sítě*, kde *unsupervised* zastupuje standartní metodu určování *target tensoru* a *supervised* je experimentální pro učení s učitelem.
- *Reward onDamage:* je velikost odměny po obdržení zranění robota.
- *Reward onNextLevel:* je velikost odměny, když robot dosáhne výtahu v módu elevator.
- *Reward onCharge:* je velikost odměny, když robot obdrží energii.
- *Reward onDecreasePower:* je velikost odměny po odečtení energie robota.
- *Reward onHeal:* je velikost odměny, když robot obdrží životy.
- *Reward pickupWeaponDelay:* je velikost odměny po zvednutí zbraně.
- *Reward onDie:* je velikost odměny po smrti robota.

Parametry se ukládají do souboru, který lze upravit v textovém editoru. Soubor je v adresáři “settings” a jméno souboru je “settings.lua”.



Obrázek 16: Okno pro nastavení odměn a parametrů paměti

7.2 Tvorba vlastní sítě

Pro tvorbu vlastní neuronové sítě slouží tlačítko z menu *Create AI*. Po kliknutí na tlačítko se zobrazí okno na obrázku 17.

Jednotlivé položky z obrázku 17 mají následující význam:

- *Name:* je to standartní "input box". Po kliknutí do něj můžeme napsat název, pod kterým se uloží nová umělá inteligence.
- *Mode:* určuje mód, pro který je umělá inteligence vytvořená.
- *Criterion:* je chybová funkce, lze vybrat z několika funkcí.
- *Reccurent network:* zde lze vybrat z několika typů rekurentních neuronových sítí. Tento typ se bude používat v neuronové síti.
- *Reccurent layers:* je počet vrstev rekurentních sítí.
- *Neuron unit:* je počet neuronů v rekurentní neuronové síti.
- *Sequence length:* je velikost sekvence vstupů do rekurentní neuronové sítě.

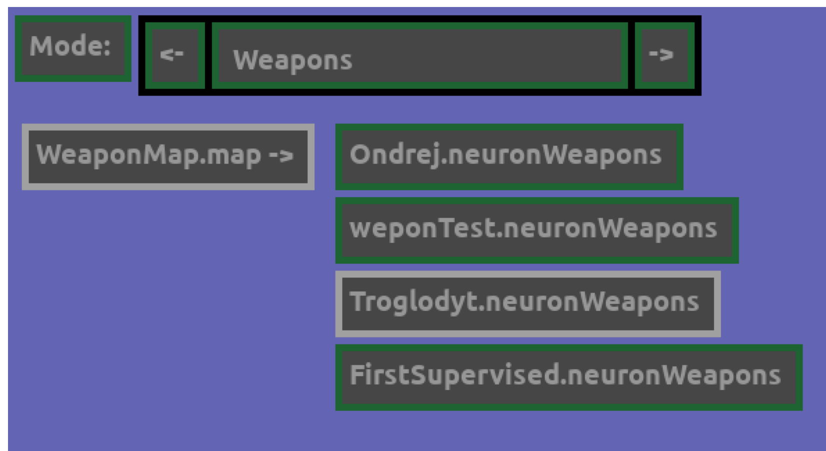
Obrázek 17: Okno pro tvorbu nové umělé inteligence

- *Batch size:* je velikost batche, se kterou se bude trénovat neuronová síť.
- *Learning rate:* je učící se konstanta.

Nově vytvořenou umělou inteligenci pak lze použít. Tyto parametry se ukládají do adresáře "neuron". Parametry můžeme upravovat v souboru se jménem složeného pomocí jména umělé inteligence, módu a postfixem "meta".

7.3 Trénování robota, testování robota a příklady natrénovaných sítí

Pro spuštění trénovacího a testovacího režimu robota slouží tlačítko *New game*. Tlačítko *Examples* je pro příklady natrénovaných sítí. Po kliknutí na tyto tlačítka se zobrazí okno podobné jako na obrázku 18.



Obrázek 18: Okno výběru prostředí hry

Na obrázku 18 první položka *mode* nastavuje mód, ve kterém se spustí prostředí. Dále se okno rozděluje na dva sloupce. První sloupec jsou mapy a druhý sloupec jsou uložené umělé inteligence. Pro spuštění prostředí musíme zakliknout mapu a umělou inteligenci. Na obrázku 18 je vybraný mód *Weapons*, mapa *WeaponMap* a umělá inteligence *Troglodyt*. Ve spodní části se nachází tlačítko *continue*, které spouští prostředí (na obrázku není vidět). Z důvodu chybějící komponenty *scrollbox* se může stát, že se položky nevejdou na obrazovku. Uložené neuronové sítě pak lze mazat z adresáře "neuron", pro smazání je potřeba smazat všechny soubory s názvem neuronové sítě.

Na obrázku 19 je představeno virtuální prostředí s uživatelským rozhraním. Uživatelské rozhraní se nachází v pravé části okna. První tři položky jsou takzvané "status bary". První zobrazuje životy robota, druhý energii a třetí počet nábojů. Náboje se v módech nepoužívají.

Další položky jsou:

- *Error:* je sečtená velikost chyb, jaká nastala v odhadu odměn při trénování.
- *Rand action:* nastavuje velikost náhodné akce v procentech. Náhoda automaticky klesá po několika trénováních.
- *Train iters:* nastavuje počet trénovacích cyklů po smrti nebo vybití robota.
- *Tlačítko Train:* přepíná mezi trénováním a evaluací. Při módu evaluace se vynechává trénování a náhoda se nastaví na nulu.
- *Score:* ukazuje, jaké úspěšnosti agent dosahuje v konkrétním kole.
- *Best score:* je nejlepší skóre agenta.

Virtuální prostředí umožňuje ovládání robota pomocí klávesnice. Člověk tedy může ovládat robota místo agenta. V tu chvíli nastává učení s učitelem. Ovládání je následující:

Obrázek 19: Okno výběru prostředí hry

- *šipka nahoru:* robot jde dopředu
- *šipka doprava:* robot se otáčí doprava
- *šipka doleva:* robot se otáčí doleva
- *E:* robot vezme zbraň

Pokud je vybrána neplatná akce, použije se akce agenta. Robot ve skutečnosti může dělat více akcí, ale v těchto módech nejsou potřeba.

8 Experimenty

Experimenty představené v této kapitole budou demonstrovat účinnost zvolených metod. Metody budou ukázány na dvou módech. Tyto módy byly představeny v kapitole *Virtuální prostředí robota*. Jako první bude ukázána metoda bez pomocí učitele a následně pak s učitelem. Při těchto experimentech se postupně bude snižovat náhoda z 98% na 10%. Hodnoty odměn byly nastaveny následovně:

- Velikost ukládané paměti byla 500 her.
- *onNextLevel*: 4.27
- *onHeal*: 0.05
- *pickupWeaponDelay*: 1.3
- *onDamage*: 0.05
- *gamma*: 0.99

Neuronová síť byla nastavená následovně:

- Chybová funkce: *SmoothL1Criterion*.
- Rekurentní síť: *FastLSTM*.
- Rekurentní vrstvy: 2.
- Velikost sekvence: 9.
- Velikost batche: 16.
- Učící konstanta: 0.0002

V této kapitole následující grafy zobrazují závislost skóre dosaženého agentem na počtu učících iterací.

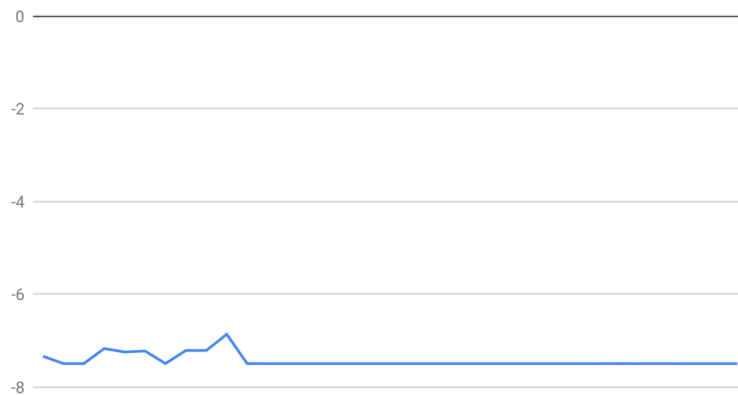
8.1 Experimentování s učením bez učitele

Experimentování bez učitele probíhalo, že jsem spustil prostředí robota a robot se učil sám plnit úkoly. Při tomto učení byla v *Reinforcement settings* nastavena proměnná *Target value* na *unsupervised*. Výsledky jsou zobrazeny pomocí grafu, který vybírá průměr skóre z každých 50 iterací učení. Tyto experimenty z hlediska učení nebyly úspěšné.

8.1.1 Mód Elevator

Byla zvolená mapa *LavaMapq.map* a odehráno 1724 epizod. Dle grafu 20 je vidět, že experiment dopadl neúspěšně. Robot se nenaučil zvyšovat skóre.

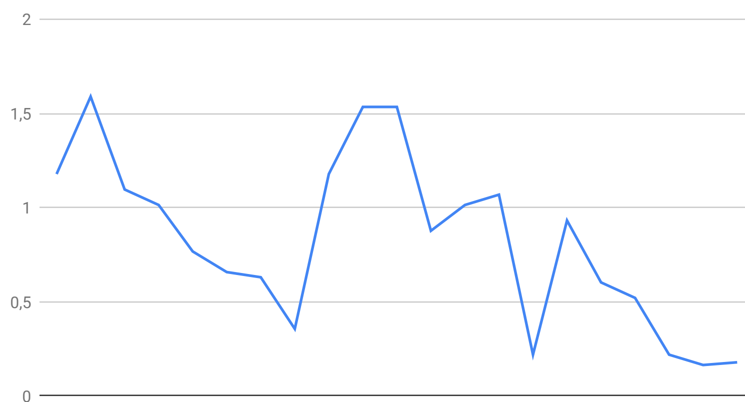
Experiment by bylo zajímavé provést na větším počtu trénovacích iterací.



Obrázek 20: Graf učení bez učitele v módu *Elevator*.

8.1.2 Mód Weapons

Byla zvolená mapa *WeaponMap.map* a odehráno 1536 epizod. Učení pro tento mód dopadlo neúspěšně, jak ukazuje graf 21. Robot se nenaučil zvedat zbraně, které se náhodně objevovaly.



Obrázek 21: Graf učení bez učitele v módu *Weapons*.

8.2 Experimentování s učením s učitelem

Experimenty probíhaly podobně jako bez učitele jen s tím rozdílem, že jsem nahrazoval strategii výběru akce určitou dobu. Tudíž jsem předpřipravil množinu dat, která se kombinovala s vytvořenými pomocí strategie výběru akce. Dále pak při tomto učení byla v *Reinforcement settings* nastavena proměnná *Target value*

na *supervised*. Výsledky jsou zobrazeny pomocí grafu, který vybírá medián skóre z každých 50 iterací učení.

8.2.1 Mód Elevator

Byla zvolená mapa *LavaMapq.map* a odehráno 750 epizod. Na grafu 22 lze vidět, že robot se naučil chodit do cíle.

Obrázek 22: Graf učení s učitelem v módu *Elevator*.

8.2.2 Mód Weapons

Byla zvolená mapa *WeaponMap.map* a odehráno 439 epizod. Na grafu 23 lze vidět, že robot se naučil zvedat zbraně.

Obrázek 23: Graf učení s učitelem v módu *Weapons*.

Závěr

Výstupem této diplomové práce byla aplikace s implementovaným virtuálním prostředím robota a umělou inteligencí. Virtuální prostředí robota bylo vytvořeno s podporou několika módů pro experimentování a umělou inteligencí. Umělá inteligence byla založená na reinforcement learning metodě. Pomocí této metody bylo dosaženo učení robota ve virtuálním prostředí. Metoda v tomto podání nebyla zcela efektivní především v *unsupervised* módu. V *supervised* módu bylo dosaženo lepšího výsledku, při kterém byl robot schopen zvyšovat skóre. Efektivnost této metody je založena na mnoha faktorech. Zajímavé by bylo dále experimentovat s nastavením neuronové sítě a vstupem do neuronové sítě.

Conclusions

Output of this master theses was application with implemented robot's enviroment and artificial intelligence. Robot's enviroment was created with some modes for experimenting around artificial intelligence. Artificial intelligence was based on reinforcement learning method. Robot was trained via this method in the enviroment. The method was not so effective in my way of implementation mainly in unsupervised mode. In supervised mode was better results robot was able to increase his score. Effectivnes of this method is based on many factors. Interesting can be more experiment with neural network and with input for the neural network.

A Obsah přiloženého CD/DVD

Na samotném konci textu práce je uveden stručný popis obsahu přiloženého CD/DVD, tj. jeho závazné adresářové struktury, důležitých souborů apod.

`image/`
: *Virtual image* pro aplikaci *Virtual box.* V *imagu* je nainstalován systém *Xubuntu 16.04* v 64bitové verzi. Po spuštění image je na ploše skript *Robots*, který spouští aplikaci.

`doc/`
: Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

`SmartRobots/`
: Kompletní aplikace s prostředím pro vývoj.

`SmartRobots/install-deps.sh`
: Skript pro nainstalování závislostí. Aplikace byla testována a vyvíjena na operačním systém Xubuntu 16.04 v 64bitové verzi, z toho důvodu vřele doporučuji použít tento systém. Od verzí Xubuntu 18.04 aplikace padá z důvodu použitého *Torch frameworku.*

`SmartRobots/src`
: Kompletní zdrojové kódy aplikace.

`readme.txt`
: Instrukce pro instalaci a spuštění programu.

U veškerých cizích převzatých materiálů obsažených na CD/DVD jejich zahrnutí dovolují podmínky pro jejich šíření nebo přiložený souhlas držitele copyrightu. Pro všechny použité (a citované) materiály, u kterých toto není splněno a nejsou tak obsaženy na CD/DVD, je uveden jejich zdroj (např. webová adresa) v bibliografii nebo textu práce nebo v souboru `readme.txt`.

Literatura

[1] *Torch, A SCIENTIFIC COMPUTING FRAMEWORK FOR LUAJIT*. Dostupný z: ⟨http://torch.ch/⟩.

[2] NICHOLAS, Leonard; WAGHMARE, Sagar; WANG, Yang; KIM, Jin-Hwa. *rnn : Recurrent Library for Torch*. 2015. Dostupný z: ⟨https://arxiv.org/abs/1511.07889⟩.

[3] BHATT, Shweta (ed.). *5 Things You Need to Know about Reinforcement Learning*. 2018. Dostupný z: ⟨https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html⟩.

[4] BROCKMAN, Greg; CHEUNG, Vicki; PETTERSSON, Ludwig, et al. *OpenAI Gym*. 2016. Dostupný také z: ⟨eprint: arXiv:1606.01540⟩.

[5] ARTHUR, Juliani (ed.). *Simple Reinforcement Learning with Tensorflow Part 0: Q-Learning with Tables and Neural Networks*. 2016. Dostupný z: ⟨https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0⟩.

[6] NIELSEN, Michael. *Neural Networks and Deep Learning*. 2015. Dostupný z: ⟨http://neuralnetworksanddeeplearning.com/⟩.

[7] MOAWAD, Assaad. *Neural networks and back-propagation explained in a simple way*. 2018. Dostupný z: ⟨https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e⟩.

[8] *Convolutional Neural Networks (CNNs / ConvNets)*. Dostupný z: ⟨http://cs231n.github.io/convolutional-networks/⟩.

[9] NGUYEN, Michael. *Illustrated Guide to LSTM's and GRU's: A step by step explanation*. 2018. Dostupný z: ⟨https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21⟩.

[10] PROCHÁZKA, Ondřej. *Integrace skriptovacího jazyka Lua do herního enginu*. 2015.

[11] ANDREW G. BARTO, Richard S. Sutton a. *Reinforcement Learning: An Introduction*. Second. Cambridge (Mass): The MIT Press, 2018. 548 s. ISBN 9780262039246.

[12] *A Beginner's Guide to Deep Reinforcement Learning*. Dostupný z: ⟨https://skymind.ai/wiki/deep-reinforcement-learning⟩.

[13] OLAH, Christopher (ed.). *Understanding LSTM Networks*. 2015. Dostupný z: ⟨http://colah.github.io/posts/2015-08-Understanding-LSTMs/⟩.

[14] WIKIPEDIA CONTRIBUTORS. *Long short-term memory — Wikipedia, The Free Encyclopedia*. 2019. [Online; accessed 31-July-2019]. Dostupný také z: ⟨https://en.wikipedia.org/w/index.php?title=Long_short-term_memory&oldid=907761627⟩.

[15] DEVENDRA SINGH CHAPLOT, Guillaume Lample a. Playing FPS Games with Deep Reinforcement Learning. 2018. Dostupný také z: ⟨https://arxiv.org/pdf/1609.05521.pdf⟩.

[16] RUDER, Sebastian. *An overview of gradient descent optimization algorithms.* 2016. Dostupný z: ⟨http://ruder.io/optimizing-gradient-descent/index.html#rmsprop⟩.

[17] GANDHI, Rohith. *A Look at Gradient Descent and RMSprop Optimizers.* 2018. Dostupný z: ⟨https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b⟩.

[18] ANDRÉ LAMOTHE, Mat Buckland a. *AI techniques for game programming.* United States of America: Stacy L. Hiquet, 2002. 481 s. ISBN 1-931841-08-X.