

Automatické řízení pro hru Trackmania

Bc. Daniel Mago

Diplomová práce
2025



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Daniel Mago
Osobní číslo: A23736
Studijní program: N0613A140022 Informační technologie
Specializace: Softwarové inženýrství
Forma studia: Prezenční
Téma práce: Automatické řízení pro hru Trackmania
Téma práce anglicky: Automated Driving for Trackmania

Zásady pro vypracování

- Vypracujte literární rešerš na danú tému.
- Preskúmajte možné vhodné modely metód umelej inteligencie pre automatické riadenie auta v hre Trackmania.
- Otestujte vybrané modely pomocou nástroja TMRL.
- Vyhodnotte a porovnajte získané výsledky.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. RAMSUNDAR, Bharath a ZADEH, Reza Bosagh. TensorFlow for deep learning: from linear regression to reinforcement learning. Beijing: O'Reilly Media, 2018. ISBN 9781491980422.
2. SUTTON, Richard S. a BARTO, Andrew G. Reinforcement learning: an introduction. Adaptive computation and machine learning. Cambridge, Mass.: MIT Press, ©1998. ISBN 9780262257053. Dostupné také z: <https://proxy.k.utb.cz/login?url=http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6267343>.
3. ŠRANKO, Jozef. Analýza možností získávání dat pro automatizované řízení ve hře Trackmania. Bakalářská práce, vedoucí Viktorin, Adam. Zlín: Univerzita Tomáše Bati ve Zlíně. Fakulta aplikované informatiky, Ústav informatiky a umělé inteligence, 2024. Dostupné také z: <http://hdl.handle.net/10563/56058>.
4. TMInterface. Online. 2023. Dostupné také z: <https://donadigo.com/tminterface/>.
5. BOUTELLER, Yann. TMRL Documentation. Online. TMRL Documentation. 2022. Dostupné z: <https://tmrl.readthedocs.io/en/latest/>.
6. GITHUB PB4GIT. Trackmania_rl_public. Online. 2023. Dostupné také z: https://github.com/pb4git/trackmania_rl_public.

Vedoucí diplomové práce: **Ing. Adam Viktorin, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **27. října 2024**
Termín odevzdání diplomové práce: **2. června 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Diplomová práca má za cieľ čitateľa oboznámiť s problematikou spätnoväzobného učenia, predstaviť mu úspešné projekty jeho uplatnenia od rôznych vedeckých tímov, a objasniť spôsob fungovania dvoch vybratých populárnych algoritmov – Deep Q-Networks a Soft Actor-Critic. Práca poskytuje detailný popis princípov funkcie týchto algoritmov, ich nedostatky a ako sa im možno vyhnúť, a taktiež podrobne popisuje proces ich praktickej implementácie pre využitie v počítačovej hre Trackmania (2020). V práci sú viaceré návody, ktoré čitateľovi objasnia, ako môže natrénované modely algoritmov otestovať na vlastnom zariadení. Výstupné dáta z tréňovaných algoritmov sú zozbierané a následne reprezentované vo viacerých informatívnych grafoch, pre ktoré sú vytvorené aj popisy trendov, ktoré sú na nich viditeľné. V prílohe práce sú manuálne vytvorené vizuálne porovnania efektivity natrénovaných modelov oproti ľudskému výkonu.

Kľúčové slová: spätnoväzobné učenie, trackmania, tmrl, sac, dqn

ABSTRACT

This diploma thesis aims to familiarize the reader with the topic of reinforcement learning, introduce them to successful projects implementing reinforcement learning made by various research teams as well as shed a light on the inner workings of two selected popular algorithms – Deep Q-Networks and Soft Actor-Critic. The thesis provides detailed descriptions of the core principles of these algorithms, their shortcomings and how to possibly avoid them, and also goes into detail in regards to practically implementing them for use in the videogame Trackmania (2020). This thesis provides various guides meant to explain the process of testing pre-trained models of the aforementioned algorithms for the reader to try on their own computer. The output data produced by the training process of these algorithms has been collected and turned into informative plots, which are accompanied by descriptions of the trends observed in them. The attachments to this thesis contain handmade visual comparisons of the effectiveness of these algorithms when put against an actual human.

Keywords: reinforcement learning, trackmania, tmrl, sac, dqn

Podakovanie

Týmto by som sa chcel poďakovať svojim rodičom za ich nikdy neprestávajúcu podporu a trpezlivosť. Tiež by som sa chcel poďakovať svojej priateľke za to, že bola počas štúdia neustále mojou oporou. Vďaku venujem aj vedúcemu práce Ing. Adamovi Viktorínovi, Ph.D. za jeho spätnú väzbu a trpezlivosť pri jej vypracovávaní.

OBSAH

ÚVOD.....	13
I TEORETICKÁ ČÁST	14
1 SÉRIA HIER TRACKMANIA	15
1.1 TRACKMANIA (2003)	15
1.1.1 Trackmania Original	15
1.2 TRACKMANIA SUNRISE	15
1.3 TRACKMANIA NATIONS A TRACKMANIA UNITED	16
1.3.1 Trackmania Nations	16
1.3.2 Trackmania United.....	16
1.3.3 Trackmania Nations & United Forever	16
1.4 TRACKMANIA 2 (TRACKMANIA ²)	16
1.5 TRACKMANIA TURBO	17
1.6 TRACKMANIA (2020)	17
2 TOOL-ASSISTED SPEEDRUN	18
2.1 ČO JE TO TOOL-ASSISTED SPEEDRUN	18
2.1.1 Význam TAS.....	19
2.2 TAS V SÉRII TRACKMANIA.....	19
2.2.1 Funkcie TMInterface.....	20
3 VYUŽITIE ALGORITMOV AI A RL V HRÁCH.....	22
3.1 ČO JE TO REINFORCEMENT LEARNING (RL)	22
3.2 ČO JE TO DEEP REINFORCEMENT LEARNING (DRL)	23
3.3 PRÍKLADY APLIKOVANIA ALGORITMOV RL V HRÁCH.....	24
3.3.1 Projekt AlphaGo.....	24
3.3.2 Projekt OpenAI Five	24
3.4 VYUŽITIE ALGORITMOV AI A RL V SÉRII TRACKMANIA	25
3.4.1 Projekt Linesight	25
3.4.2 Projekt Trackmania_AI.....	26
3.4.3 Trackmania AI vytvorené YouTube-rom Yosh	26
4 VYUŽITÉ NÁSTROJE	28
4.1 OPENPLANET	28
4.2 FRAMEWORK TMRL	28
5 ALGORITMY RL VYUŽITÉ V TEJTO PRÁCI.....	32
5.1 DEEP Q-NETWORK (DQN).....	32
5.1.1 Q-Learning	32
5.1.2 Proces tréovania	33
5.1.3 Architektúra Deep Q-Network	34

5.1.4	Deep Q-Learning	35
5.1.5	Prioritized Experience Replay	36
5.1.6	Zhrnutie DQN.....	38
5.2	SOFT ACTOR-CRITIC	38
5.2.1	Entropia v SAC, Soft Policy Iteration	39
5.2.2	Pôvodná forma SAC a jej vylepšenie.....	39
5.2.3	Polyak Averaging	40
5.2.4	Dvojica Q-Funkcií.....	41
5.2.5	Proces trénovania	42
II	PRAKTICKÁ ČASŤ	44
6	PRÍPRAVA NÁSTROJOV.....	45
6.1	TRACKMANIA 2020	45
6.2	SKRIPTOVACIA PLATFORMA OPENPLANET PRE HRU TRACKMANIA 2020.....	48
6.3	FRAMEWORK TMRL	49
6.3.1	Úpravy v súboroch TMRL	50
6.3.2	PyTorch CUDA	50
6.4	PRÍPRAVA SÚBORU PRE IMPLEMENTÁCIU VLASTNÉHO MODELU	51
7	IMPLEMENTÁCIA VLASTNÝCH MODELOV.....	53
7.1	IMPLEMENTÁCIA DQN	55
7.1.1	Definícia „action space“	55
7.1.2	Trieda DQNNetwork.....	56
7.1.3	Trieda DQNActorModule	57
7.1.4	Funkcia epsilon_scheduler().....	58
7.1.5	Trieda DQNTrainingAgent	59
7.1.6	Triedy PrioritizedMemory a PERTrainingOffline	63
7.1.7	Úprava súboru config.json	65
7.2	IMPLEMENTÁCIA SAC	68
7.2.1	Príprava konštánt pre využitie neskôr	68
7.2.2	Pomocná funkcia process_observation	69
7.2.3	Trieda MLPNetwork	70
7.2.4	Trieda LidarSACPolicy	72
7.2.5	Trieda LidarQFunction.....	75
7.2.6	Trieda LidarSACActorCritic	76
7.2.7	Trieda ImprovedSACAgent	76
7.2.8	Zapnutie zberu dát pomocou wandb	83
7.2.9	Úprava súboru config.json	84
7.3	SNAHA O IMPLEMENTÁCIU ALGORITMU PPO.....	87
8	PROCES TRÉNOVANIA.....	88
8.1	PRÍPRAVA TRÉNOVANIA	88
8.1.1	Nahratie vlastnej odmeny	92
8.1.2	Automatický zber užitočných tréningových dát pomocou Wandb	93
8.2	TRÉNOVANIE ALGORITMU DQN	95

8.2.1	Proces tréovania DQN s pozmenenými nastaveniami	98
8.3	TRÉNOVANIE ALGORITMU SAC.....	101
8.4	POROVNANIE NAJLEPŠÍCH ZÍSKANÝCH ČASOV NATRÉNOVANÝCH MODELOV	106
8.4.1	Vytvorenie programu pre hromadné premenovanie súborov.....	107
8.4.2	Najrýchlejšie časy algoritmu DQN	108
8.4.3	Najrýchlejšie časy algoritmu SAC	108
8.5	POROVNANIE ROBUSTNOSTI NATRÉNOVANÝCH MODELOV	108
8.5.1	Algoritmus DQN.....	109
8.5.2	Algoritmus SAC.....	109
8.6	POROVNANIE VÝSLEDKOV PROTI ĽUDSKÉMU ČASU	110
8.7	ZHRNUTIE SPRÁVANIA NATRÉNOVANÝCH MODELOV.....	110
ZÁVĚR		111
SEZNAM POUŽITÉ LITERATURY		112
SEZNAM OBRÁZKŮ		117
SEZNAM TABULEK.....		121
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....		122
SEZNAM PŘÍLOH.....		123

ÚVOD

Žijeme v dobe, kde skoro každá časť nášho života je do určitej miery ovplyvnená algoritmami umelej inteligencie - či už sa pýtame o vysvetlenie nejakej zložitej problematiky veľkého jazykového modelu ako ChatGPT; na sociálnych sieťach narážame na groteskné a nezmyselné obrázky vygenerované modelmi ako Midjourney, alebo aj na video platformách počúvame hlas, ktorý bol syntetizovaný modelom ako ElevenLabs. To, čo ešte pred piatimi rokmi znelo ako sci-fi, je dnes realitou. Nie je teda prekvapujúce, že tento trend zasahuje aj do herného priemyslu.

V tejto práci bude rozoberaná tematika automatizovaného ovládania v závodnej hre Trackmania. Konkrétnejšie povedané, pomocou rôznych nástrojov budú najprv z hry zbierané dôležité dáta (ako napríklad pozícia auta, rýchlosť auta, vzdialenosť od steny, rotácia kolies, atď.), ktoré sú potom prepracované do formy argumentov a informácií pre učenie rôznych algoritmov umelej inteligencie.

Teoretická časť sa zameriava na krátke vysvetlenie princípov hier v sérii Trackmania; využitie automatizovaného ovládania pomocou algoritmov umelej inteligencie v takzvaných TAS (Tool Assisted Speedrun) – teda v „speedrun“-och s pomocou nástrojov; rozboru využitých nástrojov ako aj popisu vybraných algoritmov umelej inteligencie, ktoré budú medzi sebou porovnávané.

V praktickej časti je rozoberaná extrakcia dôležitých dát, ktoré sú potom využité pri trénoch už vyššie spomenutých algoritmov. Výsledky dosiahnuté rôznymi algoritmami sú následne porovnané medzi sebou, ako aj s výsledkami získanými ľuďmi, ktorí sa zaoberajú automatizovaným ovládaním v hre Trackmania.

I TEORETICKÁ ČÁST

1 SÉRIA HIER TRACKMANIA

Herná séria Trackmania vznikla v roku 2003, kedy francúzske herné štúdio Nadeo vydalo prvý titul s rovnakým menom - Trackmania. Je to arkádová závodná hra, ktorá je jednoduchá na naučenie, no má veľmi vysoký strop zručnosti. Hlavnou pointou je závod na čas. Čím lepší čas hráč dosiahne, tým lepšiu medailu získa na konci. Následne môže svoj úspech nahráť na online rebríčky pre porovnanie s inými hráčmi. [1][6]

1.1 Trackmania (2003)

Princípom hry bolo s autom pretekať na predom zostavených tratiach od vývojárov, aby hráč získal menu nazývané „coppers“, alebo „cc“. Tieto trate sú dostupné v dvoch základných herných módoch – Race (závod) a Puzzle (hlavolam). V závodnom móde je hráč postavený proti trom autám, z ktorých každé auto má svoj vlastný čas na trati – Gold (zlato), Silver (striebro) a Bronze (bronz). Hráč vyhrá, ak porazí všetky autá. V režime hlavolamu je hráčovi daná trať, na ktorej chýbajú určité stavebné bloky, a hráč musí tieto bloky doplniť tak, aby dosiahol čo najlepší čas. Pomocou meny „coppers“ si môže hráč nakupovať rôzne stavebné bloky pre postavenie svojej vlastnej závodnej trate. Hráč má na výber z troch prostredí – Rally (vidiek), Snow (vysokohorské) alebo Desert (rýchlosť). Každé prostredie ponúka odlišné vlastnosti auta a ovládania. Reakcie kritikov na hru by sa dali zhrnúť ako ľahko nadpriemerné. Chválené bolo práve tvorenie vlastných tratí, a vysoká úroveň kreatívnej slobody, no kritizovanou bola vcelku dôležitá časť hry, a to ovládanie závodných aut. [1][2][3][4]

1.1.1 Trackmania Original

Hra Trackmania bola neskôr znovu vydaná v roku 2005 pod názvom *Trackmania Original*. Hlavným rozdielom bola zmena grafického engine-u na verziu **Trackmania Sunrise** a pridanie približne desiatich nových stavebných blokov. [2]

1.2 Trackmania Sunrise

V roku 2005 bol do série Trackmania vydaný nový titul - Trackmania Sunrise. Táto hra bola vytvorená na vylepšenom hernom engine, a priniesla lepšiu grafickú kvalitu ako aj vylepšené ovládanie. Dodatočne boli okrem pôvodných herných módov pridané aj nové módy – Platform (dokončenie trate s mnohými zložitými časťami bez toho, aby sa hráč „reštartoval“) a Crazy (hráč súperí s tzv. duchmi). [1][5]

1.3 Trackmania Nations a Trackmania United

Počas roku 2006 boli vydané dva nové tituly do série Trackmania. V januári to bol Trackmania Nations a v júni Trackmania United. [1]

1.3.1 Trackmania Nations

Táto hra bola vydaná zdarma, a slúžila hlavne ako reklama pre ESWC – Electronic Sports World Cup – majstrovstvá sveta v elektronických športoch. Väčšina funkcií bola ponechaná z predošlého titulu Trackmania Sunrise, no bolo pridané nové prostredie – Stadium (štadión). Zo základu hra obsahuje sto tratí pre jedného hráča, no hra je do značnej miery prezentovaná ako online hra. Trackmania Nations bola prvá hra v sérii, ktorá pridala tzv. „leaderboard“ – rebríček, kde hráči medzi sebou súperia o najlepší čas. Päť najlepších hráčov je potom zobrazených na domovskej stránke hry. [1] [6]

1.3.2 Trackmania United

Krátko po vydaní voľne dostupnej Trackmania Nations bola na trh uvedená Trackmania United, ktorá bola akési zlúčenie všetkých dovtedy vydaných hier Trackmania. Obsahuje sieťový systém typu peer-to-peer pre jednoduchšie vzájomné zdieľanie obsahu vytvoreného hráčmi, a zjednotený systém hodnotenia. [1] [6]

1.3.3 Trackmania Nations & United Forever

Vylepšené verzie pôvodných hier boli vydané v roku 2008. Bol vylepšený systém hernej fyziky, a hráči oboch hier mohli spolu pretekať na rovnakých herných serveroch. V roku 2010 došlo k zatvoreniu spoločnosti, ktorá pre Nadeo dodávala reklamy. To znamenalo že firma Nadeo prišla o značnú časť svojho príjmu, čo ich prinútilo do týchto hier pridať tzv. predplatné. Táto novinka však značne odpudila veľkú časť ich hernej komunity. [1] [6]

1.4 Trackmania 2 (Trackmania²)

Ďalší hlavný titul v sérii bola hra Trackmania 2, ktorá bola vydaná koncom roku 2011. Okrem už známych funkcií z predošlých hier bol pridaný aj „ManiaScript“, skriptovací jazyk, ktorý hráčom umožňuje pridať do hry úplne nové funkcie. Veľkým rozdielom od predošlých hier bol epizodický formát. V rokoch od 2011 do 2017 boli vydané 4 epizódy. Prvá epizóda, ktorá vyšla spolu s hrou, bola Canyon (kaňon), neskôr vo februári roku 2013 bola pridaná epizóda Stadium (štadión) a o pár mesiacov v júli epizóda Valley (údolie). Nakoniec až v roku 2017 bola vydaná posledná epizóda Lagoon (lagúna). [1] [7]

1.5 Trackmania Turbo

Pravdepodobne najpopulárnejší titul v sérii bol vydaný v roku 2016 pod názvom Trackmania Turbo. Najväčším dôvodom, prečo hra tak uspela bolo to, že bola vydaná na všetkých hlavných herných platformách – počítač, PlayStation 4 a Xbox One. Hráč môže pretekať na viac ako 200 tratiach. Okrem pôvodných bronzových, strieborných a zlatých medailí tu existuje ešte posledné, najťažšie ocenenie, tzv. majster trate (Track Master). Okrem už známeho editoru tratí bol pridaný aj generátor tratí, kde môže hráč náhodne vygenerovať celú trať. [1] [8]

1.6 Trackmania (2020)

Zatiaľ posledným vstupom do série je hra Trackmania vydaná v roku 2020. Je to tzv. mäkký reštart série Trackmania, prerobenie (remake) hry Trackmania Nations. Hra je voľne dostupná s možnosťou zaplatiť si predplatné pre odomknutie prístupu k mnohým funkciám, ako napríklad pravidelné online akcie, editor tratí, atď. Opäť sa vracia prostredie štadiónu, ktoré bolo prítomné vo viacerých predošlých hrách. Všetky stavebné bloky, ktoré hráči dobre poznali boli odstránené a úplne nahradené novými blokmi od základu. Okrem štandardnej asfaltovej závodnej trate alebo „poľných ciest“ boli pridané aj ľadové povrchy. Okrem prostredia štadiónu sa vracajú aj prostredia Rally, Snow a Desert (rally, sneh a púšť). Každé prostredie má svoje vlastné auto a viaceré vlastné stavebné bloky. Namiesto jednej, permanentnej, závodnej kampane bol systém zmenený na sezónne závodné kampane, z ktorých každá má 25 tratí a obmieňa sa každé 3 mesiace. Okrem sezónnych kampaní hra obsahuje aj tzv. tréningovú kampan (ktorá má tiež 25 tratí), permanentne dostupnú pre všetkých hráčov. Hráči, ktorí si zaplatili štandardné alebo klubové predplatné, majú retroaktívny prístup k všetkým predošlým sezónnym kampaniam, no časové rekordy a rebríčky na týchto starých kampaniach sú zamknuté, čiže akýkoľvek možný nový rekord nebude vzatý do úvahy. Novinkou je aj „Track Of The Day“, alebo trať dňa, ktorá je prístupná len hráčom s klubovým predplatným. Každá trať, ktorá je v Track Of The Day uvádzaná, bola vytvorená komunitou, a ručne vybratá vývojárom Nadeo. Táto trať, ako už možno zistiť z názvu, sa mení každých 24 hodín, takže rekordy, ktoré na nej hráči získajú sú veľmi prestížne, nakoľko sú časovo limitované. Od roku 2020 bolo do hry pridaných mnoho aktualizácií, vrátane nových povrchov, napr. plastové, drevené, ako aj mnoho dodatočných stavebných blokov a viacej druhov aut s rozličnými vlastnosťami. Trackmania 2020 bude využívaná pre túto prácu. [1] [9] [10]

2 TOOL-ASSISTED SPEEDRUN

Výraz „Speedrun“ by sa dal vysvetliť ako najrýchlejší možný spôsob dokončenia hry, alebo časti hry. Existuje veľa kategórií speedrun-ov, no hlavnými sú „Any%“, kde sa hráč snaží čo najrýchlejšie dokončiť hru bez ohľadu na to, či dokončil aj všetky jej nepovinné súčasti; „100%“ kde sa hráč naopak snaží hru čo najrýchlejšie dokončiť spolu so všetkými nepovinnými časťami; alebo „Low%“, kde sa hráč snaží dokončiť hru s minimálnym najnižším percentom splnenia (podľa hry môže takýto typ speedrun-u trvať značne dlhšie než Any%). [11][12]

2.1 Čo je to Tool-Assisted Speedrun

Tool-Assisted Speedrun (skrátene **TAS**) znamená to, že rekordný čas sa nesnaží získať človek organicky s využitím svojich vlastných zručností, ale využíva pre to nejaký špeciálny počítačový program. Tieto speedrun-y sú kategorizované do samostatných rebríčkov, pretože drvivá väčšina z nich nie je realisticky pre človeka dosiahnuteľná. Zvyčajne proces vytvárania takéhoto speedrun-u vyzerá tak, že hráč pomocou počítačových nástrojov manipuluje s časom v hre a manuálne zadáva vstupy, ktoré sa majú spustiť v určitý čas. Tým, že hráč má možnosť voľne manipulovať so základnými hernými mechanikami, môže teda v ideálny čas s presnosťou na stotiny sekundy vykonať sekvenciu akcií, ktoré často ľudské reflexy nemajú šancu replikovať. Tieto vstupy sú taktiež opakovateľné s perfektnou, nadľudsky konzistentnou presnosťou po každom spustení. Tvorba takýchto speedrun-ov je veľmi časovo náročná, lebo tvorca musí prechádzať cez celú hru prakticky snímok po snímku. TAS často využívajú známe chyby v kóde hry (glitch/bug). Najužitočnejšími nástrojmi pri tvorbe TAS sú „save states“ a „frame advance“.

Save states umožňujú užívateľovi okamžite uložiť hru a neskôr sa do tohto uloženého stavu vrátiť, čo je využívané prakticky ako cestovanie v čase. Umožňuje napraviť akékoľvek chyby, ktoré boli spätne objavené.

Frame advance je proces, kedy je hra postupne posúvaná po jednom snímku (frame) na príkaz používateľa. Tieto nástroje sa využívajú na maximalizáciu precíznosti. Ďalšími príkladmi využívaných nástrojov sú „RAM Search“ a „RAM Watch“. Pomocou týchto techník je monitorovaná pamäť RAM (Random Access Memory) za behu hry, a hľadajú sa v nej konkrétne hodnoty určitých premenných. Výsledkom je napríklad možnosť manipulovať so „šťastím“, teda s generátorom náhodných čísel (RNG), alebo sledovanie horizontálnej rýchlosti hráčskej postavy. [11][12][13]

2.1.1 Význam TAS

Ako bolo spomenuté, rekordy, ktoré boli dosiahnuté prostredníctvom TAS sú kategorizované samostatne, a tiež nemôžu byť vydávané za speedrun, ktorý spravil človek. To však neznamená, že TAS nie sú užitočné. Mnohokrát sa stáva, že techniky ušetrenia času nájdené počas tvorby TAS, môžu byť do určitej miery užitočné aj pre klasických speedrunner-ov, aj keď často s nižšou efektivitou. [11][12][13]

2.2 TAS v sérii Trackmania

Do dnešného dňa majú najväčšiu komunitnú podporu v hernej sérii Trackmania konkrétne hry Trackmania Nations Forever a Trackmania United Forever. To, že Trackmania je hra, kde hlavným cieľom je dosiahnuť čo najlepší čas na danej trati znamená, že aj zisk stotiny sekundy je dosť na to, aby bol prelomený rekord spred 10 rokov. Nové techniky a chyby v hre, ktoré sa dajú využiť pre časové zisky, sú nachádzané dodnes, viac ako 15 rokov po vydaní týchto hier. Pre zjednodušenie prístupu k tvoreniu TAS v hre Trackmania vytvoril člen komunity pod menom Donadigo nástroj **TMInterface**, ktorý je používaný primárne pre ukladanie, analyzovanie, modifikovanie a opakované prehratie záznamov z predošlých pretekov. Pre pokročilejších užívateľov z hľadiska TAS obsahuje funkcionality pre plnú kontrolu herných mechaník, ako editácia rýchlosti hry (spomalenie času), vizualizácia vstupov, ale napríklad aj skript „Bruteforce“, ktorý automaticky vyhľadáva ako získať potenciálne lepší čas. Automaticky simuluje tisíce pokusov, pričom v každom pokuse sú trochu pozmenené vstupy od pôvodného pokusu. Konvenčný spôsob tvorenia TAS v hre Trackmania (2020) zatiaľ neexistuje, nakoľko záznamy zo závodov sú šifrované, a sú implementované aj dodatočné bezpečnostné opatrenia. [14]

2.2.1 Funkcie TMInterface

Skript, ktorý je využívaný pre ovládanie auta, je vo forme obyčajného textového súboru s príponou .txt. Základné príkazy pre akceleráciu, brzdenie, zatáčanie, „respawn“ (znovu sa objaviť na poslednom kontrolnom bode) a reštart závodu sú:

- `press` – stlačí tlačidlo, vykoná akciu. Možné akcie:
 - o `up` – akcelerácia
 - o `down` – brzda
 - o `left` – zatočiť doľava
 - o `right` – zatočiť doprava
 - o `enter` – respawn
 - o `delete` – reštart celého závodu od začiatku

Syntax týchto príkazov je celkom jednoduchá. Pre príklad, ak by užívateľ chcel najprv akcelerovať, následne zatočiť doprava, pribrzdiť a potom zatočiť doľava, a nakoniec reštartovať závod, vyzeralo by to približne takto:

```
0 - 1200 press up
1200 - 3500 press right
3500 - 4000 press down
4000 - 5500 press left
5500 press delete
```

Toto je jeden z viacerých možných spôsobov použitia príkazov. Číslo na začiatku každého riadku špecifikuje časový rozsah odkedy dokedy má tlačidlo byť stlačené v milisekundách. Dodatočne je ešte možné mať kontrolu nad úrovňou stlačenia tlačidla pre zatáčanie pomocou analógových hodnôt. To môže užívateľ dosiahnuť tak, že namiesto využitia `press right` alebo `press left`, využije príkaz `steer`, ku ktorému pridá hodnotu z celočíselného (integer) rozsahu **[-65536, 65536]**. V prípade zápornej hodnoty auto zatočí doľava, pri pozitívnej hodnote zatočí doprava. Hodnota 0 je ekvivalentná žiadnemu zatáčaniu. Táto funkcionálna je veľmi užitočná pre precíznejšie zatáčanie, nakoľko emuluje analógové páčky na konzolovom ovládači (napr. DualShock ovládač pre PlayStation konzole). Pri obvyčajnej klávesnici má tlačidlo len dva stavy – stlačené alebo nestlačené. [14][16]

Okrem vyššie spomenutých príkazov existujú aj viaceré ďalšie, napríklad pre ukladanie vstupov vykonaných hráčom po dokončení závodu (`dump_inputs`), pre teleportovanie auta na určité XYZ súradnice (`tp`) alebo pre okamžité dokončenie závodu bez ohľadu na to, či hráč dosiahol posledný kontrolný bod (`finish`). [14][16]

Samozrejme je implementovaná funkcionálna pre vytváranie a načítavania save states. Tie majú formu binárnych súborov s príponou `.bin`. Save states obsahujú informácie o všetkých vstupoch vykonaných pred ich vytvorením, prístupné cez príkaz `recover_inputs`. [14] [15]

Okrem manuálnych save states je možné si pomocou `TMInterface` opakovane zahrať určité časti pretekov. To sa robí príkazom `replay`, ktorý je využívaný ako: `replay 5.27` – kde 5.27 je konkrétny čas v sekundách, do ktorého `TMInterface` zrýchli herný čas, a následne vráti kontrolu do rúk hráča. [14] [15] [17]

3 VYUŽITIE ALGORITMOV AI A RL V HRÁCH

Aj keď algoritmy AI a strojové učenie nie sú novinkou, ich aplikácia v sfére hier, či stolných alebo počítačových, je stále relatívne neprebádané teritórium. Najznámejšími príkladmi úspešných projektov sú AlphaGo vytvorené v roku 2016 tímom DeepMind, ktorý patrí pod spoločnosť Google alebo OpenAI Five, ktorý bol vytvorený v roku 2018 dnes už veľmi známou firmou OpenAI. [18]

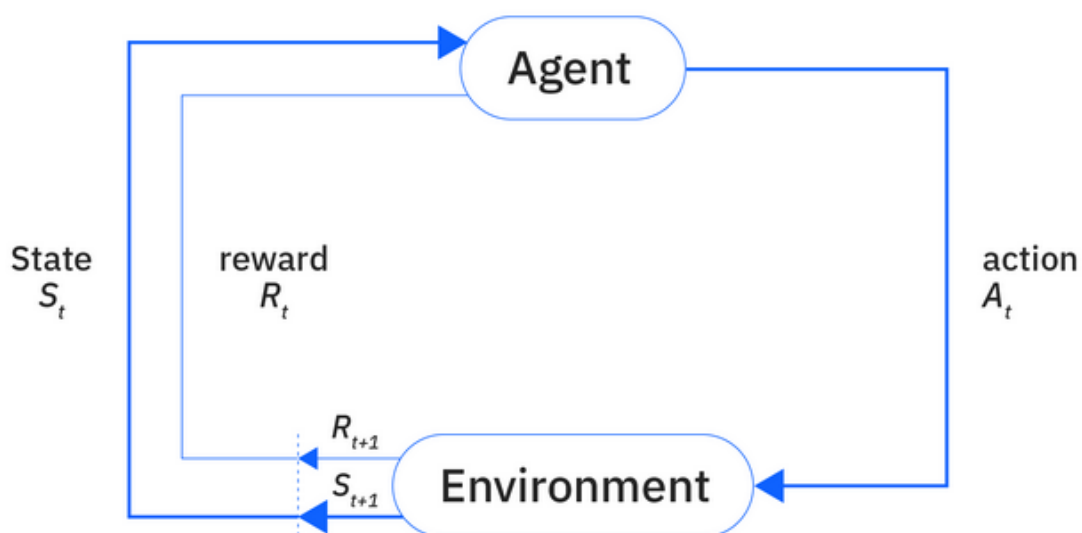
3.1 Čo je to reinforcement learning (RL)

Reinforcement learning, alebo spätnoväzobné učenie, je druh strojového učenia, ktorý je zameraný na proces rozhodovania autonómnych „agentov“. Je to jedna z troch hlavných paradigiem strojového učenia, spolu s „učením s učiteľom“ a „učením bez učiteľa“ (Supervised a Unsupervised learning). Agentom je myslený akýkoľvek systém, ktorý dokáže rozhodovať o tom, čo má spraviť a ako sa má chovať, ako odpoveď na prostredie, v ktorom sa nachádza – nezávisle od ľudského zásahu alebo inštrukcie. Zaoberá sa najmä sekvencnými problémami rozhodovania v neistých prostrediach. Ako konkrétny príklad je možné si predstaviť autonómne taxíky firmy Waymo. [27]



Obrázok 1: Autonómny taxík firmy Waymo [26]

Agent sa učí vykonávať určitý úkon procesom pokus-omyl za pomoci odmenovej funkcie – reward function. Samotné učenie agenta nastáva interakciou so svojim prostredím. Prostredie agentovi poskytuje informácie o svojom momentálnom stave (S_t) a na základe týchto informácií sa agent rozhoduje, akú akciu (A_t) by mal vykonať. Ak akcia, ktorú agent zvolil, dostane pozitívnu odozvu vo forme odmeny (R_t) od prostredia, agent je „povzbudený“, aby túto akciu vykonal v podobných budúcich stavoch. Tento proces sa opakuje pre každý nový stav. Postupom času sa teda agent učí z odmien a „trestov“ vykonávať v prostredí akcie, ktoré dosahujú nejaký cieľ. Tento vzťah medzi agentom, prostredím a nejakým cieľom je nazývaný ako Markovov rozhodovací proces (MDP). [27]



Obrázok 2: Markovov rozhodovací proces [27]

3.2 Čo je to deep reinforcement learning (DRL)

Je to metodika, ktorá kombinuje reinforcement learning s hlbokými neurónovými sieťami. Hlboké neurónové siete sa berú ako neurónové siete, ktoré obsahujú 3 a viac vrstiev. Základný princíp učenia zostáva totožný s reinforcement learning-om, no dodatočne je zakomponovaná hlboká neurónová sieť, ktorá reprezentuje stratégie (policy) alebo modely prostredia. Táto integrácia umožňuje agentom spracovať vysoko-rozmerné vstupné priestory, napríklad obrázky alebo spojité ovládacie signály, čo DRL robí veľmi populárnou možnosťou pre komplexné úlohy. [28]

3.3 Příklady aplikovania algoritmov RL v hrách

Najznámejšie príklady aplikovania RL v hrách pochádzajú od výskumných tímov patriacich pod firmy Google a OpenAI.

3.3.1 Projekt AlphaGo

AlphaGo je systém umelej inteligencie, ktorého jediným cieľom je hrať starodávnu čínsku stolovú hru Go. Dokázal poraziť ľudského svetového šampióna desať rokov pred tým, než si experti mysleli, že by to bolo možné. Boli využité dve neurónové siete, prvá – policy network – rozhodovala o ďalšom ťahu, a druhá – value network – predpokladala víťaza hry. Na začiatku AlphaGo predstavili viaceré amatérske zápasy, aby sa systém mohol naučiť, ako približne ľudia hru hrajú. Následne postavili AlphaGo proti rozličným verziám seba samej tisícky krát. Toto je známe ako reinforcement learning. Koncom roku 2015 vyhral systém AlphaGo proti trojnásobnému európskemu šampiónovi Fan Hui so skóre 5-0. V prvej štvrtine roku 2016 systém hral historický zápas proti najlepšiemu hráčovi tej dekády, Lee Sedol. Zápas sledovalo viac ako sto miliónov divákov. Systém AlphaGo tento zápas vyhral so skóre 4-1. Obrovský úspech tohto systému vydláždil cestu pre ďalšiu generáciu podobných systémov – AlphaZero, MuZero a AlphaDev. [19]

3.3.2 Projekt OpenAI Five

OpenAI Five je v porovnaní s projektom AlphaGo gigantický – každý deň proti sebe hrá 180 rokov zápasov. Funguje na báze algoritmu PPO – Proximal Policy Optimization – proximálna optimalizácia politiky, beží na 256 grafických kartách a 128,000 procesorových jadrách. Cieľom tohto systému je hrať hru Dota 2, vyvinutú herným štúdiom Valve. Systém sa skladá z piatich AI agentov, ktorí reprezentujú tím hráčov.

Natrénovaný systém bol postavený proti piatim tímom s rôznymi úrovňami skúseností:

1. Najlepší hráči zo spoločnosti OpenAI – 46. percentil (priemer)
2. Hráči, ktorí sledovali prvý zápas proti zamestnancom OpenAI, no nikdy spolu nehrali ako tím – 90. až 99. percentil (vysoká obtiažnosť)
3. Najlepší hráči zo spoločnosti Valve – 46. až 99. percentil (zmes priemeru a vysokej obtiažnosti)
4. Amatérsky tím, ktorý spolu trénuje – 93. percentil
5. Poloprofesionálny tím, ktorý spolu trénuje – 99. percentil [20]

Jedna z prvých verzií OpenAI Five, z mája 2018, vyhrala jeden zápas proti tímu 1, a prehrala druhý zápas. Verzia z júna 2018 bez väčších problémov vyhrala všetky svoje zápasy proti tímom 1 až 3. Očakávalo sa však, že táto verzia nebude mať žiadnu šancu proti tímom 4 a 5, avšak opak bol pravdou. Systém vyhral dva z prvých troch zápasov proti obom tímom. [20]

3.4 Využitie algoritmov AI a RL v sérii Trackmania

TMInterface je veľmi užitočná pomôcka pre tvorenie klasických TAS, no sama osebe neumožňuje pracovať s algoritmami umelej inteligencie. Z tohto dôvodu existuje pre hru Trackmania Nations Forever len hárška verejne dostupných projektov voľne dostupných pre využitie.

3.4.1 Projekt Linesight

Jedným z takýchto projektov je **Linesight**, ktorý bol zverejnený do podoby open-source len v júni roku 2024. Tento projekt je pravdepodobne najpokročilejším AI pre Trackmania (konkrétne Trackmania Nations Forever), a porazil už viacero veľmi prestížnych ľudských svetových rekordov. Linesight prešiel za posledných pár rokov viacerými iteráciami. Algoritmus, ktorý využíva, je vysoko upravená verzia algoritmu IQN – Implicit Quantile Network – Implicitná kvantilová sieť. Podľa slov tvorcov tohto projektu, pb4git a Agade09, bol projekt vytvorený na obyčajnom stredno-triednom PC, s jednou úpravou, a to je zväčšenie kapacity pamäte RAM na 64GB. Toto zväčšenie je potrebné pre to, aby počas tréningu dokázal algoritmus pracovať s veľkým množstvom obrázkov malého rozlíšenia (v prípade Linesight to je 120x160px pri 20 snímkach za sekundu) z hry v reálnom čase. Model nevyužíva žiadnu metodiku „look-ahead“, teda neplánuje svoje kroky dopredu. [21][22]



Obrázok 3: Ukážka toho, čo vidí model (bez textu)[22]

3.4.2 Projekt Trackmania_AI

Ďalším projektom, ktorý je zameraný na implementáciu a porovnanie rôznych algoritmov RL, je Trackmania_AI, vytvorený užívateľom AndrejGobeX. Tento projekt implementuje algoritmus SAC (Soft Actor-Critic), PPO a TD3 (Twin-Delayed Deep Deterministic Policy Gradient). Získané výsledky sú:

Tabuľka 1: Výsledky projektu Trackmania_AI

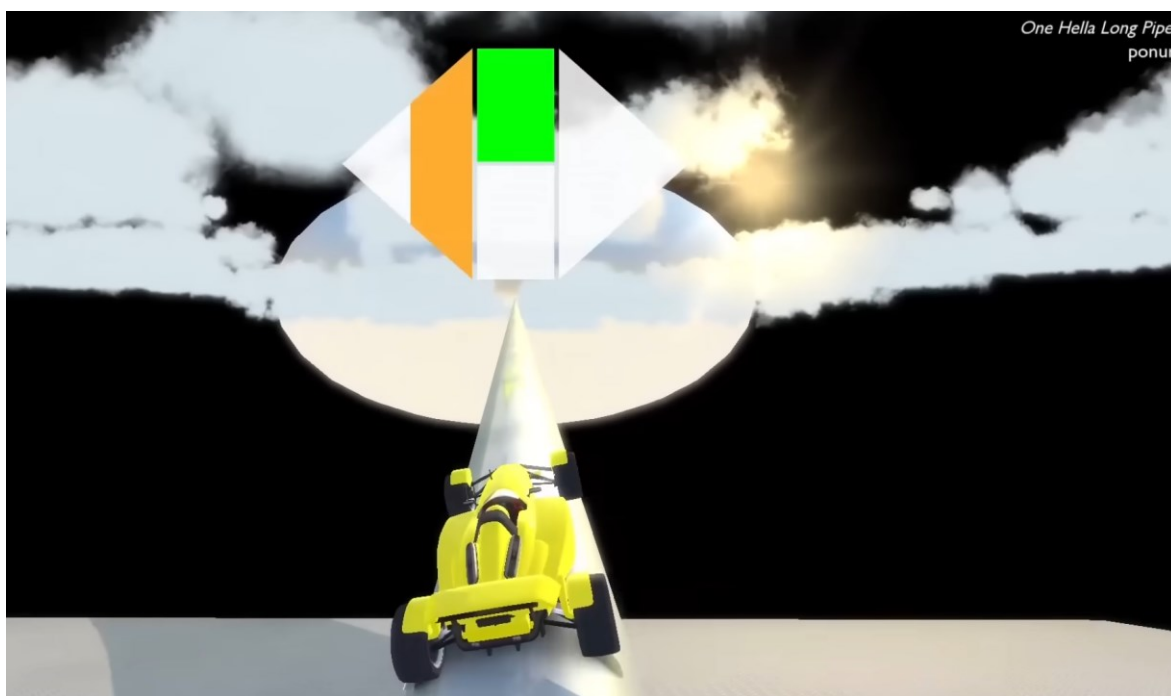
Trať	Vlastný rekord	SAC (rekord)	SAC (priemer)	PPO (rekord)	PPO (priemer)	TD3 (rekord)	TD3 (priemer)
<i>Nascar2</i>	14.348s	16.180s	18.287s	19.126s	19.367s	16.247s	17.791s
<i>Test2</i>	12.728s	14.109s	16.196s	14.174s	14.655s	17.894s	18.758s
<i>Night</i>	10.124s	10.576s	13.220s	10.668s	10.999s	10.728s	11.474s

Tento projekt využíva zjednodušenú verziu funkcie odmeny, ktorá bola vytvorená študentmi na Univerzite v Zürichu, pôvodne pre hru Gran Turismo Sport. Taktiež implementuje metódu zrkadlenia, kde pri každej novej tréningovej epizóde dochádza k zrkadleniu trate, aby sa zabránilo overfitting-u (model je zaseknutý v lokálnom minime). [23]

3.4.3 Trackmania AI vytvorené YouTube-rom Yosh

Posledný, viacej známy, projekt, ktorý bude spomenutý, je AI tréňované a vyvíjané tvorcom na platforme YouTube, Yosh. Jeho implementácia nie je verejne dostupná, a informácie o využitých metodikách sa dajú získať len z jeho videí, prípadne plateného predplatného na platforme Patreon. V jeho najnovšom videu z januára roku 2025 [24] spomenul, že sa v najnovšej iterácii inšpiroval už spomenutým projektom Linesight, takže pravdepodobne využíva algoritmus IQN alebo DQN. Experimenty, ktoré Yosh necháva svoj model podstúpiť sú neštandardné, nakoľko sa snaží model učiť, aby napríklad využívalo nedostatky a chyby v hernom engin-e pre svoj prospech, alebo porazil profesionálneho ľudského oponenta len pomocou spiatočky. Model väčšinou prechádza viacerými tréningmi, ktoré v niektorých prípadoch môžu trvať až 200 a viac hodín. Jedná sa teda o veľmi schopný model, a skoro všetky výzvy, proti ktorým je postavený, sú porazené.

Konkrétnym príkladom je trať nazvaná „One Hella Long Pipe“ (jedno pekelné dlhé potrubie), ktorá je veľmi jednoduchá – po krátkom štartovacom bloku musí hráč prejsť z obvyčajnej cesty na úzke potrubie, a dostať sa po ňom až do cieľovej línie. Ľudský svetový rekord na tejto trati je jedna hodina, päťdesiatšesť minút a päťdesiatšesť sekúnd, pretože dokázať jazdiť na takomto povrchu si vyžaduje precíznosť a trpezlivosť. Model, ktorý vytvoril Yosh po viacerých iteráciách a úpravách, dokončil túto trať s časom pätnásť minút a päťdesiatosem sekúnd, rýchlejšie o viac ako hodinu a štyridsať minút. [25]



Obrázok 4: Ukážka trate One Hella Long Pipe [25]

4 VYUŽITÉ NÁSTROJE

Nástroje, ktoré umožňujú implementáciu algoritmov RL pre hru Trackmania, sú vytvorené komunitou tejto hry, a sú voľne dostupné na stiahnutie z internetu. Tieto nástroje sú OpenPlanet a TMRL, a oba sú open-source. TMRL využíva dáta poskytnuté cez OpenPlanet pre samotné tréningovanie modelov.

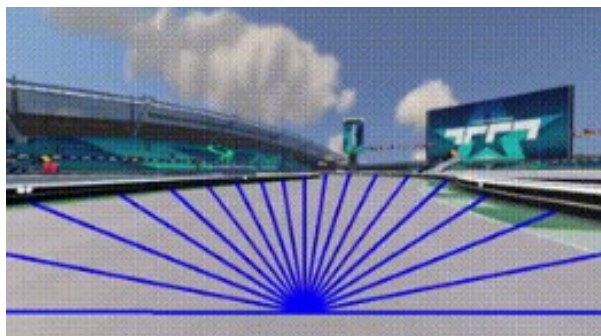
4.1 OpenPlanet

Nástroj OpenPlanet je popísaný ako skriptovacia platforma určená pre hru Trackmania (2020), Trackmania Turbo a platformu Maniaplanet. Má mnoho funkcií, ako napríklad možnosť vytvárania vlastných doplnkov, sťahovanie užitočných doplnkov vytvorených komunitou, umožňuje náhľad „pod kapotu“ hry, prístup k herným súborom a mnoho iných. Pre skripty/doplňky je využívaný skriptovací jazyk AngelScript. [35]

Táto práca využíva hlavne nástroj TMRL, takže OpenPlanet je využitý len na inštaláciu doplnkov TMRL GrabData a TMRL SaveGhost.

4.2 Framework TMRL

TMRL znamená Trackmania Reinforcement Learning. Je to framework pre programovací jazyk Python, ktorý umožňuje užívateľom trénovať algoritmy RL pre použitie v aplikáciách v reálnom čase. V základnej forme obsahuje pripravený pipeline pre hru Trackmania 2020, no tento framework je možné využiť aj pre iné hry, alebo aj pre ovládanie auta na diaľkové ovládanie. Dva možné prístupy pre spracovanie údajov z hry pre tréningovanie je takzvaný „Full environment“, ktorý využíva históriu posledných štyroch snímok z hry, a „LIDAR environment“, ktorý namiesto snímok z hry využíva pre tréningovanie históriu posledných štyroch meraní LIDAR. LIDAR v skratke meria vzdialenosť auta od okrajov cesty viacerými modrými lúčmi. [36]



Obrázok 5: Ukážka prostredia LIDAR

GitHub repozitár tohto framework-u [36] obsahuje veľmi efektívny návod pre začiatočníkov [37], ktorý pokrýva všetko od inštalácie, až po začiatok tréningovania existujúceho predpripraveného modelu, alebo jeho prípadné otestovanie s natrénovanými váhami. Okrem začiatočného návodu obsahuje aj veľmi pokročilý návod, ktorý sa zameriava na využitie framework-u ako knižnice a vytvorenie si vlastného pipeline. To pre túto prácu nebude potrebné. Pre implementácie vlastných modelov bude využitý súbor `custom_actor_module.py`, ktorý sa nachádza v priečinku `tuto/competition` hlbšie v repozitári [36] **Chyba! Nenašiel sa žiaden zdroj odkazov..** Tento súbor obsahuje z ákladnú kostru vrátane predpripravených jednoduchých funkcií `VanillaCNN` a `mlp` pre spracovanie herných dát, ako aj import všetkých užitočných knižníc a funkcií, ktoré budú využívané. Dodatočne je priložený aj popis procesu implementácie vo forme komentárov. Nie všetky z nich sú pre túto prácu užitočné, nakoľko pôvodný zmysel tohto súboru je pre ľudí, ktorí chcú svoje modely otestovať a zapojiť ich do „RoboRace“ súťaže, kde sú porovnávané výkony natrénovaných modelov medzi sebou.

```

"""
=====
COMPETITION TUTORIAL #1: Custom model and RL algorithm
=====

In this tutorial, we will customize the TrackMania training pipeline.

The tutorial works with the TrackMania FULL Gymnasium environment.
Please refer to the README on GitHub to set up this environment in config.json:
https://github.com/trackmania-rl/tmrl#full-environment

Note: This tutorial describes implementing and running a TrainingAgent along with an ActorModule.
It is relevant if you want to implement your own RL approaches in TrackMania.
If you plan to try non-RL approaches instead, this is also accepted in the competition:
just use the Gymnasium Full environment and do whatever you need,
then, wrap your trained policy in an ActorModule, and submit your entry :)

Copy and adapt this script to implement your own algorithm/model in TrackMania.
Then, use the script as follows:

To launch the Server, provided the script is named custom_actor_module.py, execute:
python custom_actor_module.py --server

In another terminal, launch the Trainer:
python custom_actor_module.py --trainer

And in yet another terminal, launch a RolloutWorker:
python custom_actor_module.py --worker

You can launch these in any order, but we recommend server, then trainer, then worker.
If you are running everything on the same machine, your trainer may consume all your resource,
resulting in your worker struggling to collect samples in a timely fashion.
If your worker crazily warns you about time-steps timing out, this is probably the issue.
The best way of using TMRL with TrackMania is to have your worker(s) and trainer on separate machines.
The server can run on either of these machines, or yet another machine that both can reach via network.
Achieving this is easy (and is also kind of the whole point of the TMRL framework).
Just adapt config.json (or this script) to your network configuration.
In particular, you will want to set the following in the TMRL config.json file of all your machines:

"LOCALHOST_WORKER": false,
"LOCALHOST_TRAINER": false,
"PUBLIC_IP_SERVER": "<ip.of.the.server>",
"PORT": <port of the server (usually requires port forwarding if accessed via the Internet)>,

If you are training over the Internet, please read the security instructions on the TMRL GitHub page.

IMPORTANT: Set a custom 'RUN_NAME' in config.json, otherwise this script will not work.
"""

```

Obrázok 6: Úvodný komentár súboru `custom_actor_module.py`

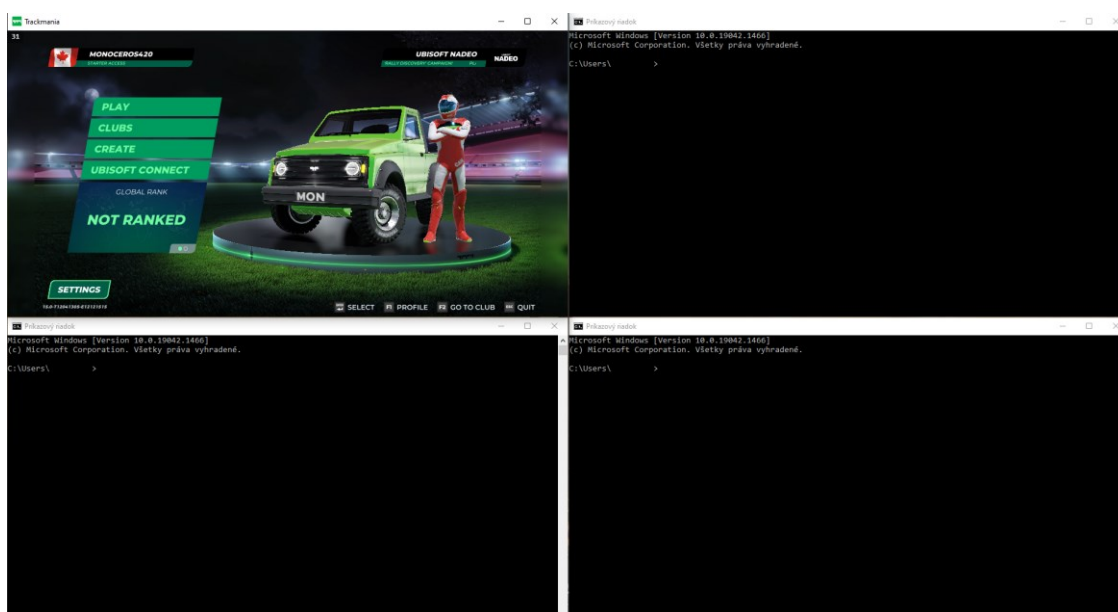
Okrem samotnej implementácie modelu je potrebné upraviť aj sprevádzajúci súbor `config.json`, v ktorom sa nachádzajú nastaviteľné hyperparametre¹ algoritmov a iné užitočné nastavenia týkajúce sa funkcionality. Základná kostra tohto súboru je tiež dostupná v repozitári, ale aj priamo v nainštalovaných súboroch framework-u, konkrétne v užívateľskom priečinku v systéme Windows, v priečinku `TmrlData/config`.

```
{
  "RUN_NAME": "SAC_4_imgs_pretrained",
  "RESET_TRAINING": false,
  "BUFFERS_MAXLEN": 500000,
  "RW_MAX_SAMPLES_PER_EPISODE": 1000,
  "CUDA_TRAINING": true,
  "CUDA_INFERENCE": false,
  "VIRTUAL_GAMEPAD": true,
  "DCAC": false,
  "LOCALHOST_WORKER": true,
  "LOCALHOST_TRAINER": true,
  "PUBLIC_IP_SERVER": "0.0.0.0",
  "PASSWORD": "==>TMRL@UseASecurePasswordHere!<==",
  "TLS": false,
  "TLS_HOSTNAME": "default",
  "TLS_CREDENTIALS_DIRECTORY": "",
  "NB_WORKERS": -1,
  "WANDB_PROJECT": "tmrl",
  "WANDB_ENTITY": "tmrl",
  "WANDB_KEY": "df28d4daa98d2df2557d74caf78e40c68adaf288",
  "PORT": 55555,
  "LOCAL_PORT_SERVER": 55556,
  "LOCAL_PORT_TRAINER": 55557,
  "LOCAL_PORT_WORKER": 55558,
  "BUFFER_SIZE": 536870912,
  "HEADER_SIZE": 12,
  "SOCKET_TIMEOUT_CONNECT_TRAINER": 300.0,
  "SOCKET_TIMEOUT_ACCEPT_TRAINER": 300.0,
  "SOCKET_TIMEOUT_CONNECT_ROLLOUT": 300.0,
  "SOCKET_TIMEOUT_ACCEPT_ROLLOUT": 300.0,
  "SOCKET_TIMEOUT_COMMUNICATE": 30.0,
  "SELECT_TIMEOUT_OUTBOUND": 30.0,
  "ACK_TIMEOUT_WORKER_TO_SERVER": 300.0,
  "ACK_TIMEOUT_TRAINER_TO_SERVER": 300.0,
  "ACK_TIMEOUT_SERVER_TO_WORKER": 300.0,
  "ACK_TIMEOUT_SERVER_TO_TRAINER": 7200.0,
  "RECV_TIMEOUT_TRAINER_FROM_SERVER": 7200.0,
  "RECV_TIMEOUT_WORKER_FROM_SERVER": 600.0,
  "WAIT_BEFORE_RECONNECTION": 10.0,
  "LOOP_SLEEP_TIME": 1.0,
  "MAX_EPOCHS": 10000,
  "ROUNDS_PER_EPOCH": 100,
  "TRAINING_STEPS_PER_ROUND": 200,
  "MAX_TRAINING_STEPS_PER_ENVIRONMENT_STEP": 4.0,
  "ENVIRONMENT_STEPS_BEFORE_TRAINING": 1000,
  "UPDATE_MODEL_INTERVAL": 200,
  "UPDATE_BUFFER_INTERVAL": 200,
  "SAVE_MODEL_EVERY": 0,
}
```

Obrázok 7: Ukážka časti obsahu súboru `config.json`

¹ Hyperparameter - nastavenie konfigurácie – ako je rýchlosť učenia alebo gamma – ktoré sú definované pred tréňovaním modelu, ktoré ho navádzajú k tomu, ako sa bude učiť

Pre proces trénovania alebo testovania implementovaného algoritmu je potrebné okrem hry spustiť tri okná príkazového riadku (cmd), a tieto okná poukladať tak, aby každé zaberalo štvrtinu obrazovky. Tri príkazové riadky sú potrebné preto, lebo tréningový proces prebieha v troch častiach – server, trainer a worker. Server je sprostredkovateľ, ktorý zaisťuje komunikáciu medzi zvyšnými dvomi časťami, získava od nich dáta a ukladá ich, a zároveň zasiela nové skúsenosti od worker-a trainer-ovi. Trainer má na starosti proces učenia, aktualizácie váh, ukladanie kontrolných bodov cez server, beží samotné tréningové slučky. Worker je „hráč“, ktorý interaguje s prostredím – hrou – a získava skúsenosti, ktoré potom odosiela serveru. [37]



Obrázok 8: Rozpoloženie okien hry a príkazových riadkov

5 ALGORITMY RL VYUŽITÉ V TEJTO PRÁCI

Pre túto prácu boli vybrané dva algoritmy reinforcement learning-u. Oba z nich už boli spomenuté v predošlých kapitolách. Implementované budú v relatívne jednoduchých formách s malými úpravami a dodatkami.

5.1 Deep Q-Network (DQN)

Deep Q-Network, alebo Deep Q-Learning je model DRL, ktorý rozširuje algoritmus Q-learning o hlbokú neurónovú sieť.

5.1.1 Q-Learning

Q-learning je off-policy metóda reinforcement learning-u, ktorá je hodnotovo založená a využíva prístup TD pre tréning jej Q-funkcie - $Q(s_t, a_t)$. Off-policy znamená, že pre „acting/inference“ – správanie, je využitá iná stratégia – „policy“, ako pre tréningovanie, aktualizovanie. Prístup TD (Temporal Difference) znamená, že Q-funkcia je aktualizovaná po každom kroku, namiesto toho, aby bola aktualizovaná až na konci tréningovej epizódy. To, že je Q-learning hodnotovo založený, sa dá vysvetliť tak, že hľadá optimálnu stratégiu nepriamo, a to tým, že trénuje Q-funkciu, ktorá udáva hodnotu každého stavu, prípadne páru stav-akcia. [29][30]

5.1.1.1 Q-Funkcia

Písmeno Q vo výraze Q-funkcia vyplýva z pojmu „Quality“ alebo kvalita. Konkrétne sa jedná o hodnotu kvality určitej akcie, ktorú si môže agent vybrať v danom stave. Je potrebné rozlíšiť hodnotu Q-funkcie od odmeny. Hodnota Q-funkcie udáva hodnotu páru stav-akcia, teda očakávanú kumulatívnu odmenu, ktorú agent dostane, ak v tomto páre stav-akcia začne a následne sa správa v súlade so stratégiou. Odmena je spätná väzba prostredia, ktorú agent dostáva po vykonaní akcie v určitom stave.

Q-funkcia je zakódovaná Q-tabuľkou – tabuľka, v ktorej každá bunka odpovedá hodnote určitého páru stav-akcia. Q-tabuľka sa dá popísať ako taká pamäť, alebo „ťahák“ Q-funkcie. [29][30]

5.1.1.2 Temporal Difference vs. Monte Carlo

Keďže RL je spôsob učenia z predošlých skúseností bez akéhokoľvek modelu prostredia, je potrebné sa rozhodnúť medzi dvoma možnosťami - prvou je metóda Monte Carlo, pri ktorej je potrebné počkať až na úplný koniec tréningovej epizódy na to, aby mohli byť aktualizované odhady hodnotových funkcií, alebo v tomto prípade Q-funkcie. Druhou možnosťou je Temporal Difference, pomocou ktorej je možné namiesto čakania na koniec epizódy hodnoty aktualizovať okamžite po určitom časovom kroku t na základe odmeny R_{t+1} , a následne opäť aktualizovať na základe odhadov nového stavu S_{t+1} . Toto je najjednoduchšia verzia TD, TD(0), kde nula reprezentuje fakt, že je potrebné čakať len na jediný krok pre vykonanie aktualizácie, no tento princíp sa vzťahuje aj na komplexnejšie algoritmy ako n-step TD. Cieľ TD (TD Target) reprezentuje cieľovú hodnotu, ku ktorej by mal agent približovať svoje odhady počas procesu tréningu. [29][30][31]

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \quad (1)$$

Nová hodnota stavu t
 Predošlý odhad hodnoty stavu t
 Rýchlosť učenia (Learning rate)

Odmena
 Diskontovaná hodnota ďalšieho stavu

Cieľ TD

Obrázok 9: Výpočet TD-target (Cieľ TD) [33]

5.1.2 Proces trénovania

Na začiatku trénovania je inicializovaná Q-tabuľka, a všetky jej hodnoty sú nastavené na nulu (toto však nie je tvrdo dané pravidlo). Pokračuje sa výberom nejakej akcie, a to pomocou stratégie „ε-greedy“. Stratégia ε-greedy má za úlohu dosiahnuť nejaký kompromis medzi prieskumom a „vykorisťovaním“ (exploration – exploitation). Prieskum znamená, že agent volí celkom náhodné akcie bez dodatočnej logiky, pre to aby preskúmal viacero stavov. K prieskumu dochádza vtedy, ak je hodnota ε vysoká, napríklad 1. Vykorisťovanie znamená, že agent bude voliť akcie, ktoré majú najvyššiu hodnotu páru stav-akcia. K vykorisťovaniu dochádza vtedy, ak je hodnota ε malá, teda $1 - \epsilon$. Hodnota ε sa mení počas trénovania. Zvyčajne začína na hodnote 1, alebo inej vysokej hodnote, no naprieč časom sa postupne znižuje na hodnotu veľmi blízkej nule. Výsledkom je to, že na úplnom začiatku bude agent síce často voliť úplne nezmyselné akcie, no to mu umožňuje rozšíriť si obzory, a nájsť tak aj užitočné akcie pre budúce pokusy. Znižovaním hodnoty ε je agentovi „nakázané“ držať sa už len týchto užitočných akcií, a to vďaka tomu, že Q-tabuľka je už v tejto chvíli naplnená zmysluplnejšími odhadmi a hodnotami. [29][30][31]

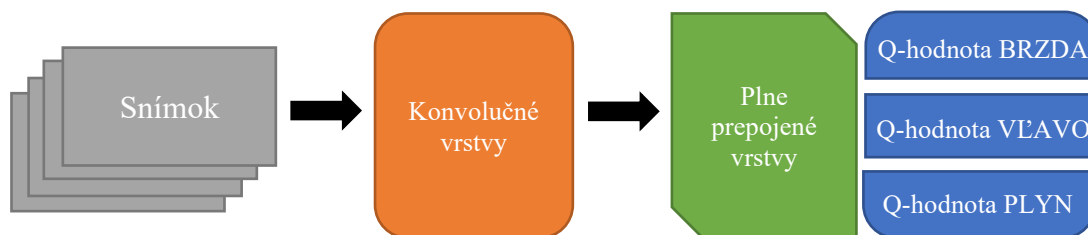
Po vybratí akcie pomocou stratégie ϵ -greedy je táto akcia vykonaná. Ako bolo spomenuté, s prístupom učenia TD sú hodnoty stratégie alebo hodnotovej funkcie - v tomto prípade Q-funkcie - aktualizované po každom jednom kroku interakcie. Pre vytvorenie TD cieľu je využitá okamžitá odmena spolu s diskontovanou optimálnou Q-hodnotou ďalšieho stavu. Diskontovaná optimálna Q-hodnota je len Q-hodnota vynásobená hodnotou γ , čo udáva, ako moc má agent prioritizovať dlhodobé odmeny namiesto okamžitých odmien. Hodnota γ môže byť napríklad 0.995 a je jednou z viacerých nastaviteľných hyperparametrov. TD cieľ je získaný nasledovne:

1. je získaná okamžitá odmena po vykonaní akcie A_t
2. pre získanie optimálnej Q-hodnoty ďalšieho stavu je využitá „greedy“ stratégia (odlišná od ϵ -greedy), ktorá vždy vyberie akciu s najvyššou Q-hodnotou

Po aktualizovaní Q-hodnoty je začatý nový stav, v ktorom je vybratá ďalšia akcia opäť pomocou stratégie ϵ -greedy, a celý proces sa opakuje. Z tohto dôvodu je Q-Learning nazývaný „off-policy“. [29][30][31]

5.1.3 Architektúra Deep Q-Network

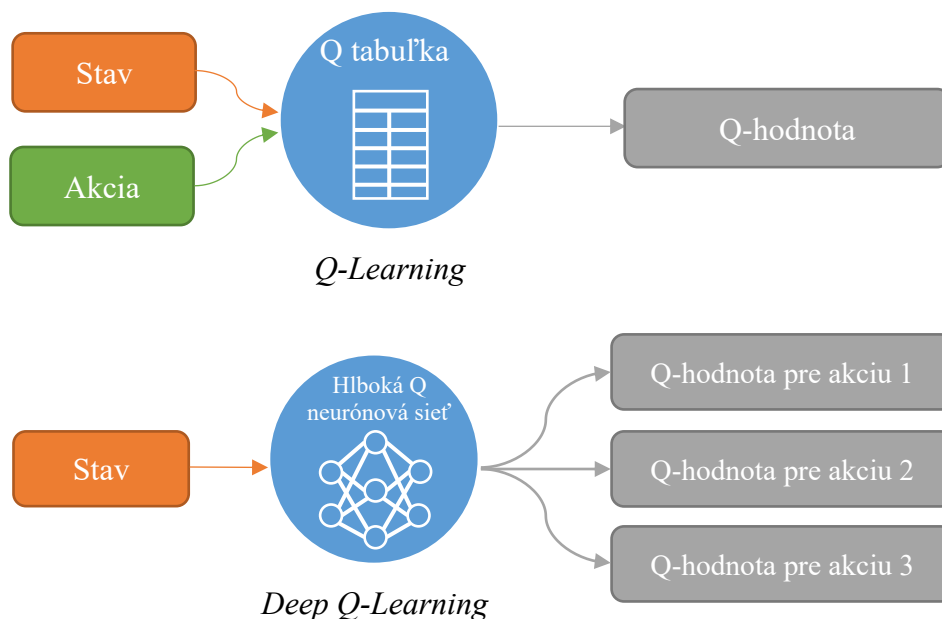
Ako vstup pre Deep Q-Network bude využitá skupinka štyroch posledných snímok z hry. Tieto snímky reprezentujú stavy, a je potrebné ich predspracovať, zjednodušiť. Hra beží v malom okienku, v rozlíšení 256x128. Framework TMRL získava a následne dodáva 4 posledné snímky z hry v tomto rozlíšení, pričom ich farbu dodatočne mení na odtiene sivej. Pre tréovanie nestačí len jeden posledný snímok z hry, nakoľko sa len z jednej snímky nedá odhadnúť smer pohybu. Po naskladaní týchto snímok na seba prechádzajú cez štyri konvolučné vrstvy, ktoré z nich získavajú dôležité informácie, ako napríklad priestorové vzťahy. Výstup z konvolučných vrstiev potom prechádza cez takzvané plne prepojené vrstvy. Plne prepojené vrstvy sú vrstvy, v ktorých každý jeden neurón v danej vrstve je prepojený so všetkými neurónmi z predošlej vrstvy. [32] Okrem snímok sú z hry zbierané taktiež informácie ako rýchlosť, prevodový stupeň, otáčky a predchádzajúce vykonané akcie. Taktiež je dodatočne obsiahnutý jednoduchý viacvrstvový perceptron.



Obrázok 10: Diagram princípu DQN

5.1.4 Deep Q-Learning

Keďže sa klasický Q-Learning spolieha na Q-tabuľku pre uchovávanie Q-hodnôt, je veľmi rozmerovo obmedzený. V určitých komplexnejších prípadoch sa stáva efektívne využitie polí a tabuliek pre reprezentáciu stavov a akcií jednoducho nereálne. Obyčajný Q-Learning nie je škálovateľný. Pre vysoko komplexné prostredia, ako napríklad snímky z hry, bude pre odhad Q-hodnôt využitá hlboká neurónová sieť. Táto sieť má na starosti odhad Q-hodnôt pre každú možnú akciu v danom stave. Zatiaľ čo obyčajný Q-Learning bral ako vstup ako stav, tak aj akciu, Deep Q-Learning si žiada len stav. [33]



Obrázok 11: Porovnanie štruktúry Q-Learning a Deep Q-Learning [40]

Pri Deep Q-Learning-u je namiesto priamej aktualizácie Q-hodnôt pre páry stav-akcia vytvorená stratová funkcia Q-strata (Q-loss), ktorá porovnáva odhad Q-hodnoty s cieľom (Q-target) a využíva gradientový zostup pre aktualizovanie váh Deep Q-Network, čo umožní lepšie odhadovanie Q-hodnôt. Gradientový zostup sa dá vysvetliť ako napríklad porovnávanie si svojich odpovedí na skúške so správnymi odpoveďami, a následné učenie sa z nich pre zlepšenie. [33]

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (2)$$

Nový odhad Q-hodnoty Predošlý odhad Q-hodnoty Rýchlosť učenia Okamžitá odmena Diskontovaný odhad optimálnej Q-hodnoty ďalšieho stavu

Cieľ TD – Q-Target

Chyba TD – Q-Loss

Obrázok 12: Vzorec odhadu Q-hodnôt [33]

Na rozdiel od obyčajného Q-Learningu má tréningový algoritmus dve fázy:

- **Zber vzoriek (Sampling)** – vykonávajú sa akcie pre získavanie skúseností a tieto skúsenosti sa ukladajú do pamäte
- **Tréning** – výber náhodnej malej skupinky skúseností a učenie sa z nej pomocou gradientového zostupu (aktualizácia váh) [33]

5.1.5 Prioritized Experience Replay

Verzia DQL, ktorá bude využitá v praktickej časti obsahuje aj PER – Prioritized Experience Replay, čo je vcelku populárne vylepšenie pôvodného algoritmu pre zvýšenie kvality tréningovania. Úlohou PER je priradovať priority skúsenostiam v pamäti agenta na základe ich užitočnosti. Pri PER je využitý parameter α , ktorého hodnota rozhoduje úroveň preferencie týchto dôležitejších skúseností namiesto náhodného výberu. Ak je hodnota α nastavená napríklad na 0.7, preferencia je bližšie k dôležitejším skúsenostiam, ale berú sa do úvahy aj menej dôležité skúsenosti. Úroveň dôležitosti je udávaná hodnotou chyby TD, no toto nie je jediná možnosť, ktorá sa dá využiť. Prístup využitia chyby TD pre udávanie priority je nazývaný „greedy“, a je dôležité, aby pri inicializácii neboli Q-hodnoty nulové. Ak by boli Q-hodnoty nulové, prechody, ktoré zatiaľ nemajú odmenu by mali nulovú, alebo skoro nulovú, chybu TD, čo by ich umiestnilo na koniec fronty, a efektívne zapríčinilo to, že by už nikdy neboli vzorkované počas procesu tréningovania. [34]

5.1.5.1 *Catastrophic Forgetting*

Katastrofické zabúdanie je koncept v problematike neurónových sietí, kde počas procesu kontinuálneho učenia dochádza k strate informácii týkajúcich sa predošlých skúseností. Napríklad, ak je model trénovaný na úlohu 1, a následne je presunutý na učenie sa úlohy 2, ak je opäť vrátený na úlohu 1, učí sa ju úplne odznova. Existujú viaceré metódy pre snahu zabrániť tomuto javu, no žiadne nemá stopercentnú efektivitu:

- Regularizácia (napríklad **EWC**)
- Architektonické riešenia (napríklad **DWA**)
- Súborové metódy (napríklad **Lifelong learning forest**)
- Techniky nácviku (napríklad **Experience replay**)
- Neurónové siete rozšírené o pamäť – MANN (napríklad **GEM**)

Jednou z týchto metód je práve PER, ktorá patrí do sekcie techník nácviku, a zlepšuje algoritmu DQN možnosť „spomínať si“ na predošlé skúsenosti.

Cieľom tréningu je minimalizovať stratovú funkciu pomocou metód, ako je gradientový zostup. Rýchlosť učenia určuje tempo, akým model aktualizuje svoje váhy počas tréningu. Konfigurácia váh modelu je reprezentáciou znalostí: matematickým odrazom toho, ako model chápe svoje tréningové údaje. Ak model upraví svoje váhy natoľko podstatne, že nové hodnoty už nie sú relevantné pre predchádzajúce úlohy, stratí schopnosť tieto úlohy vykonávať. V procese učenia sa nových úloh model „katastrofálne“ alebo úplne zabudol, ako pristupovať k starým. Skryté vrstvy medzi vstupom a výstupom neurónovej siete sa môžu časom posúvať. Keď neurónové siete uprednostňujú nové údaje pred predchádzajúcimi znalosťami, môžu nadmerne upraviť svoju váhu - namiesto rozširovania svojich znalostí model efektívne nahrádza svoje predchádzajúce znalosti novými údajmi. [49][50][51]

5.1.6 Zhrnutie DQN

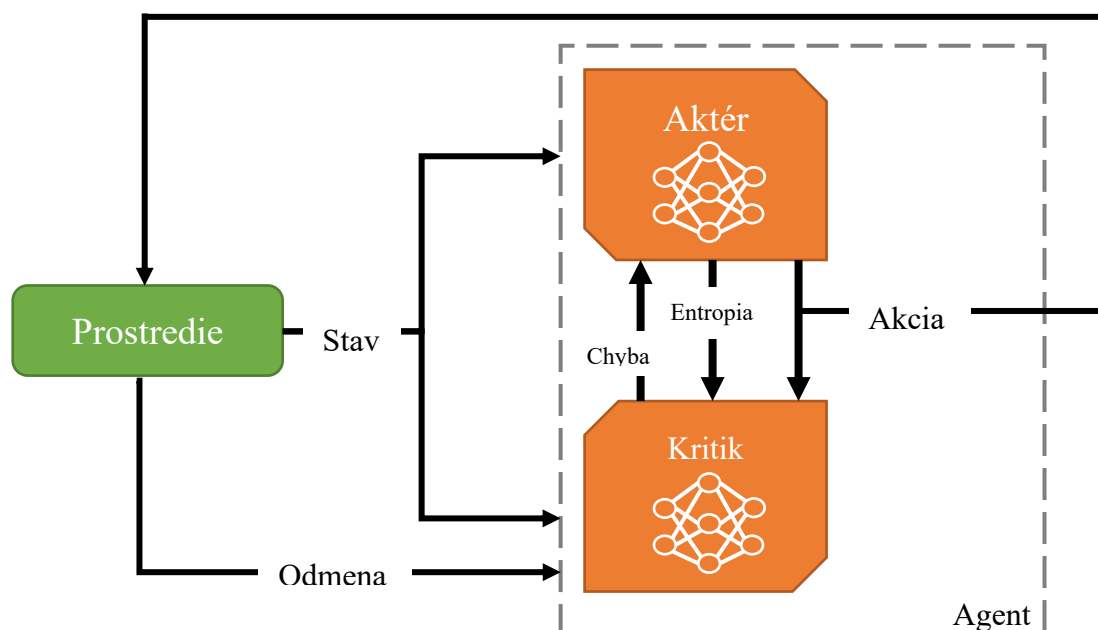
V skratke sa dá princíp funkcie DQN zhrnúť ako snaha samostatného agenta zlepšiť svoju schopnosť vykonať nejakú úlohu, a to tak, že najprv skúša úplne náhodné úkony, z ktorých získava skúsenosti a z týchto skúseností zisťuje, aké úkony mu v predošlých pokusoch získali najvyššie odmeny. Postupom času sa agent učí opakovať tieto úkony namiesto tých náhodných. Tento proces učenia je umožnený dvomi neurónovými sieťami:

- hlavná sieť, ktorej cieľom je učenie agenta a je aktualizovaná po každom tréningovom kroku, má na starosti odhad Q-hodnôt pre všetky možné úkony agenta;
- cieľová sieť, ktorá je kópiou hlavnej siete, no je aktualizovaná menej často, táto sieť má na starosti výpočet stabilného Q-cieľu, ktorý využíva hlavná sieť pre svoje trénovanie.

Cieľom tréningu je čo najviac znížiť hodnotu Q-straty, ktorá udáva rozdiel agentových momentálnych odhadov od dlhodobých ideálnych očakávaní (odhadnuté cieľovou sieťou).

5.2 Soft Actor-Critic

Soft Actor-Critic, skrátene SAC, je algoritmus RL vyvinutý na univerzite Berkeley v roku 2018. Podobne ako algoritmus DQN je založený na princípe off-policy. Veľkým rozdielom od DQN je ale fakt, že pracuje so spojitými akciami. SAC nevynakladá snahu maximalizovať len dlhodobé odmeny, snaží sa aj maximalizovať entropiu stratégie. Toto sa nazýva **Soft Policy Iteration**. [41]



Obrázok 13: Štruktúra Soft Actor-Critic [45]

5.2.1 Entropia v SAC, Soft Policy Iteration

Entropia sa dá vysvetliť mnohými spôsobmi, a definícia veľmi záleží na kontexte a aplikácii. V tomto kontexte by sa však dala entropia vysvetliť ako „jak moc nepredvídateľná je nejaká náhodná veličina?“. Ak má náhodná veličina vždy jednu hodnotu, entropia je rovná nule, lebo táto veličina nie je vôbec nepredvídateľná. Ak ale náhodná veličina môže nabráť podobu akéhokoľvek reálneho čísla s rovnomernou pravdepodobnosťou, entropia je vysoká, pretože v tomto prípade je nepredvídateľnosť obrovská. Vysoká entropia je ideálna v prípade, ak chceme algoritmus naviesť k prieskumu prostredia, nakoľko podporuje toto správanie. V porovnaní s algoritmom DQN by sme mohli entropiu prirovnáť k hodnote ϵ , ktorá v ňom podobne navádzala agenta k prieskumu (no tieto hodnoty v praxi fungujú veľmi odlišne). Entropia nabáda stratégiu – Q-funkciu – aby priradila akciám, ktoré majú rovnakú alebo skoro rovnakú Q-hodnotu, rovnaké pravdepodobnosti. Okrem toho tiež zabraňuje nechcenému správaniu, napríklad aby agent nezačal opakovať určitú akciu, ktorá zneužíva možný nedostatok v odhadnutej Q-funkcii.

Tento mechanizmus umožňuje algoritmus SAC vyhnúť sa takzvanému „brittleness problem“, čo sa v skratke dá vysvetliť ako zlyhanie naučeného správania agenta pri najmenšej zmene v prostredí, a to tak, že navádza sieť so stratégiou k prieskumu, namiesto toho, aby priradzovala vysokú pravdepodobnosť akejkoľvek jednej časti rozsahu akcií. [41]

5.2.2 Pôvodná forma SAC a jej vylepšenie

Prvá verzia algoritmu SAC bola založená na báze troch funkcií – dve Q-funkcie a samostatná funkcia stav-hodnota, označovaná ako V . Cieľ Q-siete záleží na ciele V -siete, a cieľ V -siete záleží na Q-sieti, čo znie paradoxne. Dáva to však zmysel, pretože sa snažíme vynútiť Bellmanovu konzistenciu (Bellmanova rovnica) medzi týmito dvoma funkciami. Kvôli tomuto má V -sieť cieľ, ktorý je nepriamo závislý na sebe samej, čo znamená, že tento cieľ je založený na tých istých parametroch, ktoré sa pokúšame trénovať. Z tohto dôvodu je tréning veľmi nestabilný. Riešením pre tento problém je využitie sady parametrov, ktoré sa približujú parametrom hlavnej V -siete, no s časovým oneskorením. Výsledkom je podobná myšlienka, ako pri DQN, kde cieľová sieť je menej často aktualizovaná kópia hlavnej siete. Toto však nie je jediný spôsob, ako predísť nestabilite. Druhou možnosťou je metóda „Polyak Averaging“ – Polyakove priemerovanie, ktorá priemeruje cieľovú a hlavnú sieť. [41]

Vylepšená verzia algoritmu SAC, vyvinutá opäť na univerzite Berkeley, bola úplne zbavená V-funkcie, a využíva nový parameter „temperature“ – teplota, tiež nazývaný alfa, ktorý pridáva automatické zisťovanie váhy entropie. [41]

5.2.3 Polyak Averaging

Metóda Polyakovho priemerovania bola vytvorená pre zlepšenie stability a výkonu iteratívnych optimalizačných algoritmov, ako gradientový zostup (gradient descent, spomenutý v 5.1.4). Myšlienkou je spriemerovať parametre modelu, ako váhy, naprieč viacerými iteráciami, namiesto predpokladania ich hodnôt v poslednej iterácii. Výsledkom tohto prístupu sú lepší výkon generalizácie a plynulejšia konvergencia², ako aj zníženie oscilácií a šumu počas tréningovania, hlavne ak sa pracuje so stochastickým alebo mini-batch gradientovým zostupom.

V štandardnej forme gradientového zostupu sú aktualizované parametre θ_t v každej iterácii t pomocou tohto pravidla:

$$\theta_{t+1} = \theta_t - \eta \nabla L(\theta_t) \quad (3)$$

Kde:

- θ_t reprezentuje parametre v iterácii t
- η je rýchlosť učenia
- $\nabla L(\theta_t)$ je gradient stratovej funkcie L s ohľadom na parametre

Pri tomto prístupe sú finálne parametre modelu tie, ktoré boli vypočítané v poslednej iterácii. [42]

² Konvergencia v RL znamená, že agent sa naučil stabilné správanie alebo stratégiu, ktorá sa dodatočným učením nejako viac nemení [44]

Pri Polyakovom priemerovaní sú parametre spriemerované naprieč iteráciami. Namiesto využívania iba parametrov z predchádzajúcej iterácie je vypočítaný kĺzavý priemer parametrov cez všetky iterácie podľa pravidla:

$$\theta_{\text{avg}} = \frac{1}{T} \sum_{t=1}^T \theta_t \quad (4)$$

Kde:

- θ_{avg} je spriemerovaný parameter po T iteráciách
- θ_t je parameter v iterácii t
- T je celkový počet iterácií

V tejto formulácii každá aktualizácia parametru prispieva konečnému modelu, čo môže značne vyrovnať výkyvy spôsobené šumivými gradientmi alebo oscilácie v optimalizačnom procese. [42]

5.2.4 Dvojica Q-Funkcií

Algoritmus SAC využíva dve Q-funkcie na to, aby sa mohol vyhnúť „preceňovaniu“ Q-hodnôt. Tento problém nastáva pri algoritmoch RL, ktoré sú charakteru off-policy v prípade, že sa bootstrapuje³ ako:

$$\max_{a'} Q(s', a') \quad (5)$$

V takomto prípade akékoľvek pozitívne chyby v odhade budú zveličené operátorom max, čo vedie k systematicky príliš optimistickým odhadom hodnôt. Postupom času môže tento bias destabilizovať učenie, pretože stratégia sa naháňa za neexistujúcimi vysokohodnotnými akciami, a ako následok sa hodnotové ciele posúvajú nahor.

Tomuto správaniu sa dá zabrániť tak, že sú trénované dve samostatné Q-funkcie (kritici) – $Q_1(s, a)$ a $Q_2(s, a)$, z ktorých sa využije minimum ich odhadov cieľu. Je totiž veľmi nepravdepodobné, že obe siete precenia rovnakú akciu rovnakým množstvom v rovnaký čas. [41] [43]

³ Bootstrapping – metóda, pri ktorej si agent aktualizuje svoje odhady hodnôt (napr. očakávané odmeny) pomocou vlastných momentálnych predpovedí namiesto čakania na úplný výsledok vykonaných akcií

Výber minima dáva pesimistický no presnejší cieľ, ktorý znižuje bias preceňovania:

$$y = r + \gamma(\min\{Q_1(s', a'), Q_2(s', a')\} - \alpha \log \pi(a' | s')) \quad (6)$$

Kde:

- y je bootstrap cieľ pre aktualizácie Q-funkcií
- r je okamžitá odmena po vykonaní akcie a v stave s
- γ je diskontný faktor určujúci, ako moc sa zaoberať budúcimi odmenami
- $\min\{Q_1(s', a'), Q_2(s', a')\}$ je výber minima z odhadov dvoch cieľových Q-sietí, čo dáva pesimistický a empiricky stabilnejší cieľ pre zabránenie preceňovaniu
- s', a' je nasledujúci stav a akcia získané zo súčasnej stratégie π_θ
- $-\alpha \log \pi(a' | s')$ je pridanie bonusovej entropie, čo nabáda výber akcií vedúcich k vyššej úrovni prieskumu [41] [43]

5.2.5 Proces trénovania

Štruktúra tohto algoritmu je spomenutá v samotnom názve tohto algoritmu - „Actor-Critic“. Štruktúra Actor-Critic nie je špecifická pre algoritmus SAC. Iné populárne Actor-Critic algoritmy sú napríklad A2C, A3C, DDPG. Využíva dve siete, aktéra a kritika. Úlohou aktéra je vykonávať akcie na základe pozorovania prostredia. Úlohou kritika je navádzať aktéra k lepším výsledkom na základe jeho predošlých skúseností. Oproti ostatným Actor-Critic algoritmom je SAC robustnejší, pretože vďaka Soft Policy Iteration je schopný vysporiadať sa s nepredvídateľnými situáciami. Tiež je menej náchylný na zaseknutie sa v lokálnych optimách.

Proces začína zberom dát interakciou s prostredím. Z týchto skúseností je vybratá mini-batch, ktorá sa využíva pre výpočet mäkkého TD-cieľu. Pre oboch kritikov je minimalizovaná stredná kvadratická chyba podľa $(Q_{\phi_i}(s, a) - y)^2$. Po aktualizovaní kritikov dochádza k aktualizácii aktéra na základe naučených Q-hodnôt od kritikov. Aktér sa „pýta“ kritikov, ktoré akcie sa javia ako najlepšie (po zohľadnení entropie), a upravuje svoje parametre pre zvýšenie šance vykonania týchto akcií. [41] [43] [47]

V praxi dochádza k striedaniu – jeden alebo viac gradientových krokov kritikov (pre prispôsobenie Q), nasledované jedným gradientovým krokom aktéra (pre zlepšenie π), čo ich udržuje na rovnakom replay buffer-i⁴. Minimalizuje sa strata stratégie pomocou:

$$\alpha \log \pi(a | s) - Q_{\phi_{min}}(s, a) \quad (7)$$

kde $Q_{\phi_{min}}$ je kritik s nižšou hodnotou.

Po aktualizovaní aktéra je na rade úprava teploty (α) prostredníctvom gradientového zostupu:

$$-\alpha \log \pi(a | s) - H_{target} \quad (8)$$

Kde H_{target} je cieľová hodnota entropie, ktorú by mal agent udržiavať.

Nakoniec dochádza k aktualizovaniu cieľovej siete $Q_{\bar{\phi}_i}$ pomocou Polyakovho priemerovania – soft update. [41] [43] [47]

⁴ Replay buffer ukladá a opätovne prehráva skúsenosti nazbierané agentom vďaka interakcii s prostredím [46]

II PRAKTICKÁ ČÁST

6 PRÍPRAVA NÁSTROJOV

Ako prvé je potrebné nainštalovať všetky súčasti – hru Trackmania 2020, skriptovaciu platformu OpenPlanet a framework TMRL.

6.1 Trackmania 2020

Hra Trackmania 2020 je voľne dostupná na herných platformách Ubisoft Connect [38] a Steam [39] a má približne 7GB. V tomto prípade bola hra nainštalovaná prostredníctvom platformy Steam, no aj napriek tomu je dodatočne automaticky doinštalovaná platforma Ubisoft Connect po prvom spustení, nakoľko vydavateľské štúdio Ubisoft vlastní vývojára Nadeo. Po tom, ako sa hra načíta je potrebné prejsť do nastavení (tlačidlo Settings v ľavom dolnom rohu okna).



Obrázok 14: Ukážka hlavného menu hry Trackmania 2020

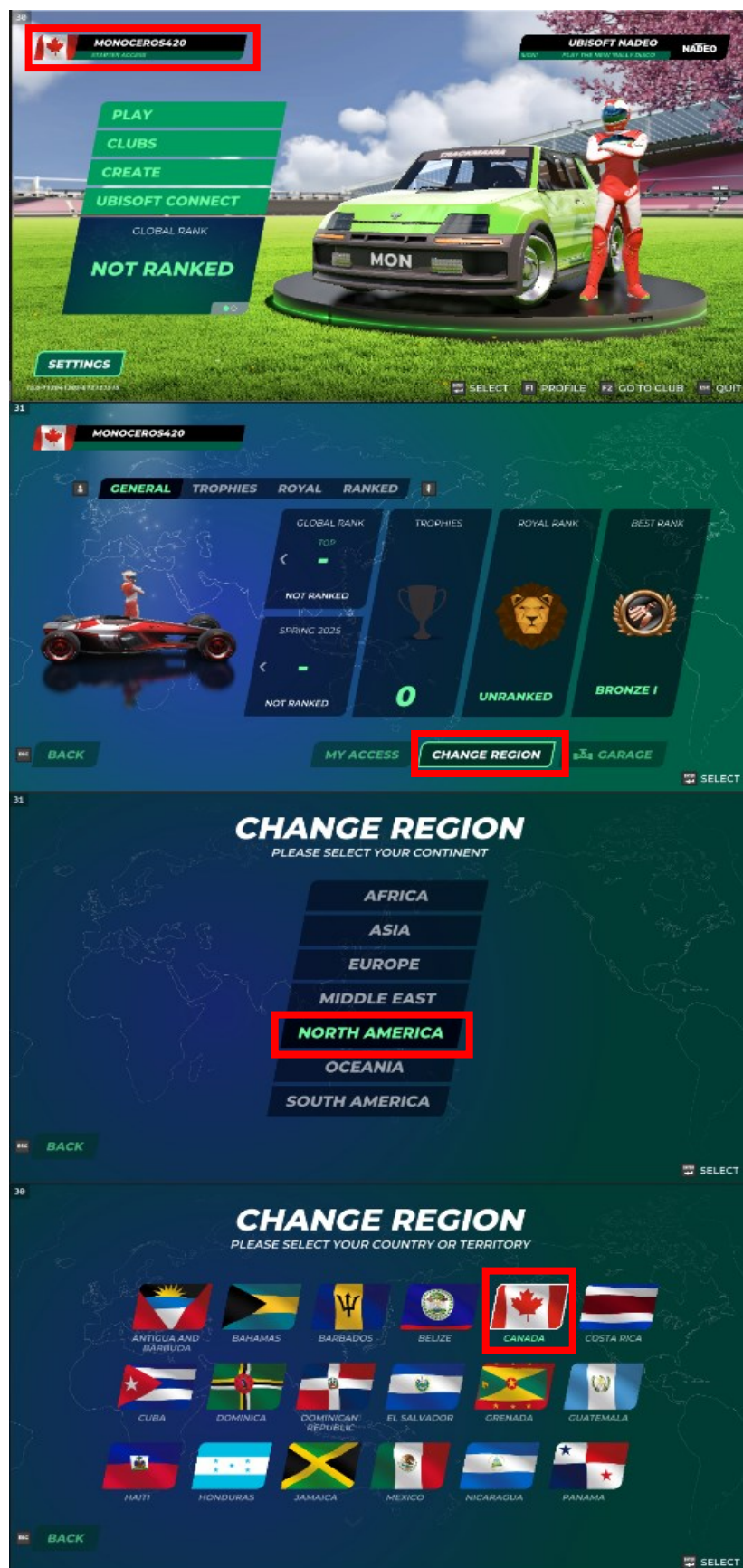
Po otvorení okna s nastaveniami je treba zvoliť kartu „Video“. V tejto karte boli všetky dostupné nastavenia prepnuté do najnižšej možnej kvality, napr. „Shaders Quality“ z „Very High“ na „Very Low“. Taktiež bolo zapnuté obmedzenie snímkovej frekvencie na 30. Tieto zmeny boli vykonané podľa odporúčenia v dokumentácii frameworku TMRL pre slabšie počítačové zostavy. Trénovanie je veľmi hardvérovo náročné, a preto je preň potrebné zaistiť čo najviac prostriedkov. Dodatočne bola v okne „Sound“ znížená hladina hlasitosti hudby na nulu, no to je len voľba preferencie.

Hru je hlavne potrebné nastaviť do režimu „Windowed“, pretože pre tréning musí byť okno hry v ľavom hornom rohu obrazovky.



Obrázok 15: Ukážka okna „Video“ v sekcii „Settings“

Dodatočne je potrebné manuálne zmeniť základný vzhľad auta, ktorý sa automaticky nastaví podľa regiónu hráča. Ako je uvedené v dokumentácii frameworku TMRL, dopredu trénovaný model SAC využíval vzhľad auta, ktorý mal kanadskú vlajku. Pre najideálnejšie podmienky na otestovanie tohto modelu bol zmenený vzhľad auta zo slovenskej vlajky na kanadskú a to tak, že sa v hlavnej ponuke hry klikne na meno hráča, zvolí možnosť „Change Region“, následne zvolí „North America“, potom „Canada“ a voľba konkrétneho regiónu ako aj typ „pilota“ vozidla už nie sú veľmi dôležité, a nemajú žiadny vplyv na efektivitu testovacieho modelu.

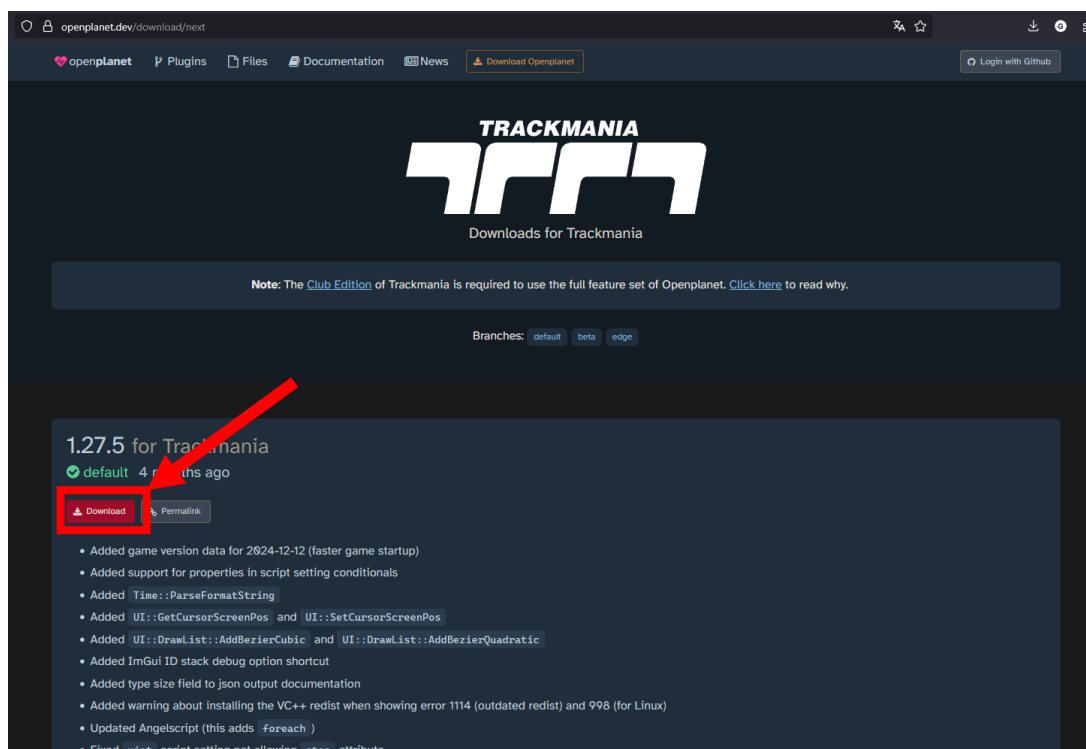


Obrázok 16: Proces zmeny regiónu pre zmenenie vzhľadu vozidla

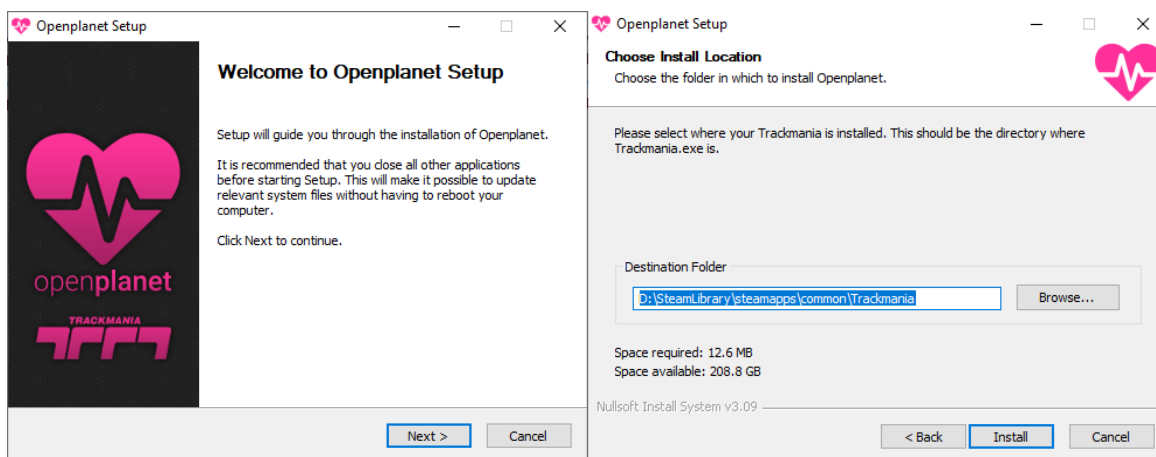
Základná príprava hry je týmto dokončená.

6.2 Skriptovacia platforma OpenPlanet pre hru Trackmania 2020

Skriptovacia platforma OpenPlanet je v kontexte tejto práce využívaná len ako sprostredkovateľ pre framework TMRL. Pomocou OpenPlanet využíva TMRL dva doplnky na získavanie informácií priamo z hry pre výpočet odmeny – TMRL Grab Data a TMRL Save Ghost. Inštalačný súbor pre OpenPlanet je dostupný na adrese [35]. Inštalácia je jednoduchá. Jediné, čo je potrebné, je spustiť stiahnutý .exe súbor; zvoliť, kde na disku je hra nainštalovaná, a nechať program nainštalovať potrebné súbory.



Obrázok 17: Stránka pre stiahnutie inštalačného súboru platformy OpenPlanet



Obrázok 18: Inštalácia platformy OpenPlanet

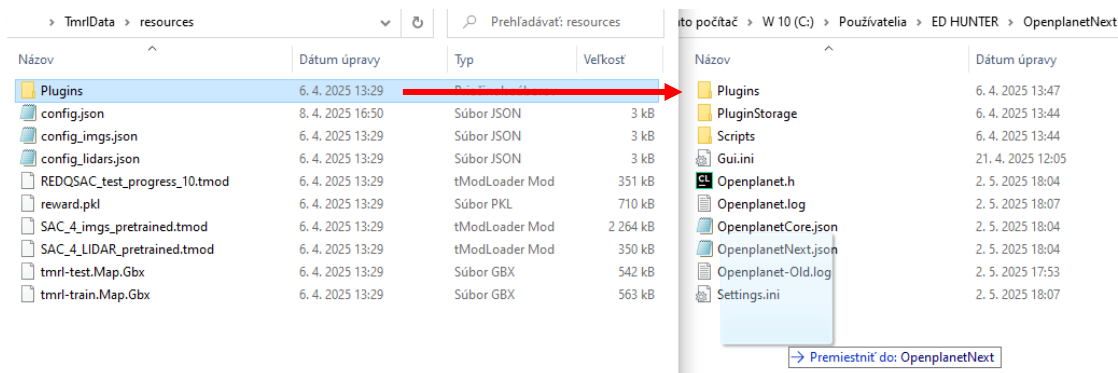
6.3 Framework TMRL

Tento framework je dostupný na odkaze <https://github.com/trackmania-rl/tmrl> no inštaluje sa v príkazovom riadku pomocou príkazu `pip install tmrl`. Ako je v požiadavkách uvedené, TMRL funguje na operačných systémoch Windows a Linux a pre funkčnosť potrebuje programovací jazyk Python, verzia 3.7 alebo novšia. Najväčšiu efektivitu dosahuje na počítačových zostavách, ktoré využívajú grafickú kartu značky NVIDIA, a to preto, lebo využíva knižnicu PyTorch, ktorá je vcelku závislá na rozhraní CUDA pre tréning modelov. Počas procesu inštalácie v príkazovom riadku bude dodatočne nainštalovaný aj ovládač pre emuláciu virtuálneho gamepadu, ktorý bude umožňovať modelu interakciu s hrou.

Po inštalácii je potrebné sa uistiť, či prebehla úspešne a všetky súčasti sú tam, kde majú byť. Ako prvé je treba spustiť príkaz `python -m tmrl -install`, ktorý validuje inštaláciu. Ďalej je odporúčené uistiť sa, že inštalácia vytvorila priečinok `TmrlData`, ktorý sa nachádza v „C:\Users\[Užívateľské meno]“. Z tohto priečinku, konkrétne z pod-priečinku „resources“ je potom potrebné skopírovať priečinok „Plugins“ do priečinku „OpenplanetNext“, ktorý sa tiež nachádza v „C:\Users\[Užívateľské meno]“.

Názov	Dátum úpravy	Typ
config	19. 4. 2025 12:30	Priečinok súborov
dataset	6. 4. 2025 13:29	Priečinok súborov
checkpoints	21. 4. 2025 16:29	Priečinok súborov
resources	6. 4. 2025 13:29	Priečinok súborov
reward	6. 4. 2025 13:29	Priečinok súborov
weights	21. 4. 2025 15:34	Priečinok súborov

Obrázok 19: Štruktúra priečinku TmrlData



Obrázok 20: Presun priečinku Plugins do priečinku OpenplanetNext

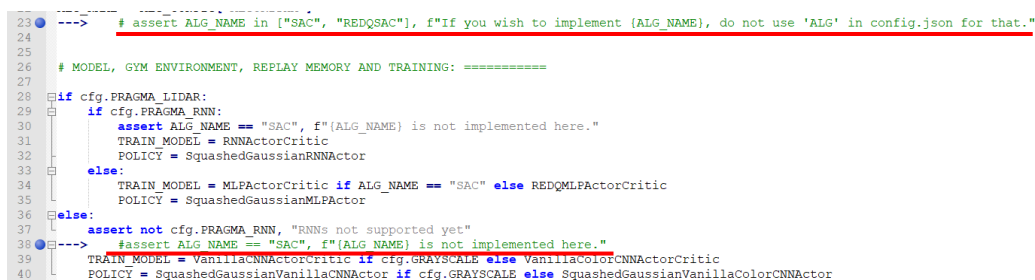
Ako posledný krok je potrebné presunúť testovaciu trať, na ktorej budú modely tréované. V priečinku resources je okrem priečinku Plugins aj súbor „tmrl-test.Map.gbx“, ktorý obsahuje túto trať. Tento súbor bude presunutý do priečinku „My maps“, ktorý sa nachádza v: C:\Users\[Užívateľské meno]\Documents\Trackmania2020\Maps.

6.3.1 Úpravy v súboroch TMRL

Z dôvodu nejasností a nekonzistencií v dokumentácii bolo potrebné vykonať dve zmeny v súbore „config_objects.py“. Tento súbor sa nachádza v:

C:\Users\[Užívateľské meno]\AppData\Local\Programs\Python\[Verzia jazyka Python]\Lib\site-packages\tmrl\config

Zmeny, ktoré je potrebné vykonať sa týkajú dvoch konkrétnych riadkov pri začiatku súboru – riadok 23 a riadok 38. K tomu, aby bol kód pre vlastné modely spustiteľný je potrebné tieto riadky označiť ako komentár pomocou symbolu #.



```

23  ---> # assert ALG_NAME in ["SAC", "REDQSAC"], f"If you wish to implement {ALG_NAME}, do not use 'ALG' in config.json for that."
24
25
26  # MODEL, GYM ENVIRONMENT, REPLAY MEMORY AND TRAINING: =====
27
28  if cfg.PRAGMA_LIDAR:
29      if cfg.PRAGMA_RNN:
30          assert ALG_NAME == "SAC", f"{ALG_NAME} is not implemented here."
31          TRAIN_MODEL = RNNActorCritic
32          POLICY = SquashedGaussianRNNActor
33      else:
34          TRAIN_MODEL = MLPActorCritic if ALG_NAME == "SAC" else REDQMLPActorCritic
35          POLICY = SquashedGaussianMLFactor
36      else:
37          assert not cfg.PRAGMA_RNN, "RNNs not supported yet"
38  ---> # assert ALG_NAME == "SAC", f"{ALG_NAME} is not implemented here."
39          TRAIN_MODEL = VanillaCNNActorCritic if cfg.GRAYSCALE else VanillaColorCNNActorCritic
40          POLICY = SquashedGaussianVanillaCNNActor if cfg.GRAYSCALE else SquashedGaussianVanillaColorCNNActor

```

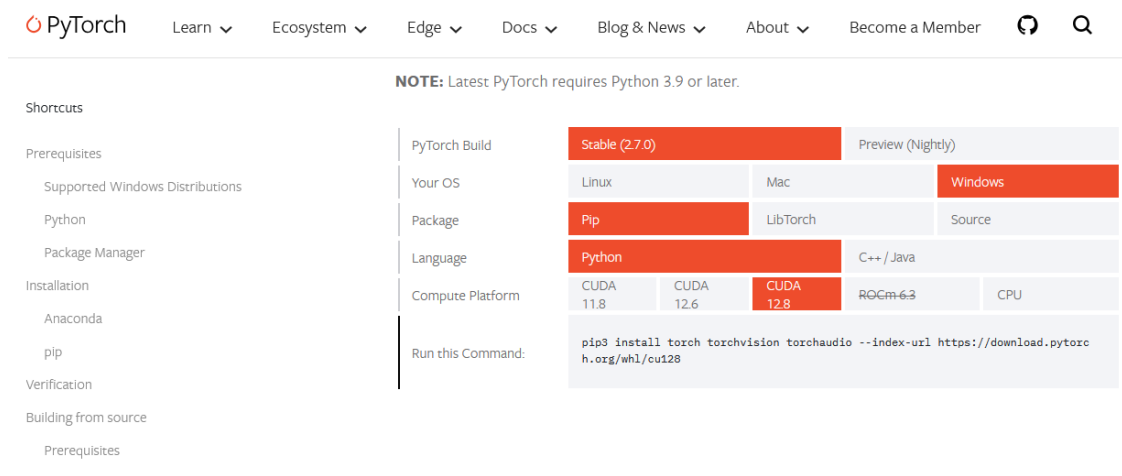
Obrázok 21: Ukážka pozície riadkov potrebných na komentovanie

Tieto riadky je treba označiť ako komentár preto, lebo pri implementácii akéhokoľvek modelu, ktorý nie je nazvaný „SAC“ alebo „REDQSAC“ (dva testovacie modely tréované tvorcom TMRL) dôjde k okamžitému ukončeniu programu. V dokumentácii nebolo nájdené žiadne lepšie riešenie. Implementácia vlastného tréningového pipeline by zaberala príliš veľa času, a vôbec nie je potrebná.

6.3.2 PyTorch CUDA

Dodatočne je potrebné nainštalovať verziu knižnice PyTorch, ktorá pracuje s rozhraním CUDA. Štandardná verzia, ktorá je stiahnutá a nainštalovaná počas inštalácie TMRL pracuje len s procesorom, a nie s grafickou kartou. PyTorch CUDA je dostupná na <https://pytorch.org/get-started/locally/> a inštaluje sa pomocou príkazu (príkazový riadok):

```
pip3 install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu128
```



Obrázok 22: Inštalčné nastavenia pre PyTorch CUDA

Inštalácia chvíľu potrvá, pretože sťahuje veľký inštalčný súbor pre sadu nástrojov CUDA, ktorý má veľkosť približne 3.3GB.

6.4 Príprava súboru pre implementáciu vlastného modelu

Pre implementáciu vlastného modelu bez potreby vytvorenia vlastného učiaceho „pipeline“ bude využívaný súbor „custom_actor_module.py“, ktorý zo základu nie je súčasťou TMRL, ale dá sa získať z GitHub repozitáru projektu TMRL na [36]. Tento súbor je primárne myslený pre užívateľov, ktorí by si chceli vytrénovať vlastný model pre takzvaný „Trackmania RoboRace League“, čo je akási súťaž slúžiaca pre porovnanie efektivity natrénovaných modelov. Táto práca sa nezapája do tejto súťaže, no súbor, ktorý je na ňu využívaný, bude užitočný, nakoľko obsahuje návod na približné implementovanie vlastného modelu aj predpripravené funkcie pre spracovávanie dát za behu hry. V pôvodnej forme obsahuje úplnú implementáciu algoritmu SAC, ktorú je potrebné odstrániť – funkcia `SACTrainingAgent`, funkcia `MyActorModule`, funkcia `VanillaCNNQFunction` a funkcia `VanillaCNNActorCritic`. Tieto funkcie vôbec nebudú potrebné, nakoľko všetky súčasti budú implementované podľa vlastnej potreby.

```

class SACTrainingAgent(TrainingAgent):
    # no-grad copy of the model used to send the Actor weights in get_actor():
    model_nograd = cached_property(lambda self: no_grad(copy_shared(self.model)))

    def __init__(self,
                 observation_space=None, # Gymnasium observation space (required argument here for your convenience)
                 action_space=None, # Gymnasium action space (required argument here for your convenience)
                 device=None, # Device our TrainingAgent should use for training (required argument)
                 model_cls=VanillaCNNActorCritic, # An actor-critic module, encapsulating our ActorModule
                 gamma=0.99, # Discount factor
                 polyak=0.995, # Exponential averaging factor for the target critic
                 alpha=0.2, # Value of the entropy coefficient
                 lr_actor=1e-3, # Learning rate for the actor
                 lr_critic=1e-3): # Learning rate for the critic

        # required arguments passed to the superclass:
        super().__init__(observation_space=observation_space,
                        action_space=action_space,
                        device=device)

        # custom stuff:
        model = model_cls(observation_space, action_space)
        self.model = model.to(self.device)
        self.model_target = no_grad(deepcopy(self.model))
        self.gamma = gamma
        self.polyak = polyak
        self.alpha = alpha
        self.lr_actor = lr_actor
        self.lr_critic = lr_critic
        self.q_params = itertools.chain(self.model.q1.parameters(), self.model.q2.parameters())
        self.pi_optimizer = Adam(self.model.actor.parameters(), lr=self.lr_actor)
        self.q_optimizer = Adam(self.q_params, lr=self.lr_critic)
        self.alpha_t = torch.tensor(float(self.alpha)).to(self.device)

    def get_actor(self):
        """
        Returns a copy of the current ActorModule.

        We return a copy without gradients, as this is for sending to the RolloutWorkers.

        Returns:
            actor: ActorModule: updated actor module to forward to the worker(s)
        """
        return self.model_nograd.actor

    def train(self, batch):
        # First, we decompose our batch into its relevant components, ignoring the "truncated" signal:
        o, a, r, o2, d, _ = batch

        # We sample an action in the current policy and retrieve its corresponding log probability:
        pi, logp_pi = self.model.actor(obs=o, test=False, compute_logprob=True)

        # We also compute our action-value estimates for the current transition:
        q1 = self.model.q1(o, a)
        q2 = self.model.q2(o, a)

        # Now we compute our value target, for which we need to detach from gradients computation:
        with torch.no_grad():
            a2, logp_a2 = self.model.actor(o2)
            q1_pi_targ = self.model_target.q1(o2, a2)
            q2_pi_targ = self.model_target.q2(o2, a2)
            q_pi_targ = torch.min(q1_pi_targ, q2_pi_targ)
            backup = r + self.gamma * (1 - d) * (q_pi_targ - self.alpha_t * logp_a2)

        # This gives us our critic loss, as the difference between the target and the estimate:
        loss_q1 = ((q1 - backup)**2).mean()
        loss_q2 = ((q2 - backup)**2).mean()
        loss_q = loss_q1 + loss_q2

        # We can now take an optimization step to train our critics in the opposite direction of this loss' gradient:
        self.q_optimizer.zero_grad()
        loss_q.backward()
        self.q_optimizer.step()

        # For the policy optimization step, we detach our critics from the gradient computation graph:
        for p in self.q_params:
            p.requires_grad = False

        # We use the critics to estimate the value of the action we have sampled in the current policy:

```

Obrázok 23: Časť kódu funkcie SACTrainingAgent

7 IMPLEMENTÁCIA VLASTNÝCH MODELOV

Pomocou pripravených nástrojov budú implementované dva modely reinforcement learning-u – algoritmus DQN a algoritmus SAC. Pre každý algoritmus bude potrebné vytvoriť upravenú verziu súboru `config.json` s hyperparametrami daného algoritmu.

Súbor `custom_actor_module.py` obsahuje triedy `VanillaCNN` a `mlp` – klasická konvolučná neurónová sieť a viacvrstvový perceptron. Tieto funkcie budú užitočné, pretože ich hlavným účelom je spracovávať informácie z obrázkov hry v reálnom čase. Prostredie, v ktorom budú algoritmy pracovať, využíva posledné štyri obrázky z hry pre získavanie informácií, takže konvolučná neurónová sieť bude mať štyri konvolučné vrstvy. Po spracovaní týchto obrázkov vznikne výstup, ktorý je potom „sploštený“ do takzvaného „vektoru vlastností“ vo forme – rýchlosť; prevodový stupeň; otáčky; dve predchádzajúce akcie, ktoré model zvolil; a preferovaná akcia pre kritiku. Tento vektor následne prechádza cez perceptron, ktorý produkuje buď jeden skalár pre výstup Q-siete alebo vektor vlastností pre siete aktérov predtým, ako použijú samostatné výstupné hlavy. V krátkosti sa jedná o jednoduchú feed-forward sieť (viaceré lineárne vrstvy + ReLU⁵), ktorej šírku a hĺbku možno ovládať pomocou zoznamu veľkostí.

Tieto triedy boli vytvorené tvorcom frameworku TMRL, a neboli implementované manuálne. Budú využívané ako pomocné funkcie v algoritme DQN. Pre algoritmus SAC bude vytvorená upravená verzia funkcie `mlp`, ktorá bude pracovať s LIDAR prostredím.

⁵ ReLU – Rectified Linear Unit, je to aktivačná funkcia, ktorá generuje nulu pre akýkoľvek negatívny vstup a výstupom je samotný vstup, ak je pozitívny, čo pomáha neurónovým sieťam učiť sa rýchlejšie a vyhnúť sa miznúcim gradientom

```

class VanillaCNN(nn.Module):
    def __init__(self, q_net):
        """
        super(VanillaCNN, self).__init__()

        self.q_net = q_net

        # Convolutional layers processing screenshots:
        # The default config.json gives 4 grayscale images of 64 x 64 pixels
        self.h_out, self.w_out = img_height, img_width
        self.conv1 = nn.Conv2d(imgs_buf_len, 64, 8, stride=2)
        self.h_out, self.w_out = conv2d_out_dims(self.conv1, self.h_out, self.w_out)
        self.conv2 = nn.Conv2d(64, 64, 4, stride=2)
        self.h_out, self.w_out = conv2d_out_dims(self.conv2, self.h_out, self.w_out)
        self.conv3 = nn.Conv2d(64, 128, 4, stride=2)
        self.h_out, self.w_out = conv2d_out_dims(self.conv3, self.h_out, self.w_out)
        self.conv4 = nn.Conv2d(128, 128, 4, stride=2)
        self.h_out, self.w_out = conv2d_out_dims(self.conv4, self.h_out, self.w_out)
        self.out_channels = self.conv4.out_channels

        # Dimensionality of the CNN output:
        self.flat_features = self.out_channels * self.h_out * self.w_out

        # Dimensionality of the MLP input:
        # The MLP input will be formed of:
        # - the flattened CNN output
        # - the current speed, gear and RPM measurements (3 floats)
        # - the 2 previous actions (2 x 3 floats), important because of the real-time nature of our controller
        # - when the module is the critic, the selected action (3 floats)
        float_features = 12 if self.q_net else 9
        self.mlp_input_features = self.flat_features + float_features

        # MLP layers:
        # (when using the model as a policy, we will sample from a multivariate gaussian defined later in the tutorial;
        # thus, the output dimensionality is 1 for the critic, and we will define the output layer of policies later)
        self.mlp_layers = [256, 256, 1] if self.q_net else [256, 256]
        self.mlp = mlp([self.mlp_input_features] + self.mlp_layers, nn.ReLU)

def forward(self, x):
    """
    if self.q_net:
        # The critic takes the next action (act) as additional input
        # act1 and act2 are the actions in the action buffer (real-time RL):
        speed, gear, rpm, images, act1, act2, act = x
    else:
        # For the policy, the next action (act) is what we are computing, so we don't have it:
        speed, gear, rpm, images, act1, act2 = x

    # Forward pass of our images in the CNN:
    # (note that the competition environment outputs histories of 4 images
    # and the default config outputs these as 64 x 64 greyscale,
    # we will stack these greyscale images along the channel dimension of our input tensor)
    x = F.relu(self.conv1(images))
    x = F.relu(self.conv2(x))
    x = F.relu(self.conv3(x))
    x = F.relu(self.conv4(x))

    # Now we will flatten our output feature map.
    # Let us double-check that our dimensions are what we expect them to be:
    flat_features = num_flat_features(x)
    assert flat_features == self.flat_features, f"x.shape:{x.shape},\
        flat_features:{flat_features},\
        self.out_channels:{self.out_channels},\
        self.h_out:{self.h_out},\
        self.w_out:{self.w_out}"

    # All good, let us flatten our output feature map:
    x = x.view(-1, flat_features)

    # Finally, we can feed the result along our float values to the MLP:
    if self.q_net:
        x = torch.cat((speed, gear, rpm, x, act1, act2, act), -1)
    else:
        x = torch.cat((speed, gear, rpm, x, act1, act2), -1)
    x = self.mlp(x)

    # And this gives us the output of our deep neural network :)
    return x

```

Obrázok 24: Kód funkcie VanillaCNN

```
def mlp(sizes, activation, output_activation=nn.Identity):
    """
    layers = []
    for j in range(len(sizes) - 1):
        act = activation if j < len(sizes) - 2 else output_activation
        layers += [nn.Linear(sizes[j], sizes[j + 1]), act()]
    return nn.Sequential(*layers)
```

Obrázok 25: Kód funkcie mlp

7.1 Implementácia DQN

Ako prvý bude implementovaný algoritmus DQN, nakoľko pracuje s „experience replay“, ktorý má zmysel pre hru ako Trackmania. Profesionálni hráči často strávia aj 200 hodín počas procesu zlepšovania sa na jednej trati, a keďže algoritmus bude tiež trénovaný na jednej konkrétnej trati, využitie algoritmu s akousi „pamäťou“ predošlých pokusov môže byť výhodou. Pre DQN budú vytvorené tri hlavné triedy – DQNNetwork, DQNActorModule a DQNTrainingAgent. Najprv je však potrebné zadať možné akcie, z ktorých si model bude vyberať.

7.1.1 Definícia „action space“

Herné vstupy sú spojitého charakteru, no DQN potrebuje, aby boli nejakým spôsobom prevedené do diskretnej formy. Toto je dosiahnuté tak, že možné vstupy vo forme spojitých trojíc - napr. (1.0, 0.0, 0.0) – plný plyn, žiadna brzda, žiadne zabáčanie; sú uložené do listu ACTIONS, kde je im pridelený diskretný index. S týmito indexmi potom pracuje model pri tréningu.

```
ACTIONS = [
    (1.0, 0.0, 0.0), # full gas only (straight)
    (1.0, 0.0, -1.0), # full gas + full steer left
    (1.0, 0.0, 1.0), # full gas + full steer right
    (1.0, 0.0, -0.5), # full gas + half steer left
    (1.0, 0.0, 0.5), # full gas + half steer right
    (0.0, 1.0, 0.0), # brake
    (0.0, 1.0, -1.0), # brake + full steer left
    (0.0, 1.0, 1.0), # brake + full steer right
]
_action_to_index = {a: i for i, a in enumerate(ACTIONS)}
```

Obrázok 26: List možných akcií, prevod zo spojitého na diskretný charakter

7.1.2 Trieda DQNNetwork

Táto trieda je vcelku krátka, jej podstatou je vziať vstupné dáta od triedy VanillaCNN, extrahuje z nich „features“⁶, a pomocou týchto features sa sieť rozhoduje, ako veľmi sa oplatia vykonať jednotlivé akcie/vstupy.

```
class DQNNetwork(nn.Module):  
    """Q-network for DQN: uses VanillaCNN for features and outputs Q-values."""  
    def __init__(self, num_actions):  
        super().__init__()  
        self.rep = VanillaCNN(q_net=False)  
        self.q_out = nn.Linear(256, num_actions)
```

Obrázok 27: Inicializácia funkcie DQNNetwork

Začína sa tým, že je do premennej „rep“ volaná trieda VanillaCNN, ktorá ako už bolo spomenuté, pracuje s poslednými štyrmi obrázkami z hry, a tieto obrázky transformuje na číselnú reprezentáciu, ktorá sa nazýva „features“. Tento proces sa nazýva „feature extraction“, alebo extrakcia vlastností. Následne je vytvorená lineárna vrstva, ktorá na vstupe berie 256 čísel, a na výstupe má toľko čísel, koľko máme akcií (počet vstupov v liste ACTIONS, v tomto prípade 8). Táto vrstva má za úlohu odhadnúť hodnotu Q pre každý možný vstup.

7.1.2.1 Funkcia forward()

Nasleduje funkcia forward, ktorá definuje akúsi cestu, ktorú dáta nasledujú naprieč sieťou. Je volaná PyTorch-om pri každom prechode dát cez sieť. Konkrétne určuje, ako bude sieť spracovávať vstup „obs“ (pozorovanie). Následne sa zasiela momentálne pozorovanie do už spomenutého VanillaCNN, kde je zredukované na 256 čísel. Na základe týchto 256 čísel sú potom odhadnuté Q-hodnoty pre každú akciu, ktorú si môže model zvoliť.

```
def forward(self, obs):  
    rep_out = self.rep(obs)  
    return self.q_out(rep_out)
```

Obrázok 28: Funkcia forward v DQNNetwork

⁶ Feature v CNN - špecifický vzor (ako okraj, roh alebo textúra), ktorý sa filter naučil detegovať vo vstupnom obrázku

7.1.3 Trieda DQNActorModule

Ako prvý krok je vytvorená DQN sieť pomocou triedy `DQNNetwork` pre odhadovanie Q-hodnôt každej akcie. Následne sú definované základné parametre hodnoty epsilon. Premenná „epsilon“ je nastavená na hodnotu 1.0, určuje počiatočnú pravdepodobnosť vybratia náhodnej akcie pre vyššiu mieru „exploration“, alebo prieskumu. Premenná `epsilon_end` určuje spodnú hranicu hodnoty epsilon-u, je nastavená na 0.01. Premenná `epsilon_decay` určuje to, ako rýchlo sa hodnota epsilon-u znižuje každým krokom.

```
class DQNActorModule(TorchActorModule):
    """ActorModule for DQN: epsilon-greedy discrete action selection."""
    def __init__(self, observation_space, action_space):
        super().__init__(observation_space, action_space)
        # build Q-network; device moved later
        self.q_network = DQNNetwork(len(ACTIONS))
        self.epsilon = 1.0
        # decay parameters
        self.epsilon_end = 0.01
        self.epsilon_decay = 0.995
```

Obrázok 29: Inicializácia DQN siete a hodnoty epsilon v triede DQNActorModule

7.1.3.1 Funkcia act()

Po inicializácii je vytvorená funkcia „act“, ktorá na začiatok zaistuje, že všetky prvky premennej „obs“ (či už tensor alebo pole NumPy) sú typu `torch.Tensor`, že sú na správnom zariadení (procesor alebo grafická karta), a že sú dátového typu `float32`.

```
def act(self, obs, test=False):
    # convert all obs tensors to float32 on the correct device
    obs = tuple(
        t.to(self.device).float() if torch.is_tensor(t)
        else torch.tensor(t, dtype=torch.float32, device=self.device)
        for t in obs
    )
```

Obrázok 30: Začiatok funkcie act v triede DQNActorModule

Následne je volaná DQN sieť, ktorá vyprodukuje vektor ôsmich Q-hodnôt. Vytvára sa premenná `idx`, ktorá odpovedá indexu vybranej akcie. Ak je zvolený testovací režim, vždy bude zvolená akcia s najvyššou Q-hodnotou. Ak ale ide o tréningový režim, podľa hodnoty epsilon je vybraná náhodná akcia, inak je zvolený prístup „greedy“, kde je zvolený index najvyššej hodnoty Q.

```
with torch.no_grad():
    # assume obs is already torch.Tensor on correct device
    q_vals = self.q_network(obs)
    if test:
        idx = torch.argmax(q_vals, dim=-1).item()
    else:
        if np.random.rand() < self.epsilon:
            idx = np.random.randint(len(ACTIONS))
        else:
            idx = torch.argmax(q_vals, dim=-1).item()
```

Obrázok 31: Logika výberu Q-hodnoty

Hodnota epsilon sa každým krokom znižuje podľa vzorca $\text{epsilon} * \text{epsilon_decay}$, s tým, že hodnota neklesne pod úroveň 0.01.

```
# decay epsilon per step
self.epsilon = max(self.epsilon_end, self.epsilon * self.epsilon_decay)
```

Obrázok 32: Znižovanie hodnoty epsilon

Nakoniec dochádza k výberu odpovedajúcej akcie z listu `ACTIONS` na základe premennej `idx`, ktorá bola definovaná v `if` podmienke vyššie.

```
throttle, brake, steer = ACTIONS[idx]
return np.array([throttle, brake, steer])
```

Obrázok 33: Výber akcie z listu

7.1.4 Funkcia `epsilon_scheduler()`

Logika týkajúca sa zmien hodnoty epsilon bola po niekoľkých iteráciách programu prepísaná funkciou `epsilon_scheduler`, v ktorej dochádza k znižovaniu hodnoty epsilon na základe prebiehajúcej tréningovej epochy – prvé dve epochy majú hodnotu epsilon nastavenú na 1.0 pre čo najvyššiu mieru prieskumu; epochy 3 až 8 postupne znižujú hodnotu epsilon z 1 na 0.01, kde zostane do konca tréningu.

```
def epsilon_scheduler(agent, epoch):
    """
    Schedule  $\epsilon$  by epoch: first 2 epochs pure exploration ( $\epsilon=1$ ),
    then linear decay to agent.epsilon_end over next 6 epochs, then floor.
    """
    if epoch < 2:
        agent.model.epsilon = 1.0
    elif epoch < 2 + 6:
        frac = (epoch - 2) / 6.0
        agent.model.epsilon = 1.0 - frac * (1.0 - agent.epsilon_end)
    else:
        agent.model.epsilon = agent.epsilon_end
```

Obrázok 34: Funkcia pre úpravu hodnoty epsilon-u na základe momentálnej epochy

7.1.5 Trieda DQNTrainingAgent

Táto trieda je, prakticky povedané, mozog celého tréningového procesu. Je v nej zadefinované, ako sa DQN sieť má trénovať a ako má zbierať skúsenosti. Trieda dedí od TrainingAgent. Po jej zadefinovaní je vytvorená premenná `model_nograd`, ktorá vytvára kópiu modelu bez gradientov, určenú pre zaslanie RolloutWorkerom. Je to bezpečná kópia, v režime „len na čítanie“ (read-only), aby worker-i pri zbieraní dát so sieťou nemanipulovali.

```
class DQNTrainingAgent(TrainingAgent):
    """DQN training algorithm."""
    # no-grad copy of model for sending to workers
    model_nograd = cached_property(lambda self: no_grad(copy_shared(self.model)))
```

Obrázok 35: Definícia triedy DQNTrainingAgent a vytvorenie kópie modelu

Nasleduje sekcia určená na inicializáciu dodatočných hyperparametrov algoritmu DQN ako *learning rate* (rýchlosť učenia), *gamma* („discount“ faktor, ako moc je zohľadnená budúca odmena), *target update interval* - po koľkých tréningových krokoch sa skopíruje hlavná sieť do cieľovej siete - ako aj Adam optimalizátoru, ktorý hýbe váhami siete na základe chyby. Taktiež tu je vytvorená premenná `model`, ktorá obsahuje hlavnú DQN sieť z triedy DQNActorModule, a premenná `model_target`, ktorá je presná kópia hlavnej siete, s tým rozdielom, že nie je aktualizovaná gradientom. Táto sieť je využívaná na stabilný výpočet cieľových Q-hodnôt.

```
def __init__(self, observation_space=None, action_space=None, device=None,
             gamma=0.995, lr=5e-4, target_update_interval=1000,
             epsilon_start=1.0, epsilon_end=0.01, epsilon_decay=0.995):
    super().__init__(observation_space, action_space, device)
    # main and target networks
    self.model = DQNActorModule(observation_space, action_space).to(self.device)
    self.model.epsilon = epsilon_start
    self.model.target = no_grad(deepcopy(self.model))
    self.gamma = gamma
    self.optimizer = Adam(self.model.q_network.parameters(), lr=lr)
    self.target_update_interval = target_update_interval
    self.epsilon_end = epsilon_end
    self.epsilon_decay = epsilon_decay
    self.steps = 0
```

Obrázok 36: Inicializácia premenných v triede DQNTrainingAgent

7.1.5.1 Funkcia get_actor()

Veľmi krátka funkcia `get_actor` len navracia sieť bez gradientov, ktorú budú využívať worker-i na zber skúseností.

```
def get_actor(self):
    # return the DQNActorModule (with no grads) for RolloutWorkers
    return self.model_nograd
```

Obrázok 37: Funkcia get_actor

7.1.5.2 Funkcia *train()*

Funkcia `train` má na starosti samotný proces tréningovania. Najprv sú definované vstupné premenné: `o` – stav, `a` – akcia, `r` – odmena, `o2` – nasledujúci stav, `d` – done-flag a posledná hodnota, ktorá nie je relevantná. Ďalej dochádza ku konvertovaniu všetkých tuplov na `float32 torch.Tensor`-y.

```
def train(self, batch):
    o, a, r, o2, d, _ = batch
    # convert observation tuples to float32 torch tensors on our device to match network dtype
    o = tuple(torch.as_tensor(feats, dtype=torch.float32, device=self.device) for feats in o)
    o2 = tuple(torch.as_tensor(feats, dtype=torch.float32, device=self.device) for feats in o2)
```

Obrázok 38: Definovanie premenných a konvertovanie tuplov na tensory

Následne je vykonaná indexácia akcií po tom, ako sú prijaté od workera pomocou `_action_to_index`. Potom je vypočítaná aktuálna Q -hodnota (`curr_q`) vykonanej akcie spolu s cieľovou Q -hodnotou na základe cieľovej siete (`target_q`) a Bellmanovej rovnice s tým, že ak tréningová epizóda skončila ($d = 1$), ďalšia odmena sa nezaráta.

$$target_q = r + \gamma(1 - d) \max_{a'} Q(s', a'; \theta^-) \quad (9)$$

Kde:

- r = aktuálna odmena,
- γ = discount faktor
- d = flag konca epizódy
- $\max_{a'} Q(s', a'; \theta^-)$ = najlepší budúci odhad

```
actions = []
for act in a.cpu().numpy():
    key = (float(act[0]), float(act[1]), float(act[2]))
    actions.append(_action_to_index[key])
actions = torch.LongTensor(actions).to(self.device)
# current Q
q_vals = self.model.q_network(o)
curr_q = q_vals.gather(1, actions.unsqueeze(1)).squeeze(1)
# target Q
with torch.no_grad():
    next_q_vals = self.model_target.q_network(o2)
    max_next_q = next_q_vals.max(1)[0]
    # build reward and done tensors on correct device/dtype
    rewards = torch.as_tensor(r, dtype=next_q_vals.dtype, device=self.device)
    dones = torch.as_tensor(d, dtype=next_q_vals.dtype, device=self.device)
    target_q = rewards + self.gamma * (1 - dones) * max_next_q
```

Obrázok 39: Indexácia akcií a výpočet `curr_q` a `target_q`

Ďalej prebieha spätná propagácia (back propagation) a aktualizácia váh podľa straty. Strata (loss) je vypočítaná ako stredná kvadratická odchýlka (mean squared error) medzi tensormi `curr_q` a `target_q`. Váhy siete sú upravované po vyčistení starých gradientov (`self.optimizer.zero_grad`) pomocou `loss.backward` – spätnej propagácie, kde PyTorch automaticky prepočítava všetky parciálne derivácie $\frac{\partial \text{loss}}{\partial \theta}$, teda ako veľmi by sa mala zmeniť každá váha siete pre zmenšenie straty – a `self.optimizer.step`, ktorý využíva gradienty z `loss.backward` na samotnú zmenu váh. Pomocou volania `.step` prechádza cez všetky parametre θ siete a upravuje ich hodnoty podľa vzorca, ktorý v zásade vyzerá ako:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial \text{loss}}{\partial \theta_i} \quad (10)$$

```
loss = F.mse_loss(curr_q, target_q)
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

Obrázok 40: Výpočet straty, spätná propagácia a aktualizácia váh

Pomocou `target_update_interval` je určené ako často sa majú presúvať váhy z hlavnej siete do tej cieľovej.

```
self.steps += 1
if self.steps % self.target_update_interval == 0:
    self.model_target.load_state_dict(self.model.state_dict())
```

Obrázok 41: Frekvencia presunu váh z hlavnej siete do cieľovej

Tým, že implementácia DQN využíva Prioritized Experience Replay (popísané v 5.1.5), je potrebné po každom kroku aktualizovať priority vzoriek podľa ich Temporal Difference error (TD-chyba) (popísané v 5.1.1.2), ktorá definuje, o koľko bola odhadovaná Q-hodnota v aktuálnom stave odlišná od lepšieho odhadu získaného aktualizáciou podľa Bellmanovej rovnice.

$$\delta = r + \gamma(1 - d) \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \quad (11)$$

Kde:

- δ = TD-chyba
- $r + \gamma(1 - d) \max_{a'} Q(s', a'; \theta^-)$ = pôvodná Bellmanova rovnica
- $Q(s, a; \theta)$ = aktuálny odhad

```

try:
    with torch.no_grad():
        # compute sample-wise TD errors
        td_errors = (target_q - curr_q).abs().cpu().numpy()
        # update buffer priorities
        self.memory.priorities[self.memory.last_indices] = td_errors + self.memory.epsilon_per
        # track max priority for new transitions
        self.memory.max_priority = max(self.memory.max_priority, td_errors.max())
except Exception:
    pass

```

Obrázok 42: Výpočet TD-chyby a úprava priorít vzoriek pre PER

Nakoniec dochádza k návratu hodnôt, kde je vrátený slovník s aktuálnou stratou a hodnotou epsilon pre umožnenie sledovania klesania miery preskúmania.

```

return dict(loss=loss.item(), epsilon=self.model.epsilon)

```

Obrázok 43: Výstupný slovník funkcie train()

```

def train(self, batch):
    o, a, r, o2, d, _ = batch
    # convert observation tuples to float32 torch tensors on our device to match network dtype
    o = tuple(torch.as_tensor(feats, dtype=torch.float32, device=self.device) for feats in o)
    o2 = tuple(torch.as_tensor(feats, dtype=torch.float32, device=self.device) for feats in o2)
    # map continuous actions back to discrete indices
    actions = []
    for act in a.cpu().numpy():
        key = (float(act[0]), float(act[1]), float(act[2]))
        actions.append(action_to_index[key])
    actions = torch.LongTensor(actions).to(self.device)
    # current Q
    q_vals = self.model.q_network(o)
    curr_q = q_vals.gather(1, actions.unsqueeze(1)).squeeze(1)
    # target Q
    with torch.no_grad():
        next_q_vals = self.model_target.q_network(o2)
        max_next_q = next_q_vals.max(1)[0]
        # build reward and done tensors on correct device/dtype
        rewards = torch.as_tensor(r, dtype=next_q_vals.dtype, device=self.device)
        dones = torch.as_tensor(d, dtype=next_q_vals.dtype, device=self.device)
        target_q = rewards + self.gamma * (1 - dones) * max_next_q
    loss = F.mse_loss(curr_q, target_q)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
    # update target network periodically
    self.steps += 1
    if self.steps % self.target_update_interval == 0:
        self.model_target.load_state_dict(self.model.state_dict())
    # Update PER priorities (if using PrioritizedMemory)
    try:
        with torch.no_grad():
            # compute sample-wise TD errors
            td_errors = (target_q - curr_q).abs().cpu().numpy()
            # update buffer priorities
            self.memory.priorities[self.memory.last_indices] = td_errors + self.memory.epsilon_per
            # track max priority for new transitions
            self.memory.max_priority = max(self.memory.max_priority, td_errors.max())
    except Exception:
        pass
    return dict(loss=loss.item(), epsilon=self.model.epsilon)

```

Obrázok 44: Celý kód funkcie train()

7.1.6 Triedy PrioritizedMemory a PERTrainingOffline

Pre zefektívnenie pamäte modelu bol implementovaný tzv. Prioritized Experience Replay (5.1.5), ktorý má schopnosť uprednostňovať skúsenosti s veľkou chybou. Premenná `alpha` udáva ako silno sú uprednostňované prechody s veľkou chybou, kde 0 = úplne náhodné vzorkovanie a 1 = úplne podľa chyby. Je jej pridelená hodnota 0.6. Premenná `epsilon_per` má veľmi malú hodnotu, jej účelom je to, aby bolo zaistené, že žiadna priorita neklesne na nulovú hodnotu a vždy existovala možnosť vzorkovať aj tie s malou chybou. Pomocou `np.zeros()` je vytvorené pole `priorities`, najprv naplnené nulami pre každý slot v buffer-i, určené pre ukladanie priorít. Pri pridaní nových prechodov je im priradená hodnota 1 – najvyššia priorita, aby mohli byť okamžite vzorkované.

```
class PrioritizedMemory(memory_base_cls):
    """Proportional Prioritized Experience Replay buffer."""
    def __init__(self, *args, alpha=0.6, **kwargs):
        super().__init__(*args, **kwargs)
        self.alpha = alpha
        self.epsilon_per = 1e-6
        # initialize priorities to zeros
        self.priorities = np.zeros(self.memory_size, dtype=np.float32)
        self.max_priority = 1.0
```

Obrázok 45: Inicializácia hodnôt v triede PrioritizedMemory

7.1.6.1 Funkcia `append_buffer()`

Funkcia `append_buffer` slúži na rozšírenie bufferu o nové prechody. Najprv zisťuje momentálnu veľkosť bufferu, potom koľko prechodov pribudlo, ďalej vypočítava indexy v poli priorít, ktoré zodpovedajú práve pridaným prechodom. Nakoniec novým prechodom nastaví najvyššiu prioritu rovnako pri inicializácii.

```
def append_buffer(self, buffer):
    old_len = len(self)                # before append
    super().append_buffer(buffer)
    new_count = len(self) - old_len    # how many you just added
    idxs = (np.arange(old_len, old_len + new_count) % self.memory_size)
    self.priorities[idxs] = self.max_priority
```

Obrázok 46: Funkcia `append_buffer`

7.1.6.2 Funkcia `sample_indices()`

Funkcia `sample_indices` pracuje s indexmi prechodov, kde najprv zisťuje aktuálny počet prechodov v bufferi, následne zvýrazní tie s vyššou prioritou tak, že ich umocní premennou `alpha`. Vypočíta sa kontrolný súčet, aby sa zistilo, či nie je suma rovná nule, a ak áno, tak sa nastaví rovnomerné pravdepodobnosti. Ak nie, dochádza k normalizácii tak, aby bol súčet 1. Nakoniec je náhodne vybraný `batch_size` indexov podľa pravdepodobností.

```
def sample_indices(self):
    # compute sampling probabilities over current buffer length
    n = len(self)
    probs = self.priorities[:n] ** self.alpha
    total = probs.sum()
    if total == 0:
        probs = np.ones_like(probs) / n
    else:
        probs /= total
    idxs = np.random.choice(n, self.batch_size, p=probs)
    return idxs
```

Obrázok 47: Funkcia `sample_indices`

7.1.6.3 Definícia iterátoru `__iter__`

Nakoniec je definovaný iterátor `__iter__`, ktorý vracia batch-e pre tréning na základe počtu tréningových krokov na „round“. Sú vybrané indexy pre mini-batch a na základe týchto indexov sú vybrané samotné prechody, z ktorých je potom vytvorený tensor pripravený pre tréning. Tieto indexy sú uložené a po tréningu sú nimi aktualizované váhy. Pomocou `yield` batch je batch daný trénerovi.

```
def __iter__(self):
    for _ in range(self.nb_steps):
        idxs = self.sample_indices()
        # use __getitem__ so info is stripped and we get exactly 6 elements per sample
        samples = [self[i] for i in idxs]
        batch = self.collate(samples, self.device)
        # store indices for priority updates
        self.last_indices = idxs
        yield batch
```

Obrázok 48: Iterátor na konci triedy `PrioritizedMemory`

Trieda `PERTrainingOffline` je rozšírenie offline trénera⁷. Umožňuje spravovanie PER. Po inicializácii pamäte a agenta z `TrainingOffline` je doplnená `agent.memory`. Do tejto pamäte sú vložené PER-buffer. Vďaka tomuto môže agent po každom tréningovom kroku aktualizovať priority podľa TD-chýb.

```
class PERTrainingOffline(TrainingOffline):
    """Trainer subclass to allow agent to update buffer priorities."""
    def __post_init__(self):
        super().__post_init__()
        # give agent access to the replay memory
        self.agent.memory = self.memory
```

Obrázok 49: Trieda `PERTrainingOffline`

7.1.7 Úprava súboru `config.json`

Tento súbor bol spomenutý v kapitole 4.2, a pre každý algoritmus, vrátane DQN, je potrebné v ňom vykonať určité zmeny. V základnej forme obsahuje hodnoty, ktoré ovládajú správanie framework-u TMRL ako takého – od nastavení konkrétneho tréningového behu, veľkosti pamäte, výberu zariadenia pre tréningovanie, nastavenie zaznamenávania informácií do databázy wandb, až po špecifické nastavenia a hyperparametre pre RL algoritmy.

7.1.7.1 Úprava obecných tréningových parametrov

Ako prvé je potrebné zmeniť niektoré nastavenia týkajúce sa tréningu ako takého. Zo základu je hodnota `MAX_EPOCHS` nastavená na obrovských 10,000. Táto hodnota bude zmenená na 10, aby proces tréningu netrval stovky hodín, ale len približne 4-5 hodín. Hodnota `ROUNDS_PER_EPOCH` udáva, koľko tréningových kôl sa má vykonať v každej epoche. Základné nastavenie hodnoty je 100, no ideálnejšia je hodnota 75.

"MAX_EPOCHS": 10000,
"ROUNDS_PER_EPOCH": 100, ➡ "MAX_EPOCHS": 10,
"ROUNDS_PER_EPOCH": 75,

Obrázok 50: Zmena hodnôt `MAX_EPOCHS` a `ROUNDS_PER_EPOCH`

Hodnoty parametrov `ENVIRONMENT_STEPS_BEFORE_TRAINING` a `RW_MAX_SAMPLES_PER_EPISODE` boli zmenené z 1,000 na 3,000, aby bolo nazbieraných viacej skúseností pred aktualizáciou.

⁷ Offline tréner - súčasť vzdelávacieho systému, ktorý aktualizuje model pomocou skupiniek predtým zhromaždených údajov o skúsenostiach namiesto učenia sa priamo z interakcií v reálnom čase

"RW_MAX_SAMPLES_PER_EPISODE": 1000, "RW_MAX_SAMPLES_PER_EPISODE": 3000,
 "ENVIRONMENT_STEPS_BEFORE_TRAINING": 1000, "ENVIRONMENT_STEPS_BEFORE_TRAINING": 3000,

Obrázok 51: Zmena hodnôt ENVIRONMENT_STEPS_BEFORE_TRAINING a RW_MAX_SAMPLES_PER_EPISODE

Dodatočne bol v tejto sekcii upravený ešte jeden parameter, a to UPDATE_MODEL_INTERVAL z hodnoty 200 na 150, aby sa Q-sieť aktualizovala častejšie počas každej epochy.

"UPDATE_MODEL_INTERVAL": 200, "UPDATE_MODEL_INTERVAL": 150,

Obrázok 52: Zmena hodnoty UPDATE_MODEL_INTERVAL

7.1.7.2 Úprava parametrov prostredia

Malé zmeny boli vykonané aj v sekcii prostredia, hlavnou z nich je zapnutie ukladania záznamov pokusov počas tréningu. Táto zmena bola vykonaná v nastaveniach RTGYM_CONFIG, ktoré sú vnorené v časti ENV. Konkrétny parameter, ktorý bol zmenený je pod interface_kwargs a volá sa save_replays. Jeho hodnota bola zmenená z „false“ na „true“.

```
"ENV": {
  "RTGYM_INTERFACE": "TM20IMAGES",
  "WINDOW_WIDTH": 256,
  "WINDOW_HEIGHT": 128,
  "IMG_WIDTH": 64,
  "IMG_HEIGHT": 64,
  "IMG_GRAYSCALE": true,
  "SLEEP_TIME_AT_RESET": 1.5,
  "IMG_HIST_LEN": 4,
  "RTGYM_CONFIG": {
    "time_step_duration": 0.05,
    "start_obs_capture": 0.04,
    "time_step_timeout_factor": 1.0,
    "act_buf_len": 2,
    "benchmark": true,
    "wait_on_done": true,
    "ep_max_length": 3000,
    "interface_kwargs": {"save_replays": true}
  }
},
```

Obrázok 53: Zapnutie zaznamenávania individuálnych pokusov počas tréningu

V sekcii RTGYM_CONFIG bola vykonaná dodatočne ešte jedna zmena. Parameter ep_max_length bol zmenený z hodnoty 1,000 na 3,000, aby boli dosiahnuté dlhšie epizódy tréningu, a nazbieralo sa tak viac skúseností.

"ep_max_length": 1000, "ep_max_length": 3000,

Obrázok 54: Zmena hodnoty ep_max_length

V časti, ktorá obsahuje parametre vzťahujúce sa na nastavenie odmeny bola vykonaná malá zmena v hodnote parametru `CONSTANT_PENALTY`. Tá bola nastavená z nuly na hodnotu `-0.01`, ktorá model penalizuje za každý krok, čo by ho malo tlačiť k tomu, aby sa snažil trať dokončiť čo najskôr.

`"CONSTANT_PENALTY": 0.0,` ➔ `"CONSTANT_PENALTY": -0.01,`

Obrázok 55: Pridanie malej penalizácie každého kroku

7.1.7.3 Úprava a pridanie hyperparametrov algoritmu

V sekcii `ALG` boli zmenené hyperparametre špecifické pre algoritmus DQN. Sú to premenné, ktoré boli popísané v 5.1:

- $\gamma = \text{GAMMA}$ ➔ diskontný faktor, ovládanie toho, ako moc je prioritizovaná budúca odmena
- Learning rate = `LR` ➔ úroveň/rýchlosť učenia, určuje, aký veľký krok je použitý vždy, keď sa menia odhady Q-hodnôt

Tieto hyperparametre už v určitej forme existovali pre algoritmus SAC, no hodnota `GAMMA` bola upravená z `0.995` na `0.999`, a namiesto troch hyperparametrov pre `LR` – `LR_ACTOR`, `LR_CRITIC`, `LR_ENTROPY` – bol pridaný jeden univerzálny parameter, ktorého hodnota bola nastavená na `0.0005`. Dodatočne bol pridaný nový parameter `TARGET_UPDATE_INTERVAL`, ktorý udáva, ako často je cieľová Q-sieť synchronizovaná s online Q-sieťou. Jeho hodnota bola nastavená na `1,500`.

```
"ALG": {
  "ALGORITHM": "DQN",
  "GAMMA": 0.999,
  "LR": 0.0005,
  "TARGET_UPDATE_INTERVAL": 1500,
```

Obrázok 56: Úprava a pridanie parametrov algoritmu

7.2 Implementácia SAC

Ako ďalší bude implementovaný algoritmus SAC, pretože je značne pokročilejší a nie je až tak veľmi závislý na perfektných definíciách hodnôt hyperparametrov. Je robustný a dokáže sa adaptovať zmenám v jeho prostredí. Pracuje so spojitým akčným priestorom, takže sú očakávané lepšie, plynulejšie výsledky po natrénovaní, na rozdiel od DQN, ktoré potrebovalo prevod spojitých akcií do diskkrétnej formy. Implementácia bola inšpirovaná pôvodnou verziou SAC v pomocnom súbore, ktorý je využívaný, no táto verzia je obohatená o viaceré vylepšenia, ktoré budú popísané v ich odpovedajúcich triedach.

7.2.1 Príprava konštánt pre využitie neskôr

Na začiatku programu, hneď po importovaní potrebných knižníc a súčastí (hlavne z knižnice PyTorch a priamo z framework-u TMRL), sú zadefinované určité konštanty – LOG_STD_MAX, LOG_STD_MIN, TARGET_ENTROPY_SCALE a INPUT_DIM spolu s HIDDEN_SIZES.

```
LOG_STD_MAX = 2
LOG_STD_MIN = -20
TARGET_ENTROPY_SCALE = 1.5

INPUT_DIM = 83
HIDDEN_SIZES = [256, 256]
```

Obrázok 57: Definícia užitočných konštánt

Prvé dve konštanty sa vzťahujú na vymedzenie hodnôt logaritmu smerodajnej odchýlky v stratégii tak. Maximálna hodnota zaisťuje to, aby sa odhad rozptylu náhodnej akcie príliš nerozbíhal, a minimálna hodnota zabraňuje odhadu rozptylu stať sa príliš malým, čo by mohlo spôsobiť numerickú nestabilitu.

Konštanta TARGET_ENTROPY_SCALE je faktor škálovania pre cieľovú entropiu, ktorá sa využíva pri automatickom doladovaní koeficientu entropie (α). Vyššia hodnota odpovedá väčšiemu dôrazu na prieskum.

Konštanta INPUT_DIM naberá hodnotu 83, pretože sa jedná o súčet 3 hodnôt: 76 – 4 posledné merania LIDAR $\times$ 19 lúčov LIDAR; 1 – stavová veličina (rýchlosť); 6 – dva posledné akčné vektory (2×3). Táto konštanta bude využívaná pri spracovaní LIDAR dát. Posledná konštanta HIDDEN_SIZES len udáva veľkosti skrytých vrstiev viacvrstvovej perceptronovej siete – 2 vrstvy, obe obsahujú 256 neurónov, takže hodnota vyzerá ako [256, 256].

7.2.2 Pomocná funkcia process_observation

Táto funkcia má za úlohu zvládnuť spracovať vstupnú pozorovaciu množinu `obs` do plochého vektora vhodného pre sieťové vrstvy.

```
def process_observation(obs):  
    state, lidar_data, act1, act2 = obs  
  
    if lidar_data.dim() == 2:  
        lidar_flat = lidar_data  
    else:  
        lidar_flat = lidar_data.unsqueeze(-1)  
  
    def ensure_2d(t):  
        return t if t.dim() >= 2 else t.unsqueeze(0)  
  
    parts = [state, lidar_flat, act1, act2]  
    parts = [ensure_2d(t) for t in parts]  
  
    x = torch.cat(parts, dim=-1)  
    return x
```

Obrázok 58: Kód funkcie process_observation

Bez úpravy sú dáta `obs` vo forme štvorice, takže ich je najprv potrebné rozdeliť na štyri premenné – `speed`, `lidar_data`, `act1` a `act2`. Prvá premenná uchováva údaje o rýchlosti, a v pôvodnej `obs` je na prvej pozícii. Druhá premenná uchováva dáta meraní LIDAR, tretia a štvrtá premenná obsahujú údaje o posledných dvoch vykonaných akciách. Je prítomná kontrolná podmienka `if`, ktorá zaisťuje, že dimenzia LIDAR dát je 2D. Ak nie sú 2D, je pridaná dodatočná dimenzia pomocou `unsqueeze(-1)`. Pomocná funkcia `ensure_2d` má za úlohu zabezpečiť, že všetky tensorové údaje sú aspoň dvojdimenzionálne, a ak nie, je im pridaná dodatočná dimenzia na začiatok. Z hodnôt je vytvorený list `parts`. Hneď na to je na všetky jeho súčasti aplikovaná pomocná funkcia `ensure_2d`. Tento list je nakoniec preformovaný do jednorozmerného vektora pomocou `torch.cat` pre umožnenie spracovania viacvrstvovým perceptronom.

7.2.3 Trieda MLPNetwork

Táto trieda implementuje samotný viacvrstvový perceptron. Dedí od `nn.Module`, čo je základná PyTorch trieda pre neurónové siete. Účelom tejto triedy je spracovať vstupné dáta – LIDAR merania, rýchlosť a predchádzajúce akcie – aby ich mohli využiť ako aktér pre výber akcií, tak aj kritici pre odhady hodnôt stavov a akcií.

```
class MLPNetwork(nn.Module):
    """
    MLP Network for processing lidar data
    """
    def __init__(self, input_dim, hidden_sizes, output_dim, activation=nn.ReLU, output_activation=nn.Identity):
        super().__init__()

        self.layers = []
        dims = [input_dim] + hidden_sizes + [output_dim]

        for i in range(len(dims)-1):
            self.layers.append(nn.Linear(dims[i], dims[i+1]))
            if i < len(dims) - 2:
                self.layers.append(activation())
            else:
                self.layers.append(output_activation())

        self.net = nn.Sequential(*self.layers)
```

Obrázok 59: Inicializácia triedy MLPNetwork

Na začiatku sú definované parametre perceptronu:

- `input_dim` → veľkosť vstupného vektora (83)
- `hidden_sizes` → zoznam veľkostí skrytých vrstiev ([256, 256])
- `output_dim` → veľkosť výstupného vektora
- `activation` → aktivačná funkcia pre skryté vrstvy (predvoľba je ReLU)
- `output_activation` → aktivačná funkcia pre výstupnú vrstvu (predvoľba je Identity)

Po určení parametrov dochádza k inicializácii siete. Následne je vytvorená prázdna premenná `layers` dátového typu list, ktorá bude využitá pre vytvorenie vrstiev perceptronu. Dodatočne je vytvorená ďalšia premenná typu list, nazvaná `dims`, ktorá všetky dimenzie všetkých vrstiev uloží do jedného listu, napr. `input_dim = 83`, `hidden_sizes=[256,256]`, `output_dim=3`; `dims = [83, 256, 256, 3]`.

Nasleduje vytvorenie vrstiev pomocou `for` cyklu, ktorý pre každú dvojicu susedných vrstiev:

- vytvorí lineárnu vrstvu `nn.Linear`,
- pridá aktivačnú funkciu
 - o ReLU pre skryté vrstvy
 - o Identity pre výstupnú vrstvu

```
for i in range(len(dims)-1):
    self.layers.append(nn.Linear(dims[i], dims[i+1]))
    if i < len(dims) - 2:
        self.layers.append(activation())
    else:
        self.layers.append(output_activation())
```

Obrázok 60: For cyklus pre tvorenie vrstiev

Nakoniec je vytvorená sekvencia, do ktorej sa všetky vrstvy spoja pomocou `nn.Sequential`, čo umožňuje automatické prepojenie vrstiev.

7.2.3.1 Funkcia *forward()*

Dodatočne je ešte zadaná funkcia `forward`, ktorá určuje, ako dáta prechádzajú sieťou. V tomto prípade je to len jednoduché `net(x)`, kde hodnota `x` postupne prechádza každou vrstvou siete.

```
def forward(self, x):
    return self.net(x)
```

Obrázok 61: Kód funkcie `forward()`

7.2.4 Trieda LidarSACPolicy

V tejto triede je implementovaný aktér algoritmu SAC. Dedí od `TorchActorModule`, ktorá slúži ako základná trieda pre aktérske moduly v TMRL. Jej parametrami sú `observation_space` – priestor pozorovaní, obsahuje informácie o stave, a `action_space` – priestor akcií, v Trackmanii to je zabáčanie, plyn a brzda. Po inicializovaní sú vytvorené dve premenné – `dim_act` a `act_limit`. Hodnota `dim_act` je 3, pretože v Trackmanii sú známe 3 akcie, ako je spomenuté vyššie. Hodnota `act_limit` naberá hodnotu maximálnej hodnoty pre akcie.

```
class LidarSACPolicy(TorchActorModule):
    """
    Actor module for SAC with Lidar inputs
    """
    def __init__(self, observation_space, action_space):
        super().__init__(observation_space, action_space)

        dim_act = action_space.shape[0] # 3 for Trackmania (steering, gas, brake)
        act_limit = action_space.high[0]
```

Obrázok 62: Inicializácia triedy LidarSACPolicy

Ďalším krokom je inicializácia hlavnej siete, ktorá bude spracovávať vstupy – `policy_backbone`, ktorej vstup je 83 dimenzií (rýchlosť, LIDAR, predošlé akcie), má [256, 256] skrytých vrstiev, a jej výstupom je 256 dimenzií. Ako ďalšia je vytvorená vrstva `mu_layer`, ktorá bude počítat' strednú hodnotu Gaussovej distribúcie μ , ako posledná je vytvorená vrstva `log_std_layer`, ktorá bude vypočítavať logaritmus smerodajnej odchýlky $\log \sigma$. Posledným krokom inicializácie je inicializácia váh na týchto dvoch vrstvách malými hodnotami, čo pomáha so stabilitou tréningu.

```
# Policy network
self.policy_backbone = MLPNetwork(INPUT_DIM, HIDDEN_SIZES, HIDDEN_SIZES[-1])
self.mu_layer = nn.Linear(HIDDEN_SIZES[-1], dim_act)
self.log_std_layer = nn.Linear(HIDDEN_SIZES[-1], dim_act)
self.act_limit = act_limit

# Initialize last layer weights with smaller values
# This helps with training stability
self.mu_layer.weight.data.uniform_(-3e-3, 3e-3)
self.mu_layer.bias.data.uniform_(-3e-3, 3e-3)
self.log_std_layer.weight.data.uniform_(-3e-3, 3e-3)
self.log_std_layer.bias.data.uniform_(-3e-3, 3e-3)
```

Obrázok 63: Inicializácia siete, dodatočných vrstiev a ich váh

7.2.4.1 Funkcia forward()

Nasleduje funkcia `forward`, ktorá prijíma dva parametre – `test`, a `compute_logprob`. Parameter `test` rozlišuje, či je sieť v testovacom alebo tréningovom režime. Ak je v testovacom režime, sieť vracia deterministické akcie, teda priamo hodnotu μ . Toto je ideálne, ak sa vykonáva zhodnotenie trénovaného agenta pri finálnom testovaní. Ak je sieť v tréningovom režime, sieť vracia stochastické akcie, čo je využívané počas procesu trénovania. Parameter `compute_logprob` je tiež určený pre rozlíšenie testovacieho alebo tréningového režimu. Výpočet logaritmu pravdepodobnosti je dôležitý len pri trénovaní, pretože počíta entropiu stratégie, aktualizuje parameter α a využíva sa pri výpočte straty aktéra.

Na začiatku funkcie `forward` je volaná funkcia `process_observation` na vstupné dáta `obs`, a jej výstup je uložený do premennej `x`. Tento výstup následne prechádza hlavnou sieťou `policy_backbone`. Výstup hlavnej siete je využitý pre výpočet parametrov distribúcie – `mu`, `log_std` a `std`, pomocou vrstiev zadaných v predchádzajúcej časti – `mu_layer` a `log_std_layer`. Výstup `log_std` je pred výpočtom obyčajného `std` ešte obmedzený konštantami `LOG_STD_MIN` a `LOG_STD_MAX`.

```
def forward(self, obs, test=False, compute_logprob=True):
    """Forward pass for policy"""
    # Process inputs
    x = process_observation(obs)

    # Backbone network
    net_out = self.policy_backbone(x)

    # Action head
    mu = self.mu_layer(net_out)
    log_std = self.log_std_layer(net_out)
    log_std = torch.clamp(log_std, LOG_STD_MIN, LOG_STD_MAX)
    std = torch.exp(log_std)
```

Obrázok 64: Začiatok funkcie `forward` v `LidarSACPolicy`

Ďalším krokom je vzorkovanie akcií z normálneho rozdelenia `pi_distribution`. Ako bolo spomenuté, ak je parameter `test` nastavený na hodnotu `true`, vyberajú sa deterministické akcie, no ak je nastavený na `false`, vykonávajú sa stochastické akcie pomocou reparametrizačného triku⁸.

Ak je parameter `compute_logprob` nastavený na `true`, je vypočítaná hodnota `logp_pi`, teda logaritmus pravdepodobnosti.

Hodnota `pi_action` je normalizovaná tak, aby bola v rozsahu `[-1, 1]`, a následne je škálovaná na rozsah akcií. Výstupné premenné tejto funkcie sú `pi_action` a `logp_pi`.

```
# Sample from normal distribution
pi_distribution = Normal(mu, std)

if test:
    # Deterministic at test time
    pi_action = mu
else:
    # Reparameterization trick for backprop
    pi_action = pi_distribution.rsample()

# Compute log probability if needed
if compute_logprob:
    logp_pi = pi_distribution.log_prob(pi_action).sum(axis=-1)
    # Correction for tanh squashing
    logp_pi -= (2 * (np.log(2) - pi_action - F.softplus(-2 * pi_action))).sum(axis=1)
else:
    logp_pi = None

# Apply tanh squashing to ensure actions in [-1, 1]
pi_action = torch.tanh(pi_action)
pi_action = self.act_limit * pi_action

# Remove batch dimension if necessary
pi_action = pi_action.squeeze()

return pi_action, logp_pi
```

Obrázok 65: Vzorkovanie akcií vo funkcii forward

7.2.4.2 Funkcia `act()`

Funkcia `act` vypočíta akcie pre dané vstupné dáta `obs` pomocou funkcie `forward`, a následne ich vykonáva. Návratová hodnota je pole `numpy` naplnené akciami.

```
def act(self, obs, test=False):
    """
    Compute action from observation (implementation required by TMRL)
    """
    with torch.no_grad():
        a, _ = self.forward(obs=obs, test=test, compute_logprob=False)
        return a.cpu().numpy()
```

Obrázok 66: Výpočet akcií z pozorovania

⁸ Reparametrizačný trik je metóda, ktorá umožňuje namiesto vzorkovania náhodnej veličiny z normálneho rozdelenia s daným priemerom a rozptylom separovať náhodnosť ako samostatný vstup, ktorý nie je treba optimalizovať [48]

7.2.5 Trieda LidarQFunction

Táto trieda implementuje Q-funkciu, ktorej úlohou je odhadovať hodnoty stavu a akcie. Pri inicializácii sú vytvorené tri časti siete:

- `q_input_net` → spracováva LIDAR pozorovania, vytvára reprezentáciu stavu; vstup 83 dimenzií, výstup 256 dimenzií
- `q_action_net` → spracováva akcie, transformuje akcie na rovnakú dimenziu ako stav; vstup 3 dimenzie, výstup 256 dimenzií
- `q_output_net` → spracováva kombinovanú reprezentáciu stavu a akcie, vypočítava finálnu Q-hodnotu; výstup 1 dimenzia – 1 Q-hodnota

Podobne ako pri `LidarSACPolicy`, váhy poslednej lineárnej vrstvy `q_output_net` sú inicializované s drobnými hodnotami.

```
class LidarQFunction(nn.Module):
    """Q-function network for lidar inputs"""
    def __init__(self, observation_space, action_space):
        super().__init__()

        action_dim = action_space.shape[0]

        # Q networks - we'll use a different architecture with action added later
        self.q_input_net = MLPNetwork(INPUT_DIM, [HIDDEN_SIZES[0]], HIDDEN_SIZES[0])
        self.q_action_net = nn.Linear(action_dim, HIDDEN_SIZES[0])
        self.q_output_net = MLPNetwork(HIDDEN_SIZES[0], [HIDDEN_SIZES[1]], 1)

        # Initialize last layer weights with smaller values
        last_layer = self.q_output_net.layers[-2] # Get the last linear layer
        last_layer.weight.data.uniform_(-3e-3, 3e-3)
        last_layer.bias.data.uniform_(-3e-3, 3e-3)
```

Obrázok 67: Inicializácia Q-funkcie

7.2.5.1 Funkcia forward()

Funkcia `forward` je jednoduchá – najprv sú spracované vstupy pomocou `process_observation`, potom je spracovaný stav pomocou `q_input_net` a akcie pomocou `q_action_net`. Následne sú stav a akcie skombinované súčtom do premennej `combined`, na ktorej je aplikovaná ReLU aktivačná funkcia. Kombinovaná reprezentácia je spracovaná `q_output_net`, ktorej výsledkom je jedna Q-hodnota. Na konci je odstránená posledná dimenzia pomocou `squeeze`.

```
def forward(self, obs, act):
    """Forward pass for Q-function"""
    # Process observation
    x = process_observation(obs)

    # Process state and action separately and combine
    obs_features = self.q_input_net(x)
    act_features = self.q_action_net(act)

    # Combine with ReLU activation
    combined = F.relu(obs_features + act_features)

    # Final output layer
    q_value = self.q_output_net(combined)

    return torch.squeeze(q_value, -1)
```

Obrázok 68: Funkcia forward v triede LidarQFunction

7.2.6 Trieda LidarSACActorCritic

Táto trieda je veľmi stručná – implementuje aktéra a dvoch kritikov. Dedí od `nn.Module`, rovnako ako `LidarQFunction` a `MLPNetwork`. Sú volané triedy `LidarSACPolicy` pre aktéra a `LidarQFunction` dvakrát, pre dvoch kritikov. Ako argumenty akceptujú `observation_space` a `action_space`, teda pozorovací a akčný priestor.

```
class LidarSACActorCritic(nn.Module):
    """
    Actor-critic module for the SAC algorithm with lidar inputs
    """
    def __init__(self, observation_space, action_space):
        super().__init__()

        # Policy network (actor)
        self.actor = LidarSACPolicy(observation_space, action_space)

        # Value networks (critics)
        self.q1 = LidarQFunction(observation_space, action_space)
        self.q2 = LidarQFunction(observation_space, action_space)
```

Obrázok 69: Kód triedy LidarSACActorCritic

7.2.7 Trieda ImprovedSACAgent

Trieda `ImprovedSACAgent` obsahuje „mozog“ celej implementácie, pretože v nej dochádza k tréningu ako takému. Volá inicializáciu rodičovskej triedy `TrainingAgent`. Na jej začiatku sú inicializované všetky základné parametre algoritmu SAC, ako aj parametre týkajúce sa jeho vylepšení – automatické nastavovanie entropie, obmedzenia gradientov, škála odmeny, a iné. Okrem parametrov je inicializovaný aj hlavný model aktér-kritik a je presunutý na správne zariadenie (v tomto prípade GPU). Zároveň je vytvorený cieľový model – target network – pre stabilnejšie učenie.

```

class ImprovedSACAgent(TrainingAgent):
    """
    Improved SAC Training Agent with additional enhancements:
    - Automatic entropy tuning
    - Gradient clipping
    - Learning rate scheduling
    - Weighted Bellman backup for stability
    """

    # no-grad copy of the model for actor retrieval
    model_nograd = cached_property(lambda self: no_grad(copy_shared(self.model)))

    def __init__(self,
                  observation_space=None,
                  action_space=None,
                  device=None,
                  model_cls=LidarSACActorCritic,
                  gamma=0.99,
                  polyak=0.995,
                  alpha=0.2,
                  lr_actor=1e-3,
                  lr_critic=1e-3,
                  lr_alpha=1e-4,
                  auto_alpha=True,
                  grad_clip=None,
                  reward_scale=1.0,
                  buffer_warmup=1000,
                  weight_decay=0.0):

        # Required arguments for superclass
        super().__init__(observation_space=observation_space,
                         action_space=action_space,
                         device=device)

        # Initialize model
        model = model_cls(observation_space, action_space)
        self.model = model.to(self.device)
        self.model_target = no_grad(deepcopy(self.model))

        # Hyperparameters
        self.gamma = gamma
        self.polyak = polyak
        self.reward_scale = reward_scale
        self.buffer_warmup = buffer_warmup
        self.grad_clip = grad_clip
        self.training_steps = 0

```

Obrázok 70: Inicializácia triedy ImprovedSACAgent

Po inicializácii hyperparametrov je implementovaná logika automatického nastavovania entropie `auto_alpha`. Najprv je vypočítaná cieľová entropia ako `-dimenzie(akcií) * TARGET_ENTROPY_SCALE`. Parameter α sa ukladá v tvare logaritmu pre lepšiu numerickú stabilitu. Premenná `alpha_t` ukladá aktuálnu hodnotu α formou `exp(log_alpha)`. Nakoniec je vytvorený Adam optimalizátor využívaný pre aktualizáciu hodnoty α . Výsledkom je udržiavanie správnej miery prieskumu, adaptívne prispôsobenie entropie počas procesu tréningu a vyhýba sa viacerým problémom spojeným s fixnou hodnotou α . Ak je parameter `auto_alpha` nastavený na hodnotu `False`, algoritmus pracuje s fixnou hodnotou α .

```

# Automatic entropy tuning
self.auto_alpha = auto_alpha
if auto_alpha:
    self.target_entropy = -np.prod(action_space.shape) * TARGET_ENTROPY_SCALE
    self.log_alpha = nn.Parameter(torch.zeros(1, device=self.device))
    self.alpha_t = torch.exp(self.log_alpha).detach()
    self.alpha_optimizer = Adam([self.log_alpha], lr=lr_alpha)
else:
    self.alpha_t = torch.tensor(float(alpha)).to(self.device)

```

Obrázok 71: Implementovanie automatického nastavenia entropie

Ako ďalšie sú vytvorené Adam optimalizátory pre aktéra aj kritikov. Najprv je však potrebné získať parametre oboch kritikov a zreťaziť ich pomocou `itertools.chain`. Výsledkom je stabilnejšie trénovanie a lepšia konvergencia.

```

# Optimizers
self.q_params = itertools.chain(self.model.q1.parameters(), self.model.q2.parameters())
self.pi_optimizer = Adam(self.model.actor.parameters(), lr=lr_actor, weight_decay=weight_decay)
self.q_optimizer = Adam(self.q_params, lr=lr_critic, weight_decay=weight_decay)

```

Obrázok 72: Vytvorenie optimalizátorov pre aktéra a kritikov

Poslednou časťou inicializácie je plánovač pre hodnoty rýchlosti učenia (LR). Jeden plánovač je vytvorený pre rýchlosť učenia aktéra a jeden pre rýchlosť učenia kritikov. Ich funkciou je znížiť rýchlosť učenia, keď prestane dochádzať k znižovaniu straty. Minimálna hodnota LR je nastavená na 0.00001 (1e-5). Toto správanie pomáha vyhnúť sa lokálnym minimám.

```

# Learning rate scheduler
self.pi_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    self.pi_optimizer, mode='min', factor=0.5, patience=6000,
    threshold=0.01, min_lr=1e-5
)
self.q_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    self.q_optimizer, mode='min', factor=0.5, patience=6000,
    threshold=0.01, min_lr=1e-5
)

```

Obrázok 73: Implementácia plánovačov rýchlostí učenia

7.2.7.1 Funkcia `get_actor()`

Funkcia `get_actor` len vracia kópiu aktuálnej verzie aktéra bez gradientov, využívanú pre zhodnotenie stratégie a vykonávanie akcií v prostredí.

```

def get_actor(self):
    """
    Returns a copy of the current ActorModule.
    """
    return self.model_nograd.actor

```

Obrázok 74: Funkcia `get_actor` v ImprovedSACAgent

7.2.7.2 Funkcia *train()*

Vo funkcii `train` je najprv potrebné zvýšiť hodnotu počítadla tréningového kroku o 1, čo sa zopakuje každým volaním tejto funkcie. Prvým krokom je rozbalenie batch-u na viaceré premenné:

- `obs` → obsahuje dáta o pozorovaní
- `a` → obsahuje dáta o akcií
- `r` → obsahuje dáta o odmene
- `o2` → obsahuje dáta o ďalších pozorovaniach (rovnaká štruktúra ako `obs`)
- `d` → done flag, určuje, či je epizóda ukončená

Premenná `obs` a `a` sú hneď potom rozdelené na časti pre úpravu odmeny. Sú z nich extrahované informácie o rýchlosti, plyne a brzde – `forward_speed`, `throttle`, `brake`.

```
def train(self, batch):
    """
    Executes a training iteration from batched training samples
    """
    self.training_steps += 1

    # Unpack the batch
    obs, a, r, o2, d, _ = batch
    speed_batch, _, _, _ = obs
    forward_speed = speed_batch[:, 0]
    throttle      = a[:, 1]
    brake         = a[:, 2]
```

Obrázok 75: Začiatok funkcie `train`

Po extrakcii týchto informácií dochádza k úprave odmeny, kde je odmenená vysoká rýchlosť faktorom 1.5, penalizované brzdenie faktorom 0.5 ako aj využívanie spiatočky faktorom 0.35. Hraničné hodnoty pre rýchlosť sú nastavené na 0.1, pre brzdenie a spiatočku na 0. Konečná odmena je získaná vzorcom:

$\text{odmena rýchlosti} - \text{penalta za brzdenie} - \text{penalta za spiatočku}$

Výsledok je uložený do premennej `r_shaped`.

```
speed_bonus      = 1.5 * forward_speed.clamp(min=0.1)
brake_penalty    = 0.5 * brake.clamp(min=0)
reverse_penalty  = 0.35 * (-throttle).clamp(min=0)

r_shaped = (speed_bonus - brake_penalty - reverse_penalty)
```

Obrázok 76: Úprava odmeny

Následne je skontrolovaný stav `buffer_warmup`-u, ktorý odštartuje proces tréovania až po získaní nejakých prvých skúseností.

```
# Only start training after collecting some initial experience
if self.training_steps < self.buffer_warmup:
    return {"skipping_training": 1.0}
```

Obrázok 77: Kontrola hodnoty `buffer_warmup`

Ďalší krok je vzorkovanie akcií, výpočet logaritmov pravdepodobnosti a výpočet Q-hodnôt. Podľa stratégie sú generované nové akcie, a kritici počítajú Q-hodnoty pre aktuálne páry stav-akcia.

Ako ďalšie sú vypočítané cieľové Q-hodnoty pomocou cieľovej siete. Najprv sú vzorkované budúce akcie `a2` a `logp_a2`, potom sú na nich volané cieľové siete `q1` a `q2`, ktoré vypočítajú cieľové Q-hodnoty. Z týchto Q-hodnôt je vybraté minimum, a nakoniec je vypočítaná hodnota `backup`, čo je vlastne časť Bellmanovej rovnice, a reprezentuje cieľovú hodnotu.

```
# Sample actions and compute log probabilities
pi, logp_pi = self.model.actor(obs=obs, test=False, compute_logprob=True)

# Compute Q-values
q1 = self.model.q1(obs, a)
q2 = self.model.q2(obs, a)

# Compute target Q-values
with torch.no_grad():
    # Sample next actions
    a2, logp_a2 = self.model.actor(o2)

    # Target Q-values from both networks
    q1_pi_targ = self.model_target.q1(o2, a2)
    q2_pi_targ = self.model_target.q2(o2, a2)

    # Use minimum of both Q-values (double Q-learning)
    q_pi_targ = torch.min(q1_pi_targ, q2_pi_targ)

    # Compute target with entropy term
    backup = r + self.gamma * (1 - d) * (q_pi_targ - self.alpha_t * logp_a2)
```

Obrázok 78: Vzorkovanie a výpočet cieľových Q-hodnôt

Ďalej dochádza k výpočtu straty kritikov pomocou metódy smooth L1 loss (tiež známa ako Huberova strata, je robustnejšia než metóda strednej kvadratickej chyby), aktualizácii kritikov a k obmedzeniu gradientov. Najprv sú vynulované gradienty pred novým výpočtom, následne je vypočítaný gradient straty a potom sú obmedzené veľkosti gradientov na hodnotu `grad_clip` (ak je nenulová). Pomocou `q_optimizer.step()` sú aktualizované parametre pomocou optimalizátoru. Na základe straty je potom aktualizovaná hodnota rýchlosti učenia kritikov pomocou plánovača.

```

# Critic loss
loss_q1 = F.smooth_l1_loss(q1, backup)
loss_q2 = F.smooth_l1_loss(q2, backup)
loss_q = loss_q1 + loss_q2

# Update critics
self.q_optimizer.zero_grad()
loss_q.backward()
if self.grad_clip is not None:
    torch.nn.utils.clip_grad_norm_(self.q_params, self.grad_clip)
self.q_optimizer.step()

# Update learning rate based on loss
self.q_scheduler.step(loss_q)

```

Obrázok 79: Výpočet straty a aktualizácia kritikov

Dôležitou časťou tréningového procesu je aktualizácia aktéra a hodnoty entropie, no pre udržanie stability je ideálne to spraviť len počas každého druhého tréningového kroku. Taktiež je potrebné dočasne „zmraziť“ Q-siete, aby nedochádzalo ku konfliktom gradientov medzi kritikmi a aktérmi a k problému s pohybujúcim sa cieľom (moving target problem). Proces aktualizácie aktéra je veľmi podobný s aktualizáciou kritikov, takže je najprv vypočítaná Q-hodnota oboch kritikov, vybratá tá menšia z nich, a na základe nej je spolu s hodnotou entropie a Q-hodnoty vypočítaná strata aktéra. Aktér sa učí maximalizovať Q-hodnoty a entropiu. Samotná aktualizácia prebieha rovnakými krokmi ako u kritikov – vynulovanie gradientov, spätná propagácia, obmedzenie gradientov a volanie optimalizátoru. Tiež je aktualizovaná hodnota rýchlosti učenia pre aktéra, podľa potreby.

```

# Actor and entropy updates only every other step for stability
if self.training_steps % 2 == 0:
    # Temporarily freeze Q-networks to avoid computing gradients for them
    for p in self.q_params:
        p.requires_grad = False

    # Actor loss
    q1_pi = self.model.q1(obs, pi)
    q2_pi = self.model.q2(obs, pi)
    q_pi = torch.min(q1_pi, q2_pi)
    loss_pi = (self.alpha_t * logp_pi - q_pi).mean()

    # Update actor
    self.pi_optimizer.zero_grad()
    loss_pi.backward()
    if self.grad_clip is not None:
        torch.nn.utils.clip_grad_norm_(self.model.actor.parameters(), self.grad_clip)
    self.pi_optimizer.step()

    # Update actor learning rate based on loss
    self.pi_scheduler.step(loss_pi)

    # Unfreeze Q-networks
    for p in self.q_params:
        p.requires_grad = True

```

Obrázok 80: Zmrazenie Q-sietí a aktualizovanie aktéra

Po vykonaní týchto úkonov môžu byť Q-siete opäť odmrzené a pripravené na ďalšiu aktualizáciu kritikov.

Po aktualizácii aktéra je čas na automatické nastavenie entropie. Najprv je vypočítaná hodnota α a jej strata. Proces aktualizácie hodnoty α je opäť identický s procesom aktualizácie aktéra a kritikov - vynulovanie gradientov, spätná propagácia, obmedzenie gradientov a volanie optimalizátoru. Aktualizovaná hodnota α je uložená do premennej `alpha_t`, ktorá je potom využitá pri výpočte straty aktéra a kritika. Podmienka `else` káže, aby bola premenná `loss_pi` (strata aktéra) naplnená nulovými hodnotami keď nedochádza k aktualizácii.

```
# Automatic entropy tuning
if self.auto_alpha:
    # Alpha loss
    alpha = self.log_alpha.exp()
    alpha_loss = -(self.log_alpha * (logp_pi + self.target_entropy).detach()).mean()
    # Update alpha
    self.alpha_optimizer.zero_grad()
    alpha_loss.backward()
    self.alpha_optimizer.step()
    self.alpha_t = alpha.detach()
else:
    # Fill with placeholder values when not updating
    loss_pi = torch.tensor(0.0)
```

Obrázok 81: Automatické nastavenie entropie

Nakoniec je aktualizovaná cieľová sieť pomocou Polyakovho priemerovania. Výstupom funkcie sú užitočné štatistiky o procese tréningu, ktoré budú pomocou wandb spracované do grafov. Tieto hodnoty sú zobrazované po každom tréningovom kole v príkazovom riadku.

```
# Update target network with polyak averaging
with torch.no_grad():
    for p, p_targ in zip(self.model.parameters(), self.model_target.parameters()):
        p_targ.data.mul_(self.polyak)
        p_targ.data.add_((1 - self.polyak) * p.data)

# Return training metrics for logging
ret_dict = {
    'loss_actor': loss_pi.detach().item(),
    'loss_critic': loss_q.detach().item(),
    'q1_mean': q1.mean().detach().item(),
    'q2_mean': q2.mean().detach().item(),
    'alpha': self.alpha_t.item(),
    'actor_entropy': -logp_pi.mean().detach().item(),
    'pi_lr': self.pi_optimizer.param_groups[0]['lr'],
    'q_lr': self.q_optimizer.param_groups[0]['lr'],
}

return ret_dict
```

Obrázok 82: Aktualizácia cieľovej siete a výstupné metriky

Parametre algoritmu SAC sú nastavené pomocou `training_agent_cls`. Najprv sú mu dané informácie, ktoré triedy má využívať. Hodnoty parametrov využité pre tréning sú:

- `gamma` → **0.99** – diskontný faktor
- `polyak` → **0.995** – hodnota Polyakovho priemerovania
- `auto_alpha` → **True** – zapnutie automatického nastavenia hodnoty α
- `lr_actor` → **3e-4** - rýchlosť učenia pre aktéra (0.0003)
- `lr_critic` → **3e-4** - rýchlosť učenia pre kritikov (0.0003)
- `lr_alpha` → **3e-4** – rýchlosť učenia pre hodnotu α (0.0003)
- `grad_clip` → **0.5** – obmedzenie gradientov
- `reward_scale` → **0.1** – škála odmeny
- `buffer_warmup` → **1000** – začať tréning po získaní 1000 vzorkov
- `weight_decay` → **1e-4** – malá L2 regularizácia (0.0001)

```
training_agent_cls = partial(
    ImprovedSACAgent,
    model_cls=LidarSACActorCritic,
    gamma=0.99,
    polyak=0.995,
    auto_alpha=True, # Automatic entropy adjustment
    lr_actor=3e-4,
    lr_critic=3e-4,
    lr_alpha=3e-4,
    grad_clip=0.5, # Gradient clipping for stability
    reward_scale=0.1, # Scale rewards for better training dynamics
    buffer_warmup=1000, # Start training after collecting 1000 samples
    weight_decay=1e-4 # Small L2 regularization
)
```

Obrázok 83: Nastavenie parametrov tréningového agenta

7.2.8 Zapnutie zberu dát pomocou wandb

Pre spracovanie metrík do formy grafov na stránke wandb je potrebné spustiť uchovávanie dát z trénera. Po inicializácii trénera je pri jeho spustení vo funkcii `__main__` potreba zavolať funkciu `run_with_wandb`, ktorá je súčasťou framework-u TMRL. Ako argumenty potrebuje `wandb_entity` – užívateľské meno, `wandb_project` – názov projektu, do ktorého dáta ukladať, a `wandb_run_id` – unikátny identifikátor, ktorý je vygenerovaný automaticky. Dodatočne je nastavená vlajka `x_disable_stats`, ktorá vypína zber systémových metrík (napríklad takty grafickej karty), aby boli obmedzené faktory spomaľujúce tréningový proces.

```

if args.trainer:
    my_trainer = Trainer(training_cls=training_cls,
                        server_ip=server_ip_for_trainer,
                        server_port=server_port,
                        password=password,
                        security=security)
    custom_wandb_settings = wandb.Settings(x_disable_stats=True)
    wandb.setup(settings=custom_wandb_settings)
    my_trainer.run_with_wandb(entity=wandb_entity, project=wandb_project, run_id=wandb_run_id)

```

Obrázok 84: Spustenie zberu dát do wandb

7.2.9 Úprava súboru config.json

Mnoho parametrov zostalo rovnakých ako pri algoritme DQN, no došlo k viacerým zmenám, a to hlavne v sekcii algoritmických parametrov. Niektoré hodnoty boli však zmenené aj v obecných nastaveniach.

7.2.9.1 Úprava obecných parametrov

Ako prvé je dôležité zmeniť parameter RUN_NAME na ľubovoľný názov. Pre tento algoritmus bol po viacerých iteráciách nastavený názov „SAC_TEST_RUN-13-LOGGING-DEBUG“. Hodnota RW_MAX_SAMPLES_PER_EPISODE zostáva konzistentná s algoritmom DQN – 3,000. Ostatné hodnoty až po parametre MAX_EPOCHS a ROUNDS_PER_EPOCH sú totožné s pôvodnými hodnotami. Tieto dve hodnoty boli nastavené na 20 a 55, pretože SAC potrebuje značne viac času na efektívne učenie.

"MAX_EPOCHS": 10000, ➡ "MAX_EPOCHS": 20,
 "ROUNDS_PER_EPOCH": 100, ➡ "ROUNDS_PER_EPOCH": 55,

Obrázok 85: Nastavenie hodnôt MAX_EPOCHS a ROUNDS_PER_EPOCH

Taktiež bol zmenený parameter TRAINING_STEPS_PER_ROUND z pôvodných 200 na 1,500, aby bolo algoritmu umožnené nazbierať viac skúseností počas každého tréningového kola. Pre porovnanie, priemerná dĺžka kola pri algoritme DQN bola približne 10 sekúnd, zatiaľ čo pri algoritme SAC to bolo až 40 sekúnd.

"TRAINING_STEPS_PER_ROUND": 200, ➡ "TRAINING_STEPS_PER_ROUND": 1500,

Obrázok 86: Nastavenie hodnoty TRAINING_STEPS_PER_ROUND

Parameter `ENVIRONMENT_STEPS_BEFORE_TRAINING` bol o niečo znížený z 1,000 na 800, aby proces tréningu začal trochu skôr. Parameter `UPDATE_MODEL_INTERVAL` zostáva konzistentný s algoritmom DQN na hodnote 150 pre častejšie aktualizovanie modelu.

`"ENVIRONMENT_STEPS_BEFORE_TRAINING": 1000,` ➡ `"ENVIRONMENT_STEPS_BEFORE_TRAINING": 800,`

Obrázok 87: Nastavenie hodnoty `ENVIRONMENT_STEPS_BEFORE_TRAINING`

Posledným zmeneným obecným parametrom, je parameter `MEMORY_SIZE`, ktorý bol nastavený z pôvodných 1,000,000 na 700,000 po viacerých iteráciách sledovania. Po dokončení tréningu dosiahla hodnota tohto parametru výšku 670,000, takže rezervovať viacero miest bolo zbytočné.

`"MEMORY_SIZE": 1000000,` ➡ `"MEMORY_SIZE": 700000,`

Obrázok 88: Nastavenie hodnoty `MEMORY_SIZE`

7.2.9.2 Nastavenie parametrov pre algoritmus

V pôvodnej forme súboru `config.json` sú už pripravené parametre pre algoritmus SAC, no verzia algoritmu, ktorá bola v tejto práci implementovaná, ich má o niečo viac, takže ich je potrebné pridať. Prvé parametre, ktoré sa zhodujú v oboch verziách, sú `GAMMA`, `POLYAK`, `LR_ACTOR`, `LR_CRITIC`, `LR_ENTROPY`, `ALPHA`, `LEARN_ENTROPY_COEF` a `TARGET_ENTROPY`. Toto sú základné parametre, ovládajúce rýchlosť učenia, diskontný faktor, hodnotu pre Polyakove priemerovanie a ovládanie automatického nastavovania hodnoty entropie. Hodnota `GAMMA` a `POLYAK` sú z pôvodných 0.995 nastavené na 0.99.

`"GAMMA": 0.995,`
`"POLYAK": 0.995,` ➡ `"GAMMA": 0.99,`
`"POLYAK": 0.99,`

Obrázok 89: Úprava hodnôt `GAMMA` a `POLYAK`

Parametre pre rýchlosti učenia aktéra, kritika a entropie/alpha boli zvýšené z pôvodných 0.00001 a 0.00005 a 0.0003 na 0.0003 pre všetky tri.

`"LR_ACTOR": 0.00001,`
`"LR_CRITIC": 0.00005,`
`"LR_ENTROPY": 0.0003,` ➡ `"LR_ACTOR": 0.0003,`
`"LR_CRITIC": 0.0003,`
`"LR_ALPHA": 0.0003,`

Obrázok 90: Úprava hodnôt rýchlostí učenia

Parametre `LEARN_ENTROPY_COEF` a `TARGET_ENTROPY` majú na starosti automatické nastavovanie hodnoty entropie. V základnom stave je prvý parameter nastavený na hodnotu `false`, takže je potrebné ho zmeniť na `true`. Druhý parameter má pôvodne udanú fixnú hodnotu, pretože automatické nastavovanie je vypnuté, preto je hodnota zmenená z fixnej hodnoty `-0.5` na `auto`. Dodatočne je pridaný nový parameter – `TARGET_ENTROPY_SCALE`, ktorý udáva, ako moc entropie by sa mal agent snažiť dosiahnuť pri automatickom nastavovaní entropie. V skratke núti agenta udržiavať väčšiu úroveň prieskumu (exploration). Je nastavený na hodnotu `1.5`.

```
"LEARN_ENTROPY_COEF": false,
"TARGET_ENTROPY": -0.5,
"LEARN_ENTROPY_COEF": true,
"TARGET_ENTROPY": "auto",
"TARGET_ENTROPY_SCALE": 1.5,
```

Obrázok 91: Nastavenie hodnôt automatického ovládania entropie

Hodnota parametru `ALPHA` je nerelevantná, keďže automatické nastavovanie entropie je v tejto implementácii zapnuté. No ak by bolo vypnuté, tento parameter by mal hodnotu `0.2` na rozdiel od pôvodných `0.01`.

Novými parametrami sú `WEIGHT_DECAY`, `GRAD_CLIP`, `REWARD_SCALE` a `BUFFER_WARMUP`. Parameter `WEIGHT_DECAY` má za úlohu aplikovať L2 regularizáciu počas optimalizačného kroku a snaží sa tak zabrániť problému overfitting-u a povzbudzuje nižšie hodnoty váh neurónových sietí. Parameter `GRAD_CLIP` má na starosti obmedziť veľkosti gradientov a parameter `REWARD_SCALE` je použitý pre škálovanie upravených odmien. Parameter `BUFFER_WARMUP` udáva, koľko vzoriek má byť zozbieraných pred začatím tréningového procesu.

```
"WEIGHT_DECAY": 0.0001,
"GRAD_CLIP": 0.5,
"REWARD_SCALE": 0.1,
"BUFFER_WARMUP": 1000,
```

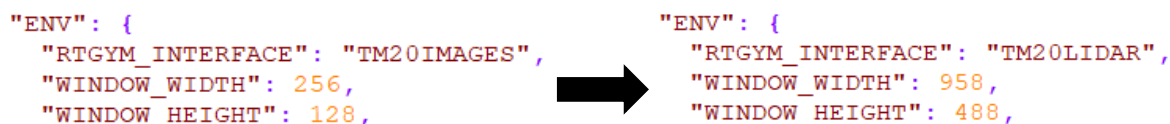
Obrázok 92: Pridanie vlastných parametrov

7.2.9.3 Úprava parametrov prostredia

V parametroch prostredia je vykonaná závažná zmena na rozdiel od algoritmu DQN. Keďže algoritmus SAC je značne komplexnejší, jeho tréning by mohlo zabráť aj niekoľko dní pri zložitejšom prostredí. Preto bolo zmenené prostredie z klasických celých snímok hry na prostredie LIDAR meraní. Presnosť konečného modelu bude o niečo horšia, no rýchlosť tréningu sa zvýši niekoľkonásobne. Preto sú prvé tri parametre v tejto sekcii zmenené na základe inštrukcií v repozitári TMRL nasledovne:

- RTGYM_INTERFACE → z TM20IMAGES na TM20LIDAR
- WINDOW_WIDTH → z 256 na 958
- WINDOW_HEIGHT → z 128 na 488

Týmto je zmenené prostredie, s ktorým algoritmus pracuje.



The diagram illustrates the modification of environment parameters. On the left, a JSON object for 'ENV' is shown with 'RTGYM_INTERFACE' set to 'TM20IMAGES', 'WINDOW_WIDTH' set to 256, and 'WINDOW_HEIGHT' set to 128. A large black arrow points to the right, where the same JSON object is shown with 'RTGYM_INTERFACE' changed to 'TM20LIDAR', 'WINDOW_WIDTH' changed to 958, and 'WINDOW_HEIGHT' changed to 488.

```
"ENV": {  
  "RTGYM_INTERFACE": "TM20IMAGES",  
  "WINDOW_WIDTH": 256,  
  "WINDOW_HEIGHT": 128,  
}
```

```
"ENV": {  
  "RTGYM_INTERFACE": "TM20LIDAR",  
  "WINDOW_WIDTH": 958,  
  "WINDOW_HEIGHT": 488,  
}
```

Obrázok 93: Zmena prostredia

Ostatné parametre sú totožné s tými, ktoré boli využité pre algoritmus DQN.

7.3 Snaha o implementáciu algoritmu PPO

Pôvodným cieľom práce bolo implementovanie a porovnanie troch algoritmov – DQN, SAC a PPO. Po vytvorení programu, ktorý implementoval algoritmus PPO, sa však objavil veľký problém, ktorý pri implementácii predošlých dvoch algoritmov nebol okamžite zrejmý. Framework TMRL v svojej základnej forme vôbec nepodporuje algoritmy typu „on-policy“, takže implementácia algoritmu PPO nepripadá v úvahu. Algoritmy DQN a SAC sa učia z hrstky predošlých skúseností, ktoré zozbierala niekedy odlišná, predchádzajúca stratégia, zatiaľ čo algoritmus PPO zbiera skúsenosti prostredníctvom momentálnej stratégie, na ktorých následne vykonáva viaceré optimalizačné kroky. Off-policy opätovne pracuje so starými skúsenosťami, ukladá ich pre neskoršie použitie, no on-policy potrebuje vždy čerstvé skúsenosti. Pre schopnosť tento algoritmus implementovať by bolo potrebné prekopáť celý pipeline, ktorý je pre proces tréningu využívaný. Takýto úkon je mimo rozsahu tejto práce, a teda pôvodný nápad porovnania troch algoritmov bol opustený.

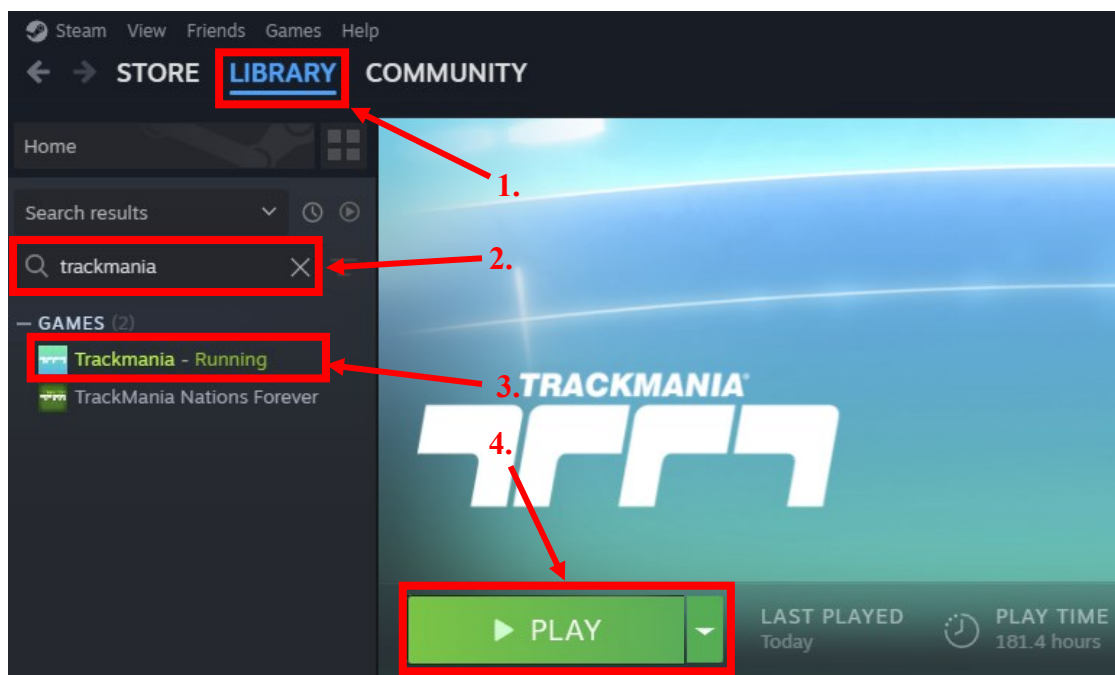
8 PROCES TRÉNOVANIA

Proces trénovania je najdôležitejšia časť tvorenia modelu RL. Framework TMRL umožňuje tento proces spustiť aj na viacerých počítačových zariadeniach, no vzhľadom na to, aké modely sú využívané, to nebude potrebné.

8.1 Príprava trénovania

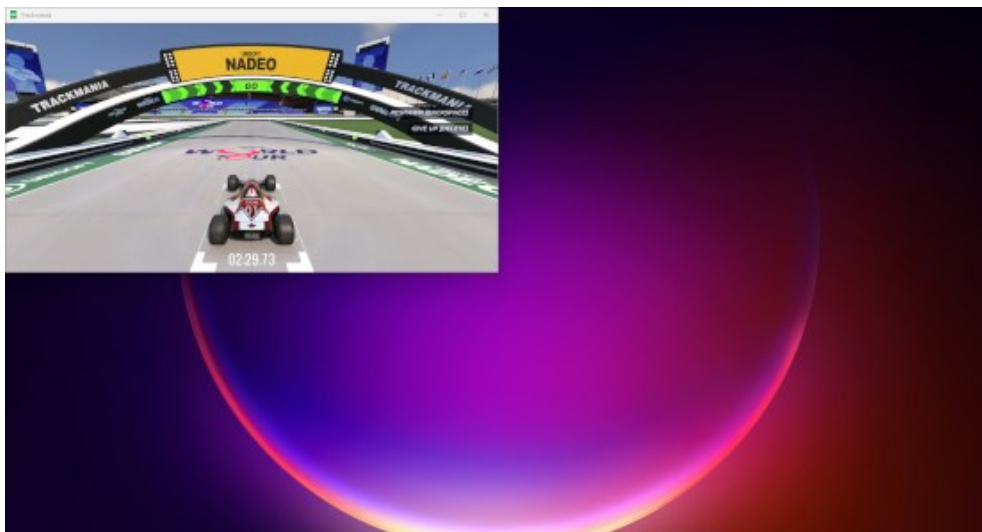
Ako prvé je potrebné spustiť hru cez platformu, na ktorej bola nainštalovaná. V prípade tejto práce je to platforma Steam, no je možné hru spustiť ako cez platformu Steam, tak aj platformu Ubisoft Connect.

Na platforme Steam je po prihlásení potrebné prejsť do sekcie „Library“, teda knižnice hier. V tejto sekcii je políčko pre vyhľadávanie, takže doňho možno napísať „Trackmania“.



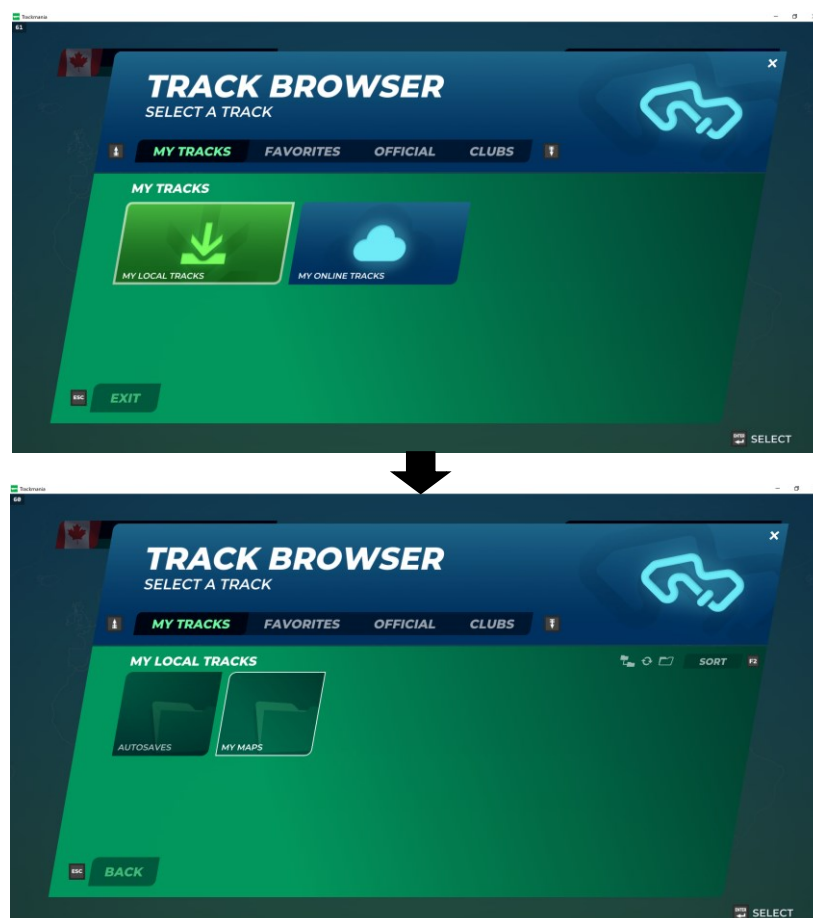
Obrázok 94: Spustenie hry Trackmania cez platformu Steam

Po načítaní hry je potrebné jej okno umiestniť do ľavého horného rohu obrazovky, pretože nástroj, ktorý využíva framework TMRL pre získavanie obrázkov z hry sa spolieha na toto umiestnenie. Ak hra z nejakého dôvodu nie je v režime okna namiesto celej obrazovky aj po nastaveniach v kapitole 6.1, stačí stlačiť kombináciu tlačítok Alt + Enter, čo hru nastaví do režimu okna.



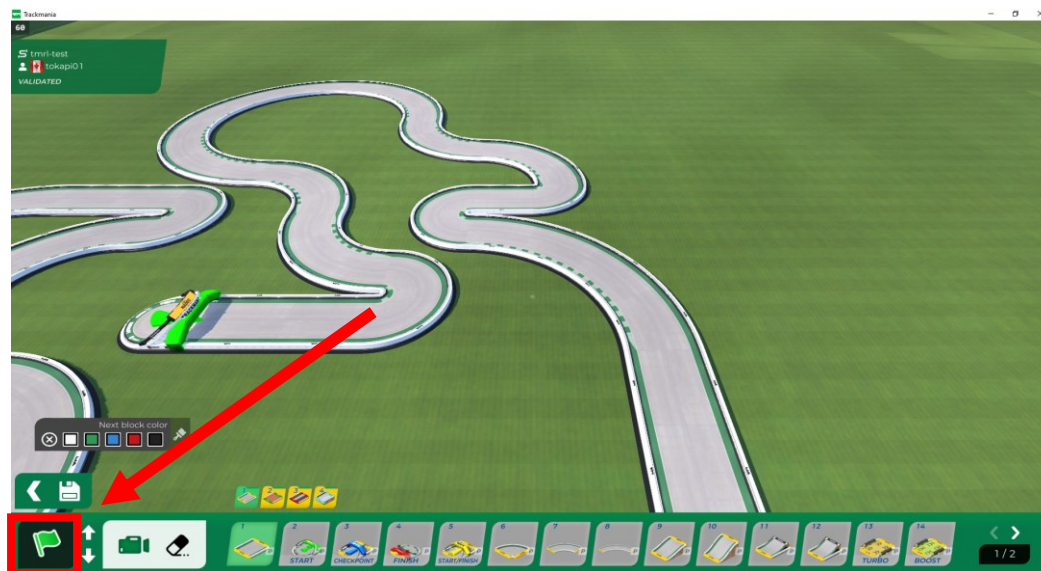
Obrázok 95: Premiestnenie herného okna do ľavého horného rohu [37]

Po nastavení okna je potrebné sa dostať k trati, na ktorej bude tréning prebiehať. V hlavnej ponuke hry je treba kliknúť na položku „Create“. V sekcii „Create“ je položka „Track editor“, teda editor tratí, v ktorej je možnosť „Edit a track“. V tejto ponuke je potreba kliknúť na položku „My Local Tracks“, a ďalej na položku „My Maps“.



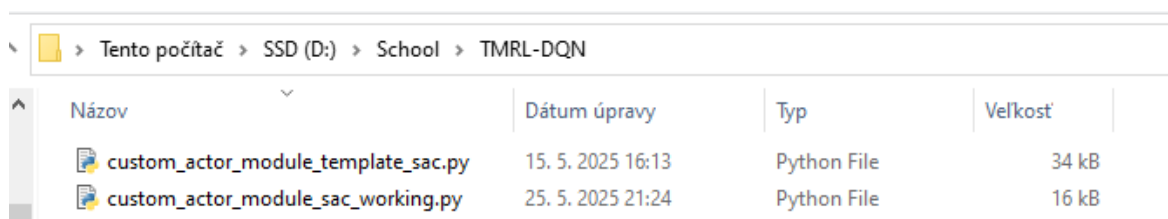
Obrázok 96: Umiestnenie testovacej trate

V tejto sekcii už je na výber konkrétna trať nazvaná tmrl-test. Keďže súbor obsahujúci testovaciu trať je dostupný len lokálne, prístup k nemu je potrebné získať týmto spôsobom. Trať sa zobrazí v pohľade zhora, pretože základne je v režime editácie. Aby sa na nej dalo jazdiť, je nutné kliknúť na ikonku malej zelenej vlajky v ľavom dolnom rohu obrazovky.



Obrázok 97: Spustenie testovacej trate

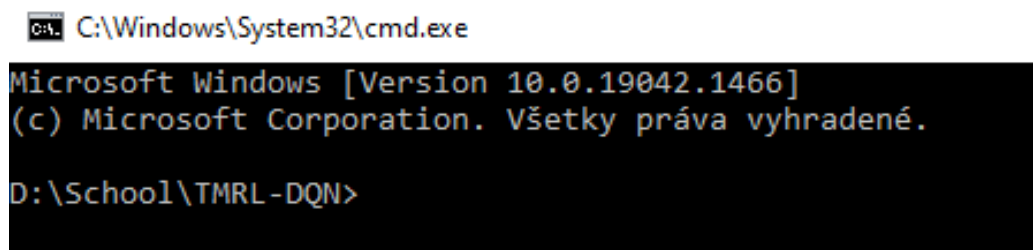
Príprava hry je týmto dokončená, a zostáva už len otvoriť tri okná príkazového riadku. To možno jednoducho vykonať otvorením priečinku, kde sú uložené zdrojové súbory algoritmov. Kliknutím do políčka, kde je uvedená cesta k súboru je možné napísať „cmd“ a stlačiť tlačidlo Enter, čo otvorí príkazový riadok nastavený na túto cestu na disku.



Obrázok 98: Priečink so zdrojovými súbormi

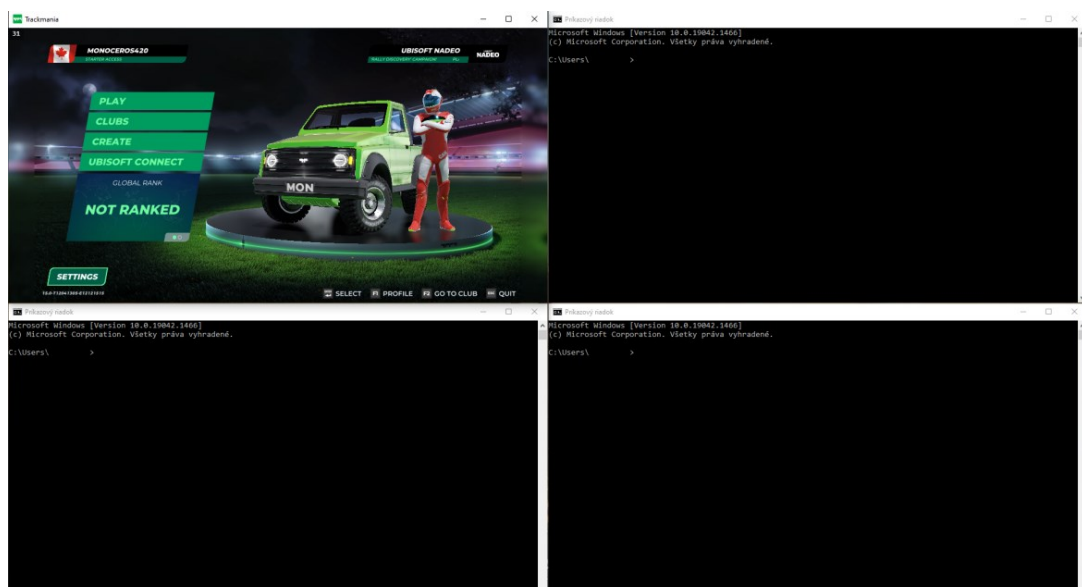


Obrázok 99: Napísanie „cmd“ do poľa s cestou k priečinku



Obrázok 100: Ukážka otvoreného okna s príkazovým riadkom

Pre otvorenie troch takýchto okien stačí iba opätovne kliknúť na políčko s cestou k súboru a stlačiť Enter. Tieto tri okná je potrebné uložiť do zvyšných troch štvrtín obrazovky, tak ako to je ukázané na konci kapitoly 4.2.



Obrázok 101: Rozpoloženie okna hry s oknami príkazových riadkov

Do týchto príkazových riadkov je potrebné napísať tieto príkazy:

- python [názov zdrojového súboru] --server
- python [názov zdrojového súboru] --trainer
- python [názov zdrojového súboru] --worker

Poradie spustenia prakticky nezáleží, no dobrým zvykom je postupovať v poradí, v akom sú uvedené.

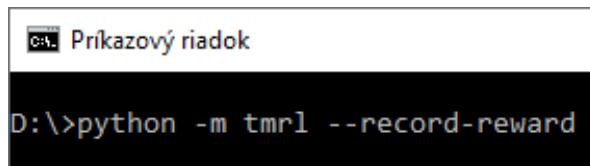
Po spustení týchto príkazov začne proces automaticky, a do jeho konca nie je potrebný žiadny užívateľský vstup. Pred začiatkom je však jedna zmena, ktorá môže nastať – v prípade ak algoritmus pracuje s plným TM20IMAGES prostredím, pohľad na auto, ktorý je využitý je vzadu a nad autom (ako v obrázku 95). V takomto prípade nie je od užívateľa potrebné nič. Ak ale algoritmus využíva prostredie TM20LIDAR, pohľad na auto je potrebné zmeniť, a to stlačením tlačidla 3 na klávesnici. Pravdepodobne bude potrebné ho stlačiť viac ako jeden krát, pretože existujú dve verzie tohto pohľadu – na kapote, kde je viditeľná predná časť auta, a aj tachometer; alebo alternatíva, v ktorej nie je vidieť žiadna časť auta, len trať. Táto alternatívna verzia je potrebná pre prostredie TM20LIDAR.



Obrázok 102: Ukážka správneho pohľadu pre prostredie TM20LIDAR

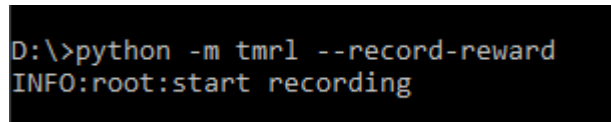
8.1.1 Nahratie vlastnej odmeny

Algoritmy RL sa síce učia sami z vlastných skúseností, no potrebujú nejakú spätnú väzbu o tom, či sa pohybujú na trati správnym smerom. Toto je úlohou „reward track“, ktorá je vlastne len „nahrávka“ úspešného dokončenia trate človekom. V súboroch framework-u TMRL je prítomná už predom pripravená „reward track“ od tvorca framework-u. Ak si chce ale užívateľ vytvoriť vlastnú, môže tak urobiť príkazom `python -m tmrl --record-reward` v príkazovom riadku s otvorenou hrou. V príkazovom riadku sa po vykonaní príkazu zobrazí správa, že nahrávanie začalo. Užívateľ potom môže s autom manuálne jazdiť, a ideálne by sa mal snažiť získať dobrý čas, aby mali algoritmy dobrý východiskový bod pre tréning. Po dokončení trate sa v príkazovom riadku zobrazia štatistiky o vytvorenej nahrávke, napr. koľko kontrolných bodov bolo zaznamenaných. Výstupný súbor je vo formáte `.pkl` a nachádza sa v priečinku `TmrlData/reward`.



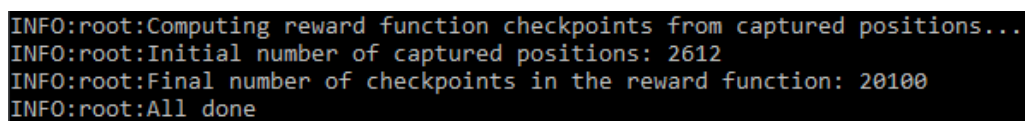
```
CA: Příkazový řádek
D:\>python -m tmrl --record-reward
```

Obrázok 103: Spustenie nahrávania vlastnej odmeny



```
D:\>python -m tmrl --record-reward
INFO:root:start recording
```

Obrázok 104: Hláska pre začiatok nahrávania



```
INFO:root:Computing reward function checkpoints from captured positions...
INFO:root:Initial number of captured positions: 2612
INFO:root:Final number of checkpoints in the reward function: 20100
INFO:root:All done
```

Obrázok 105: Konečný výpis štatistík nahranej odmeny

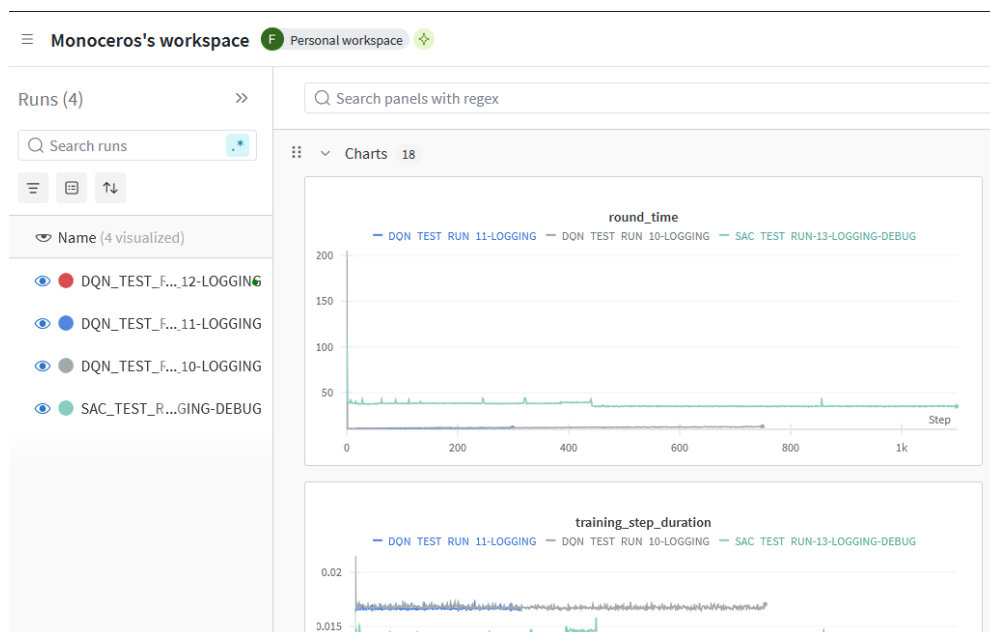
8.1.2 Automatický zber užitočných tréningových dát pomocou Wandb

Tréningové dáta sa v základnom tvare zobrazujú len v príkazovom riadku, na ktorom bol spustený proces `--trainer`. Tieto dáta ale pri dlhších tréningových časoch môžu byť prepísané, pretože príkazový riadok má určitý limit toho, čo sa dá v okne zobrazit'. Preto framework TMRL obsahuje jednoduchý spôsob, ako tieto dáta dlhodobo ukladať na jednom mieste, a rovno ich aj spracováva do grafov. Nástroj, ktorý je na tento účel využívaný sa nazýva Wandb (Weights and Biases), a pre jeho spustenie je sú potrebné dve hlavné zmeny – pridanie jedného riadku do zdrojového kódu algoritmu; úprava niektorých parametrov v súbore `config.json`. Zmena v kóde bola popísaná v 7.2.8. Zmena v `config.json` sa týka parametrov `WANDB_PROJECT`, `WANDB_ENTITY` a `WANDB_KEY`. Prvý parameter bol nazvaný ako „tmrl-dp-mago“, pretože tak sa nazýva vytvorený projekt na stránke wandb. Druhý parameter bol skopírovaný zo stránky, a je ekvivalentný automaticky generovanému užívateľskému menu, v tomto prípade to je „monoceros-univerzita-tom-e-bati-ve-zl-n-„. Posledný parameter sa týka API kľúča, ktorý je potrebný pre umožnenie prístupu k projektu cez príkazový riadok. Ten v texte práce uvedený nebude. Teoreticky je možné tento zber dát spustiť aj bez zmeny týchto parametrov, no dáta by boli v takom prípade ukladané do verejného projektu tvorcu framework-u TMRL, kde sú viditeľné pre všetkých, a je možné, že po určitom čase by boli údaje vymazané.

```
"WANDB_PROJECT": "tmrl-dp-mago",  
"WANDB_ENTITY": "monoceros-univerzita-tom-e-bati-ve-zl-n-"  
"WANDB_KEY": "
```

Obrázok 106: Nastavenie užívateľských údajov pre wandb

Aby bol získaný prístup k tomuto projektu prostredníctvom príkazového riadku, je potrebné využiť príkaz „wandb login“. Ten si od užívateľa vyžiada, aby skopíroval do príkazového riadku API kľúč, ktorý bol spomenutý. Po jeho skopírovaní sa zobrazí hláška, že prihlásenie bolo úspešné, a zobrazia sa následne údaje, ako v parametri WANDB_ENTITY a WANDB_PROJECT. Po úspešnom vykonaní týchto krokov budú všetky tréningové údaje ukladané na stránke wandb, kde budú prístupné na stránke „projects“, v položke „tmrl-dp-mago“, alebo akýkoľvek názov projektu bol zvolený.

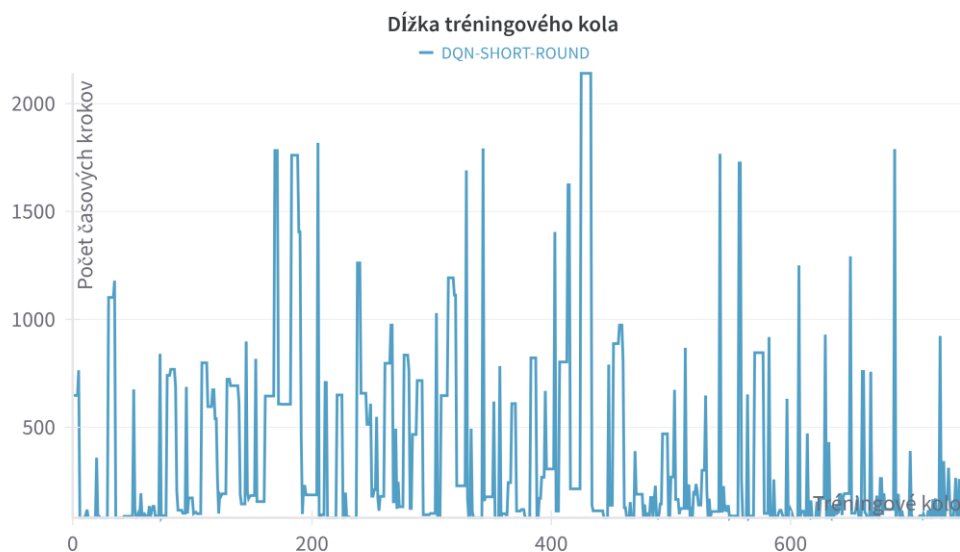


Obrázok 107: Zobrazovanie zbieraných tréningových dát na stránke wandb

8.2 Trénovanie algoritmu DQN

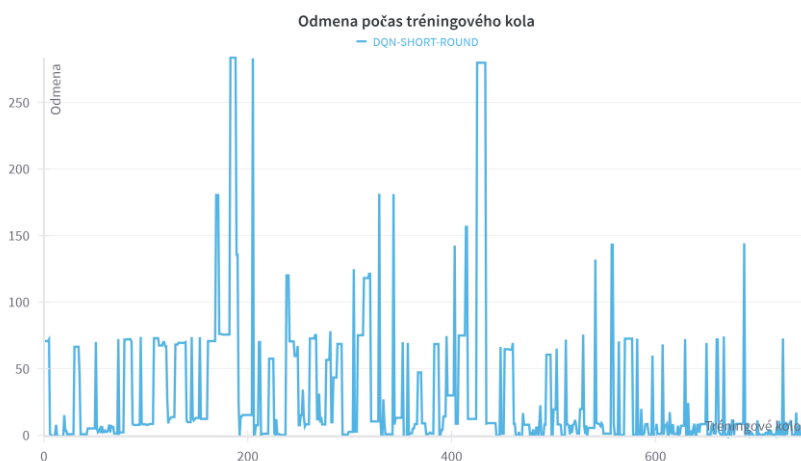
Trénovanie algoritmu DQN prebieha prekvapivo rýchlo, ak sa zoberie ohľad na to, že využíva plné prostredie so snímkami z hry. Finálna iterácia tohto algoritmu nevyužíva žiadne vlastné zlepšenie odmeny, napríklad penalizácia pomalejšej jazdy, alebo odmenenie rýchlejšej jazdy, ani iné populárne zlepšenia základnej verzie algoritmu, ako Double DQN (okrem PER). Celkový čas tréningu po prvé úspešné ukončenie celej trate je približne jedna až dve hodiny, no čas, ktorý je potrebný pre naučenie konzistencie je bližšie 4-6 hodinám.

Z dôvodu veľmi obmedzeného rozsahu možných akcií, z ktorých si agent môže vybrať (kvôli potrebe diskrétného vstupu), je možno krátko po spustení tréningu vidno sľubné výsledky. Často sa stane, že už v priebehu prvej tréningovej epochy sa dokáže agent veľmi priblížiť cieľovej čiare trate, a niekedy ju aj zvládne dokončiť. Je však nutné podotknúť, že toto správanie je extrémne nekonzistentné, a môže sa stať, že hneď po úspešnom dokončení trate začne model narážať do prvej steny pri štartovacím bode. Postupom prvých 4-5 epoch sa táto nekonzistencia postupne znižuje vďaka implementácii PER, ktorá zvykne prioritizovať lepšie pokusy s väčšou odmenou. Správanie, ktoré zostáva konzistentné naprieč celým tréningom je náhodné narážanie do stien po oboch stranách trate. Je ťažké presne odhadnúť, prečo k tomuto dochádza, no jedno logické vysvetlenie je príliš obmedzený akčný priestor. Model má prístup len k plným alebo polovičným hodnotám pre zabávanie, takže v niektorých sekciách testovacej trate by si možno vyžiadal presnejšiu kontrolu nad točením kolies.



Obrázok 108: Graf dĺžky tréningového kola naprieč tréningom

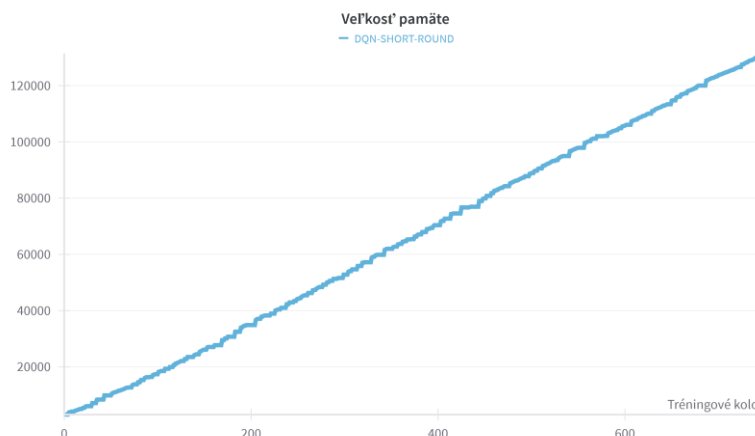
Na obrázku 103 je možno vidieť, ako sa vyvíjala dĺžka tréningových epizód počas procesu tréningu. Je veľmi ľahké všimnúť si spomenutú nekonzistenciu výsledkov. Čiastočne to dáva zmysel, aspoň po krok 400, pretože dovtedy je ešte hodnota premennej epsilon celkom vysoká, no aj po jej postupnom zmenšení po tomto bode nie je vidno zlepšenie. Najvyššie vrcholy približujúce sa a prekračujúce hodnotu 2,000 tiež odpovedajú neintuitívnemu neefektívnemu riešeniu, ktoré DQN často zneužíva – neustále voliť akciu, kde auto ide dopredu a zároveň doprava. Tento spôsob často zaručí, že sa po dlhšej dobe auto dostane cez cieľovú čiaru, no za cenu veľmi zlého času dokončenia trate.



Obrázok 109: Graf odmeny počas tréningového kola

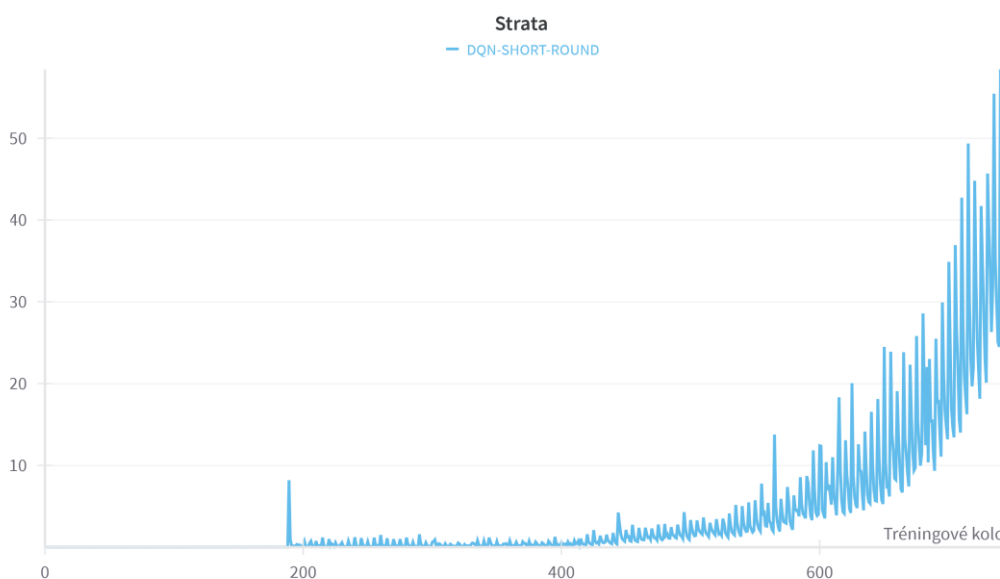
Charakter grafu na obrázku 104 je miestami podobný s grafom z obrázku 103, a dá sa pomocou neho zistiť, v ktorých pokusoch asi model dokázal dokončiť celú trať. To

odpovedá najvyšším bodom na grafe. V ideálnom prípade by týchto vyšších miest malo byť omnoho viac.



Obrázok 110: Graf veľkosti pamäte

Veľkosť pamäte stúpa počas priebehu tréningu celkom lineárne, a je možno sledovať, že pôvodný limit parametru `MEMORY_SIZE` 1,000,000 je naozaj obrovský.



Obrázok 111: Strata Q-funkcie

V grafe na obrázku 106 je zobrazená strata Q-funkcie, na ktorej je vidno, že prvé 2 epochy majú stratu veľmi blízko nule (v rádoch tisícín), no postupným znižovaním hodnoty epsilon zapríčini to, že proces učenia veľmi rýchlo stráca stabilitu a dochádza k divergencii. Toto je pravdepodobne výsledok príliš vysokej hodnoty rýchlosti učenia, ako aj príliš krátkych

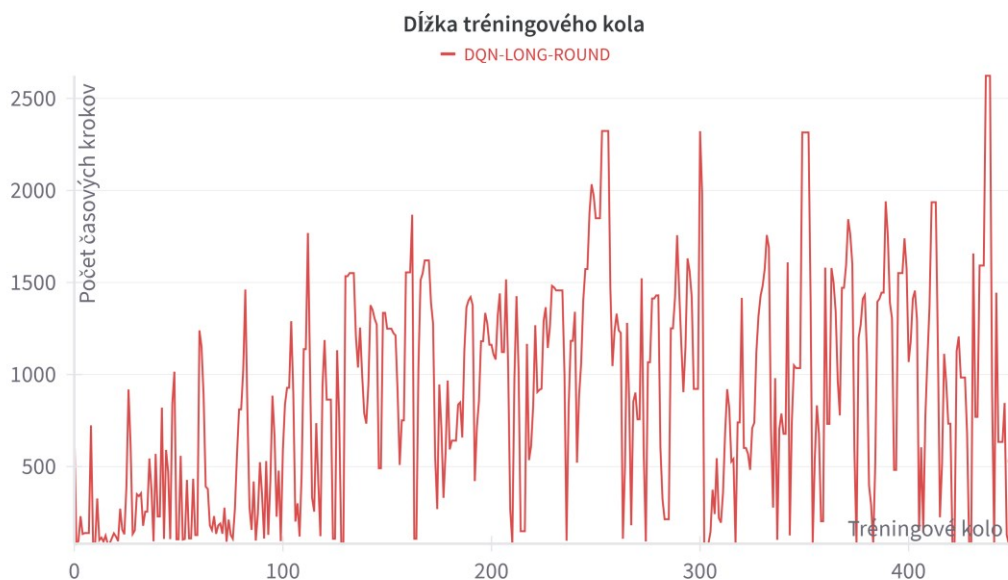
tréninových kôl. Celkový proces tréningu trval približne 2 a pol hodiny, no ako je vidno z obrázku 106, model si nezachoval dostatočne vysokú úroveň „inteligencie“.

8.2.1 Proces tréningovania DQN s pozmenenými nastaveniami

Ako pokus bol proces tréningovania pre DQN spustený po druhý krát, s tým, že parametre, ktoré najpravdepodobnejšie zapríčiňovali stratu vedomostí boli pozmenené na nižšie hodnoty. Najväčšou zmenou bol parameter rýchlosti učenia, ktorého hodnota bola zmenená z 0.0005 na 0.0003. Toto by malo pomôcť znížiť šancu, alebo aspoň oneskoriť, vznik veľkých oscilácií v hodnote straty Q-funkcie. Taktiež bola o niečo znížená hodnota `epsilon_decay` z 0.995 na 0.99. V súbore `config.json` boli pozmenené 3 hodnoty – najdôležitejšou zmenou je zvýšenie hodnoty parametru `TRAINING_STEPS_PER_ROUND` z 200 na 600, čo efektívne strojnásobí dĺžku tréningového kola z ~10 sekúnd na ~30 sekúnd. Tiež bola nastavená hodnota `MEMORY_SIZE` na 350,000.

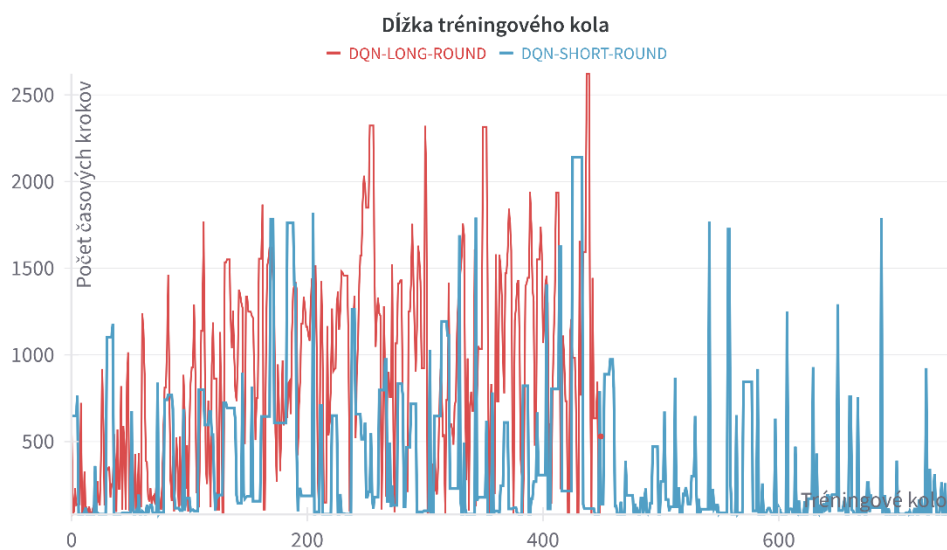
Nakoniec bola zmenená hodnota `ENVIRONMENT_STEPS_BEFORE_TRAINING` z hodnoty 3,000 na 500 ktorá len zapríčiní to, že tréning začne skôr.

Tréning bežal po dobu šiestich epoch, a s predĺženým časom kola trval približne 5 hodín.

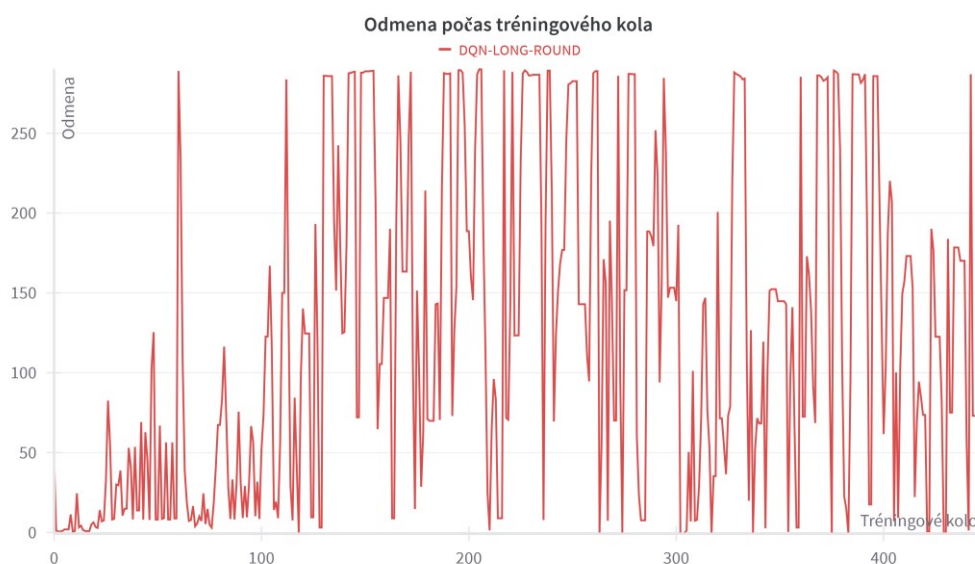


Obrázok 112: Dĺžka tréningového kola pri DQN s alternatívnymi hodnotami

Z grafu na obrázku 107 je možné si všimnúť omnoho viac špicov v dĺžke kola, čo priamo odpovedá vyššej schopnosti dokončiť trať. V tejto iterácii došlo k prvému dokončeniu trate už v prvých tridsiatich minútach tréningu. Je však stále vidno veľkú mieru nekonzistencie.

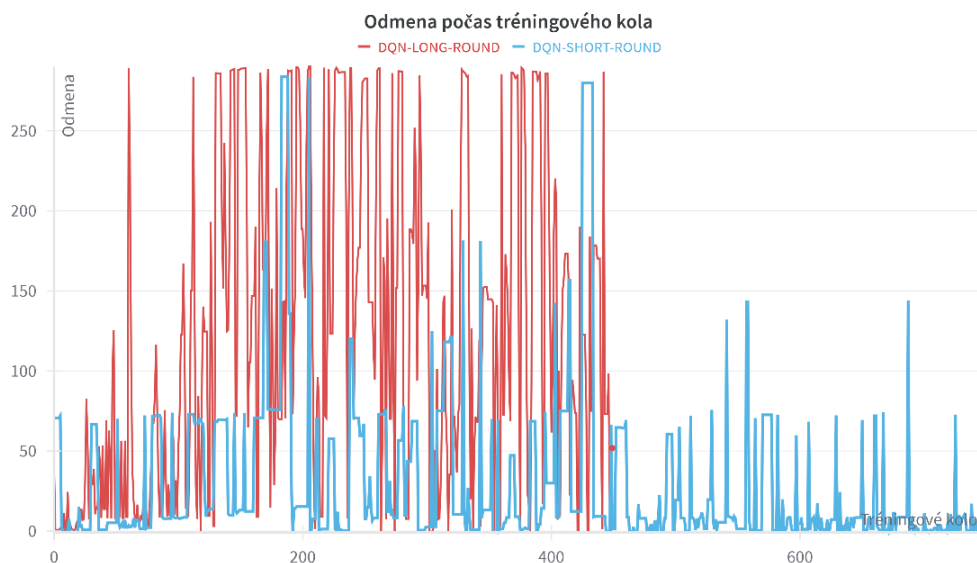


Obrázok 113: Porovnanie dĺžky tréningového kola medzi oboma verziami



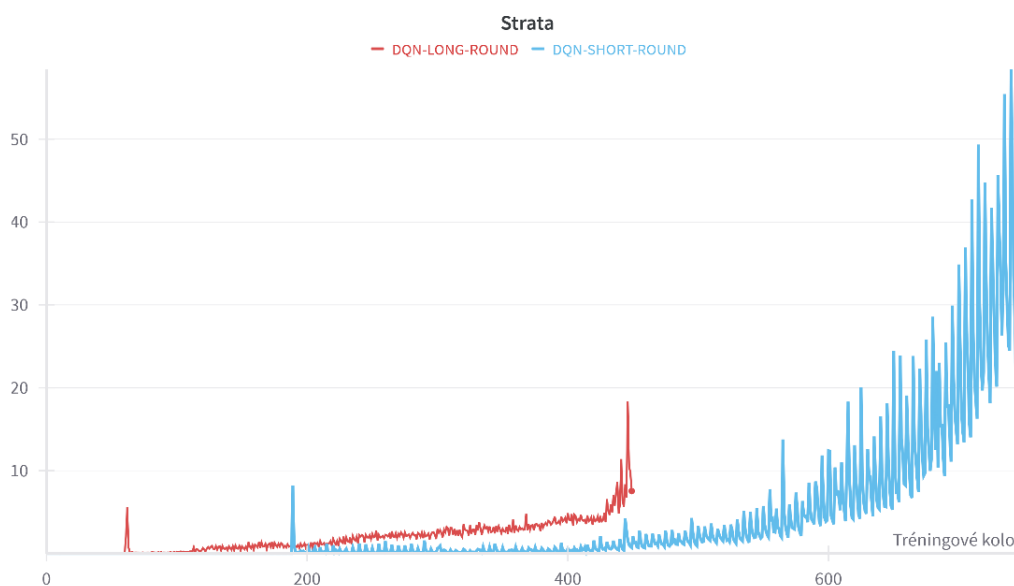
Obrázok 114: Výška odmeny počas tréningového kola pri alternatívnom DQN

V grafe odmeny je možné vidieť trend o niečo väčšej konzistencie ešte viac, počet úspešných dokončení trate sa zvýšil približne štvornásobne. Tento trend je ešte viac viditeľný pri priamom porovnaní oboch verzií.



Obrázok 115: Porovnanie odmeny oboch verzií DQN

Čo sa problému so stratou Q -funkcie týka, zmena parametrov ho nevyriešila, ale značne posunula jeho prvý prejav. Ku koncu dvojhodinového tréningu prvej verzie hodnota straty explodovala do výšok až skoro 60, no po piatich hodinách druhej verzie vystúpila nanajvýš na hodnotu 18. Je možné, že dodatočné pozmenenie hodnôt algoritmu by toto správanie ešte viac posunuli.



Obrázok 116: Porovnanie straty Q-funkcie medzi oboma verziami DQN

8.3 Trénovanie algoritmu SAC

Proces trénovania algoritmu SAC zabral približne 12 hodín, a to aj napriek tomu, že využíva značne jednoduchšie prostredie v podobe meraní LIDAR na rozdiel od celých snímok z hry, ktoré využívalo DQN. Celkovo môže byť proces trénovania finálnej verzie implementácie označený ako veľký úspech. Na rozdiel od DQN má vcelku pomalý rozbeh, no krátko po tom, ako narazí na lepšiu skúsenosť ju dokáže konzistentne replikovať a zlepšovať svoj prístup. Taktiež pri SAC nie je vidno degradáciu vo výsledkoch ani po 12 hodinách trénovania. Je možné, že ak by tréning pokračoval aj dlhšie, model by sa ešte dokázal o niečo málo zlepšiť.

Tento výsledok bol relatívne očakávaný. Algoritmus SAC je sám o sebe značne pokročilejší od DQN, a implementácia popísaná v tejto práci má viaceré vylepšenia od pôvodnej formy.



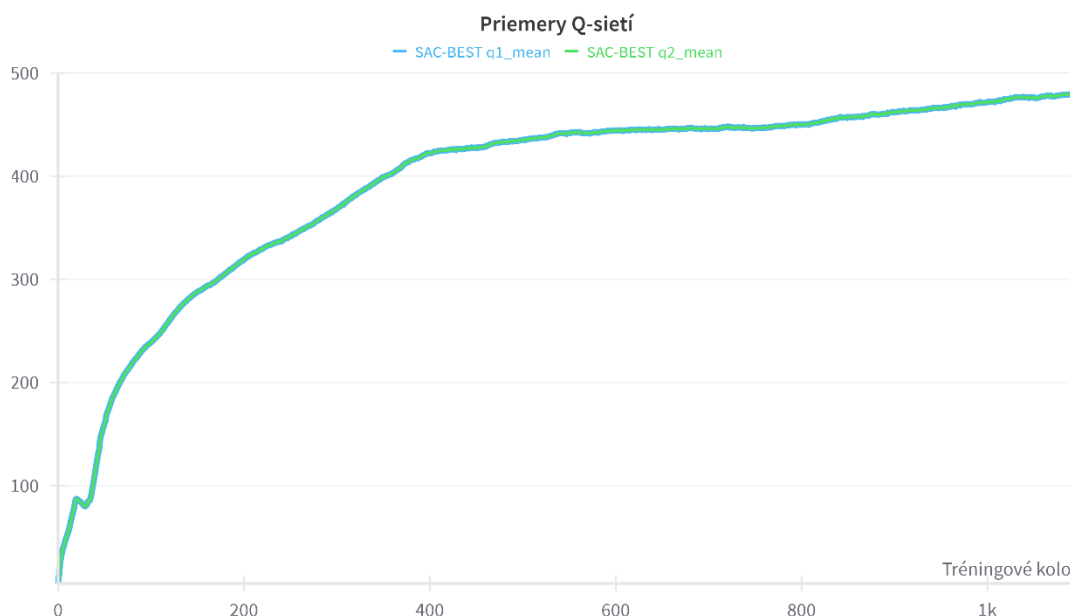
Obrázok 117: Graf straty kritika

Z grafu na obrázku 112 je možné sledovať vcelku stabilný trend straty u Q-sietí kritikov. Počas prvých tréningových epoch je viditeľný relatívne prudký výstup, no po objavení nadpriemernej stratégie sa začala hodnota stabilizovať pod úrovňou 20. Jej postupné zvyšovanie od kola 800 naznačuje potenciál pre dodatočné zlepšenie v prípade, že by tréning bežal dlhšie.



Obrázok 118: Graf straty aktéra

Trend grafu na obrázku 113 môže síce vyzerat' drasticky, no opak je pravdou. Hodnoty, ktoré je vidno, sú trocha prehnané, pravdepodobne kvôli tomu, ako je implementovaná logika úpravy odmeny, no progresívne negatívne hodnoty sú dobrým znamením toho, že si aktér vylepšuje svoju stratégiu.



Obrázok 119: Priemery hodnôt Q-sietí

Trend grafu na obrázku 114 je inverzný voči grafu na obrázku 113, a to preto, lebo medzi hodnotou straty aktéra, a medzi priemernými hodnotami Q-sietí je priama korelácia.

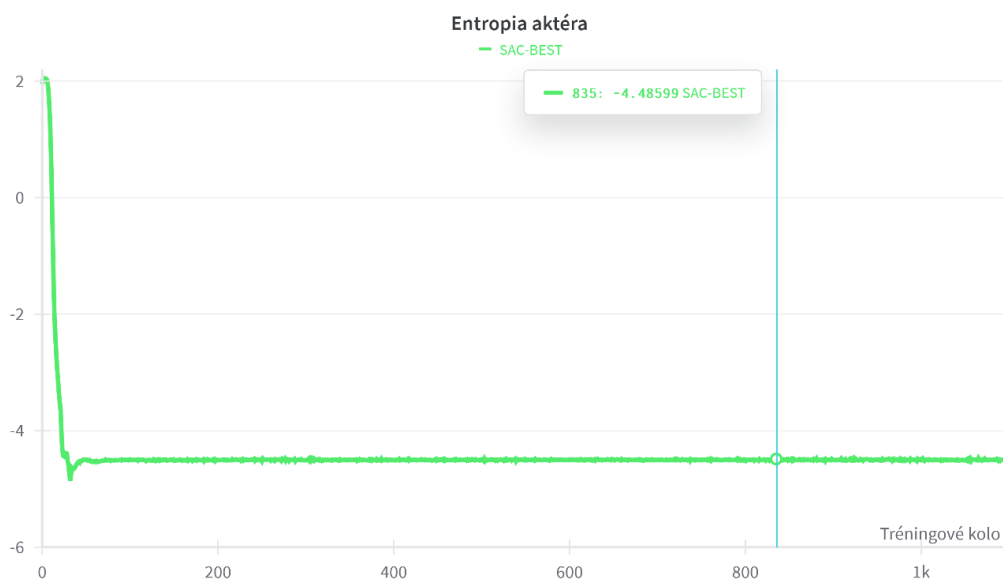
Hodnoty sú v tomto grafe tiež relatívne vysoké z podobného dôvodu ako pri strate aktéra. Taktiež je možné sledovať, že priemery siete Q_1 a priemery siete Q_2 majú medzi sebou rozdiely v rádoch stotín až tisícín.

Pri rýchlostiach učenia bol implementovaný plánovač, a medzi znížením hodnoty rýchlosti učenia kritika a aktéra je značný rozdiel. Kritik upadol už počas prvej epochy, zatiaľ čo aktér sa držal na polovičnej hodnote až do polovice tréningového procesu. Tento trend sa zhoduje s realitou a očakávaniami. Kritici – Q-funkcie – konvergujú pred aktérom. Keďže aktér sa učí z odhadov kritikov, rýchlejšia konvergencia kritikov im umožňuje ustáliť sa na lepších odhadoch. Stálejšie a stabilnejšie odhady Q-hodnôt znamenajú efektívnejšie učenie pre aktéra.

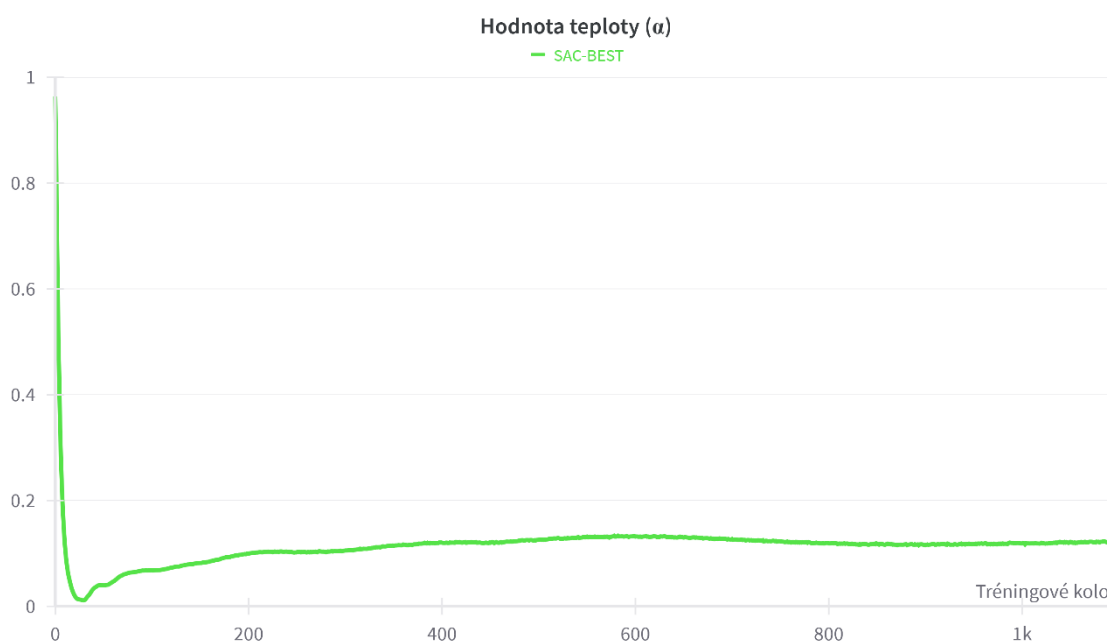


Obrázok 120: Porovnanie rýchlosti poklesu hodnôt LR medzi aktérom a kritikmi

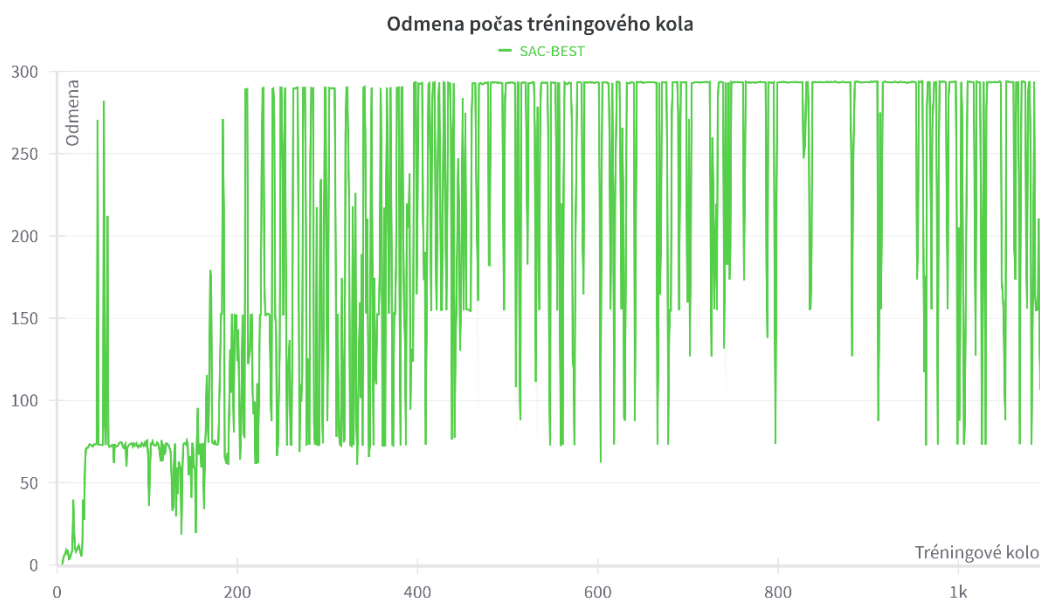
Keďže implementácia SAC využíva automatické nastavovanie hodnoty entropie, boli zbierané aj dáta o entropií aktéra, spolu s dátami o veľkosti hodnoty α (teplota), pretože tieto hodnoty sú prepojené - α udáva váhu entropie, alebo ako moc by sa mal agent snažiť preferovať náhodnosť oproti očakávanej odmene. Hodnota α začína pri 0.97, no rapídne klesá k nule hneď na začiatku tréningu. Rovnaký trend je možné sledovať pri hodnote entropie aktéra, ktorá začína na hodnote 2 a rýchlo klesá na hodnotu skoro -5, a následne sa stabilizuje pri hodnote -4.5 po zbytok tréningu.



Obrázok 121: Entropia aktéra algoritmu SAC

Obrázok 122: Hodnota teploty – α – algoritmu SAC

Pri grafe odmeny v obrázku 116 je vidno zlomový bod, kedy bola objavená nadpriemerná stratégia, a krátko na to začala byť postupne viacej a viacej konzistentne dosiahnutá. Taktiež je možné pozorovať malé zhoršenie v konzistencii medzi kolom 800 a kolom 1,000, čo odpovedá zvýšeniu hodnoty straty kritika. Výsledný model však dosahuje konzistenciu približne 70-80% z hľadiska dokončenia testovacej trate.



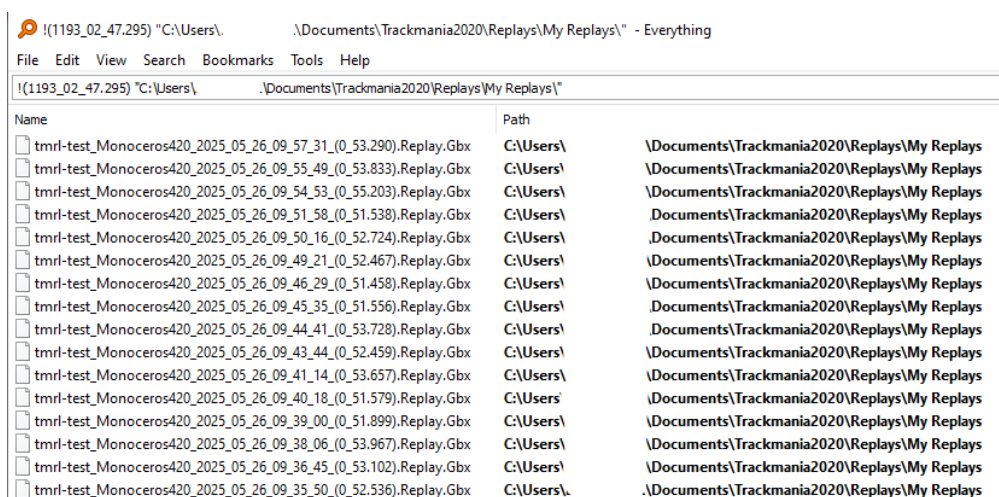
Obrázok 123: Odmena získaná počas tréningového kola pri algoritme SAC

8.4 Porovnanie najlepších získaných časov natrénovaných modelov

Ďalším dôležitým údajom o efektívnosti natrénovaných modelov je to, ako rýchlo dokázali trať dokončiť. Poznámka: časy, ktoré sú viditeľné nižšie, sú v skutočnosti o jednu sekundu dlhšie, pretože časťou tréningovej pipeline je sekunda dlhé oneskorenie pred prenechaním kontroly modelu.

Názvy súborov, ktoré obsahujú záznamy úspešných dokončení trate sú veľmi dlhé, takže najprv pomocou programu Everything sú vyfiltrované neúspešné pokusy pomocou príkazu:

```
!(1193_02_47.295) "C:\Users\[užívateľské-  
meno]\Documents\Trackmania2020\Replays\My Replays\"
```



Obrázok 124: Vyfiltrovanie neúspešných pokusov v programe Everything

Prvá časť príkazu udáva, že sa nemajú v danom priečinku zobrazovať súbory, ktoré v svojom názve obsahujú špecifický text - (1193_02_47.295). Súbory, ktoré zostali zobrazené sú už len pokusy, kde došlo k úspešnému dokončeniu trate. Čísla v zátvorke pred súborovou príponou .Replay.Gbx udávajú, aký čas bol dosiahnutý počas pokusu, napríklad (0_53.290).

Po zoradení vyfiltrovaných výsledkov podľa dátumu je možné izolovať len pokusy vykonané poslednou iteráciou algoritmu SAC. Tieto súbory môžu byť skopírované a presunuté do nového priečinku pre ďalšie spracovanie.

8.4.1 Vytvorenie programu pre hromadné premenovanie súborov

Keďže stredná časť názvu súboru obsahuje dátum a čas, kedy bol súbor vytvorený, zoradenie podľa názvu pre zistenie najkratšieho času fungovať nebude. Pre účel hromadného spracovania názvov súborov bol vytvorený jednoduchý program v jazyku Python, nazvaný `replay-renamer.py`. Program bol uložený k skopírovaným súborom záznamov, kde bol pomocou príkazového riadku spustený. Jeho účelom je hromadne premenovať všetky súbory v priečinku z pôvodného názvu len na čísla v zátvorke, ktorá je časťou pôvodného názvu.

```
import os
import re

def rename_files(folder_path):
    for filename in os.listdir(folder_path):
        file_path = os.path.join(folder_path, filename)
        if os.path.isfile(file_path):
            # Extract content from parentheses using regex
            match = re.search(r'\([^\)]+\)', filename)
            if match:
                new_name = match.group(1)
                new_name = new_name.replace('_', '-')
                if filename.endswith('.Replay.gbx'):
                    new_filename = f'{new_name}.Replay.gbx'
                else:
                    _, ext = os.path.splitext(filename)
                    new_filename = f'{new_name}.Replay{ext}'
                new_file_path = os.path.join(folder_path, new_filename)
                if file_path != new_file_path:
                    if os.path.exists(new_file_path):
                        print(f"Warning: '{new_filename}' already exists. Skipping '{filename}'")
                    else:
                        os.rename(file_path, new_file_path)
                        print(f"Renamed: '{filename}' -> '{new_filename}'")
            else:
                print(f"No parentheses found in: '{filename}', skipping")

folder_path = input("Enter the folder path: ").strip()

if os.path.exists(folder_path) and os.path.isdir(folder_path):
    rename_files(folder_path)
    print("\nProcessing complete!")
else:
    print("Invalid folder path!")
```

Obrázok 125: Program pre hromadné premenovávanie súborov

Po spustení programu je v príkazovom riadku vytvorený krátky výstup, ktorý oznamuje užívateľovi, či došlo ku konfliktom s premenovávaním, napríklad duplicitné súbory.

Výstupom je zorganizovaný priečinok so súbormi.

> SAC-REPLAYS

Názov	Dátum úpravy	Typ
 0_50.429.Gbx	26. 5. 2025 5:55	Súbor GBX
 0_50.994.Gbx	26. 5. 2025 10:23	Súbor GBX
 0_51.006.Gbx	26. 5. 2025 6:06	Súbor GBX
 0_51.050.Gbx	26. 5. 2025 7:38	Súbor GBX
 0_51.083.Gbx	26. 5. 2025 8:34	Súbor GBX
 0_51.092.Gbx	26. 5. 2025 5:25	Súbor GBX
 0_51.095.Gbx	26. 5. 2025 8:58	Súbor GBX
 0_51.258.Gbx	26. 5. 2025 7:15	Súbor GBX
 0_51.271.Gbx	26. 5. 2025 7:06	Súbor GBX
 0_51.379.Gbx	26. 5. 2025 8:39	Súbor GBX
 0_51.458.Gbx	26. 5. 2025 9:46	Súbor GBX
 0_51.492.Gbx	26. 5. 2025 6:46	Súbor GBX
 0_51.498.Gbx	26. 5. 2025 5:31	Súbor GBX
 0_51.527.Gbx	26. 5. 2025 10:32	Súbor GBX
 0_51.538.Gbx	26. 5. 2025 9:51	Súbor GBX
 0_51.555.Gbx	26. 5. 2025 6:18	Súbor GBX

Obrázok 126: Premenované súbory v priečinku

8.4.2 Najrýchlejšie časy algoritmu DQN

Po spracovaní názvov sa dá zistiť, že najrýchlejší čas, ktorý získal algoritmus DQN, je 56.763 sekúnd. Celkový počet úspešných pokusov, ktoré algoritmus DQN dosiahol, je 83. Najpomalší úspešný pokus bol dlhý 2 minúty a 5 sekúnd. Priemerný čas úspechu je 1 minúta a 14 sekúnd.

8.4.3 Najrýchlejšie časy algoritmu SAC

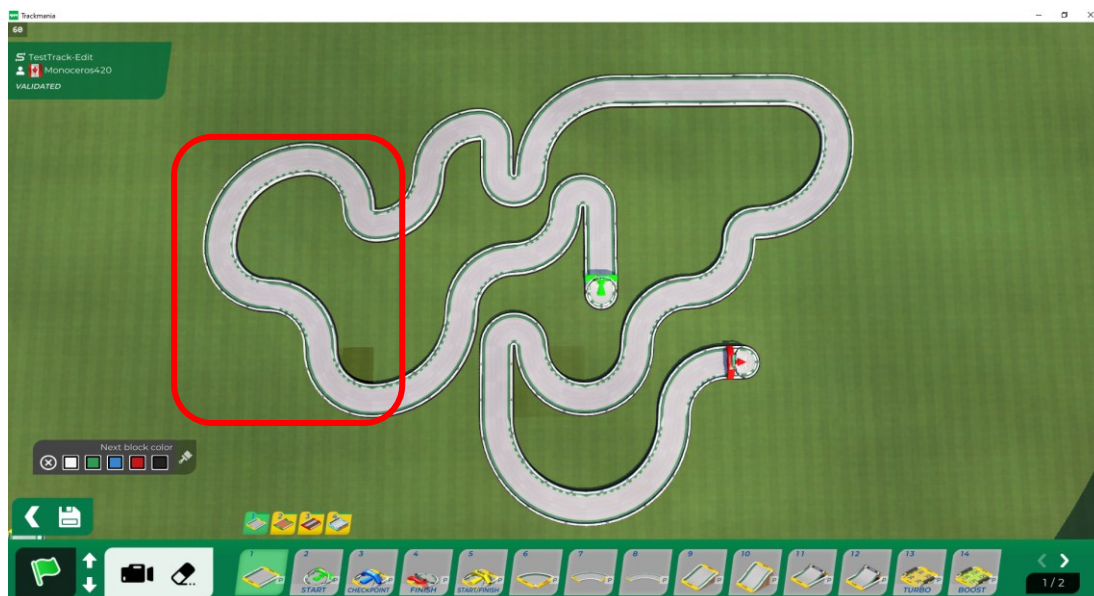
Najrýchlejší čas, ktorý získal algoritmus SAC, je 50.429 sekúnd. Celkový počet úspešných pokusov, ktoré algoritmus SAC dosiahol, je vďaka značne dlhšiemu tréningu až 590. Najpomalší čas pokusu pri algoritme SAC bol 1 minúta a 51 sekúnd, a priemerný čas úspešného pokusu bol 55 sekúnd.

8.5 Porovnanie robustnosti natrénovaných modelov

Pre toto porovnanie bola vytvorená alternatívna verzia testovacej trate, na ktorej boli vykonané vcelku drastické zmeny. Účelom je sa dozvedieť, či sa natrénované modely dokážu s týmito zmenami vysporiadať bez dodatočného pretrénovania, a stále úspešne dokončiť trať. Pre tento účel bola potrebná aj nová nahrávka odmeny, pretože bez nej sú kolá v určitých sekciách mapy ukončené automaticky príliš skoro, kvôli prílišnej deviácii od pôvodnej nahranej odmeny. Zmeny na trati sú vyznačené červeným obdĺžnikom.



Obrázok 127: Pôvodný vzhľad trate



Obrázok 128: Upravený vzhľad trate

Po nahratí novej odmeny pomocou príkazu `python -m tmrl --record-reward` je potrebné otvorenie dvoch príkazových riadkov nastavených na cestu k zdrojovému súboru programu algoritmu.

8.5.1 Algoritmus DQN

Efektívnosť algoritmu DQN sa ku koncu jeho poslednej tréningovej epochy znížila až príliš moc na to, aby dokázal dokončiť aj neupravenú trať. Hneď na začiatku trate, ešte d'aleko od pozmenenej časti, má model vysokú úroveň neistoty, a náhodne volí akcie, ktoré mu zabraňujú v progrese. Ani po niekoľkých desiatkach pokusov nebol tento algoritmus schopný dokončiť trať. Ak by bol algoritmus pretrénovaný s tým, že by bol proces tréningu manuálne ukončený pred degradovaním jeho schopností, je pravdepodobné, že by sa model dokázal po nejakom čase vysporiadať so zmenami. V jeho súčasnej podobe to však nedokáže.

8.5.2 Algoritmus SAC

Pri testovaní robustnosti algoritmu SAC sa dokázal natrénovaný model vysporiadať so zmenou na trati bez väčších problémov. Hneď na prvý pokus bol model schopný úspešne navigovať naprieč zmenenou traťou, a dokončil ju len o približne 6 sekúnd pomalšie, ako priemerný čas na neupravenej trati. Takýto výsledok jasne potvrdzuje robustnosť natrénovaného modelu.

8.6 Porovnanie výsledkov proti ľudskému času

Algoritmy, ktoré boli v tejto práci implementované a natrénované, nie sú príliš pokročilé, no aj tak dosiahli veľmi úctyhodné výsledky. Ani jeden z modelov sa však nepribližuje „najlepšiemu“ času, ktorý bol využitý ako odmena pre tréning.

Tabuľka 2: Porovnanie najazdených časov

Trat' \ Vodič	Človek	DQN (krátky tréning)	DQN (dlhý tréning)	SAC
<i>tmrl-test</i>	00'38"708	1'28"970	00'56"763	00'50"429
<i>TestTrack-Edit</i>	00'44"578	Nedokončil	Nedokončil	00'58"849

8.7 Zhrnutie správania natrénovaných modelov

Po natrénovaní oboch modelov je možné na nahrávkach sledovať veľmi odlišné stratégie jazdenia.

Povaha algoritmu DQN je veľmi nepredvídateľná a nekonzistentná, no v náročnejších sekciách testovacej trate máva často lepšiu, viacej optimálny prístup – napríklad v prudkých 180° zákrutách väčšinou pribrzdí a zvolí agresívnejšiu jazdnú líniu, a tiež sa nebojí využiť narazenie do steny trate pre svoju výhodu.

Toto správanie je presne opačné pri algoritme SAC, pretože zatiaľ čo má veľmi dobré jazdné línie v rovinkách a jemnejších zákrutách, náročnejšie sekcie, ako prudké zákruty, mu robia problémy. Často v nich prudko spomaľuje a nevolí agresívne jazdné línie, pravdepodobne kvôli tomu, že sa presnejšie drží nahratej odmeny. Každopádne bezpečnejší prístup, ktorý si zvolil algoritmus SAC dáva väčší zmysel, pretože priemerný čas úspešného kola je o skoro 15 sekúnd rýchlejší od algoritmu DQN, nehovoriac o jeho konzistencii.

Tieto pozorovania sú najlepšie viditeľné v priložených videách.

ZÁVĚR

Táto práca sa zaoberala témou automatického riadenia v hrách, konkrétne v hre Trackmania (2020). Účelom bolo oboznámiť sa s princípmi funkcie algoritmov hlbokého spätnoväzobného učenia, špecificky s algoritmami Deep Q-Network a Soft Actor-Critic.

V teoretickej časti práce bol vykonaný krátky úvod do problematiky RL ako aj do série hier Trackmania, podrobný rozbor implementovaných algoritmov, ako aj viacerých dodatočných tém z problematiky deep RL, týkajúce sa rozoberaných algoritmov – katastrofické zabúdanie, metóda Temporal Difference a Monte Carlo, Polyakove priemerovanie, koncept Q-funkcií a viaceré iné. Na záver teoretickej časti boli popísané aj nástroje, ktoré boli pre vypracovanie tejto práce využité, hlavne nástroj TMRL.

V praktickej časti práce boli najprv pripravené všetky potrebné súčasti – hra Trackmania, platforma OpenPlanet a framework TMRL, a aj dodatky ako PyTorch CUDA. Ďalej boli od úplných základov implementované vybrané algoritmy DQN a SAC – od prípravy súboru, nastavenie konfigurácie, úprava herných nastavení, až po detailné vysvetlenie samotného programu a procesu tréningovania. Výstup algoritmov bol zozbieraný pomocou nástroja Wandb, ktorý poskytol užitočné grafy. Tieto grafy pomohli vizualizovať informácie popísané v teoretickej časti, no taktiež na nich bolo vidieť nedostatky v konečných implementáciách. Na konci praktickej časti boli zhodnotené a porovnané časy dokončenia testovacej trate medzi ľudským výkonom, a najlepším výkonom natrénovaných modelov.

Cieľom tejto práce bolo hlavne pokúsiť sa implementovať populárne algoritmy z oblasti reinforcement learning-u. Tento cieľ bol splnený a to s dobrými výsledkami.

SEZNAM POUŽITÉ LITERATURY

- [1] SWAN, Cameron. *Game Rant*. Online. Game Rant. 2022. Dostupné z: <https://gamerant.com/trackmania-2023-reboot-history/>. [cit. 2025-02-22].
- [2] LUKWISNIE a IBADATNAMES9902. *TrackMania 2003 (game)*. Online. Nadeo Wiki. 2014. Dostupné z: [https://nadeo.fandom.com/wiki/TrackMania_\(game\)](https://nadeo.fandom.com/wiki/TrackMania_(game)). [cit. 2025-02-22].
- [3] NAVARRO, Alex. *TrackMania Review*. Online. GameSpot. 2004. Dostupné z: <https://www.gamespot.com/reviews/trackmania-review/1900-6101887/>. [cit. 2025-02-22].
- [4] METACRITIC. *TrackMania (2003) critic reviews*. Online. Metacritic. Dostupné z: <https://www.metacritic.com/game/trackmania-2003/critic-reviews/>. [cit. 2025-02-22].
- [5] HYLISFAN07 a FLANQER. *Trackmania Sunrise*. Online. Trackmania Wiki. 2020. Dostupné z: https://trackmania.fandom.com/wiki/Trackmania_Sunrise. [cit. 2025-02-23].
- [6] 62.252.128.16. *TrackMania*. Online. In: Wikipedia, The Free Encyclopedia. 2004. Dostupné z: <https://en.wikipedia.org/wiki/TrackMania>. [cit. 2025-02-23].
- [7] MRRANDOMCUBER, et al. *TrackMania 2*. Online. In: Wikipedia, The Free Encyclopedia. 2013. Dostupné z: https://en.wikipedia.org/wiki/TrackMania_2. [cit. 2025-02-23].
- [8] FLANQER, et al. *Trackmania Turbo (2016)*. Online. Trackmania Wiki. 2020. Dostupné z: [https://trackmania.fandom.com/wiki/Trackmania_Turbo_\(2016\)](https://trackmania.fandom.com/wiki/Trackmania_Turbo_(2016)). [cit. 2025-02-23].
- [9] ZXCVBNM, et al. *Trackmania (2020 video game)*. Online. In: Wikipedia, The Free Encyclopedia. 2021. Dostupné z: [https://en.wikipedia.org/wiki/Trackmania_\(2020_video_game\)](https://en.wikipedia.org/wiki/Trackmania_(2020_video_game)). [cit. 2025-02-23].
- [10] FLANQER. *Trackmania (2020)*. Online. Trackmania Wiki. 2020. Dostupné z: [https://trackmania.fandom.com/wiki/Trackmania_\(2020\)](https://trackmania.fandom.com/wiki/Trackmania_(2020)). [cit. 2025-02-23].
- [11] SGRUNT, et al. *Welcome To TASVideos*. Online. TASVideos. 2010. Dostupné z: <https://tasvideos.org/WelcomeToTASVideos>. [cit. 2025-02-25].
- [12] NEZNÁMY. *Glossary*. Online. SpeedRunsLive. Dostupné z: <https://www.speedrunslive.com/rules-faq/glossary>. [cit. 2025-02-25].

- [13] KARNOFSKY, Holden. *Tool-assisted speedrunning*. Online. Cold Takes. 2021. Dostupné z: <https://www.cold-takes.com/tool-assisted-speedrunning/>. [cit. 2025-02-25].
- [14] DONADIGO. *TMInterface*. Online. Donadigo. 2021. Dostupné z: <https://donadigo.com/tminterface/>. [cit. 2025-02-26].
- [15] DONADIGO. *TMInterface Guide*. Online. Donadigo. 2021. Dostupné z: <https://donadigo.com/tminterface/save-states>. [cit. 2025-02-26].
- [16] DONADIGO. *TMInterface Guide*. Online. Donadigo. 2021. Dostupné z: <https://donadigo.com/tminterface/commands>. [cit. 2025-02-26].
- [17] DONADIGO. *TMInterface Guide*. Online. Donadigo. 2021. Dostupné z: <https://donadigo.com/tminterface/fast-forward>. [cit. 2025-02-26].
- [18] DEEPCHECKS COMMUNITY BLOG. *Reinforcement Learning Applications: From Gaming to Real-World*. Online. Deepchecks. 2023. Dostupné z: <https://www.deepchecks.com/reinforcement-learning-applications-from-gaming-to-real-world/>. [cit. 2025-02-27].
- [19] DEEPMIND. *AlphaGo*. Online. Google DeepMind. 2016. Dostupné z: <https://deepmind.google/research/breakthroughs/alphago/>. [cit. 2025-02-27].
- [20] OPENAI. *OpenAI Five*. Online. OpenAI. 2018. Dostupné z: <https://openai.com/index/openai-five/>. [cit. 2025-02-27].
- [21] AGADE09; PB4GIT a ĎALŠÍ. *Linesight*. Online. GitHub. 2024. Dostupné z: <https://github.com/Linesight-RL/linesight>. [cit. 2025-05-16].
- [22] LINESIGHT. *I Trained an AI for 2 Years on Trackmania. It's Breaking Records.; YouTube*. Online. Dostupné z: <https://www.youtube.com/watch?v=cUojVsCJ51I>. [cit. 2025-05-16].
- [23] ANDREJGOBEX. *TrackMania_AI*. Online. GitHub. 2023. Dostupné z: https://github.com/AndrejGobeX/TrackMania_AI. [cit. 2025-05-16].
- [24] YOSH. *AI learns to exploit a glitch in Trackmania*. Online. 2025. Dostupné z: <https://www.youtube.com/watch?v=NUI6QikjR04>. [cit. 2025-05-16].
- [25] YOSH. *AI beats multiple World Records in Trackmania*. Online. 2024. Dostupné z: <https://www.youtube.com/watch?v=kojH8a7BW04>. [cit. 2025-05-16].

- [26] SMITH, Thomas. *What It's Really Like to Ride in a Waymo Self-Driving Car*. Online. Bay Area Telegraph. 2024. Dostupné z: <https://bayareatelegraph.com/2024/08/21/what-its-really-like-to-ride-in-a-waymo-self-driving-car/>. [cit. 2025-05-17].
- [27] MUREL, Jacob a KAVLAKOGLU, Eda. *What is reinforcement learning?* Online. IBM. 2024. Dostupné z: <https://www.ibm.com/think/topics/reinforcement-learning>. [cit. 2025-05-17].
- [28] LI, Yuxi. *Deep Reinforcement Learning: An Overview*. Online. ArXiv.org. 2017. Dostupné z: <https://arxiv.org/abs/1701.07274v6>. [cit. 2025-05-17].
- [29] SIMONINI, Thomas a ĎALŠÍ. *Deep RL Course - Introducing Q-Learning*. Online. HuggingFace.co. 2024. Dostupné z: <https://huggingface.co/learn/deep-rl-course/unit2/q-learning>. [cit. 2025-05-18].
- [30] PATICCHIO, Alessandro. *Introduction to Reinforcement Learning: Temporal Difference, SARSA, Q-Learning*. Online. Towards Data Science. 2022. Dostupné z: <https://towardsdatascience.com/introduction-to-reinforcement-learning-temporal-difference-sarsa-q-learning-e8f22669c366/>. [cit. 2025-05-18].
- [31] SIMONI, Thomas a ĎALŠÍ. *Deep RL Course - Monte Carlo vs Temporal Difference Learning*. Online. HuggingFace.co. 2024. Dostupné z: <https://huggingface.co/learn/deep-rl-course/unit2/mc-vs-td>. [cit. 2025-05-18].
- [32] SIMONI, Thomas a ĎALŠÍ. *Deep RL Course - The Deep Q-Network (DQN)*. Online. HuggingFace.co. 2024. Dostupné z: <https://huggingface.co/learn/deep-rl-course/unit3/from-q-to-dqn>. [cit. 2025-05-19].
- [33] SIMONI, Thomas. *Deep RL Course - The Deep Q-Learning Algorithm*. Online. Vydavatel' nenájdenny. 2024. Dostupné z: <https://huggingface.co/learn/deep-rl-course/unit3/from-q-to-dqn>. [cit. 2025-05-19].
- [34] SCHAUL, Tom; QUAN, John; ANTONOGLU, Ioannis a SILVER, David. *Prioritized Experience Replay*. Online. ArXiv.org. 2015. Dostupné z: <https://arxiv.org/abs/1511.05952v4>. [cit. 2025-05-19].
- [35] CODECAT, Melissa; MAIER, David B.; FAEJR a SYLAE. *Openplanet*. Online. OpenPlanet.dev. 2016. Dostupné z: <https://openplanet.dev/>. [cit. 2025-05-19].
- [36] BOUTEILLER, Yann. *TMRL Repository*. Online. GitHub. 2021. Dostupné z: <https://github.com/trackermania-rl/tmrl>. [cit. 2025-05-19].

- [37] BOUTEILLER, Yann. *TMRL Repository - Get Started*. Online. GitHub. 2021. Dostupné z: https://github.com/trackmania-rl/tmrl/blob/master/readme/get_started.md. [cit. 2025-05-19].
- [38] UBISOFT a NADEO. *Trackmania (2020)*. Online. Ubisoft Store. 2020. Dostupné z: <https://store.ubisoft.com/eu/trackmania-starter-access/5e8b58345cdf9a12c868c878.html>. [cit. 2025-05-21].
- [39] UBISOFT a NADEO. *Trackmania (2020)*. Online. Steam Store. 2023. Dostupné z: <https://store.steampowered.com/app/2225070/Trackmania/>. [cit. 2025-05-21].
- [40] SIMONI, Thomas. *Deep RL Course - From Q-Learning to Deep Q-Learning*. Online. HuggingFace.co. 2024. Dostupné z: <https://huggingface.co/learn/deep-rl-course/unit3/deep-q-algorithm>. [cit. 2025-05-22].
- [41] V.KUMAR, Vaishak. *Soft Actor-Critic Demystified*. Online. Medium. 2019. Dostupné z: <https://medium.com/data-science/soft-actor-critic-demystified-b8427df61665>. [cit. 2025-05-23].
- [42] ROBINSON, Joseph Ph.D. *Mastering Polyak Averaging: A Hidden Gem for Smoother and Faster Convergence in Optimization*. Online. Medium. 2025. Dostupné z: <https://jvision.medium.com/mastering-polyak-averaging-a-hidden-gem-for-smoother-and-faster-convergence-in-optimization-676657c72348>. [cit. 2025-05-24].
- [43] ACHIAM, Josh a ABBEEL, Pieter. *Soft Actor-Critic*. Online. OpenAI. 2018. Dostupné z: <https://spinningup.openai.com/en/latest/algorithms/sac.html>. [cit. 2025-05-24].
- [44] NEZNÁMY. *What is bootstrapping in reinforcement learning?* Online. Milvus. 2024. Dostupné z: <https://milvus.io/ai-quick-reference/what-is-bootstrapping-in-reinforcement-learning>. [cit. 2025-05-24].
- [45] AN, Qing; DICK, Chris a SEGARRA, Santiago. *A Deep Reinforcement Learning-Based Resource Scheduler for Massive MIMO Networks*. Online. ResearchGate. 2023. Dostupné z: https://www.researchgate.net/publication/373835519_A_Deep_Reinforcement_Learning-Based_Resource_Scheduler_for_Massive_MIMO_Networks. [cit. 2025-05-24].
- [46] BUCKMAN, Jacob. *How To Think About Replay Memory*. Online. Jacob Buckman. 2021. Dostupné z: <https://jacobbuckman.com/2021-02-13-how-to-think-about-replay-memory/>. [cit. 2025-05-24].

- [47] YANN BOUTEILLER. *Soft Actor-Critic: a beginner-friendly introduction*; YouTube. Online. Dostupné z: <https://www.youtube.com/watch?v=LN29DDIHp1U>. [cit. 2025-05-24].
- [48] JAYAKODY, Dilith. *The Reparameterization Trick - Clearly Explained*. Online. Dilith Jayakody. 2023. Dostupné z: <https://dilithjay.com/blog/the-reparameterization-trick-clearly-explained#reparameterization-trick>. [cit. 2025-05-25].
- [49] BELCIC, Ivan a STRYKER, Cole. *What is catastrophic forgetting?* Online. IBM. 2025. Dostupné z: <https://www.ibm.com/think/topics/catastrophic-forgetting>. [cit. 2025-05-28].
- [50] KIRKPATRICK, James; PASCANU, Razvan; RABINOWITZ, Neil a A ĎALŠÍ. *Overcoming catastrophic forgetting in neural networks*. Online. PNAS. 2017. Dostupné z: <https://www.pnas.org/doi/abs/10.1073/pnas.1611835114>. [cit. 2025-05-28].
- [51] ZHANG, Tiantian; LIANG, Bin a YUAN, Bo. *Catastrophic Interference in Reinforcement Learning: A Solution Based on Context Division and Knowledge Distillation*. Online. ArXiv.org. 2021. Dostupné z: <https://arxiv.org/abs/2109.00525v2>. [cit. 2025-05-28].

SEZNAM OBRÁZKŮ

Obrázok 1: Autonómny taxík firmy Waymo [26]	22
Obrázok 2: Markovov rozhodovací proces [27]	23
Obrázok 3: Ukážka toho, čo vidí model (bez textu)[22]	25
Obrázok 4: Ukážka trate One Hella Long Pipe [25]	27
Obrázok 5: Ukážka prostredia LIDAR	28
Obrázok 6: Úvodný komentár súboru custom_actor_module.py	29
Obrázok 7: Ukážka časti obsahu súboru config.json	30
Obrázok 8: Rozpoloženie okien hry a príkazových riadkov	31
Obrázok 9: Výpočet TD-target (Cieľ TD) [33]	33
Obrázok 10: Diagram princípu DQN	35
Obrázok 11: Porovnanie štruktúry Q-Learning a Deep Q-Learning [40]	35
Obrázok 12: Vzorec odhadu Q-hodnôt [33]	36
Obrázok 13: Štruktúra Soft Actor-Critic [45]	38
Obrázok 14: Ukážka hlavného menu hry Trackmania 2020	45
Obrázok 15: Ukážka okna „Video“ v sekcii „Settings“	46
Obrázok 16: Proces zmeny regiónu pre zmenenie vzhľadu vozidla	47
Obrázok 17: Stránka pre stiahnutie inštalačného súboru platformy OpenPlanet	48
Obrázok 18: Inštalácia platformy OpenPlanet	48
Obrázok 19: Štruktúra priečinku TmrlData	49
Obrázok 20: Presun priečinku Plugins do priečinku OpenplanetNext	49
Obrázok 21: Ukážka pozície riadkov potrebných na komentovanie	50
Obrázok 22: Inštalačné nastavenia pre PyTorch CUDA	51
Obrázok 23: Časť kódu funkcie SACTrainingAgent	52
Obrázok 24: Kód funkcie VanillaCNN	54
Obrázok 25: Kód funkcie mlp	55
Obrázok 26: List možných akcií, prevod zo spojitého na diskretný charakter	55
Obrázok 27: Inicializácia funkcie DQNNetwork	56
Obrázok 28: Funkcia forward v DQNNetwork	56
Obrázok 29: Inicializácia DQN siete a hodnoty epsilon v triede DQNActorModule	57
Obrázok 30: Začiatok funkcie act v triede DQNActorModule	57
Obrázok 31: Logika výberu Q-hodnoty	57
Obrázok 32: Znižovanie hodnoty epsilon	58
Obrázok 33: Výber akcie z listu	58
Obrázok 34: Funkcia pre úpravu hodnoty epsilon-u na základe momentálnej epochy	58

Obrázok 35: Definícia triedy DQNTrainingAgent a vytvorenie kópie modelu.....	59
Obrázok 36: Inicializácia premenných v triede DQNTrainingAgent	59
Obrázok 37: Funkcia get_actor	59
Obrázok 38: Definovanie premenných a konvertovanie tuplov na tensory	60
Obrázok 39: Indexácia akcií a výpočet curr_q a target_q	60
Obrázok 40: Výpočet straty, spätná propagácia a aktualizácia váh	61
Obrázok 41: Frekvencia presunu váh z hlavnej siete do cieľovej	61
Obrázok 42: Výpočet TD-chyby a úprava priorít vzoriek pre PER.....	62
Obrázok 43: Výstupný slovník funkcie train()	62
Obrázok 44: Celý kód funkcie train()	62
Obrázok 45: Inicializácia hodnôt v triede PrioritizedMemory	63
Obrázok 46: Funkcia append_buffer.....	63
Obrázok 47: Funkcia sample_indices	64
Obrázok 48: Iterátor na konci triedy PrioritizedMemory	64
Obrázok 49: Trieda PERTrainingOffline.....	65
Obrázok 50: Zmena hodnôt MAX_EPOCHS a ROUNDS_PER_EPOCH	65
Obrázok 51: Zmena hodnôt ENVIRONMENT_STEPS_BEFORE_TRAINING a RW_MAX_SAMPLES_PER_EPISODE	66
Obrázok 52: Zmena hodnoty UPDATE_MODEL_INTERVAL.....	66
Obrázok 53: Zapnutie zaznamenávania individuálnych pokusov počas tréningu	66
Obrázok 54: Zmena hodnoty ep_max_length.....	66
Obrázok 55: Pridanie malej penalizácie každého kroku	67
Obrázok 56: Úprava a pridanie parametrov algoritmu	67
Obrázok 57: Definícia užitočných konštánt.....	68
Obrázok 58: Kód funkcie process_observation	69
Obrázok 59: Inicializácia triedy MLPNetwork.....	70
Obrázok 60: For cyklus pre tvorenie vrstiev.....	71
Obrázok 61: Kód funkcie forward().....	71
Obrázok 62: Inicializácia triedy LidarSACPolicy	72
Obrázok 63: Inicializácia siete, dodatočných vrstiev a ich váh	72
Obrázok 64: Začiatok funkcie forward v LidarSACPolicy	73
Obrázok 65: Vzorkovanie akcií vo funkcii forward	74
Obrázok 66: Výpočet akcií z pozorovania.....	74
Obrázok 67: Inicializácia Q-funkcie	75
Obrázok 68: Funkcia forward v triede LidarQFunction	76

Obrázok 69: Kód triedy LidarSACActorCritic.....	76
Obrázok 70: Inicializácia triedy ImprovedSACAgent.....	77
Obrázok 71: Implementovanie automatického nastavenia entropie	78
Obrázok 72: Vytvorenie optimalizátorov pre aktéra a kritikov	78
Obrázok 73: Implementácia plánovačov rýchlostí učenia.....	78
Obrázok 74: Funkcia get_actor v ImprovedSACAgent.....	78
Obrázok 75: Začiatok funkcie train	79
Obrázok 76: Úprava odmeny	79
Obrázok 77: Kontrola hodnoty buffer_warmup	80
Obrázok 78: Vzorkovanie a výpočet cieľových Q-hodnôt	80
Obrázok 79: Výpočet straty a aktualizácia kritikov.....	81
Obrázok 80: Zmrazenie Q-sietí a aktualizovanie aktéra.....	81
Obrázok 81: Automatické nastavenie entropie	82
Obrázok 82: Aktualizácia cieľovej siete a výstupné metriky	82
Obrázok 83: Nastavenie parametrov tréningového agenta	83
Obrázok 84: Spustenie zberu dát do wandb.....	84
Obrázok 85: Nastavenie hodnôt MAX_EPOCHS a ROUNDS_PER_EPOCH	84
Obrázok 86: Nastavenie hodnoty TRAINING_STEPS_PER_ROUND	84
Obrázok 87: Nastavenie hodnoty ENVIRONMENT_STEPS_BEFORE_TRAINING	85
Obrázok 88: Nastavenie hodnoty MEMORY_SIZE	85
Obrázok 89: Úprava hodnôt GAMMA a POLYAK.....	85
Obrázok 90: Úprava hodnôt rýchlostí učenia	85
Obrázok 91: Nastavenie hodnôt automatického ovládania entropie	86
Obrázok 92: Pridanie vlastných parametrov.....	86
Obrázok 93: Zmena prostredia	87
Obrázok 94: Spustenie hry Trackmania cez platformu Steam	88
Obrázok 95: Premiestnenie herného okna do ľavého horného rohu [37]	89
Obrázok 96: Umiestnenie testovacej trate	89
Obrázok 97: Spustenie testovacej trate	90
Obrázok 98: Priečink so zdrojovými súbormi	90
Obrázok 99: Napísanie „cmd“ do poľa s cestou k priečinku.....	90
Obrázok 100: Ukážka otvoreného okna s príkazovým riadkom.....	91
Obrázok 101: Rozpoloženie okna hry s oknami príkazových riadkov	91
Obrázok 102: Ukážka správneho pohľadu pre prostredie TM20LIDAR	92
Obrázok 103: Spustenie nahrávania vlastnej odmeny	93

Obrázok 104: Hláška pre začiatok nahrávania.....	93
Obrázok 105: Konečný výpis štatistík nahranej odmeny.....	93
Obrázok 106: Nastavenie užívateľských údajov pre wandb	94
Obrázok 107: Zobrazovanie zbieraných tréningových dát na stránke wandb	94
Obrázok 108: Graf dĺžky tréningového kola naprieč tréningom	96
Obrázok 109: Graf odmeny počas tréningového kola	96
Obrázok 110: Graf veľkosti pamäte.....	97
Obrázok 111: Strata Q-funkcie	97
Obrázok 112: Dĺžka tréningového kola pri DQN s alternatívnymi hodnotami	98
Obrázok 113: Porovnanie dĺžky tréningového kola medzi oboma verziami	99
Obrázok 114: Výška odmeny počas tréningového kola pri alternatívnom DQN	99
Obrázok 115: Porovnanie odmeny oboch verzií DQN	100
Obrázok 116: Porovnanie straty Q-funkcie medzi oboma verziami DQN	101
Obrázok 117: Graf straty kritika	102
Obrázok 118: Graf straty aktéra.....	102
Obrázok 119: Priemery hodnôt Q-sietí	103
Obrázok 120: Porovnanie rýchlosti poklesu hodnôt LR medzi aktérom a kritikmi	104
Obrázok 121: Entropia aktéra algoritmu SAC.....	104
Obrázok 122: Hodnota teploty – α – algoritmu SAC	105
Obrázok 123: Odmena získaná počas tréningového kola pri algoritme SAC.....	105
Obrázok 124: Vyfiltrovanie neúspešných pokusov v programe Everything	106
Obrázok 125: Program pre hromadné premenovávanie súborov.....	107
Obrázok 126: Premenované súbory v priečinku	107
Obrázok 127: Pôvodný vzhľad trate	108
Obrázok 128: Upravený vzhľad trate.....	109

SEZNAM TABULEK

Tabuľka 1: Výsledky projektu Trackmania_AI.....	26
Tabuľka 2: Porovnanie najazdených časov	110

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
AI	Artificial Intelligence
DQN	Deep Q-Network
SAC	Soft Actor-Critic
PPO	Proximal Policy Optimization
IQN	Implicit Quantile Network
TAS	Tool-Assisted Speedrun
LR	Learning Rate
TD	Temporal Difference
CNN	Convolutional Neural Network
MLP	Multilayer Perceptron
LiDAR	Light Detection and Ranging
PER	Prioritized Experience Replay
TMRL	TrackMania Reinforcement Learning
RAM	Random-Access Memory
RNG	Random Number Generator

SEZNAM PŘÍLOH

Príloha P I: Diplomova-Praca-TM.rar

PŘÍLOHA P I: NÁZEV PRVNÍ PŘÍLOHY

Grafy

|_____ Grafy – DQN

|_____ DQN-Long-Round

|_____ *episode-length-train-dqn-long.png*

|_____ *episode-return-long-dqn.png*

|_____ DQN-Short-Round

|_____ *episode-return-short-dqn.png*

|_____ *loss-short-dqn.png*

|_____ *memory-len-short-dqn.png*

|_____ *round-time-short-dqn.png*

|_____ Porovnanie

|_____ *episode-length-train-comparison.png*

|_____ *episode-return-comp.png*

|_____ *loss-comp.png*

|_____ Grafy – SAC

|_____ *alpha-sac.png*

|_____ *entropy-act-sac.png*

|_____ *loss-actor-sac.png*

|_____ *loss-critic-sac.png*

|_____ *lr-sac.png*

|_____ *q-mean-sac.png*

|_____ *reward-sac.png*

Konfiguračné súbory

|_____ *config-dqn.json*

|_____ *config-sac.json*

Nahrávky

|_____ DQN

|_____ *DQN-Train-Short-Start-Montage.mp4*

|_____ *DQN-Train-Short-Mid-Montage.mp4*

|_____ *DQN-Train-Short-End-Montage.mp4*

|_____ *DQN-Train-Long-Start-Montage.mp4*

|_____ *DQN-Train-Long-Mid-Montage.mp4*

|_____ *DQN-Train-Long-End-Montage.mp4*

|_____ SAC

|_____ *SAC-Train-Start-Montage.mp4*

|_____ *SAC-Train-Mid-Montage.mp4*

|_____ *SAC-Train-End-Montage.mp4*

|_____ *SAC-Alt-Track-Blind.mp4*

|_____ *Human-vs-SAC-vs-DQN.mp4*

Odmena

|_____ *reward-dqn.pkl*

|_____ *reward-longtrack.pkl*

|_____ *reward-sac.pkl*

Tréningový kontrolný bod

|_____ DQN

|_____ *DQN_TEST_RUN_10-LOGGING_t.tcpt*

|_____ *DQN_TEST_RUN_12-LOGGING_t.tcpt*

|_____ SAC

|_____ *SAC_TEST_RUN-13-LOGGING-DEBUG_t.tcpt*

Trénované váhy

|_____ DQN

|_____ *DQN_TEST_RUN_10-LOGGING.tmod*

|_____ *DQN_TEST_RUN_10-LOGGING_t.tmod*

|_____ *DQN_TEST_RUN_12-LOGGING.tmod*

|_____ *DQN_TEST_RUN_12-LOGGING_t.tmod*

|_____ SAC

|_____ *SAC_TEST_RUN-13-LOGGING-DEBUG.tmod*

|_____ *SAC_TEST_RUN-13-LOGGING-DEBUG_t.tmod*

Využité trate

|_____ *TestTrack-Edit.Map.Gbx*

|_____ *tmrl-test.Map.Gbx*

Zdrojové súbory

|_____ *custom_actor_module_dqn.py*

|_____ *custom_actor_module_sac.py*