VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

HLBOKÉ UČENIE AI V HERNÝCH PROSTREDIACH

DEEP LEARNING AI IN GAME ENVIRONMENTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **KRISTIÁN GLÓS**
AUTHOR

VEDOUCÍ PRÁCE **Ing. TOMÁŠ POLÁŠEK,**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Glós Kristián**
Program: Informační technologie
Název: **Hluboké učení AI v herních prostředích**
Deep Learning AI in Game Environments
Kategorie: Umělá inteligence

Zadání:

1. Seznamte se s herním enginem Unity v rámci tvorby herních prostředí.
2. Seznamte se s algoritmy hlubokého učení v oboru AI a vyberte vhodné kandidáty pro učení agentů herního prostředí.
3. Vytvořte vhodné herní scény v Unity zahrnující rozličná prostředí. Implementujte zvolené algoritmy a experimentuje s jejich parametry.
4. Prozkoumejte efektivnost algoritmů v různých scénářích a výsledky porovnejte.
5. Prezentujte výsledky pomocí plakátu a krátkého videa.

Literatura:

- Arulkumaran, Kai, et al. "A brief survey of deep reinforcement learning." *arXiv preprint arXiv:1708.05866* (2017).
- Floreano, Dario, Peter Dürr, and Claudio Mattiussi. "Neuroevolution: from architectures to learning." *Evolutionary intelligence* 1.1 (2008): 47-62.
- Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." *arXiv preprint arXiv:1509.06461* (2015).
- Dále dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a částečně bod 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Polášek Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 30. října 2020

Abstrakt

Práca sa zaoberá analýzou algoritmov hlbokého učenia a ich schopností splňovať úlohy v herných prostrediach vývojového prostredia Unity. Ďalej sa pokúša vyhľadávať a špecifikovať možné využitia hlbokého učenia a neurónových sietí pre využitie pri vývoji počítačových hier. Využívame na to posilované učenie, imitačné učenie, a neuroevolúciu, pričom sa posilované učenie používalo počas celého vývoja hernej scény. Vyhodnocovanie a analýza prebiehali púštaním sietí v rôznych podmienkach herných scén a iných faktorov.

Abstract

This thesis is focused on analysing deep learning algorithms and their ability to complete given tasks implemented in game environments created via the Unity game engine. Secondary objective was to research and specify possible use-cases of deep learning during game development. The algorithms used fall into Reinforcement learning, Imitation learning and Neuroevolution, while Reinforcement learning was used throughout the whole game scene development cycle. Analysis and results were collected through training the networks in different game scene states and other factors.

Kľúčové slová

Hlboké učenie, Strojové učenie, Umelá inteligencia, Posilované učenie, Imitačné učenie, GAIL, Q-učenie, PPO, knihovňa ML-agents, Unity, Herné prostredia, Agenti, Strojové učenie, odmeny, odmeny a tresty, Genetické algoritmy, neuroevolúcia, NEAT, Evolučné stratégie, Markovské rozhodovacie procesy, neurónové siete

Keywords

Deep Learning, Machine learning, AI, Reinforcement Learning, Imitation Learning, GAIL, Q-Learning, Proximal Policy Optimalization, ML-agents, Unity, Game environments, Agents, rewards and penalties, Genetic Algorithms, Neuroevolution, Neuroevolution of Augmented Topologies, Evolution Strategies, Markov Decision Process, Neural network

Citácia

GLÓS, Kristián. *Hlboké učenie AI v herných prostrediach.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Polášek,

Hlboké učenie AI v herných prostrediach

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Tomáša Poláška.
Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Kristián Glós
19. mája 2021

Poďakovanie

Ďakujem Ing. Tomášovi Poláškovi sa poskytnutie príležitosti na vypracovanie práce a poskytovanie odbornej pomoci.

Obsah

Kapitola 1

Motivácia a ciele práce

Hlboké učenie je veľmi populárnou témou v dnešnej dobe a užíva si hromadnej adopcie do rôznych kútov technologického sveta. Jedno z týchto odvetví v ktorom sa táto veda v posledných rokoch rozširuje [7] je aj vývoj počítačových hier. Verím, že hľadanie a definovanie všetkých možných prípadov využitia je prospešné k budúcemu vývoju hier, či už ako náhrada tradičnej umelej inteligencie využívanej po celé dekády, alebo ako nástroj na zrýchlenie a zjednodušenie pri vytváraní herných scén a ich logík.

Cieľom práce bolo navhrnúť herné prostredia nad ktorými sa implementujú rôzne algoritmy hlbokého učenia. Výsledkom je analýza daných algoritmov, ich vlastností, možnosť využitia v daných úlohách. Existujú totižto rôzne výhody a nevýhody na ktoré sa dá pri práci s danými algoritmami naraziť, či naopak vyhnúť, čo naznačuje potrebu správneho výberu. Tieto parametre boli vyjadrené praktickou formou v hernom engine Unity, kde hrali úlohu ako logika poháňajúca AI na dokončenie hernej úlohy, či už pri úlohe lineárneho charakteru alebo náhodne vygenerovaného prostredia. Analyzované takýmto postupom boli populárnejšie algoritmy ako sú Reinforcement Learning a Neuroevolution, či taktiež Imitation Learning ktorý bol jednoducho implementovateľný do už existujúceho modelu, a kompatibilný s RL ako možnosť kompromisu za ziskom určitých výhod. Knižnice s podporou analyzovaných algoritmov sú voľne dostupné a samotné algoritmy jednoduché na implementáciu, ak využijeme ich silné stránky a vyvarujeme sa úlohám u ktorých nepodávajú dobré výsledky oproti iným možnostiam.

Kapitola 2

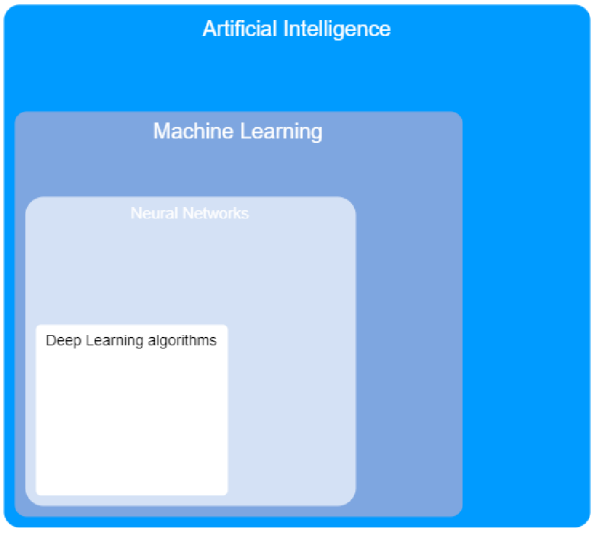
Teória

Umelá inteligencia hrá v hernom priemysle obrovskú rolu už od jeho počiatku. Od ovládania entít, po kontrolu herných mechaník a systémov, až k pseudo-náhodnej generácií herných prostredí. Tieto vykonávajú pevne definované úlohy, a sú obmedzené pevne stanovenými pravidlami. Ako príklad môže byť AI v protivníkoch single-player FPS hier. Tieto dokážu automaticky zasiahnuť protivníka každou vystrelenou guľkou v momente ako im to hra dovolí, no zámerne sa do nich programujú pravidlá, ktoré im nepovoľujú operovať na plný potenciál. Z toho môžu vyplývať rôzne problémy, ako že sa tento protivník nespráva realisticky a tým narušuje imerziu, teda vcítenie sa hráča do hry všetkými smyslami. Iným obmedzením je to, že ak by sme sa snažili tradičnými algoritmami AI donútiť počítať hrať šach perfektne, len prvý ťah by trval dlhšie, ako už existujúca história celého vesmíru[23]. Na obidva prípady existuje mnoho riešení, a práve jedno z nich je implementácia hlbokého učenia. V prvom prípade sa počítač môže naučiť hrať ako človek, operovať pod rovnakými princípmi a byť poháňaný rovnakou motiváciou. V druhom prípade nie sú algoritmy hlbokého učenia viazané svojim kódom tak, ako tradičné postupy. To viedlo k tomu, že mladá, trojročná neurónová sieť Leela Chess Zero dokázala poraziť [8] 12 ročný tradičný chess engine ktorý bol najlepším hráčom na svete zvaný Stockfish či byť konzistentne druhým najlepším hráčom v dlhších herných formátoch.

Taktiež už existujú aj vývojové využitia hlbokého učenia, ako je technológia DLSS od firmy Nvidia, ktorá dokáže hru na nižšom rozlíšení vykresľovať na rozlíšení vyššom ako toto natívne pomocou hlbokých neurónových sietí, čo sa v skutočnosti prejavuje ako bonusový výkon v hrách pri minimálnej strate grafickej integrity [14]. Dokonca už existujú patenty [15] v ktorých hlboké učenie nahrádza samotného hráča. Nie je teda príliš kontroverzné povedať, že hlboké učenie bude v budúcnosti neustále viacej neoddeliteľnou súčasťou vývoja počítačových hier. V tejto kapitole si vyjasníme v čom by takáto integrácia mohla spočívať.

2.1 Umelá inteligencia alebo hlboké učenie?

Najlepšou cestou ako si predstaviť spojitosť umelej inteligencie a hlbokého učenia je systém matriošiek. Strojové učenie je oblasťou umelej inteligencie. Hlboké učenie potom je jednou z oblastí strojového učenia, ktoré využívajú neurónové siete [11].



Obr. 2.1: Metódy učenia

2.2 Strojové učenie

Strojové učenie je oblasť umelej inteligencie zaoberajúca sa konceptom schopnosti počítačového programu adaptovať a zlepšovať sa k jemu určenej úlohe za využitia nazbieraných vedomostí, skúseností, a iných vonkajších vjavov bez ľudského zásahu. Algoritmy sú cvičené na obrovskom množstve dát k vyhľadávaniu vzorov a spojitostí na rozhodnutia a dokonca predpovede riešení prichádzajúcich nových dát [5].

2.3 Neurónové siete

Pre človeka je jednoduché rozpoznať vzory a motívy, ako je napríklad písané písmo. Preto si ani neuvedomujeme, ako zložité to reálne je a za čo všetko by sme mali byť vďačný superpočítaču ktorý sa nachádza v našej lebke. Totižto hneď ako oddelíme milióny rokov tréningu rozpoznávať malé detaily a prideľovať k ním arbitrárne významy, hneď by bolo zložité rozpoznať čísla alebo písmená nachádzajúce sa či na papieri alebo niekde na fotke. To je len jeden z príkladov, čo ale vyžadujeme od algoritmov hlbokého učenia. Namiesto toho, aby sme počítaču presne určili postup pravidiel alebo príkazov čo tvorí číslo 9, len na to aby sme narazili na vysoké množstvo výnimiek alebo iných protikladov, mu ukážeme mu ukážeme tisícky či milióny ukážkových obrázkov, ktoré ľudia za číslo 9 považujú, a počítač necháme vyvodiť si závery a doslova sa naučiť význam pár čiar na obrázku [13].

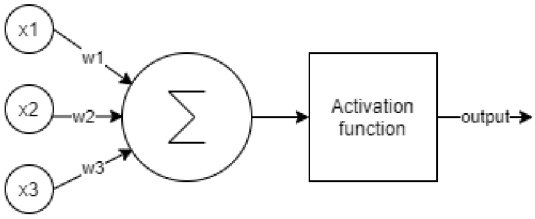
2.3.1 Perceptron

Neurónové siete sa tento problém snažia vyriešiť napodobnením funkcionality ľudského mozgu, respektíve zavedením neurónových uzlov a ich prepojení. Prvým príkladom takýchto umelých, simulovaných neurónov a ich väzieb boli perceptrony. Tie boli naprogramované už v 50. až 60. rokoch vedcom Frankom Rosenblatt [21]. Ich základom je séria jedného alebo viac binárnych vstupov s pridelenými váhami, ktoré určujú ako dôležitá je hodnota ktorú držia. Vypočíta sa hodnota sumy vstupov násobeních ich váhou, a perceptron predá na výstup hodnotu binárneho charakteru na základe určenej hranice, taktiež zvanej aktivačná

funkcia. Tá môže byť matematicky zapísaná:

$$\text{výstup} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ hranica} \\ 1 & \text{if } \sum_j w_j x_j > \text{ hranica} \end{cases}$$

kde w je váha vstupu, x je jeho hodnota a j vyjadruje počet vstupov.



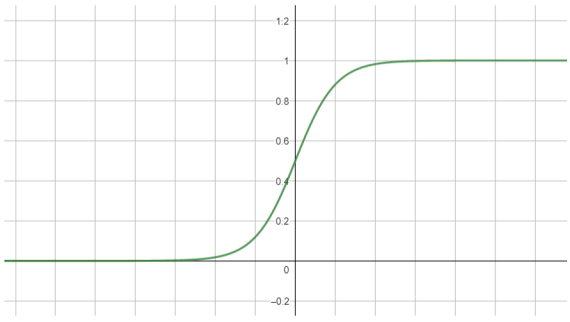
Obr. 2.2: Perceptron

2.3.2 Sigmoid neuron

Problém perceptronov vzniká v ich binárnom charaktere. Predstavme si to na príklade. Človek sa rozhoduje kúpiť si auto na základe úspor. Ak jeho úspory presahujú hodnotu 20000 eur, auto si kúpi. To sa dá vyjadriť perceptronom s jedným vstupom—úspory. Ak má tento človek 50000 eur, auto si bez problémov kúpi. Rovnako si ho ale bezproblémovo kúpi ak má 20000 eur, no keď má 19999 eur a 99 centov, v tomto momente je to auto príliš drahé a auto si nekúpi [24]. Toto naznačuje, že perceptron je príliš náchylný na minimálne zmeny blízko jeho hranice, čo potom aj znamená, že sa tieto dokážu propagovať do celého systému a ovplyvniť ho. Na to boli vyvinuté sigmoid neuróny. Tie od perceptronov najmä líši ich aktivačná funkcia, a teda to, že výstupom môže byť každé reálne číslo medzi 0 a 1. Toto číslo potom predstavuje pravdepodobnosť, že sa neurón aktivuje [24]. Názov sigmoid neuron tiež spočíva z faktu, že aktivačná funkcia je vlastne logistickou funkciou s esovitým (sigmoidným tvarom), zapísaná:

$$\mathrm{y} = \frac{1}{1 + \exp\left(-\sum_j w_j x_j - b\right)}$$

kde x,w,j sú rovnaké ako pri perceptrone, a b je **bias**, ktorý nahrádza úlohu hranice.



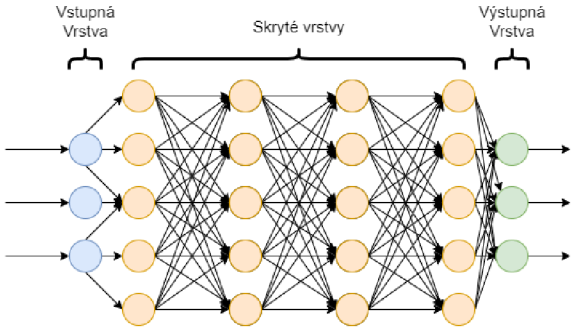
Obr. 2.3: Sigmoid funkcia

2.4 Hlboké učenie

Hlboké učenie je kategória strojového učenia zaoberajúca sa neurónovými sieťami s väčšou hĺbkou—mnoho neurónových vrstiev, okrem základných vstupových a výstupových vrstiev existujú **skryté vrstvy**. Ak má neurónova sieť takýchto vrstiev viac, jedná sa o hlbokú neurónovú sieť **DNN** (Deep Neural Network), a teda spadá pod vedu hlbokého učenia.

2.4.1 Skrytá vrstva

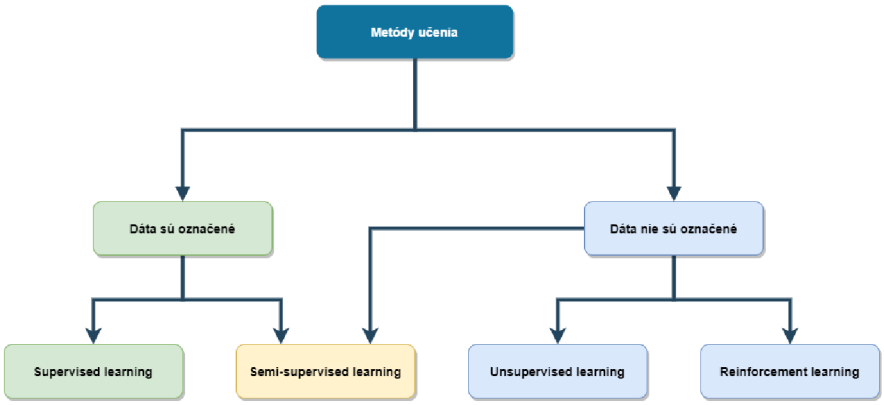
Je vlastne vrstva neurónov, ako napríklad sigmoid neurónov, každá táto vrstva v sieti vykonáva nelineárne transformácie dát, ktoré sme sieti predali ako vstupné. Taktiež sa každá špecializuje na jednu úlohu v sieti, napríklad u konvolučných sietí môže jedna vrstva identifikovať tvár, ďalšia oči a ďalšia nos, a predať pravdepodobnosť ich nálezu ďalším vrstvám. Výstupy každej vrstvy sa potom dajú spojiť k tomu, aby sme dokázali na fotke napríklad identifikovať človeka [3].



Obr. 2.4: model DNN

2.4.2 Typy učenia

Neurónové siete sa dajú trénovať rôznymi metódami, každá so svojimi výhodami a nevýhodami, využívané na rôzne účely. Hlavný rozdiel medzi nimi spočíva v akom stave predáme informácie sieti. Existujú 4 hlavné kategórie do ktorých ich je možno podľa tohto kritéria rozdeliť.



Obr. 2.5: Rozdelenie metód hlbokého učenia

2.4.3 Supervised learning

Supervised learning, teda učenie s učiteľom, spadá do kategórie učenia ktoré majú k dispozícii označené dáta. Sieť využíva trénovací dataset so správnymi odpoveďami v štádiu učenia, až do kým nedokáže vykonávať predpovede na nových, neoznačených dátach. Z toho ale nastáva potreba manuálne označiť dáta, čo môže byť celkom časovo náročné, nakoľko datasety pre neurónové siete ako sme už spomínali dokážu obsahovať až milióny príkladov. Pomocou tejto metódy dokážeme vytvoriť prevažne dva modely, klasifikačné a regresné.

Klasifikačný model

Klasifikačné modely sú populárne najmä vo svete konvolučných neurónových sietí učených na rozpoznávanie znakov a textu (OCR), ľudí na fotke a podobných úloh typu „rozpoznanie objektu na obrázku“. Model spočíva v tom, že jeho vstupom je konečná množina diskrétnych hodnôt, tiež zvaných triedy (classes), ktoré potom budú aj všetky možnosti výsledku ktoré bude možno zo siete dostať. Ak teda sieti dáme dataset iba so značkami pes a mačka, nedokázala by sama od seba identifikovať iné zviera na novom obrázku a len by dokázala vrátiť odpoveď toho, čomu sa podľa siete viacej podobá. Sieť dokážeme považovať za vyspelú v momente, ak dosiahneme nami určenú presnosť výsledkov na nových, neoznačených dátach. Tú dokážeme definovať pomocou stratovej funkcie **loss function** ktorá vypočítava chybu/stratu v sieti a podľa ktorej sieť dokážeme ďalej optimalizovať. Stratových funkcií je viacej typov, ako napríklad *Cross-entropy, Hinge, Mean Absolute Error* a podobne.

Regresný model

Regresné modely sú populárne tam, kde sa klasifikačných nedá využiť. Pracujú totižto so spojitými hodnotami namiesto diskrétnych, a teda ich predpoveďou nebýva jedna odpoveď ale zvyčajne odhad výsledku. Ich využitie si dokážeme predstaviť v predpovedaní veku osoby, ceny domu, či uhľu pod ktorým sa nosník zohne pri aplikácií presne určenej sily naň. To tiež znamená že sieť hodnotíme podľa odchýlky od očakávaného výsledku, a nie presnosťou správne odhadnutých výsledkov. Je teda potrebné si vybrať na túto úlohu správne stratové funkcie [2].

2.4.4 Semi-Supervised learning

Teda v doslovnom preklade čiastočne kontrolované učenie je učenie, kde je sieti poskytnutá len časť dát ktoré su označené, a nadmerná väčšina zostáva neoznačených. Využíva sa napríklad v prípadoch kde je dataset príliš veľký na označenie všetkých príkladov. Aj malá časť označených dát dokáže výrazne zvýšiť presnosť sietí s neoznačenými datasetmi. Jedna z populárnych implementácii tejto metódy sa nazýva General Adversarial Network [19].

General Adversarial Network

Funguje na základe pridelenia protivníka sieti. Tento protivník je vlastne ďalšia sieť, a obe sú postavené proti sebe v boji o prekabátenie tej druhej. Jedna sieť, zvaná generátor, má za úlohu vytvoriť nové dáta podobajúce sa už existujúcim, a druhá sieť, diskriminátor, má zase za úlohu zistiť, či sa jedná o pravý alebo falošný príklad dát.

2.4.5 Unsupervised learning

Učenie bez učiteľa slúži tam, kde sú dáta príliš zložité na označenie, neexistujú dostatočné označené datasety, či je označovanie inak obmedzené. V tomto prípade sú sieti predávané žiadnym spôsobom označené dáta bez explicitného cieľa alebo správnych odpovedí s ktorými sa musí vysporiadať sama. Sieť sa teda potom snaží analyzovať štruktúru predávaných dát, a získavaním užitočných vlastností podľa ktorých tieto dáta dokáže triediť. Triedenie môže prebiehať rôznymi spôsobmi podľa našej potreby.

Clustering

Clustering, teda zhlukovanie, spočíva vo vyhľadávaní a vytváraní zhlukov dát, teda skupín dát s podobnými vlastnostiami a vzormi. Populárny algoritmus zhlukovania je K-means, kde sa vyberú centrálne dáta, teda „vzory“ celého zhluku, a postupne sa k tomuto zhluku prideľujú dáta, ktoré mu majú najbližšie. Po každej iterácií sa tieto vzory prepočítavajú na hodnotu najbližšiu priemeru, a algoritmus postupuje do vtedy, pokiaľ sa už dáta v zhlukoch nepresúvajú. Využitie môže byť napríklad v biológií, identifikáciou tried zvierat, či dokonca žijúceho rodokmeňu.

Vyhľadávanie anomálií

Sieť nám týmto spôsobom dokáže vytriediť extrémy v datasete, respektíve dáta, ktoré určitými vlastnosťami alebo vzormi nesúhlasia so zvyškom našich dát.

Asociácia

Sieť vyhľadáva asociácie medzi vlastnosťami alebo vzormi, napríklad návrhy nových hier na základe histórie hernej knižnice hráča ako služba herných platforiem a podobne [19][27].

2.5 Reinforcement Learning

Metóda posilovaného učenia, či učenia so spätnou väzbou vyžaduje vlastnú sekciu, nakoľko je to jeden z prístupov ktoré sme sa rozhodli vybrať pre našu analýzu. Spadá do kategórie metód kde sa sieti nepredávajú dáta so žiadnym označením, no líši sa od učenia bez učiteľa tým, že sa nesnaží medzi dátami hľadať žiadnu štruktúru ani spojitosti. Sieť sa namiesto toho učí napríklad interakciou prostredia a **agenta**[2.6].

Agent

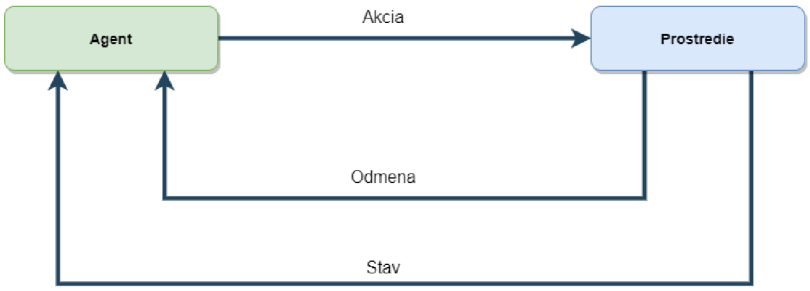
Agent je v odvetví umelej inteligencie akákoľvek entita, ktorá dokáže so svojím prostredím komunikovať a ho vnímať, pomocou senzorov, a taktiež v tomto prostredí konať, pomocou aktuátorov. Learning agent, s ktorým sa stretneme práve napríklad u Reinforcement learning, je agent, ktorý predstavuje neurónovú sieť schopný učiť sa zo svojich skúseností a adaptovať sa na situáciu v ktorej sa nachádza [1].

Podstata počítačových hier spočíva v systéme odmien za rôzne akcie. Ak hráme hru štýlu FPS, akciou je streliť nepriateľa, a odmenou je to, že sme ho zneškodnili. V turn-based stratégiách je cieľom vyhrať súboj za použitia zdrojov, teda akcií poskytovaných hráčovi. Pri adventure hrách môže byť cieľom vylúštenie hádanky a tým otvorenie truhly s odmenou, niekedy dokonca s ikonickou zvučkou, ktorá upevní resp. zvýši pocit odmeny. Ak

napríklad ale v týchto úlohách zlyháme, odmenu nedostaneme dokonca niekedy je naším trestom Game Over screen. Je v ľudskej povahe nerád prehrávať, a preto je prirodzené sa týmto omylom v budúcnosti vyhnúť, a učiť sa z našich skúseností, aby sme sa zlepšili.

Práve takto funguje aj posilované učenie. Agent má k dispozícií určité dáta o svojom prostredí, má pevne stanovené akcie s ktorými dokáže toto prostredie ovplyvňovať, a dostáva spätnú väzbu na základe jeho vykonaných akcií. Jeho cieľom je maximalizovať svoje odmeny, a práve preto ako my má motiváciu zbierať ich a vyhýbať sa trestom.

Jednou z výziev takéhoto prístupu je balancia medzi „exploration“ a „exploitation“. Exploration: prieskum, respektíve hľadanie nových stavov, je hlavným zdrojom učenia sa agenta. Exploitation: využitie svojich znalostí a vykonávanie akcií, ktoré agentovi v minulosti priniesli odmenu. Dilema spočíva v tom, že agent sa musí rozhodnúť medzi vykonávaním akcií, ktoré už pozná, a zostávať len v stavoch, ktoré už má osvedčené, alebo skúšať nové stavy, za pomoci nevyskúšaných akcií a riskovať stratu odmien či dokonca získanie trestov na to, aby mohol nájsť možné lepšie riešenia alebo nové ešte nezískané odmeny ktoré v budúcnosti agentovi prospejú. Agent sa ale nezaobíde bez obidvoch možností, bez toho aby nezlyhal v svojej úlohe. Ak by sa len držal akcií, ktoré pozná, nič by sa nenaučil, a uviazol by v stave najmenšieho odporu o ktorom si popíšeme neskôr, no taktiež nemôže stále skúšať len nové veci bez toho, aby aplikoval svoje vedomosti k vyberaniu vhodných akcií v správnych momentoch [28].



Obr. 2.6: Model interakcie agenta s prostredím u Reinforcement learning

Markovské rozhodovacie procesy

MDP (Markov Decision Process) často slúžia ako modely pre úlohy posilovaného učenia. Pomáhajú nám tieto problémy riešiť a definovať ich formálnou cestou. MDP sú definované ako usporiadaná pätica/tuple (S, A, P_a, R_a, γ) kde

- S je konečná množina všetkých stavov
- A je konečná množina všetkých akcií; $A_s \in A$ kde A_s je konečná množina všetkých akcií možných v stave s
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ je pravdepodobnosť že akcia a v stave s v čase t bude viesť do stavu s' v čase $t + 1$
- $R_a(s, s')$ je odmena za dostanie sa do stavu s' zo stavu s za vykonania akcie a
- $\gamma \in [0, 1]$, mieru dôležitosti, taktiež niekedy vynechanú, si čoskoro predstavíme [9].

2.5.1 Komponenty

Posilované učenie sa okrem samotného prostredia a agenta skladá z nasledujúcich komponentov: Policy, Reward, Value function a Model [28].

Reward

Odmena je cieľom celého posilované učenia a tiež motivácia za učením sa. V každom kroku trénovania, definovaný momentom kedy má agent na výber znova vykonať určitú akciu, sa predá agentovi číselná hodnota predstavujúca jeho odmenu. Môžme si túto hodnotu predstaviť ako endorfíny alebo receptory bolesti ovplyvňujú živočíchy. Endorfíny predstavujú kladné hodnoty, škálovateľné svojou intenzitou, naopak bolesť predstavuje záporné hodnoty, respektíve tresty. Agent sa podobne snaží vyhýbať bolesti a získavať odmeny, a to priamo svojimi vykonávanými akciami. Tieto akcie sa potom propagujú do agentovej taktiky, kde akcie s vysokými odmenami v danom stave sa do nej vložia, a akcie s nízkym stropom odmien, ba dokonca trestov sa v taktike nahradia novým prístupom. Ak teda je celkovým cieľom agenta maximalizovať jeho kumulatívnu odmenu, tak to dokážeme pomocou MDP zapísať ako zisk očakávaných hodnôt

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

kde

- G je zisk
- R_t je očakávaná odmena v časovom kroku, rovnako ako stav S_t závislá na predchádzajúcom stave S_{t-1} a vykonanej akcii A_{t-1}
- T je posledným krokom v čase

tu ale spočíva problém v tom, že u mnoho úloh nedokážeme jednoducho definovať budúce stavy, a teda aj odmeny v týchto časových intervaloch, a taktiež môže T predstavovať nekončiaci sa čas, pri čom by sme museli počítať nekonečnú postupnosť očakávaných odmien kde by bol náš zisk $G_t = \infty$ [28]. To dokážeme riešiť implementáciou hodnoty γ ktorá predstavuje discount rate, mieru znižovania dôležitosti budúcich odmien, respektíve znižovania očakávaných hodnôt mierov simulujúcou neurčitosť predstavovanú v budúcnosti v závislosti na čase. Potom náš konečný zisk teda vypadá nasledovne

$$G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1}$$

čím viac sa blíži hodnota ktorú nadobúda miera γ číslu 1, tým viacej agent myslí vpred, a kontrastne, ak γ nadobúda hodnotu 0, potom sa agent snaží vybrať akciu ktorá ho prenesie do najlepšieho budúceho stavu.

Policy

Policy, jedným možným prekladom aj taktika, definuje správanie sa agenta v danom čase. Dá sa to predstaviť ako mapovanie stavov vnímaných agentom na akcie, ktoré v týchto stavoch vykoná. Povedzme, že je v izbe úplná tma. Čo napadne človeka ako prvé vykonať

na to, aby videl? Lokalizovať spínač a rozsvietiť. Táto taktika môže predstavovať jednoduchú vyhľadávaciu tabuľku ale tiež môže zaberať rôzne zložité výpočty, záleží na počte faktorov v prostredí s ktorými sa agent v daný moment musí vysporiadať. Celkovo ale taktiky zvyčajne bývajú stochastikého charakteru, čo znamená, že stále povoľujeme agentovi experimentovať aj v stavoch, ktoré sú mu známe, oproti deterministickým, ktoré presne definujú akciu ktorú vždy agent v danom stave vykoná. Taktiku dokážeme zapísať pomocou MDP ako $\pi(a|s)$, pravdepodobnosť že agent vykoná akciu a v stave s [28].

Cieľom siete ako takej je nájsť optimálnu taktiku, teda Optimal Policy. Čo robí jednu taktiku lepšiu ako druhú sa dá definovať:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \forall s \in S$$

Čo znamená, že taktika prináša lepšie alebo rovnaké očakávané zisky definované v v_π vo všetkých možných stavoch s v množine S a potom najlepšia, optimálna taktika v_* je:

$$v_*(s) = \max_\pi v_\pi(s)$$

Optimálnych taktík takto existuje vždy jedna alebo viac.

Value function

Value function je v podstate funkcia, ktorá má na starosti oceňovanie **celkového** stavu, v ktorom sa agent nachádza. Ohodnocuje agenta na základe možných získateľných odmien v budúcnosti agenta od daného momentu. Príkladom môže byť spokojnosť človeka byť v určitej situácii. Ak je človek spokojný, je väčšia pravdepodobnosť, že sa vyhne radikálnym zmenám a bude chcieť zostať v stave v ktorom sa nachádza. Ak sa naopak nachádza v nevýhodnej situácii, chce sa z nej čo najskôr dostať, niekedy pomocou drastických opatrení. Hodnoty, ktoré sa pomocou value funkcie priradzujú stavom, sú pevne viazané na odmeny, no nie len odmeny poskytované týmto stavom, ale aj stavmi, do ktorých je možné sa v budúcnosti dostať vykonaním určitých akcií. Stav nemusí na prvý pohľad byť pre agenta podstatný, no z neho dokáže získať odmeny, ktoré by mu inak neboli dostupné. Výstupom value funkcie je teda **očakávaný zisk** . Zisk, ako sme si už povedali, závisí na odmenách a teda pároch akcia-stav, respektíve ako sa agent správa, čo tiež rieši Policy. Vieme teda povedať, že Value function je závislá od Policy. MDP nám definujú dva druhy value funkcií, State-value function a Action-value function.

- **State-value function:**udáva hodnotu očakávaného zisku E v stave s pri nasledovaní taktiky π od času t, zapísaná ako:

  $$v_\pi(s) = E\left[G_t \mid S_t = s\right]$$

  podobne ale ako u Policy, existuje optimálna State-value function, ktorú zdielajú všetky optimálne taktiky, definovanú:

  $$v_*(s) = \max_\pi v_\pi(s)$$

  respektíve State-value funkcia je optimálnou, ak má maximálne očakávané zisky vo všetkých stavoch s za využitia akejkoľvek taktiky π

- **Action-value function:**udáva hodnotu očakávaného zisku E vykonania akcie a v stave s pri nasledovaní taktiky π od času t, taktiež zapísaná ako:

$$q_\pi(s, a) = E\left[G_t \mid S_t = s, A_t = a\right]$$

Action-value function sa taktiež nazíva Q-function, kde Q značí kvalitu state, action páru. Tiež ale musí existovať Q-funkcia ktorá sa dá považovať za optimálnu, zdielanú všetkými optimálnymi taktikami. Takú môžeme nájsť nasledovne:

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

to jesť pre všetky dvojice state-action (a, s) má najväčší očakávaný zisk vykonaním akcie a v stave s za akejkoľvek taktiky π.

Na to, aby Q-funkcia bola optimálna, ale musí tiež splňovať Bellmanovu rovnicu optimálnosti ktorá hovorí:

$$q_*(s, a) = E\left[R_{t+1} + \gamma \max_{a'} q_*\left(s', a'\right)\right]$$

Pre každý pár state-action (s, a) v čase t je očakávaný zisk zo stavu s za vykonania akcie a pri nasledovaní *optimálnej* taktiky, teda hodnotu Q-funkcie pre túto taktiku, musí byť očakávaná odmena vykonania akcie a v stave s, teda R_{t+1} **plus** maximálny očakávaný zisk zrazený hodnotou γ získaného z akéhokoľvek ďalšieho možného páru state-action (s', a'), kde za využitia optimálnej taktiky je s' stav, z ktorého je možné vykonať najlepšiu nasledujúcu akciu a' v čase $t + 1$ [18] [12].

Model

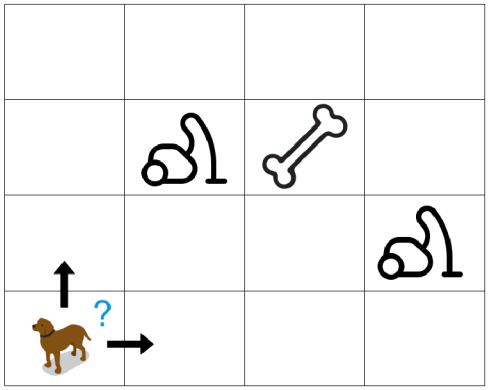
Posledným komponentom je model. Presnejšie sa jedná o model prostredia s ktorým agent komunikuje, ktorý umožňuje predvídať budúce stavy prostredia na základe vykonaných akcií. Je to teda model stavov prostredia a na ne viazaných odmien. Tento model nemusí byť vždy dostupný, a preto existujú dve vetvy posilovaného učenia: Model-based a Model-free. Kde Model-based prístupy dokážu predvídať očakávané zisky a nasledujúce stavy prostredia, Model-free prístupy toto k dispozícií nemajú. Z toho vyplýva že model-based prístupy plánujú viacej do predu, kde model-free sa učia štýlom pokus-omyl [28]. Problém v Model-based prístupoch ale spočíva v tom, že okrem taktiky si agent taktiež musí vytvoriť objektívny model svojho prostredia. To vytvára ďalší potenciálny bod zlyhania, kde si agent vytvoril model, ktorý je v reálnom prostredí neefektívny. V súčastnosti sa z väčšiny najmä kvôli relatívnej jednoduchosti oproti tomuto prístupu sa používajú model-free algoritmy [20].

2.5.2 Q-learning

Je model-free Reinforcement learning algoritmus založený na hľadaní optimálnej taktiky na základe učenia sa z Q-funkcií, teda Action-value funkcií. V trénovaní agenta sa implementujú takzvané epizódy, čo sú etapy tvorené buďto pevným počtom krokov (definovaných vykonaním akcie agenta) alebo iného ukončenia, napr. úspešné či neúspešné zavŕšenie úlohy agenta, po ktorých sa Q funkcia upravuje na základe získaných vedomostí z tejto epizódy, no v určitých prípadoch dokonca aj po stanovenom počte krokov. Hovoríme teda, že Q-funkcia má iteratívny charakter. Na začiatku sú všetky Q hodnoty párov $Q(s, a)$ nastavené

na 0, čo reprezentuje že agent o svojom prostredí nevie nič. Postupne, agent vykonáva akcie, získava vedomosti, a upravuje svoj Q-table za účelom hľadania optimálnej taktiky. Môžme si to predstaviť na jednoduchom prostredí, kde je naším agentom pes [2.7]. Ten má za úlohu dostať sa ku kosti, ktorá so sebou nesie kladnú odmenu a ukončí epizódu, a vyhnúť sa vysávačom, ktoré rovnako epizódu ukončia, ale pes dostane trest. Jeho možnými akciami sú pohyby: Hore, dole, doprava a doľava. To, ako agent na začiatku samotné akcie vyberá, sme už trochu načrtli, keď sme spomenuli problém exploration vs exploitation. Problém dokážeme vyriešiť implementovaním prístupu epsilon-greedy. Ten hovorí:

$$\text{vybraná akcia} = \begin{cases} \max Q_t(a) & 1-\epsilon \\ \text{náhodná akcia} & \epsilon \end{cases}$$



Obr. 2.7: Ako si vyberie agent akciu?

Hodnotu ϵ vyberáme na základe doby trénovania. Na začiatku vyberieme hodnotu 1, čo znamená, že agent vždy preferuje prieskum svojho prostredia. Hodnotu postupne znižujeme tak, že sa chamtivosť agenta zvyšuje. Začne viac a viac vykonávať len akcie, čo ho prinášajú do priaznivých stavov a k odmenám, ktoré už má v tabuľke Q-hodnôt.

Ako teda samotné Q-hodnoty prepočítavame? Vráťme sa späť k príkladu. Agent už má za sebou pár neúspešných epizód, preskúmal už stavy označené modrou farbou a zlyhal v stavoch červených. Momentálne sa nachádza sa v stave, o ktorom si nemyslí, že je moc priaznivý. Rozhodol sa ale vybrať akciu, ktorá ho dostala ku kosti. Dostane odmenu a ukončí epizódu [2.8]. Je teda čas prepočítať Q-hodnoty.

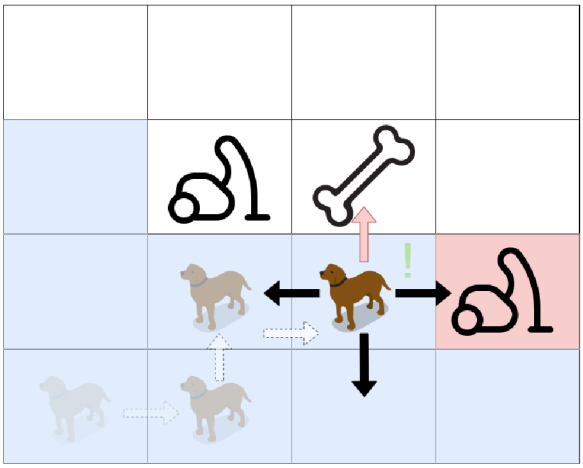
Pristupujeme nasledovne:

$$\text{New } Q(s,a) = Q(s,a) + \alpha\left[R(s,a) + y \max Q'\left(s',a'\right)\right] - Q(s,a)\big]$$

kde:

- α je learning rate, teda miera učenia. Predstavuje koeficient, ktorý hovorí ako moc sa zmení už naučená Q hodnota v danom stave, resp. ako moc je ochotný upraviť hodnotu na základe nových zistení
- $R(s,a)$ odmena v stave s za vykonania akcie a
- $Q(s,a)$ už naučená Q-hodnota
- y discount rate

- $\max Q'(s', a')$ maximálna očakávana odmena pri stave s' a všetkých možných akciách v tomto stave.



Obr. 2.8: Agent si musí zapamätať informácie o stave po výbere novej akcie

V tomto momente sa Q-hodnota stala vyššou, nakoľko vieme, že z tohto stavu je možné úspešne dokončiť úlohu. To znamená že sa agent bude rozhodovať pristúpiť do tohto stavu častejšie, najmä ak klesne hodnota ϵ [17].

2.5.3 Proximal Policy Optimization

V projekte používame najmä metódu posilovaného učenia zvanú Proximal Policy Optimization, v skratke PPO. Vznikla ako nadstavba algoritmu TRPO (trust region policy optimization firmou OpenAI. Tento prístup je značný svojou jednoduchosťou a efektivitou v porovnaní s najlepšími známymi algoritmami posilovaného učenia [16]. Algoritmus sa snaží upravovať svoju Policy konzervatívnym spôsobom. Agent počíta výhodu svojho stavu Advantage funkciou, ktorá sa dá predstaviť ako rozdiel odhadu získanej odmeny a reálneho výsledku. Ak je výhoda kladná hodnota, agent má väčšiu pravdepodobnosť vykonať akciu v budúcnosti, a opakom ak je hodnota záporná. Hodnota potom prechádza cieľovou funkciou tohto algoritmu na to, aby sa nepropagovali do systému príliš extrémne zmeny ktoré by mohli znateľne narušiť už existujúcu Policy agenta.

2.5.4 Imitation learning

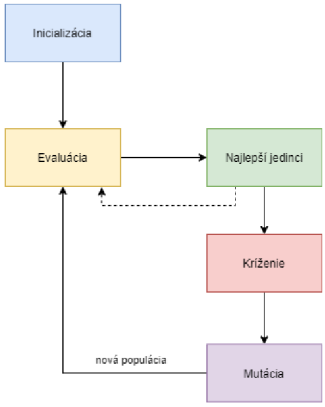
Imitation learning je druhom posilovaného učenia, kde okrem toho, aby sa agent učil len z odmien ku ktorým sa dostal po prvý krát náhodnou cestou, dokážeme naučiť agenta pomocou inštrukcií poskytnutých expertom, podobne ako u Supervised learning. Z toho často vyplýva, že užívame rovnakých výhod a nevýhod ako u tejto kategórie hlbokého učenia. Výhodou teda je, že dokážeme agenta učiť efektívnejšie a často rýchlejšie, no musíme sa uistiť, že náš dataset, respektíve inštruktáž, je optimálny a objektívny, inak môžme sieť naviesť k zlej ceste už od začiatku. Tieto algoritmy bývajú často nadstavbou posilovaného učenia, a to už buď z nevyhnutnosti [4] alebo preto, že v mnoho úlohách nechceme byť obmedzený ľudským faktorom, a sieť trénujeme k superľudským výkonom. To, ako sa agent snaží napodobniť danú inštruktáž, tiež známe ako **demo** rozdeľuje Imitation learning do viacerých prístupov, ako behavioral cloning, reverse reinforcement learning a Generative Adversarial Imitation Learning (GAIL).

GAIL

GAIL ako už názvu vyplýva sú založené na princípe GAN algoritmov. Existujú teda dve siete, v tomto prípade sieť „Learner“ predstavujúca diskriminátor, a sieť „Expert“ predstavujúca generátor, ktorý je naším agentom. Generátor má u GAIL za úlohu vytvárať čisto na základe ukážok poskytnutých demoštráciou páry action-state pre diskriminátora, ktorý má za úlohu rozpoznať či ukážka prišla z demonštrácie alebo z generátora [6]. Diskriminátor vracia odmenu na základe podobnosti falošnej ukážky s ukážkou z demonštrácie. Agent sa túto odmenu snaží maximalizovať, pričom diskriminátor je neustále striktnejší. Maximalizácia odmeny no tiež znamená, že sa agent snaží prežívať dlhšie, aby mohol vygenerovať viacej akcií podobných ukážke. To zavádza konflikt pri úlohách, kde je cieľom dostať sa do cieľa čo najrýchlejšie. Preto je vhodné prístup kombinovať s viac tradičnými algoritmami posilovaného učenia so silnejšími signálmi odmeny [10].

2.6 Evolučné stratégie

Evolučné stratégie sú triedou genetických algoritmov. Sú inšpirované prirodzenou evolúciou organizmov, ktorú dokážeme pozorovať na Zemi [22]. Táto inšpirácia sa prejavuje v samotnej štruktúre týchto algoritmov. Princípom sú generácie jedincov súperiacich v určitej úlohe, kde namiesto value funkcií sú ohodnocované na základe fitness funkcií. Z generácie sa vyberú najlepší jedinci, vybraný najvyššou fitness hodnotou, a podľa rôznych techník ako mutácia či kríženie sa ich gény propagujú do ďalších generácií. Tieto prístupy sa neustále preukazujú ako konkurencieschopné algoritmom posilovaného učenia [26] taktiež sú extrémne jednoduché na paralelizáciu a ľahšie na výpočetnú silu.



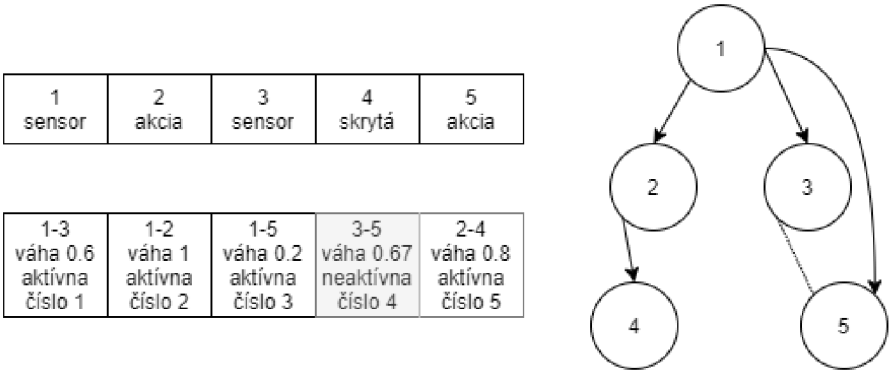
Obr. 2.9: Model evolučných stratégií

2.6.1 Neuroevolution of Augmented Topologies

Neuroevolúcia rieši problém, ktorý sa pýta či je možné využiť neurónové siete u evolučných stratégií. Neuroevolúcia augmentovaných topológií sa pýta či je možné vyvinúť samotnú neurónovú sieť. To so sebou prináša rôzne výzvy: Existuje genetická reprezentácia topológií tak, aby sme ich mohli medzi sebou krížiť a mutovať? Je spôsob, ako nové siete s rozvíjajúcou sa architektúrou chrániť pred príliš skorým vyradením? Tieto problémy riešime pomocou NEAT nasledovne:

Encoding

Genómy jednotlivých sietí delíme do génov neurónov a génov ich spojení. Každý neurón je očíslovaný, a každá spojitosť má uložené dva neuróny ktoré spája, plus unikátne číslo predstavujúce poradie jej objavenia a váhu aktivácie. Naďalej aj hodnotu boolean, ktorá určuje či je gén v inštancii siete, tiež zvanej fenotyp, aktívny alebo nie.



Obr. 2.10: Genóm a fenotyp NEAT algoritmu, predstavujúci uloženie neurónovej siete

Mutácia

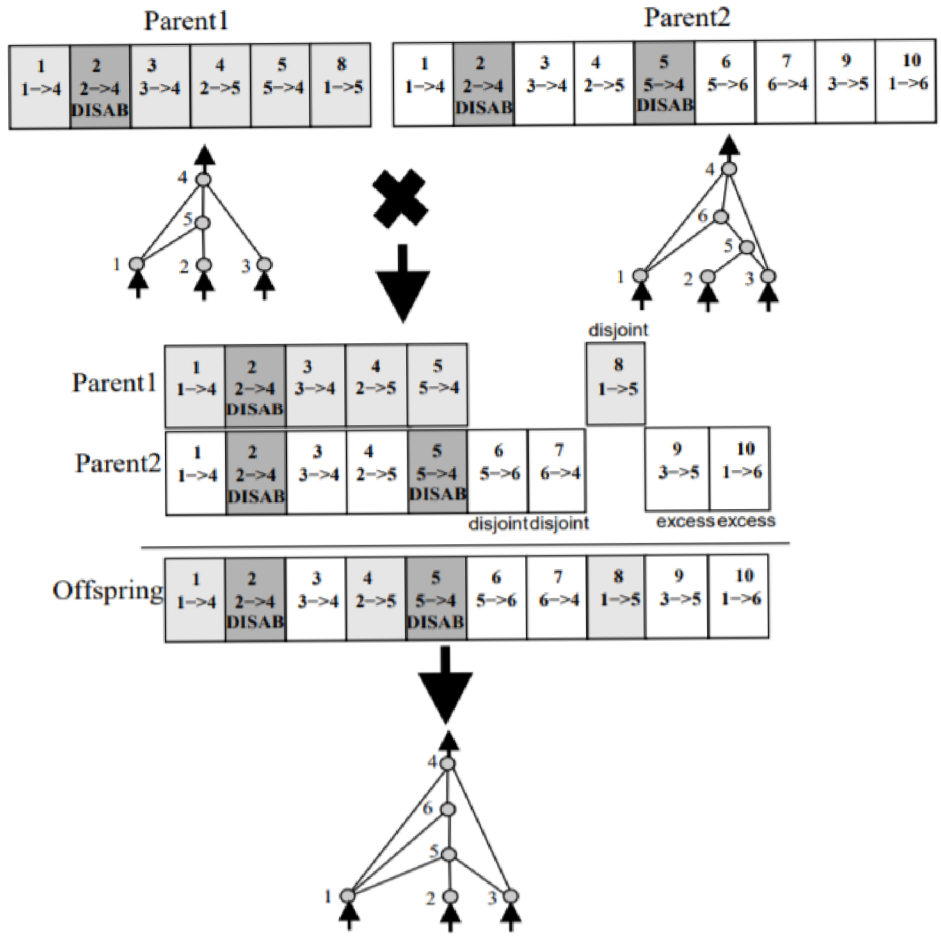
Mutovať dokážeme pomocou jednotlivých akcií: Vytvorenie nového neurónu, vytvorenie nového spojenia, aktivácia/deaktivácia spojenia, a úprava váhy aktivácie.

Kríženie

Kríženie v NEAT prebieha na základe dvoch jedincov s rôznym genómom. Krížia sa len gény spojení, pretože si dokážeme odvodiť gény neurónov ak akékoľvek v potomkovi chýbajú a priradiť mu ich. Dôležité sú tu čísla objavenia spojenia. Všetky spojenia ktoré majú rodičia rovnaké sa vyberú náhodne, teda ich váhy a to či sú aktivované, no ak sú spojenia ktoré má vždy len jeden rodič, rozdeľujeme ich medzi oddelené a prebytky. Prebytky sú tie, ktoré číslom objavenia presahujú najvyššie číslo objavenia u druhého rodiča. Ak sú oddelené alebo prebytočné spojenia rodiča ktorý je podľa fitness funkcie dominantný, snažíme sa tieto spojenia aktivovať ak je možné [2.11].

Selekcia

Selekcia v NEAT prebieha trochu iným štýlom. Namiesto toho, aby sme vybrali skupinu najsilnejších jedincov vyberáme najsilnejších jedincov jednotlivých druhov. Do akého druhu patrí fenotyp definujeme podľa jeho genómu. Ak je dostatočne rozdielny, jedná sa o nový druh siete, ktorý potom súperí len so svojim druhom. Takto riešime problém chránenia mladých sietí [25].



Obr. 2.11: Kríženie rodičov u NEAT. Zdroj: [25]

Kapitola 3

Implementácia

V tejto kapitole sa zaoberieme prvotným návrhom základných scien, na ktorých budeme agentov rôznych algoritmov učiť, a taktiež postup ich úprav, zmien a vyvolané reakcie neurónových sietí od koncepcie až do súčastného stavu. Taktiež tu preberieme rôzne prekážky, na ktoré agenti pri vykonávaní daných úloh narazili, čo ich spôsobilo a aké kroky boli implementované k tomu, aby sme ich prekonali.

3.1 Návrh

Ako počiatočnú šablónu práce sme využili ukážkovej hernej scény poskytovanej knižnicou ML-Agents[1]. Tá využíva štandardných tried Unity zvaných GameObject, a ich preddefinovaných fyzikálnych vlastností (gravitácia,trenie). Jeden z týchto objektov taktiež reprezentuje našeho trénovaného agenta, a ten využíva triedy priamo z knižnice; Agent. Ten s hernou scénou komunikoval len na základe minimálnych parametrov ako sú koordináty okolitých objektov a za úlohu mal posunúť guľu ku kocke na rovnej ploche. Možnosti akcií agenta boli obmedzené na pohybovanie guľou pridaním vektoru sily. Pozitívnymi odmenami boli dotyk kocky, čiže úspech epizódy, a trestami boli existenčný trest ktorý mal za úlohu nútiť agenta dokončiť úlohu čo najskôr, a pád z hracej plochy čo sa počítalo ako neúspešný koniec epizódy. Tento návrh slúžil čisto len na ukážkové účely, nakoľko bola scéna tak jednoduchá že sa sieť dostala do finálneho vyriešeného stavu za pár minút, no bol využiteľný ako šablóna na ďalší progres.

3.2 Unity

[2] Herná scéna je tvorená 3D hernými objektmi, každý tvorený vlastným Mesh objektom, ktorý reprezentuje ako sú poskladané polygónmi a zobrazené na ploche. Tie sú ovládané preddefinovanými fyzikálnymi javmi na základe triedy pridelenej objektu zvanej RigidBody. Tá simuluje kolízie či gravitáciu, čo nám pomôže vytvoriť pestré herné prostredia, s ktorými agent dokáže mať rôzne interakcie.

[1] www.github.com/Unity-Technologies/ml-agents
[2] www.unity.com

3.3 Agent

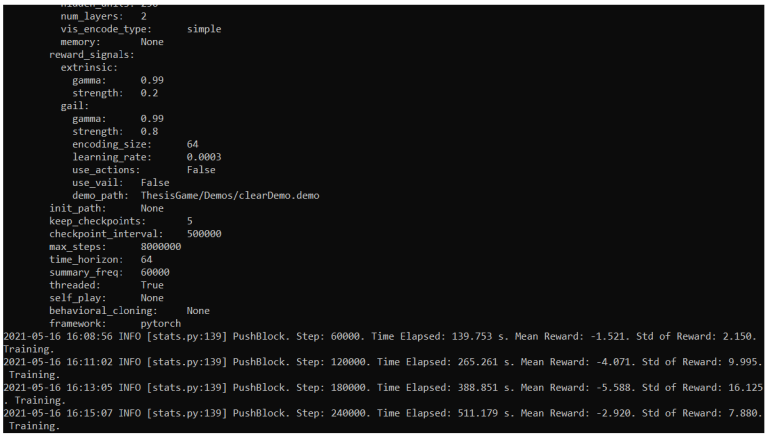
Tvorba agenta pomocou ML-Agents sa skladá z dvoch častí. Musíme si vybrať GameObject, do ktorého hernej logiky pomocou komponentu skript implementujeme triedu Agent, a jej potrebné metódy, a taktiež musíme objektu pridať komponenty „Behavior Parameters“ a „Decision Requester“.

3.4 Trénovanie sietí

Na trénovanie agentov knižnica v minulosti využívala Tensorflow, no v novších verziách knižnice sa štandard zmenil na využívanie PyTorch. Na úspešné spustenie trénovania musíme mať k dispozícií našu hernú scénu, v ktorej máme presne definovaného agenta implementujúceho všetky potrebné funkcie k učeniu, a zoznam hyperparametrov siete zapísané do súboru typu yaml. Ten obsahuje napríklad použitú metódu učenia, ako napríklad PPO, alebo SAC, konštanty ako discount rate a learning rate, či dĺžku samotného učenia. Väčšinu sme nastavili pomocou poskytnutej šablóny v example scéne. Pre učenie siete pomocou grafickej karty sme tiež potrebovali využiť CUDA drivers od firmy Nvidia, aby naša grafická karta dokázala s PyTorch komunikovať priamo. Či je všetko v poriadku zistíme priamo na príkazovom riadku v ktorom sieť aj spustíme.

3.5 Získavanie výsledkov

ML-Agents podporuje Tensorboard ako nástroj na analýzu stavu siete. Tým je možné získať rýchly prehľad čo sa týka úspešnosti agenta, dĺžky trénovania a podobne. Sú však veci ktoré sa dajú získať len priamym pozorovaním siete, ako problémy typu reward-farming, a dávať pozor aby sa tieto nepropagovali do konečných výsledkov. Informácie o skóre agenta sú taktiež vypisované v príkazovom riadku spolu s ubehnutými krokmi [3.1].



Obr. 3.1: Výpis nastavených hyperparametrov a progres cvičenia dostupný u príkazového riadku

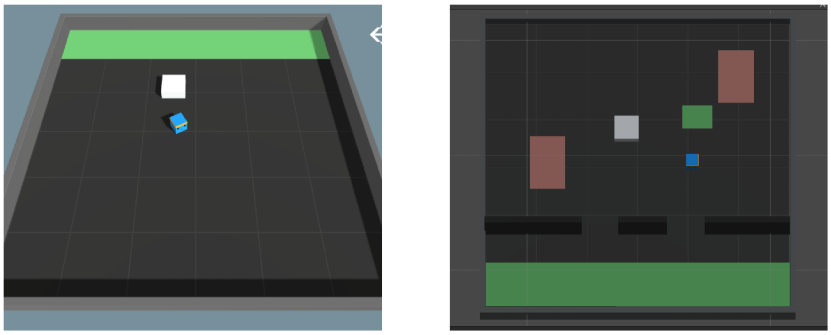
3.6 Scéna

Na to, aby sme dokázali algoritmy analyzovať a porovnávať efektívnym a objektívnym spôsobom, potrebujeme hernú scénu upraviť na toľko, aby ju aj najjednoduchší algoritmus

hľadania najkratšej cesty nedokázal za pár minút úspešne dokončiť a perfektne optimalizovať. Toho sme dosiahli postupne a vo viacerých krokoch, čo nám umožnilo presne sledovať ako sa dokáže algoritmus (v tomto prípade Reinforcement Learning) adaptovať pri čoraz sa zvyšujúcej zložitosti.

3.6.1 Zložitosť

Na začiatok je dobré si presne definovať čo je myslené zložitosťou hernej scény v tomto kontexte. Pre naše účely sme definovali zložitosť nasledovne: s akým počtom unikátnych a presne rozlíšiteľných úloh a prekážok sa musí agent v hernej scéne vysporiadať, aby sa dostal k úspešnému zakončeniu a finálnej odmene? Taktiež je treba brať v úvahu to, že nie každá prekážka je pre sieť rovnako náročná prekonať. Napríklad pridanie nového, povinného kroku na dokončenie úlohy je oveľa zložitejšie, ako pridať krátku stenu medzi začiatkom a cieľom. Príkladom môže byť naša začiatočná scéna: Ak si definujeme cieľ agenta ako pohyb z jedného bodu v scéne na druhý bez akýchkoľvek iných faktorov ako hernú scénu so složitosťou 1, potom je jasné, že scéna obsahujúca miesto, na ktoré agent musí dotlačiť iný objekt, pri čom sa musí vyhýbať stenám a priepastiam, až po čom má konečne možnosť nájsť a dostať sa do cieľovej pozície je ako mnohokrát viacej špecifická tak aj mnohonásobne zložitejšia.



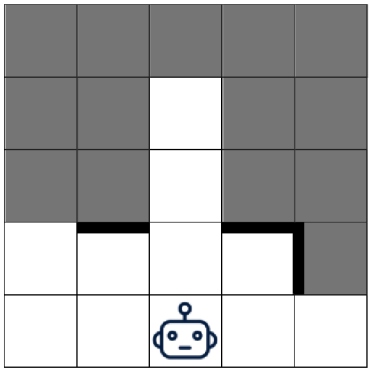
(a) Počiatočná podoba scény **(b)** Finálna podoba scény

Obr. 3.2: Úprava scény na analýzu algoritmov

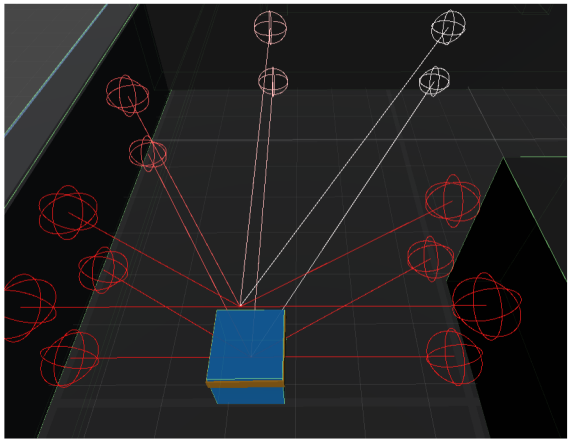
3.6.2 Komunikácia so scénou

Ako sme už spomenuli, agent so scénou na počiatku komunikoval čisto len na základe koordinátov rôznych objektov v hernej scéne. To ale značne obmedzuje komplexitu, ktorú agent dokáže prekonať čo sa týka úloh. To spočíva z faktu, že koordináty herných objektov v nadmernej väčšine len naznačujú, kde sa nachádza stred tohoto objektu, a nie informácie o jeho dimenziách či iných vlastností. Potrebovali sme teda vybrať iný systém zbieraní dát zo scény pre agenta, ktoré budú fungovať ako de facto vstupy pre neurónovú sieť. Tu je k dispozícií hneď niekoľko možností: Scénu vziať z top-down perspektívy pri čom by agent ovládal GameObject z tretej osoby. Tu by sme mohli využiť rôznych taktík ako ušetriť na výpočetných zdrojoch renderovaním len zjednodušenej verzie scény alebo len sledovaním objektov potrebných na dokončenie úlohy, no narazili by sme na jednu negatívnu stránku takéhoto prístupu, a to že agent by mal vždy priamu informáciu o mieste, kde sa nachádza jeho ďalší krok úlohy. To v praxi znamená, že nemôžeme agenta naučiť explorácii, čo je vo svete počítačových hier obrovský faktor a taktiež výrazne znižuje efektívnu zložitosť scény.

Príkladom môže byť stena, ktorá stojí medzi cieľom agenta a jeho aktuálnou pozíciou. Z top down perspektívy agent vie rozpoznať, že jeho cieľ sa za stenou nachádza, prípadne že scéna za stenou pokračuje a mal by ju obísť. Tento problém by sa teoreticky dal riešiť implementovaním funkcionality ako „Fog of war“[3.3] ktorá by zakryla všetko v scéne kde agent nemá práve dohľad, ale stále by to nebolo tak efektívne ako prístup ktorý sme sa nakoniec rozhodli použiť, Raycasting[3.4]. Raycasting nám umožňuje agenta viacmenej zjednotiť s objektom ktorého ovláda, pretože priamo získava len informácie o tom kde sa nachádza, a o blízkom okolí. Toto motivuje agenta aktívne prehľadávať scénu v ktorej sa nachádza a informácie nadobúdať týmto spôsobom. Taktiež nám dovoľuje implementovať rôzne úlohy, ktoré by pre agenta inak boli príliš jednoduché, alebo inak nevhodné.



Obr. 3.3: Ukážka princípu fog of war, hrubé čiary sú steny a šedé políčka sú mimo vízie agenta



Obr. 3.4: Raycasting vízia agenta, agent vidí prekážky ak lúč ktorý vysiela s nimi koliduje. Dokáže ich rozlíšiť podľa ich značiek

3.7 Development

Vysvetlili sme si, čo bolo motiváciou scénu vytvoriť zložitejšiu a čo vlastne zložitosť scény pre nás znamená. Teraz sa zaujmeme tým ako agent na zvýšenú zložitosť reagoval a ako sa s ňou vyrovnával. Taktiež si popíšeme ako boli rôzne kroky implementované a akú úlohu splňujú.

3.7.1 Existenčný trest

Existenčný trest bol zachovaný nakoľko efektívne znižuje možnosť reward-farming problémov a rovnako núti agenta konať efektívne, scénu dokončiť čo najskôr a výkon optimalizovať. Trest sa aplikuje u každého kroku agenta, nezávisle od toho, či agent v danom kroku taktiež získal odmenu. Implementovaný je v podstate len ako záporné číslo, čo ho líši od odmeny. Celkový trest/odmena za krok sa teda počíta ako:

$$R_{total} = R_{exist} + \sum_{i=1}^{n} R_i$$

Kde R_{total} je celková odmena, R_{exist} existenčný trest a n počet získaných odmien agentom v kroku. Ak je výsledné číslo záporné, jedná sa o trest, ak je kladné, agent v kroku získal odmenu.

3.7.2 Block

Prvá vec implementovaná po zmene agenta na RayCast perspektívu bola implementácia interaktívneho kvádra, s ktorým sa agent musí určitým spôsobom vysporiadať. V našom prípade je kváder, v kóde zvaný Block ovládateľný len ak ho agent úmyselne uchopí. To sa pre neurónovú sieť prejavuje ako kompletne nová akcia, ktorú je možné vykonať. Táto akcia „uchop Block“ je veľmi špecifická, čo znamená, že ju agent väčšinu času vykonáva bezvýsledkovo—nie je v prítomnosti bloku. Agent ale vždy musí minúť v kroku akciu buď na pohyb, alebo na uchopenie kvádra. Nakoľko máme definovaný existenčný trest, je potreba za úspešné chytenie kvádra dať dostatočnú odmenu, aby sme motivovali agenta túto akciu používať pred tým ako sa sieť naučí na čo vlastne slúži. Akcia taktiež spomaluje pohyb agenta, čo ho núti block upustiť čo najskôr. Odmena musí byť vybraná vhodne, pretože ak je príliš vysoká, môže skončiť u reward-farming, kde agent perpetuálne chytá a púšťa Block, ale nie nízka tak, aby agent prestal akciu používať. Strop minima tejto odmeny sme vyriešili ľahko, a to pomocou implementácie ďalšieho objektu, zvaného pressurePlate. Tým sme dokázali interakciu s kvádrom spraviť povinnú, a ak je to prvý zdroj odmien v hernej scéne, tak sa po naučení sieť túto odmenu pokúsi získať takmer vždy čo najskôr. Odmenu sme teda definovali nasledovne: Ak by agent dostal za krok existenčný trest, tak potom, ak má úspešne uchopený block, jeho trest je polovičný, teda jeho odmena sa rovná:

$$R = |R_{exist}| * 0.5$$

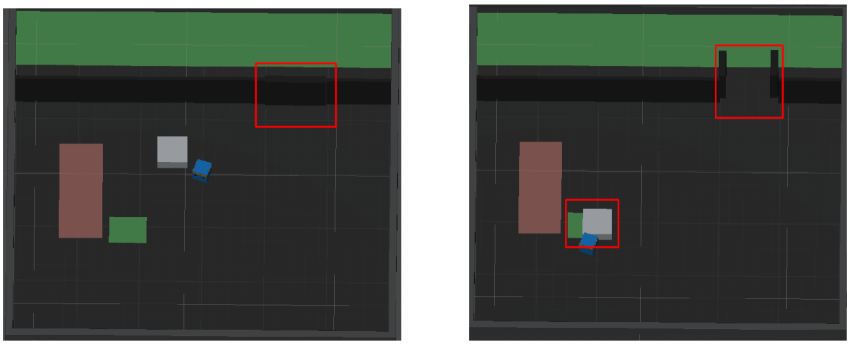
3.7.3 Pressure Plate

GameObject PressurePlate slúži v kombinácii s už spomenutým Block objektom ako prvý cieľ ktorý sme pre agenta vytvorili, a to dotlačiť kváder do trigger zóny tohto objektu. O to sa stará vstavaná funkcia pre herné objekty v Unity onTriggerEnter. PressurePlare je pre agenta malá plocha zvýraznená od podlahy hernej scény zelenou textúrou a vlastným tag menom. V tomto štádiu to bol finálny cieľ siete za ktorý agent získal odmenu a úspešne ukončil epizódu. Odmena za dokončenie tejto úlohy musela byť väčšia ako odmena za držanie kvádra na toľko, aby bolo pre agenta preferovanejšie snažiť sa úlohu dokončiť čo najskôr. Tento cieľ bol porovnateľný s cieľom dostať sa do cieľa bez žiadneho kvádra, no len pridanie tohto objektu bolo dostatočné na zvýšenie zložitosti scény v celkom veľkom zmysle. Agent musí lokalizovať Block, prísť dostatočne blízko na uchopenie, Block chytiť, lokalizovať Pressure Plate, dotlačiť Block na miesto tohto herného objektu. Už v tomto štádiu sme našli

minimálne 5 unikátnych úloh ktoré agent musí vykonať v presne definovanom poradí. Aký bol vplyv na časy optimalizácie úlohy budeme analyzovať v nasledujúcej kapitole.

3.7.4 Gate

Ďalším míľnikom bola implementácia skupiny herných objektov zvaná Gate. Tá spočíva v kombinácií stien tvoriacich bránu, slúžiacich ako prekážka agentovi dostať sa do nového konečného cieľa úlohy, ktorý sa za bránou nachádza. Brána je otvorená až po tom, ako agent Block prenesie na PressurePlate. To funguje na základe animácie pridelenej objektu gate pomocou komponentu zvaného Animator, kde namiesto ukončenia úlohy pri kolízii kvádra a PressurePlate objektu zmeníme boolean hodnotu zodpovednú za aktiváciu animácie otvárania brány. Animácia steny otočí okolo bodu v rohu tak, aby simulovali otvorenie brány, toto agentovi dovolí nájsť a dostať sa do nového cieľu.



(a) Zatvorená brána

(b) Brána otvorená po dokončení predchádzajúcej úlohy

Obr. 3.5: Implementácia brány v scéne

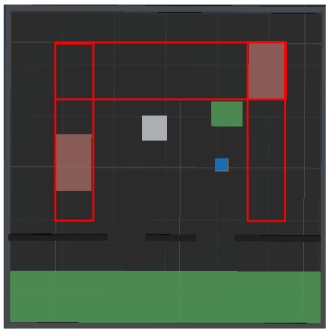
3.7.5 GoalArea

Slúži ako finálny objekt u ktorého pri kolízií s agentom sa epizóda ukončí s úspechom a agent dostane odmenu za dokončenie úlohy. Táto odmena by mala byť väčšia ako všetky možné odmeny ktoré agent dokáže získať za celú jeho aktivitu v epizóde, aby sme sa vyhli reward-farming, ale nie tak zbytočne vysoká, že vedľajšie odmeny sú pre agenta nezaujímavé. Taktiež po implementácií tohto objektu musíme upraviť odmenu za PressurePlate tak, aby ju bolo možné dostať len jeden krát za epizódu, inak môže nastať, že agent začne zbierať odmeny za to, že sa neustále dotýka s kvádrom PressurePlate objektu.

3.7.6 Náhodnosť

V tomto štádiu už je sieť dostatočne zložitá na našu analýzu, no ak chceme algoritmy naučiť byť samostatné a viac schopné, musíme implementovať do našej scény určitú mieru náhodnosti. To zaručujeme viacerými spôsobmi. Na začiatku generácie scény pre každú epizódu sa dejú nasledovné veci: Herná scéna sa rotuje náhodne v intervaloch 90 stupňov. Už týmto dokážeme zaručiť, že si agent nemôže len jednoducho zapamätať pozíciu úloh, no musí ich aktívne vyhľadávať. Taktiež ale implementujeme dva druhy herných objektov, ktoré sú pre agenta prekážkami.Prvým druhom sú steny, tie dynamicky menia pozíciu brány, a teda aj cestu ktorou musí prejsť agent do finálneho cieľa. Ďalším druhom sú diery. Tie

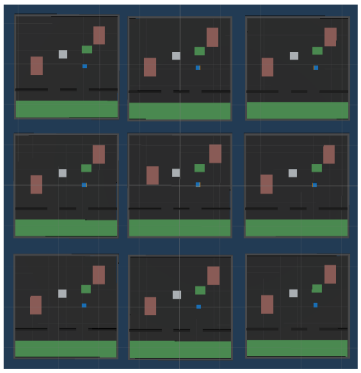
sú podobne ako PressurePlate a GoalArea len určité miesta na ploche scény, pri dotyku s ktorými agent dostane trest a epizóda sa skončí neúspechom. Narozdiel od stien sa cez ne dá vidieť, ale na druhú stranu pre agenta prestavujú väčšie riziko a zaberajú väčšiu plochu. Náhodnosť týchto je zaručená tak, že sa môžu vygenerovať v troch rôznych oblastiach mapy, kde nie je možné agentovi zabrániť úlohu dokončiť. Ak sa náhodou priepasť vygeneruje príliš blízko objektu Gate, vyberie sa ďalšia náhodná pozícia.



Obr. 3.6: všetky oblasti v ktorých sa môžu diery generovať

3.7.7 Paralelizácia

Po implementácií faktorov, ktoré sme prebrali, sa zložitosť scény stala dostatočne vysokou, že na to, aby agent dosiahol prvého úspechu, to môže sieti trvať hodiny. Z toho tiež vyplývajú nechcené situácie, ako neschopnosť siete získať ďalšiu odmenu, a optimalizovať odmeny ktoré schopný získať bol. Tie budeme riešiť v ďalšej kapitole. Jedna z vecí, ktoré ponúka knižnica ML-Agents ako potenciálne riešenie je možnosť paralelizácie siete. To znamená, že namiesto jedného agenta ktorý sa učí scénu hrať, ich existuje niekoľko. Limitom je v podstate len výpočetná sila počítaču na ktorom sieť beží. Paralelizácia sa implementuje v celku intuitívne, je potreba s hernej scény vytvoriť Unity Prefab, čo je vlastne znovu použiteľná kópia celej našej vytvorenej hernej scény. Tú potom viackrát nakopírujeme s rôznymi odchýlkami v koordinátoch ako GameObjects na rovnakej úrovni ako naša originálna scéna. Pri využívaní tejto funkcionality je treba dávať pozor, aby sa dynamické umiestňovanie objektov, ako napríklad v skriptoch scény dialo na základe LocalPosition, ktorý zaručuje, že sa tieto objekty správne umiestnia do svojich špecifických scén[3.7].



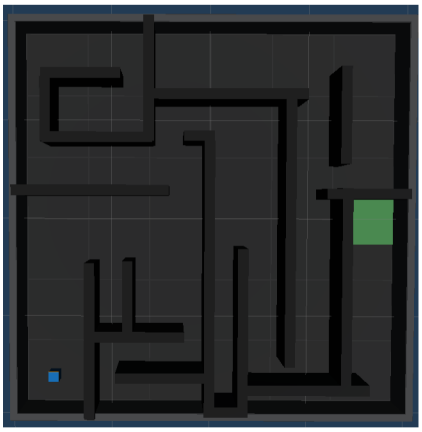
Obr. 3.7: Paralelizácia jednej scény na 9 scén, každá obsahujúca agenta prispievajúceho do spoločnej siete

3.7.8 Imitation learning implementácia

Ďalším možným prístupom, ako sieť scénu úspešne naučiť, je zmena algoritmu. ML-Agents poskytuje podporu ešte jedného, stavaného na podobné účely. Ak je scéna pre Reinforcement Learning príliš náročná alebo by bol systém odmien príliš zložitý či zdĺhavý, existuje spôsob, ako tieto problémy obísť, a to je pomocou imitačného algoritmu. Imitačný algoritmus má viacero variácií, no my sme vybrali pre naše účely GAIL—Generative Adversarial Imitation Learning. Dovôdom bola jednoduchá implementácia do našej scény a možnosť kombinovať s Reinforcement Learning pomocou jednoduchej úpravy konfiguračného súboru. Tam sme do skupiny reward_signals, zodpovednú za získavanie odmien pridali pasáž gail, definujúcu možnosť agent môže získavať odmeny na základe RL a taktiež na základe GAIL. Na implementáciu je potreba nahrať demonštráciu ako scénu natrénovať. K tomu musíme mať v scéne implementované heuristické ovládanie, teda v scéne mať implementované vstupy ktoré dokáže využívať človek. To sa preukazuje v kóde ako mapovanie každej možnej akcia agenta na napríklad klávesu. Tým dokážeme scénu ovládať a demonštráciu nahrať pomocou komponentu u agenta zvaného Demonstration Recorder.

3.7.9 Labyrinth

Pre širšiu variáciu scén na porovnávanie algoritmov sme vytvorili plne novú scénu, ktorá funguje ako bludisko. Na rozdiel od predchodzej scény má len jeden cieľ, a to je dostať sa na koniec tohto bludiska. V tom mu ale prekážajú steny a slepé uličky. Na prvý pohľad táto scéna vypadá jednoducho, no pre určité algoritmy ako RL môže byť náročná, ako si ukážeme bližšie v ďalšej kapitole [4.2.2]. Je totižto zložité v takomto jednoduchom prostredí vytvoriť systém odmien a trestov na základe agentových akcií. Nakoniec sme pre Reinforcement Learning vzali celkom prirodzený prístup. Sieť má pole koordinátov o určitej veľkosti, kde si ukladá pokryté miesta ktoré agent už prešiel. Tieto miesta teda simulujú pamäť agenta. Každá spomienka sa dá vyjadriť ako pokrytý štvorec okolo agenta, neprekrývajúci sa s inými spomienkami. Ak agent získa novú skúsenosť, teda sa dostane na koordinát mapy na ktorom sa ešte nenachádzal, a ktorý nie je súčasťou jednej z jeho spomienok, je odmenený. Ak agent stráca čas na mieste ktoré má v pamäti, jeho existenčný trest je vyšší, čo ho motivuje byť stále na pohybe. Dĺžka toho poľa koordinátov predstavuje veľkosť pamäti agenta, a ak sa tejto dĺžky presiahne, najstaršie spomienky sa zmažú—agent zabudol z akého smeru prišiel.



Obr. 3.8: Paralelizácia jednej scény na 9 scén, každá obsahujúca agenta prispievajúceho do spoločnej siete

Generácia

Steny predstavujú u nášho labyrintu určitú mieru náhodnosti. Existujú 4 možné cesty do cieľa v bludisku, no k dispozícii v danej epizóde je vždy len jedna. Ostatné steny sú pri generácii epizódy predĺžené a presunuté tak, aby boli všetky ostatné cesty zablokované. To pomáha ďalej motivovať u agenta exploráciu, pretože si nemôže ľahko memorovať cestu do cieľa. Scéna sa taktiež ako u minulej otáča v 90 stupňových intervaloch, prinášajúc dokopy 16 možných variácií bludiska, dostatočné na naše účely.

3.7.10 Neuroevolution implementácia

Pre našu novú scénu sme mohli použiť algoritmus, ktorý by u starej bol príliš nevhodný na implementáciu. To je kvôli tomu, že fitness funkcia pre hernú scénu s danou zložitosťou by bola až príliš komplikovaná a neintuitívna. Napr. jedna možnosť by bola fitness agenta počítať na základe vzdialenosti od určitých cieľov a taktiež ich postupnosti, no to kompletne ruší podstatu scény, ktorá spočíva v potrebe agenta tieto objekty vyhľadať a v nešpecifikovanom poradí vykonať. U scény labyrintu sa tento problém nenachádza, stačí, ak implementujeme systém pamäti implementovaný u RL ako fitness funkciu. Neuroevolúcia nie je natívne podporovaná knižnicou ML-Agents. Je síce možné implementovať vlastné algoritmy využitím ich low-level python API, no tvorba genetických algoritmov nebola zámerom práce. Na implementáciu sme využili faktu, že u herných objektov je logika, teda ich skripty sú tvorené v jazyku C#. Neuroevolúcia v C# už je riešená voľne dostupnou knižnicou SharpNEAT. Našli sme port tejto knižnice priamo do Unity, zvaný UnityNEAT, ktorá priamo SharpNEAT integruje do herného engine [3]. Knižnica poskytuje example scénu podobne ako ML-Agents, kde agent trénuje jazdiť po ceste. Parametre siete ako napríklad velkosť populácie či dĺžka epizód sa dajú upravovať v skripte zvanom Optimizer. Tomu pridelíme GameObject ktorý bude naším trénovaným agentom. Skript Coroutiner sa stará o to, aby sme agentov mohli trénovať viacero naraz, teda generačne. Prevzaté z knižnice boli pasáže kódu ovládajúce sieť, pričom mi sme zmenili pasáže zodpovedné za získavanie fitness a vložili do vlastnoručne vytvorenej scény.

Skúšobná scéna

Skúšobná scéna v UnityNEAT funguje na základe jednej veľkej dráhy poskladanej z obdĺžnikových častí. Tie sú očíslované na základe ich vzdialenosti od začiatku trate. Agentov cieľ je získať čo najviac bodov za určitý čas, teda dokončiť čo najviac kôl na dráhe. Sieť sa najprv naučí auto na trati ovládať a postupne optimalizuje svoju dráhu. Fitness funkcia funguje na základe vzdialenosti od cieľa, teda na ktorej časti trate sa agent nachádza, počtu kôl ktoré dokončil, a do koľkých stien v priebehu narazil.

GUI

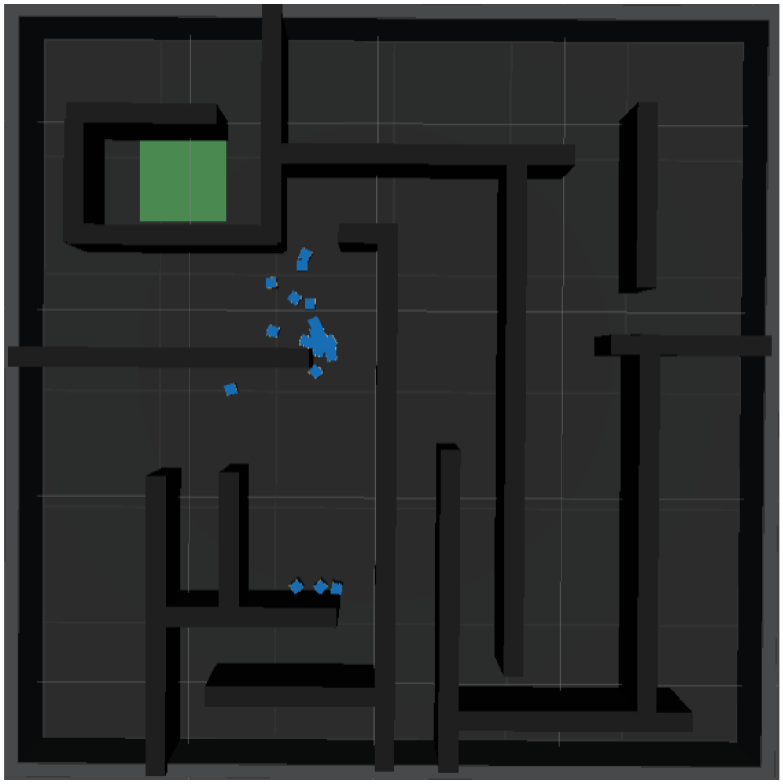
Súčasťou scény je aj jednoduché užívateľské rozhranie s ktorým sieť dokážeme spustiť, pozastaviť alebo zobraziť najlepších jedincov siete.

3.7.11 Implementácia do labyrinth scény

Na využitie knižnice pre naše účely sme využili kostru example scény. Z nej sme dokázali využiť napojenie na NEAT sieť a taktiež vstupy pre sieť—opäť sa využíva Raycasting

[3] www.github.com/lordjesus/UnityNEAT author: Daniel Jallov

prístup na videnie agenta, a taktiež jeho akcie. Tie sú podobné akciám implementovaných v ML-Agents no podobajú sa viacej bežiacemu autu, kde agent má plynulý pohyb pred s rozbehom, a otáča sa s podobným polomerom ako malé auto. To je v podstate pre naše účely dostatočné a môžme z toho získať aj inú perspektívu scény. Čo sme zmenili je ale fitness funkcia, ktorú ako sme už spomenuli sme využili rovnakú fitness funkciu ako u Reinforcement Learning algoritmu, s tým, že ak agent nezíska dostatočnú odmenu na to, aby bola fitness hodnota kladná, tak ju zarovnáme na 0.



Obr. 3.9: Implementácia neuroevolúcie do scény labyrintu

Kapitola 4

Analýza a vyhodnotenie

V tejto kapitole si povieme nielen o výsledkoch analýzy algoritmov čo sa týka času, úspešnosti a podobných vlastností. Taktiež využijeme skúseností s implementáciou, problémy na ktoré sme narazili a rôzne objavy. Tým načrtneme potencionálne využitia pre neurónové siete vo vývoji počítačových hier alebo priemysel ako taký. Hlavným zámerom bolo porovnávať vhodnosť algoritmov k špecifickým úlohám a definovať ako presne vplývajú rôzne faktory na úspešnosť sietí.

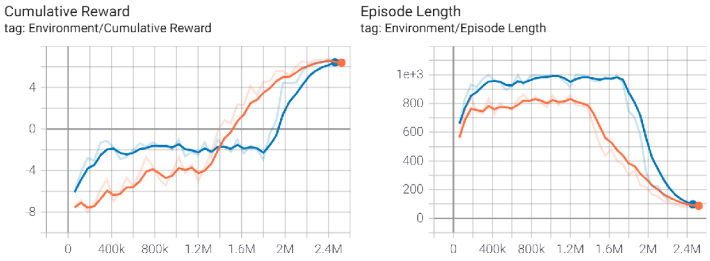
4.1 Analýza sietí

Táto sekcia je určená na analýzu a prieskum schopností rôznych algoritmov vysporiadať sa s danými úlohami v scéne. Nakoľko nebolo cieľom siete donútiť vyriešiť všetky úlohy, čo by bolo možné procesom zdĺhavých optimalizácií buď systému odmien alebo fitness funkcie, je taktiež neúspech validným výsledkom z ktorého môžme rovnako vyvodiť záver ako z úspechu.

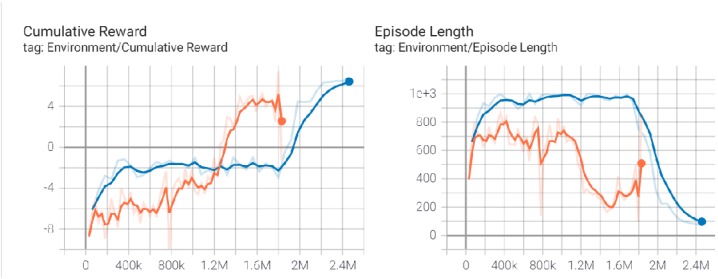
4.1.1 Dopad imitačného učenia na učenie posilované

Existujú dva spôsoby ako pomocou imitačného učenia zefektívniť učenie posilované. Jedným je zapnúť imitačné posilovanie na začiatku a postupom času ho vypnúť, alebo ho nechať aktivované po celú dobu učenia. Oba spôsoby vo výsledku učenie zefektívnia, no prinášajú so sebou rôzne nevýhody. U spôsobu kedy sme postupne od imitačného učenia upúšťali, pozorujeme rýchlejšie učenie ktoré ale v porovnaní s posilovaným učením konverguje. Výhoda tohto prístupu je, že aj keď značne nezrýchlime dokončenie trénovania siete, začíname s kratšími časmi dokončenia úloh, a rýchlejšie agenta dostaneme cez časť tréningu kde sa agent pokusom a omylom snaží porozumieť úlohu. Výsledkom je väčšia pravdepodobnosť že sa agent nestratí v lokálnom minime, a teda je trochu menej náchylný na neideálne nakonfigurované systémy odmien.

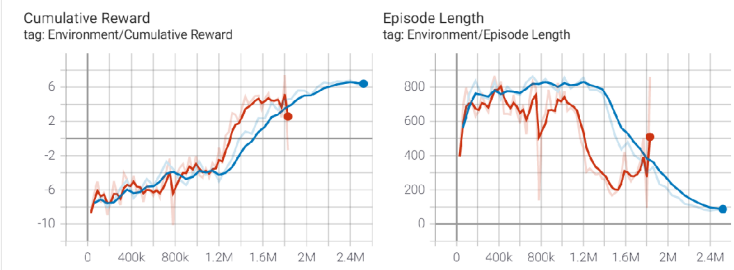
V porovnaní s týmto spôsobom je spôsob kedy neupustíme od imitačného učenia ešte rýchlejší vo svojom tréningu. Tu ale bojujeme s horšou optimalizáciou úloh, nakoľko naše úlohy sú tvorené pre dokončenie v čo najrýchlejšom čase ako výsledok implementácie existenčného trestu. To je ako sme už spomenuli v rozpore s odmenami, ktoré imitačné učenie získava za vytváranie čo najviac akcií. Z toho taktiež vyplýva vyššia nestabilita, kde sa môže taktika narušiť ak imitačnú časť skúsime v neskorších štádiách vypnúť (koniec krivky udržaného učenia v grafoch). Tento spôsob, ako aj nami nepoužiteľný spôsob plného imitačného učenia, je preferovaný ak nepotrebujeme úlohy optimalizovať k rýchlosti či efektívnosti.



(a) imitačné učenie upustené



(b) Imitačné učenie udržané



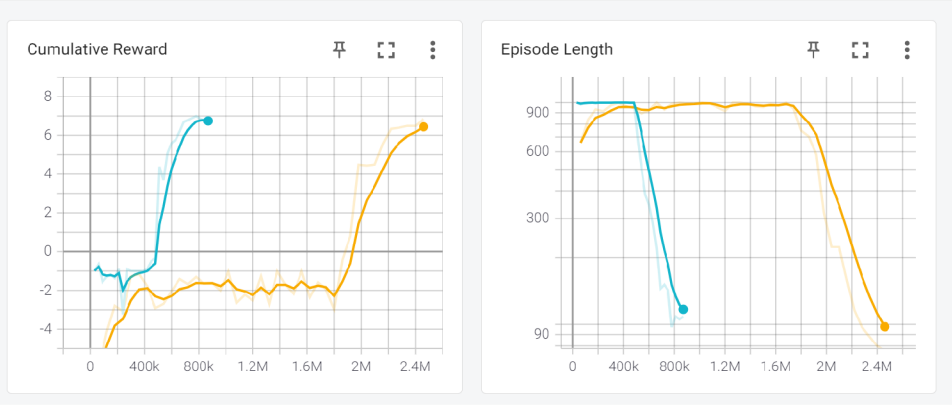
(c) porovnanie variácií udržaného imitačného učenia (červená) a upusteného (modrá)

Obr. 4.1: Grafy znázorňujúce efekty spôsobov imitačného učenia na učenie posilované

4.1.2 Dopad zložitosti

Zložitosť úlohy ktorú poskytuje scéna hrá veľkú rolu na výkon algoritmov učenia. Ako možno zložitosť pochopiť sme si vysvetlili v implementácií [3.6.1]. Nakoľko sme postupne hernú scénu vyvijáli počas analyzovania sietí, máme k dispozícii dáta ktoré môžme využiť na zmeranie tohto dopadu.

Na obrázku [4.2] sú scény v konečnom štádiu vývoja. Oranžová čiara reprezentuje štatistika finálneho vzhľadu scény. Modrá čiara reprezentuje rovnakú scénu bez dier a náhodných prvkov. Obe sú trénované posilovaným učením paralelizovaným na 9 scén. Obe mali možnosť získať rovnakú možnú maximálnu odmenu. Na prvý pohľad vidíme že kompletná scéna je veľmi pravdepodobne riadovo komplexnejšia. V tabuľke [4.1] vidíme, že časovo zabralo algoritmu vyriešiť celú úlohu takmer trikrát dlhšie ako úlohu bez náhodných prvkov, a bolo potrebných nad 2.8-krát viac krokov. Z grafu tiež dokážeme vyčítať, že agent prestal padať do priepastí okolo 400-tisíc krokov v tréningu, kde vidíme že krivka stúpla na hodnotu odmeny tam, kde začala krivka scény bez dier. Môžme teda usúdiť, že za väčšinu zvýšenia komplexity môžu náhodné prvky.



Obr. 4.2: Grafy kompletnej scény (oranžová) a scény bez priepastí a náhodných prvkov (modrá)

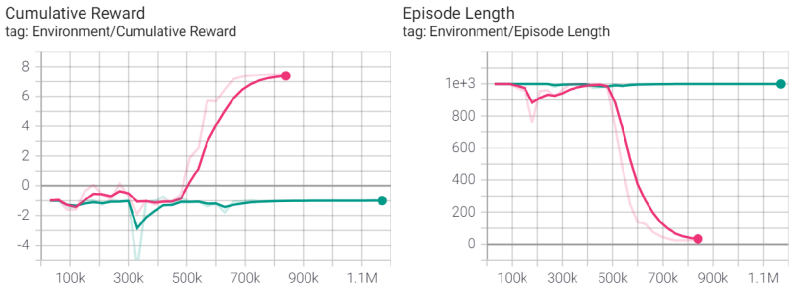
Pre referenciu bola konečná dĺžka epizód pre siete pod 76 akcií, a získaná odmena nad 6.6.

	čas	krokov
nekompletná	27m 15s	870 000
kompletná	81m 42s	2 460 000
zmena [%]	199.82	182.76

Tabuľka 4.1: Porovnanie kompletnej scény a scény bez priepastí a náhodných prvkov

Náhodnosť

Náhodnosť v scéne je asi najväčším faktorom tvorenia zložitosti pre neurónové siete. Vyplýva to z neistoty, ktorú implementuje ako už do systému odmien pre posilované učenie, tak do nepresnosti inštruktáže pre učenie imitačné. Agent totižto zrazu nemusí získavať odmeny za rovnaké akcie, spočívajúcej čisto na šanci. Je totižto zložité si nájsť spojitosti v neustále sa meniacom systéme ak nevieme podmienky týchto zmien a nedokážeme presne určiť čo nás doviedlo k úspechu respektíve neúspechu.



Obr. 4.3: Graf scéna bez náhodnosti (ružová) a s náhodnosťou (zelená)

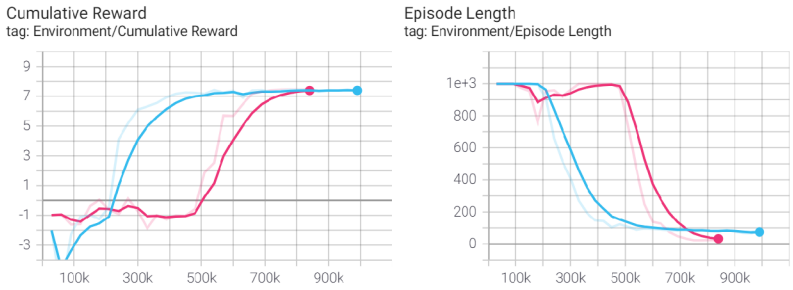
V grafe [4.3] využívame rovnakú, extrémne zjednodušenú scénu s posilovaným učením. V tej má agent dotlačiť block do cieľa a prekážajú mu len steny. V jednej scéne sa šírka týchto stien nemení, a otvor je stále na rovnakom mieste. Taktiež je ťažšie navigovať tlačený blok na ďalšie vzdialenosti, takže celková zložitosť môže byť vyššia ako u dokončenej úlohy.

V ďalšej scéne sa ale ich šírka mení, a teda otvor k cieľu sa taktiež presúva. V scéne bez náhodnosti sa úloha optimalizovala po 22 minútach. Pri inklúzii náhodnosti je optimalizácia v nedohľadnu. Agentovi sa v tejto scéne občas podarí úlohu dokončiť, ale kedže je cesta k úspechu rozdielna v ďalšej epizóde, nedokáže správne aplikovať svoje skúsenosti. Často takéto učenie vyplýva v zaseknutí sa na lokálnom minime - agent akceptuje odmeny či tresty ktoré získa a nesnaží sa vyhľadávať iné možnosti.

U takéhoto druhu úlohy nemusí pomôcť ani imitačné účenie, pretože nie sme schopný agentovi reálne ukázať všetky možnosti umiestnenia otvoru. Potom sa agent snaží napodobňovať akcie v nesprávnych situáciách, čo nesúhlasí s odmenami získanými signálom posilovaného učenia [4.4].

Obr. 4.4: Graf scéna bez náhodnosti (ružová) a s náhodnosťou za využitia imitačného učenia (oranžová)

Naopak, pri odstránení náhodnosti má imitačné učenie ešte väčšiu výhodu, skrátením času učenia o takmer polovicu[4.5][tabuľka 4.2].



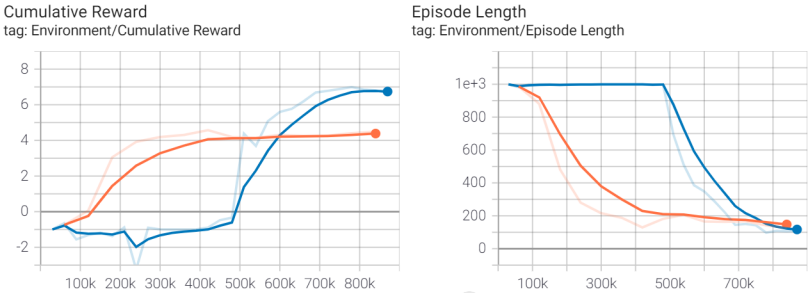
Obr. 4.5: Implementácia imitačného učenia (bledomodrá) nad zjednodušenou scénou bez priepastí a náhodnosti, v porovnaní s posilovaným učením (ružová)

	čas
posilované	27m 15s
posilované + imitačné	14m 15s
zmena [%]	-47.71%

Tabuľka 4.2: Čas implementácie imitačného učenia k posilovanému do scény bez náhodnosti

Sekvenčnosť

Ďalší faktor ovplyvňujúci zložitosť úloh je sekvenčnosť—počet akcií, ktoré agent musí vykonať za sebou v presnom poradí. To je celkom jednoznačné, keďže väčšina posilovaného učenia spočíva na pokus-omyl princípe. Čím viacej akcií je vyžadovaných za sebou, tým menšia šanca je že na ne agent náhodne narazí [4.6]. Preto sa musíme spoľahnúť na robustne vytvorené odmenové systémy, ktoré agenta nútia do situácii kde sú tieto stavy pravdepodobné.

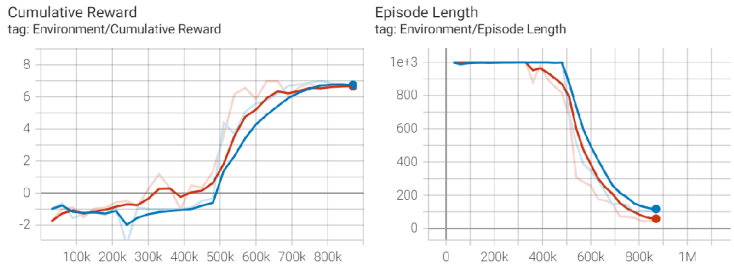


Obr. 4.6: Graf úlohy s jednotným cieľom (oranžová) a sekvenčným cieľom (modrá) bez náhodnosti

4.1.3 Dopad paralelizácie

Paralelizácia ako sme už spomenuli dokáže ovplyvniť rýchlosť učenia. Graf [4.7] vidíme že je takmer rovnaký, s rozdielom, že neparalelizované učenie má malú výhodu v efektívnosti krokov. To vedie k rýchlejšej konvergencií k maximu čo sa týka krokového hľadiska, a kratším epizódam. Za dôsledok to má chaos implementovaný viacerými agentmi ktorých logika sa neupravuje dostatočne často. Modrá sieť predstavuje zjednodušenú scénu z [4.2].

Ak ale graf vezmeme z časového hľadiska, zistíme, že výsledné hodnoty sme dostali pre sieť učenú bez paralelizácie v čase 1 hodina 20 minút 5 sekúnd, a pre paralelizáciu 9 agentmi za 27 minút 15 sekúnd. Výsledný časový rozdiel bol -65,97%, takmer dve tretiny menej v prospech siete s paralelizáciou.



Obr. 4.7: Grafy posilovaného učenia bez (červená) a s paralelizáciou (modrá)

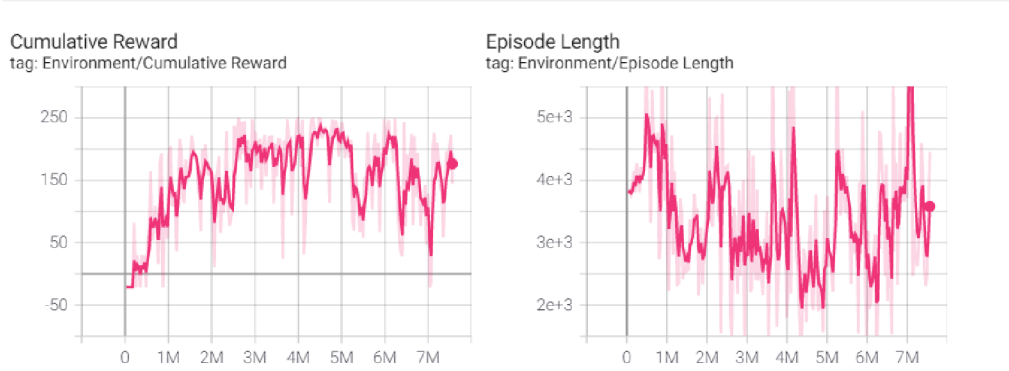
	čas
jednotlivec	80m 5s
paralelizácia	27m 15s
zmena [%]	-65.97

Tabuľka 4.3: Ušetrený čas paralelizácie v porovnaní s jednotlivcom

4.1.4 Labyrinth

Reinforcement Learning

Labyrinth je scénou ktorá obsahuje vysokú sekvenčnosť, a zložitú implementáciu odmenového systému. Výsledkom je, že posilované učenie je nevhodné pre túto úlohu. Scéna je tak zložitá, že je v pre agenta v podstate už jedno, či do scény implementujeme náhodnosť, pretože tak či tak sa mu podarí úlohu dokončiť občas čírou náhodou. Výsledkom bývajú extrémne šumivé grafy bez žiadneho pozorovateľného progresu [4.8].



Obr. 4.8: Výsledky učenia labyrintu s posilovaným učením

Neuroevolution

Neuroevolúcia si vedie v tejto úlohe oveľa lepšie. Po spustení siete dokáže NEAT algoritmus vyhľadať koniec labyrintu kdekoľvek v scéne a optimalizovať trasu k nemu počas minút. To je možné vďaka tomu, že pri paralelizácii neuroevolúcie je zásadný rozdiel oproti paralelizácii u posilovaného učenia. U posilovaného učenia paralalizácia len znamená, že sa agentova logika duplikuje n-krát, a v podstate premýšľajú rovnako s menšími rozdielmi na základe náhodnosti v ich taktike. Pri neuroevolúcií môže byť každý jedinec rozdielny, a najsilnejší prežijú. Nakoľko najsilnejších vyberieme na základe toho, ako veľa labyrintu pokryjú (simulovaná pamäť [3.7.9]), tak sú v prvých generáciách vybraní najlepší prieskumníci. Potom, ak jeden z nich nájde cieľ bludiska, ihneď sa stáva najsilnejším jedincom. Ten sa potom len propaguje sieťou až kým sa všetci jedinci generácií dokážu dostať do cieľa.

4.2 Playtesting

Playtesting je nevyhnutná časť vývoja v hernom priemysle. Spočíva to z faktu, že akokoľvek sa vývojár snaží navigovať hráča, nedokáže vziať v úvahu nekonečné permutácie faktorov v hre a ľudskom zmýšľaní. Často existuje viac spôsobov ako riešiť problémy, a výber vhodného je zvyčajne subjektívny. Výsledkom teda môže byť že vývojári nemusia súhlasiť s hernou komunitou na tom, ako hrať vlastnú hru.

Druhým dôvodom testovania hry už počas vývoja je hľadanie chýb. Nakoľko sa herný priemysel neustále rozvíja a implementujú sa nové technológie a nápady, ktoré zaberajú čoraz viacej času a nákladov, a hry sa stávajú väčšími a komplexnejšími, je obrovský potenciál na vzniknutie problémových sekcií v rôznych častiach hernej funkcionality. Ako sme už spomenuli tak tieto chyby sú často nemožné odchytiť vývojármi hier nakoľko majú presne

predom premyslenú cestu ako hru hrať od začiatku do konca, ktorá je zvyčajne v poriadku, no kedykoľvek by sme len trochu odbočili od tohto „Vyšlapaného chodníka“ môžme naraziť na mnoho neúmyselných nedopatrení. V tejto časti v súčastnosti už prestáva byť efektívna manuálna kontrola, nakoľko testeri sú tiež ľudia, ktorí majú určité spôsoby na riešenie problémov a taktiež limitovaný čas. Nakoniec členov testovacieho týmu býva v desiatkách a hráčov hry môže byť tisíce či dokonca milióny, tak je zrejmé, že takáto skupina ľudí nedokáže prejsť všetky prípady a spôsoby hrania.

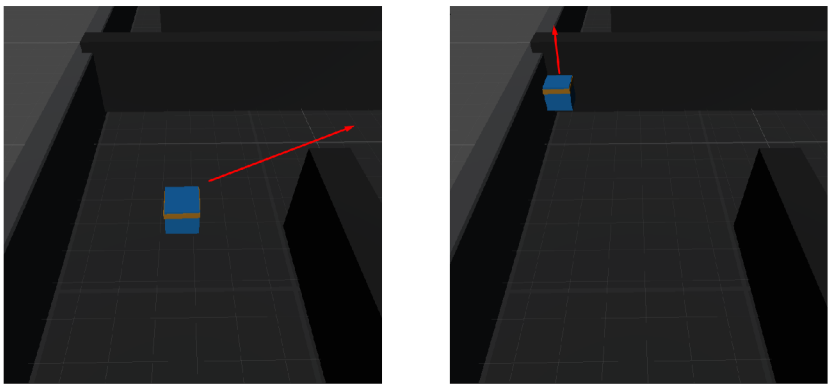
4.2.1 Vyhľadávanie exploitov

Nakoľko počítač nie je potrebné kompenzovať v hodinách, a nie je viazaný ľudským premýšlaním, je na prvý pohľad perfektný kandidát na testovanie. To spočíva z niekoľkých faktorov: Reinforcement learning je metóda, ktorou dokážeme stanoviť presné koncové ciele pasáží hry. Spočíva to v tom, že akoukoľvek cestou hráč rieši úlohu, zvyčajne býva definované miesto alebo cieľ, ku ktorému musia všetky spôsoby konvergovať. Predstavme si miestnosť s priepasťou v strede. Na jednom konci je hráč, na druhom je cieľ. Určený spôsob dostania sa k cieľu je využitie akcie skoku, alebo vyriešenie hádanky na vytvorenie mostu na prekročenie. Vnútorná fyzika herného enginu je ale zložitá, a existujú spôsoby, ako zneužívať rôznej geometrie prostredia na získanie dostatku rýchlosti na prekročenie priepasti v jedinom snímku hry. V takomto prípade nedokážeme hráčovi zabrániť priepasť prejsť nakoľko máme limitovaný počet kalkulácií na zistenie toho či máme hráča eliminovať na základe nachádzania sa vo vzduchu alebo podobne.

4.2.2 Prípady

Analýza labyrintu

V prípade labyrintu má hráč prechádzať prostredie a prehľadávať možné cesty do cieľa. Herný engine Unity ale poskytuje možnosť agentovi „šplhať“ v rohoch, kde sa stretávajú dve steny. Je za potreby precízne ovládanie, no je to možné. Agent vytvorený pomocou neuroevolúcie túto chybu odhalil behom sekúnd. To mu dovolilo maximalizovať odmeny získané dostaním sa na koniec bludiska, nakoľko ho celé obišiel. Riešením bolo zmrazenie agentovej pozície na osi y v hernej scéne.



(a) Očakávaná cesta

(b) Cesta vzatá agentom neuroevolúcie

Obr. 4.9: Racionálny postup očakávaný človekom a postup agenta hlbokého učenia

Originálna scéna

V originálnej scéne bolo viacej príkladov, u ktorých bol agent schopný odhaliť naprogramované nedostatky v scéne a tým obísť požadované úlohy.

- Rýchlosť animácie otvárania/zatvárania dverí: Agent dokázal ihneď po reštartovaní scény dokončiť úlohu bez toho, aby dokončil sekvenciu aktivovania dverí. Riešením bolo zrýchlenie animácie uzatvárania dverí.
- Gravitačné vlastnosti v scéne: Agentova rýchlosť v porovnaní s jeho trením a gravitačnou silou pôsobiacu na jeho objekt nebola dostatočná na to, aby vypadol zo scény. Namiesto toho dieru „preletel“ a dostal sa do cieľa. Riešením bolo upravenie gravitačnej sily v scéne a fyzikálnych vlastností agenta.
-

4.2.3 Regresné testovanie stavov a funkcionalít

Ak sa spoločne s hrou vyvíja aj neurónová sieť, je možné túto sieť automaticky aplikovať do vyskúšania funkcionality nových implementácií. V našej bakalárke sme na to narazili mnohokrát:

- Ako test fyzikálnych javov: Agent nebol schopný kvôli hmotnosti kvádra ho schopný odtlačiť, museli sme ho zľahčiť, aby sieť mohla postupovať.
- Pri implementácii pressure plate: Pressure plate nebol aplikovaný ako trigger ale ako colission box, čo znamenalo že agent sa naň nemohol dostať, nedokázal otvoriť dvere, a sieť nám nezobrazila žiadny progres v učení.
- Pri implementácii gate: Brána nemala správne nastavenú animáciu, neotvárala sa. Sieť sme našli v bezcieľnom stave po dotlačení bloku na pressure plate. Pri ďalšom pokuse bola animácia nesprávne aplikovaná, otvárali sa len jedny vráta brány. To agentovi neprekážalo, no bolo to možné pozorovať počas učenia
- Počas náhodnej generácie: generovali sa nemožné kombinácie pre sieť, čo sa ukázalo v grafe tréningu siete.

4.2.4 Zoznam problémov a prekážok počas implementácie

Reward farming

Problém sa preukazuje ako neočakávané ale opakované a optimalizované správanie sa agenta počas trénovania neurónovej siete. Zvyčajne znamená chybu v odmenovom systéme, no môže znamenať aj reálny dizajnový prehľad v hre (najmä ak sa jedná o hru s cieľom získať vysoké skóre). Riešením býva zníženie bodovej odmeny, zavedenie trestu, alebo opravenie nečakanej hernej funkcionality zneužívanej agentom. Pri našej implementácii sa objavili:

- pri implementácii kvádru do scény: Kedykoľvek agent uchopil block, dostal odmenu. Túto odmenu dokola zbieral opakovaným úchopom kvádra.
- pri implementácii pressure plate do scény: Podobne ako u prvého prípadu agent zbieral odmenu opakovaným posúvaním kvádra z a na plochu pressure plate.

- pri implementácii krátkeho systému pamäte do labyrint scény: Agent si pre seba vytvoril okrúhlu trasu dostatočné krátku na to, aby zbieral odmenu za navštívenie novej oblasti labyrintu ihneď po zabudnutí na danú oblasť.

Local minimum

Môže vypadať ako úplná neočakávané alebo minimálne správanie sa agenta. Agent často môže vypadať, že nevykonáva žiadne úlohy. Je spôsobený neschopnosťou siete získať nové odmeny. Prejaviť sa môže na základe implementačnej chyby na strane hry: Nefunguje funkcionalita hry, nesprávna geometria hernej scény, ale aj na strane odmenového systému: ďalšia odmena je príliš ďaleko alebo nepravdepodobná k naučeniu sa.

- pri implementácii labyrintu najprv nefungoval systém odmeny za pamäť, tak agent stál na mieste.
- pri implementácii labyrintu nefungoval cieľ, takže o neho agent nemal záujem a len zbieral body
- pri implementácii príliš vysokej náhodnej generácie v originálnej scéne sa agent nedokázal vysporiadať so zložitosťou scény a úlohu vzdal, držiac block za minimálnu odmenu

Kapitola 5

Záver

Cieľom práce bolo implementovať a analyzovať algoritmy hlbokého učenia do herných prostredí v Unity. K tomu bolo nutné si preštudovať problematiku strojového učenia, neurónových sietí, a rôznych vied v tejto vetve umelej inteligencie. Z tých boli naštudované postupy hlbokého učenia ako učenie s učiteľom, bez a najmä posilované učenie. Ďalej sme sa zoznámili s algoritmami imitačného učenia, ako je GAIL, a neuroevolúcie, ako NEAT. Spomenuté algoritmy sme po oboznámení sa implementovali do herného engine Unity pomocou rôznych knižníc, ako ML-agents a UnityNEAT. Pre naše účely bola treba modifikácia týchto knižníc. K implementácii boli taktiež potrebné znalosti vytvárania herných prostredí v Unity, ako už z dizajnovej stránky tak programovacej k vytváraniu skriptov ktoré slúžili ako médium neurónových sietí. Výsledkom je implementácia scén na ktorých je možné dané algoritmy pomocou knižníc spúšťať a analyzovať ich výsledky v rôznych kategóriách ako stanovených nami. Na záver sme načrtli pár možných dôvodov a návrhov na využitie hlbokého učenia k vývoju hier, ktoré vyplynuli zo skúseností získaných využívaním a analýzou týchto prístupov učenia. Bakalársku prácu je možné rozšíriť o rôzne druhy herných scén, analyzovať úpravu hyperparametrov sietí, pozorovanie iných algoritmov či dokonca postupov učenia a konkrétnejšiu aplikáciu tejto vedy ku komerčným účelom v hernom priemysle.

Literatúra

[1] Bansal, S. *Agents in Artificial Intelligence* [online]. August 2019 [cit. 2021-05-10]. Dostupné z: https://www.geeksforgeeks.org/agents-artificial-intelligence/.

[2] Brownlee, J. *Difference Between Classification and Regression in Machine Learning* [online]. Máj 2019 [cit. 2021-05-10]. Dostupné z: https://machinelearningmastery.com/classification-versus-regression-in-machine-learning/.

[3] DeepAI. *Hidden Layer* [online]. 2021 [cit. 2021-05-10]. Dostupné z: https://deepai.org/machine-learning-glossary-and-terms/hidden-layer-machine-learning.

[4] Ho, J. a Ermon, S. Generative adversarial imitation learning. *ArXiv preprint arXiv:1606.03476*. 2016.

[5] IBM Cloud Education. *Machine Learning* [online]. Júl 2020 [cit. 2021-05-10]. Dostupné z: https://www.ibm.com/cloud/learn/machine-learning.

[6] Debnath, S., Devos, A., Heiden, E., Julian, R. a Khatana, F. *Humanoid Imitation Learning from Diverse Sources* [online]. 2017 [cit. 2021-05-10]. Dostupné z: https://uscresl.github.io/humanoid-gail/.

[7] Electronic Arts Inc. *Imitation Learning with Concurrent Actions in 3D Games* [online]. 2021 [cit. 2021-05-10]. Dostupné z: https://www.ea.com/seed/news/seed-imitation-learning-concurrent-actions.

[8] Doggers, P. *Leela Chess Zero Beats Stockfish 106-94 In 13th Chess.com Computer Chess Championship* [online]. 2020 [cit. 2021-05-10]. Dostupné z: https://www.chess.com/news/view/13th-computer-chess-championship-leela-chess-zero-stockfish.

[9] Wikipedia. *Markov decision process* [online]. 2021 [cit. 2021-05-10]. Dostupné z: https://en.wikipedia.org/wiki/Markov_decision_process.

[10] Unity Technologies. *ML-Agents Toolkit Overview* [online]. 2021 [cit. 2021-05-10]. Dostupné z: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md#imitation-learning.

[11] Kavlakoglu, E. *AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference?* [online]. Máj 2020 [cit. 2021-05-10]. Dostupné z: https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks.

[12] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I. et al. Playing atari with deep reinforcement learning. *ArXiv preprint arXiv:1312.5602*. 2013.

[13] Nielsen, M. *Using neural nets to recognize handwritten digits* [online]. December 2019 [cit. 2021-05-10]. Dostupné z: http://neuralnetworksanddeeplearning.com/chap1.html.

[14] NVIDIA Corporation. *NVIDIA DLSS* [online]. 2021 [cit. 2021-05-10]. Dostupné z: https://www.nvidia.com/cs-cz/geforce/technologies/dlss/.

[15] Osman, S., Stafford, J. R. a Rico, J. F. *Automated artificial intelligence (AI) control mode for playing specific tasks during gaming applications*. Google Patents, január 12 2021. US Patent 10,888,788.

[16] Schulman, J., Klimov, O., Wolski, F., Dhariwal, P. a Radford, A. *Proximal Policy Optimization* [online]. Júl 2017 [cit. 2021-05-10]. Dostupné z: https://openai.com/blog/openai-baselines-ppo/.

[17] ADL. *An introduction to Q-Learning: reinforcement learning* [online]. September 2018 [cit. 2021-05-10]. Dostupné z: https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/.

[18] Deeplizard. *Reinforcement Learning - Goal Oriented Intelligence* [online]. 2018 [cit. 2021-05-10]. Dostupné z: https://deeplizard.com/learn/video/rP4oEpQbDm4.

[19] Salian, I. *SuperVize Me: What's the Difference Between Supervised, Unsupervised, Semi-Supervised and Reinforcement Learning?* [online]. August 2018 [cit. 2021-05-10]. Dostupné z: https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/.

[20] OpenAI. *Part 2: Kinds of RL Algorithms* [online]. 2018 [cit. 2021-05-10]. Dostupné z: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

[21] Rosenblatt, F. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[22] Salimans, T., Ho, J., Chen, X., Sidor, S. a Sutskever, I. Evolution strategies as a scalable alternative to reinforcement learning. *ArXiv preprint arXiv:1703.03864*. 2017.

[23] Shannon, C. E. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*. Taylor & Francis. 1950, zv. 41, č. 314, s. 256–275.

[24] Kumar, N. *Sigmoid Neuron — Building Block of Deep Neural Networks* [online]. Marec 2019 [cit. 2021-05-10]. Dostupné z: https://towardsdatascience.com/sigmoid-neuron-deep-neural-networks-a4cd35b629d7.

[25] Stanley, K. O. a Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation*. MIT Press. 2002, zv. 10, č. 2, s. 99–127.

[26] Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O. et al. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *ArXiv preprint arXiv:1712.06567*. 2017.

[27] Hajaj, Y. *Introduction to Supervised, Semi-supervised, Unsupervised and Reinforcement Learning* [online]. Október 2020 [cit. 2021-05-10]. Dostupné z: `https://www.baeldung.com/cs/machine-learning-intro`.

[28] Sutton, R. S. a Barto, A. G. *Reinforcement learning: An introduction*. 2. vyd. MIT press, 2018. ISBN 9780262193986.