UNIVERSITY OF BUCHAREST

FACULTY OF
MATHEMATICS AND
INFORMATICS

COMPUTER SCIENCE SPECIALIZATION

BACHELOR'S THESIS

REINFORCEMENT LEARNING ÎN JOC 3D (ROCKET LEAGUE): FRAM

Graduate Student

Oană Alexandru Andrei

Scientific Coordinator

Radu-Ștefan Mincu

Bucharest, June 2023

Rezumat

Această teză explorează antrenamentul și performanța unui agent antrenat prin Reinforcement Learning (RL), care învață să joace jocul video Rocket League, concentrându-se pe modul de joc 1v1. În contextul optimizării performanței agentului într-un mediu atât de complex și dinamic, algoritmul selectat pentru antrenare este *Proximal Policy Optimization (PPO-Clip)*. Studiul evidențiază evoluția agentului în cadrul unui mediu simulat, obținută exclusiv prin self-play. În cadrul lucrării sunt prezentate metodele utilizate pentru a dezvolta un agent competent să joace Rocket League, într-un timp relativ scurt și cu resurse computaționale limitate.

Abstract

This thesis explores the training and performance of an agent trained through Reinforcement Learning (RL), that learns to play the video game Rocket League, with a focus on the 1v1 game mode. In the context of optimizing the agent's performance in such a complex and dynamic environment, the algorithm selected for training is *Proximal Policy Optimization (PPO-Clip)*. The study highlights the agent's evolution within a simulated environment, achieved solely through self-play. The paper presents the methods used to develop a competent Rocket League agent, in a relatively short amount of time and with limited computational resources.

Cuprins

Capitolul 1

Introducere

1.1 Reinforcement Learning

Reinforcement Learning este una dintre cele 3 paradigme principale în Machine Learning, alături de Supervised Learning și Unsupervised Learning. În cazul problemelor rezolvate prin Supervised Learning, modelul primește un set de date, alături de predicții/clasificări corecte, scopul fiind reducerea erorii de predicție/clasificare a modelului. În cazul Unsupervised Learning, agentul primește doar un set de date, scopul urmărit fiind găsirea anumitor tipare sau asocieri în setul de date. Probleme populare unde este folosită învățarea nesupervizată sunt clustering și detecția de anomalii [1].

Problemele de Reinforcement Learning constau în antrenarea agenților în scopul maximizării unui obiectiv, într-un mediu de învățare. Obiectivul agentului este recompensa cumulativă primită într-o perioadă de timp. Diferența principală dintre Reinforcement Learning și alte domenii din Machine Learning constă în sursa datelor de intrare primite de agent. În cazul Reinforcement Learning, modelul nu primește un set de date sau informații deja etichetate, însă datele sunt generate în mod automat de către agent prin experimentarea în mediul de antrenare. Agentul acționează conform strategiei sale curente (policy), strategia fiind actualizată iterativ pentru maximizarea funcției de obiectiv. Astfel, este creat un "feedback loop" între agent și mediu, ilustrat in **Figura 1.1**. Agentul acționează asupra mediului și mediul trimite înapoi un semnal, reprezentând recompensa pentru starea în care se află [2].

Algoritmi de Reinforcement Learning au fost utilizați cu succes pentru a obține agenți performanți în jocuri precum: Go [3], Dota 2 [4], Șah [5], jocuri Atari [6]. De asemenea, metodele de Reinforcement Learning sunt explorate și în domenii precum robotică, vehicule autonome, modele de limbaj natural.

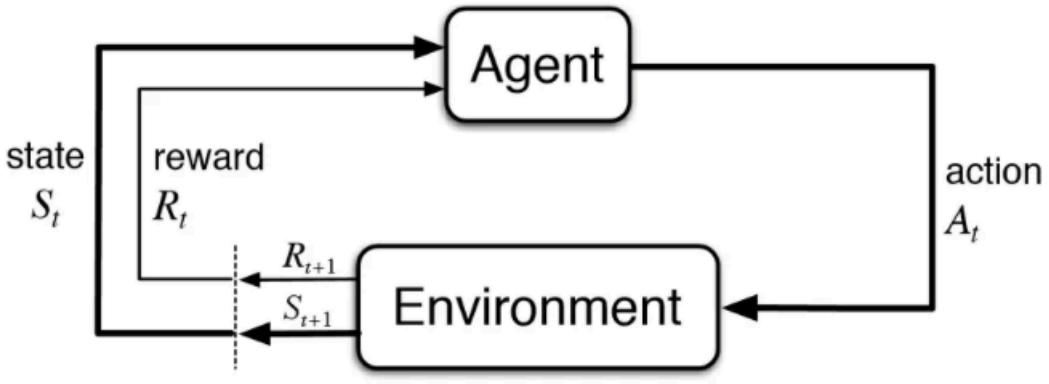

Figura 1.1: RL feedback loop

Algoritmii de Reinforcement Learning pot fi clasificați în 2 categorii: model-based și model-free [7].

Un algoritm model-based presupune ca agentul să aibă acces sau să învețe un model al mediului de antrenare. Un model al mediului este o funcție ce prezice tranzitiile dintre stări și recompensele primite. Acest algoritm permite agentului să planifice acțiuni, folosind metode de căutare pentru a observa consecințele acțiunilor posibile în starea curentă. În practică, este dificil de implementat un model perfect al mediului, iar învățarea mediului de către agent poate rezulta in performanță scăzută în mediul real. Un exemplu antrenat cu succes folosind un astfel de algoritm este AlphaZero, în șah [7].

Algoritmii model-free sunt cei în care agentul nu dispune de un model al mediului de antrenare și nici nu încearcă să învețe caracteristicile acestuia. Aceștia se clasifică mai departe în 2 categorii, în funcție de abordarea aleasă pentru obiectul optimizării: optimizatori de policy (strategie) sau algoritmi tip Q-Learning [7].

Algoritmii de tip Q-Learning presupun învățarea funcției ce estimează recompensa maximă posibilă în starea s, dacă agentul execută acțiunea a și urmează cea mai bună strategie mai departe. Optimizarea acestui estimator este efectuată de obicei ”off-policy”, adică datele utilizate pentru optimizare nu sunt neapărat colectate prin strategia curentă de acțiuni. Din acest motiv, Q-Learning este considerat eficient, întrucât poate reutiliza experiențe anterioare pentru optimizarea estimării. Pe de altă parte, faptul că optimizarea nu este realizată în mod direct pentru policy-ul agentului este un factor ce introduce instabilitate față de algoritmii care optimizează în mod direct strategia [7].

Algoritmii de tip Policy Optimization implică optimizarea directă a policy-ului agentului. Acești algoritmi învață în mod explicit o funcție de policy ce returnează acțiunea efectuată de agent pentru o stare primită. Parametrii policy-ului sunt de obicei optimizați pentru maximizarea funcției de obiectiv. Adițional, mulți algoritmi de acest tip optimizează și o funcție de evaluare ce estimează o valoare pentru o stare primită. Această funcție este utilizată în specificarea obiectivului urmărit pentru actualizarea policy-ului [8]. Algoritmul de învățare utilizat în cadrul acestei lucrări (PPO) este de tip Policy Gradient, încadrându-se în această categorie.

1.2 Rocket League

Rocket League este un joc video lansat în iulie 2015, ce poate fi descris ca "fotbal cu mașini propulsate" [9]. În joc există 2 echipe rivale: echipa albastră și echipa portocalie, iar echipa câștigătoare este cea care a înscris cele mai multe goluri după 5 minute de joc. În cazul în care scorul este egal după 5 minute, se joacă în mod suplimentar până una dintre echipe marchează un gol. Arena în care sunt desfășurate meciurile include 2 porți, pereți și un tavan, aceasta fiind ilustrată in **Figura 1.2**. Baza pereților este curbată, fiind ușor de accesat de către mașini, iar mingea rămâne mereu în interiorul arenei. Jucătorii pot colecta boost din 6 boost-uri mari a câte 100 boost sau din 28 de boost-uri mici, fiecare oferind 12 boost. Toate boost-urile se găsesc pe teren și, odată colectate de către un jucător, reapar după 10 secunde (boost-uri mari) sau 4 secunde (boost-uri mici) [10]. Jocul începe într-una dintre cele 5 poziții de kick-off, cu mingea în centrul terenului și jucătorii având 34 boost [11]. Modurile de joc competitive de tip "soccar" sunt 1v1, 2v2 și 3v3.

Fiecare jucător controlează o singură mașină, care are unul dintre cele 6 hitbox-uri standard. Mașinile nu diferă în performanță, însă fiecare hitbox are dimensiuni diferite, fapt ce amplifică complexitatea jocului [12]. Pentru această lucrare, agentul folosește exclusiv hitbox-ul de "Octane", fiind cel mai popular hitbox din joc. Acesta este ilustrat în **Figura 1.3**.

În ceea ce privește controlul mașinii, aceasta poate fi controlată în mare parte ca un vehicul cât timp este pe sol, și ca un avion cât timp este în aer. Pentru a putea zbura, jucătorul controlează simultan ruliul, girația, tangajul și tracțiunea mașinii (prin boost). Adițional, mașina poate sări de 2 ori pentru a "decola" sau pentru a se propulsa într-o anumită direcție. Cât timp se află pe perete, mașina este "prinsă" de acesta, însă odată ajunsă pe tavan, aceasta se desprinde. De asemenea, după un anumit prag de viteză, mașina este considerată în regim "supersonic". În cazul în care un jucător se află în regim "supersonic", acesta poate distruge mașina unui oponent prin impact. Odată distrusă, o

Figura 1.2: Arena Rocket League: DeadEyeCanion Oasis.

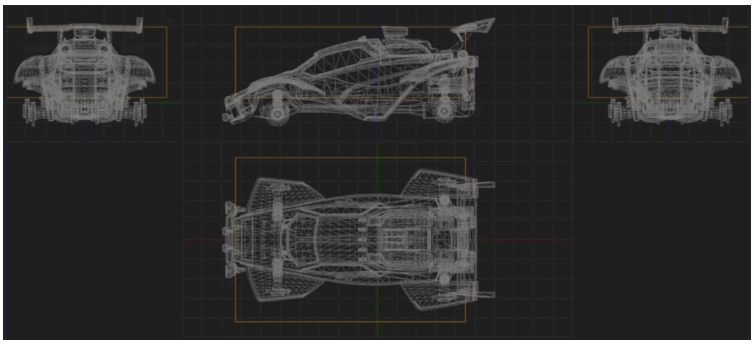

Figura 1.3: Hitbox Octane [13]

mașină revine în joc după 3 secunde într-o poziție defensivă, pe partea stângă sau dreaptă a terenului [14]. Astfel, singurul factor aleatoriu ce influențează rezultatul unui meci de Rocket League este poziția în care reapare mașina după ce a fost distrusă.

Jucătorii de Rocket League înlanțuiesc secvențe de acțiuni pentru a executa ”mecanici” specifice, în scopul de a câștiga. Printre mecanicile comune se numără:

- flip direcțional (săritură + direcție): folosit în mișcare, dar și pentru a șuta
- half-flip (flip în spate + air roll): pentru a întoarce poziția mașinii
- single-jump (săritură): folosit pentru propulsie pe sol / decolare
- double-jump (2x săritură): folosit pentru propulsie rapidă
- air-spin: spin în aer în jurul axei y, altitudinea este păstrată prin boost și tangaj

- wave-dash (săritură + direcție înspre suprafața de contact): folosit pentru ”prinderea” rapidă de suprafețe la aterizare
- mecanici de recuperare: nu reprezintă o mecanică în sine, dar înaglobează toate manevrele prin care mașina este readusă rapid în poziția dorită (aterizări, sărituri rapide de pe o suprafață pe cealaltă, wave-dash-uri etc.)

Pe lângă partea mecanică a jocului, jucătorii de Rocket League execută strategii și decizii rapide pentru a obține avantajul în cadrul unui meci. Fiind un mediu complex și dinamic, Rocket League este pe de-o parte un mediu dificil de învățat, dar și o oportunitate de a prezenta progresul posibil prin Reinforcement Learning.

1.3 Motivație

Agenții robotici au un impact semnificativ în multe jocuri, grație progreselor impresionante atinse. În șah, de exemplu, agenții robotici au depășit limita umană de performanță, cei mai buni dintre ei fiind estimați la un rating de peste 3500 elo [15]. În comparație, cel mai mare rating atins vreodată de un jucător uman este de 2882 elo, obținut de Magnus Carlsen în Mai 2014 [16]. Datorită acestei discrepanțe de performanță, roboții sunt utilizați în asistarea oamenilor atât pentru analiza jocurilor, cât și pentru antrenament, oferind idei noi atât în faza teoretică a meciului de șah (deschidere), cât și ulterior. De exemplu, meciul pentru campionatul mondial de șah a fost analizat de Mari Maeștri, alături de Stockfish, pentru evaluarea pozițiilor și a mișcărilor jucătorilor. Un alt exemplu îl reprezintă Pluribus, un agent AI dezvoltat de cercetătorii AI de la Facebook în parteneriat cu Universitatea Carnegie Mellon. Pluribus este primul agent AI care a reușit să învingă jucători profesioniști într-o competiție de poker, fiind antrenat și prin Reinforcement Learning [17].

În Rocket League, cei mai performanți agenți dezvoltați sunt antrenați prin Reinforcement Learning. Botul Nexto, în special, a atins o performanță excelentă la jocul pe sol, cuplat cu viteza instantă de reacție și cu ”flickuri” precise. Cu toate acestea, deși ”flickurile” și preluările lui Nexto pot fi considerate superumane, rating-ul estimat al acestuia este în jur de Grand Champion 1, top 0.5% din jucători [18]. Ulterior lansării lui Nexto, au apărut alți agenți cu performanță îmbunătățită, precum Seer. Aceștia nu pot fi publicați, din cauza problemelor întâmpinate de Rocket League cu folosirea roboților în mod abuziv în jocuri competitive online. Din acest motiv, este dificil de estimat rank-ul maxim atins de agenții artificiali în Rocket League, după apariția lui Nexto. Din sursele video limitate, Seer a reușit să câștige împotriva unor jucători de SSL, deci se pare că rank-ul lui ar putea depăși Grand Champion 2/3, top 0.2%. Cu toate acestea, chiar și

boți performanți precum Seer și Nexto pot fi exploatați din cauza jocului aerian deficitar, a gestionării slabe a boost-ului și a predictibilității deciziilor luate. Adițional, agenții au obținut cea mai bună performanță în modul 1v1, performanța lor în 3v3 fiind încă neclară. Din sursele video limitate, Nexto pare să obțină o performanță similară și în 3v3, însă este evident că jocurile de echipă introduc un nivel superior de complexitate pentru roboți [19].

Datorită rezultatelor obținute prin Reinforcement Learning în jocuri, dar și a faptului că agenții robotici încă nu sunt la nivelul jucătorilor profesioniști în Rocket League, dezvoltarea unui agent competent în Rocket League prezintă atât provocări tehnice, cât și potențial de inovație. Decizia de a limita antrenarea la modul de 1v1 are scopul de a simplifica obiectivul agentului, dezvoltarea unui agent capabil să joace 2v2 sau 3v3 fiind considerată prea dificilă pentru timpul alocat. Această lucrare prezintă procesul complet de implementare a unui astfel de robot, performanța obținută și dificultățile întâmpinate. De asemenea, agentul ar putea fi utilizat ca un instrument de învățare pentru jucătorii umani, ajutându-i să-și îmbunătățească strategiile și performanțele prin antrenament împotriva unui adversar artificial. Chiar dacă agentul nu obține o performanță impresionantă, acesta poate fi utilizat pe post de partener de antrenament pentru un jucător uman de rank similar, în special dacă jucătorul dorește să învețe din punctele forte ale agentului. De exemplu, unul dintre cei mai buni jucători de 1v1 din Rocket League a jucat împotriva lui Nexto, urmărind să își îmbunătățească jocul la sol [20].

Capitolul 2

Implementare

Pentru a implementa un agent capabil să joace Rocket League, prin Deep Reinforcement Learning, au fost urmăriți următorii pași:

1. Formalizarea problemei într-un cadru matematic - *Proces de decizie Markov*
2. Configurarea mediului de învățare
3. Antrenarea modelului
4. Evaluarea modelului

Acest proces de dezvoltare este inspirat din lucrări precum Seer [21] și lucrarea originală PPO [22], în cadrul cărora agenții au fost antrenați cu succes. În cele ce urmează, vor fi prezentați pașii de implementare enumerați anterior.

2.1 Formalism

Pentru a utiliza un algoritm de Reinforcement Learning, o reprezentare comună a problemei este ca un *Proces de decizie Markov*. Beneficiul principal al acestei reprezentări constă în proprietatea Markov, conform căreia acțiunile luate sunt independente de istoricul tranzitiilor dintre stări. Tratarea problemei de învățare în Rocket League prin acest cadru presupune modelarea unei observații expresive pentru agent, în care să se regăsească toate informațiile necesare efectuării unei acțiuni în starea respectivă. Un beneficiu adițional constă în simplificarea problemei și în reducerea resurselor computaționale necesare stocării stărilor precedente.

2.1.1 Terminologie

Pentru a formaliza problema în termeni matematici, este util să definim o parte din termenii specifici problemelor de Reinforcement Learning, ce vor fi utilizați pe parcursul lucrării [8]:

- Agent - Factor ce influențează mediul prin acțiuni
- Mediu - Spațiul în care agentul ia acțiuni și primește recompense
- Pas - Marchează o singură tranziție dintr-o stare s_t în starea s_{t+1}
- Stare - O descriere completă a mediului, după t pași
- Observație - O descriere parțială a mediului, după t pași; percepția agentului
- Recompensă - Un scor primit de agent pentru fiecare stare în care se află
- Episod/Traiectorie - O secvență de stări și acțiuni, încheiată într-o stare terminală
- Stare terminală - Ultima stare dintr-un episod
- Factor de discount - Parametru care penalizează recompensele întârziat
- Policy - Funcția de decizie a agentului, returnează o acțiune pentru o stare primită
- Funcție de evaluare - Estimarea "valorii" unei stări sau a unei perechi (stare, acțiune)

2.1.2 Proces de decizie Markov

Procesul de decizie Markov este reprezentat printr-un cvintuplu, $\langle S, A, R, P, \rho_0 \rangle$, unde [8]:

- S este setul tuturor stărilor valide,
- A este setul tuturor acțiunilor valide,
- $R : S \times A \times S \to \mathbb{R}$ este funcția de recompensă, cu $r_t = R(s_t, a_t, s_{t+1})$,
- $P : S \times A \to \mathcal{P}(S)$ este funcția de probabilitate de tranziție, cu $P(s'|s, a)$ fiind probabilitatea de a trece în starea s' dacă începi în starea s și iei acțiunea a,
- ρ_0 este distribuția stării de început.

Un sistem astfel definit este considerat **Markov** dacă îndeplinește următoarea condiție: Tranzițiile depind doar de starea și acțiunea cea mai recentă, și nu de stări sau acțiuni anterioare. În termeni formali, pentru orice pas t, stare s_t și acțiune a_t în cadrul unui episod, funcția de tranziție este exprimată prin:

$$f(s_t, a_t) = s_{t+1} \tag{2.1}$$

Utilizarea acestui model matematic permite estimarea valorii unei stări oarecare, sau a unei acțiuni oarecare a, executată în starea s, prin intermediul ecuațiilor *Bellman*:

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi, s'\sim P}[r(s,a) + \gamma V^{\pi}(s')] \tag{2.2}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{s'\sim P}[r(s,a) + \gamma \mathbb{E}_{a'\sim\pi}[Q^{\pi}(s',a')]], \tag{2.3}$$

unde:

- π este policy-ul actual
- $V^{\pi}(s)$ este funcția de evaluare a stării s sub policy-ul π. Aceasta reprezintă valoarea estimată a recompensei cumulative, începând din starea s și urmând policy-ul π.
- s este starea curentă a agentului.
- a este acțiunea luată de agent.
- $\mathbb{E}$ este valoarea estimată
- P este o funcție care returnează probabilitatea de a ajunge în diferite stări următoare, pentru o anumită stare și acțiune.
- $r(s,a)$ este recompensa primită după efectuarea acțiunii a în starea s.
- γ este factorul de discount. Este un număr între 0 și 1 care reduce valoarea recompenselor viitoare.
- $Q^{\pi}(s,a)$ este funcția de evaluare a acțiunii. Reprezintă valoarea așteptată a recompensei cumulative pentru efectuarea acțiunii a în starea s și urmând strategia curentă π după aceea.
- s' este starea următoare după ce a fost executată acțiunea curentă.
- a' este acțiunea următoare luată conform policy-ului curent π. [8]

2.2 Mediu de antrenare

După alegerea modelului matematic, pasul următor constă în implementarea acestuia în cod. În acest scop, sunt utilizate biblioteca RLGym și programul BakkesMod.

RLGym este o bibliotecă Python prin care jocul Rocket League este tratat ca un mediu de tip OpenAI Gym, în scopul antrenării prin Reinforcement Learning. Concret, RLGym expune metode ce facilitează configurarea mediului de învățare, precum: `step()` pentru a efectua o tranziție între stări, `make()` pentru a crea mediul și `reset()` pentru a reseta mediul într-o stare inițială [23].

BakkesMod este un program ce poate modifica setările jocului Rocket League. Prin utilizarea programului, devin disponibile opțiuni care nu există în mod standard, precum: slidere pentru gravitație, opțiuni cosmetice, joc accelerat etc. Adițional, funcționalitatea programului poate fi extinsă prin plugin-uri. Pentru a controla jocul în timp real, RLGym comunică printr-un plugin de BakkesMod [23].

De asemenea, procesul de învățare poate fi accelerat cu ajutorul plugin-ului RLGym pentru BakkesMod. Astfel, jocul poate rula de 100 de ori mai repede decât în mod normal. În plus, RLGym permite lansarea mai multor instanțe de Rocket League în paralel, pentru a acumula mai rapid experiență în mediul de antrenare. **Figura 2.1** ilustrează sistemul de transmitere a informației în mediul de antrenare.

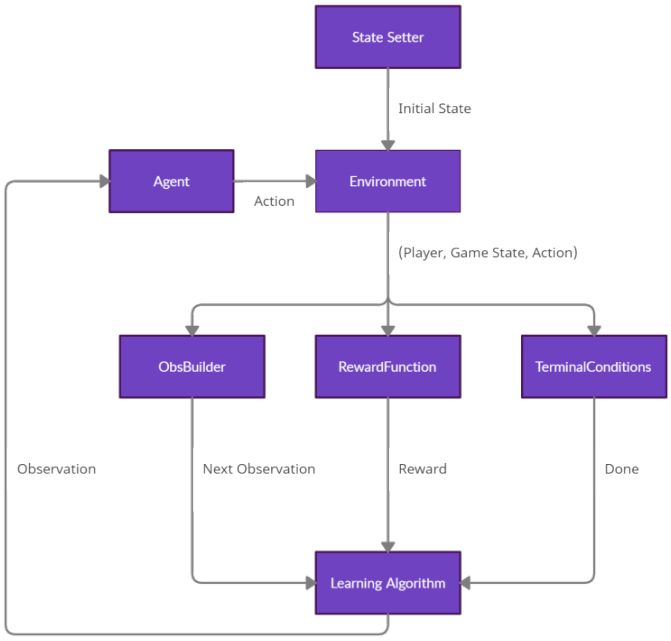

Figura 2.1: Proces de învățare prin recompensă [23]

Implementarea mediului de antrenare constă în **Fragmentele de cod 2.1** și **2.2**.

Fragment de cod 2.1: "Biblioteci folosite"

```
import torch
from rlgym.envs import Match
from rlgym.utils.reward_functions.common_rewards import
    VelocityPlayerToBallReward, RewardIfClosestToBall
from stable_baselines3 import PPO
from stable_baselines3.common.callbacks import CheckpointCallback
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.vec_env import VecMonitor, VecNormalize,
    VecCheckNan
from stable_baselines3.ppo import MlpPolicy

from rlgym.utils.obs_builders import AdvancedObs
from rlgym.utils.state_setters import RandomState, DefaultState
from rlgym.utils.terminal_conditions.common_conditions import
    TimeoutCondition, NoTouchTimeoutCondition, \
     GoalScoredCondition
from rlgym_tools.sb3_utils import SB3MultipleInstanceEnv
from rlgym.utils.reward_functions.common_rewards.misc_rewards import
    EventReward
from rlgym.utils.reward_functions.common_rewards.ball_goal_rewards
    import VelocityBallToGoalReward
from rlgym.utils.reward_functions import CombinedReward

from ReinfLearningBot.environment_config_objects.action_parser import
    CustomActionParser
from ReinfLearningBot.environment_config_objects.observation_builder
    import CustomObs

from enum import Enum
from ReinfLearningBot.environment_config_objects.reward_function import
     CustomBallPlayerDistanceReward
```

Fragment de cod 2.2: ”Configurarea mediului de învățare”

```
def get_match():
    return Match(
        team_size=1,
        reward_function=CombinedReward(
            (
                VelocityPlayerToBallReward(use_scalar_projection=True),
                VelocityBallToGoalReward(use_scalar_projection=True),
                EventReward(
                    goal=1000.0,
                    concede=-1000.0,
                    save=100.0,
                    shot=50.0,
                    demo=20.0,
                    touch=10,
                    boost_pickup=0.5
                ),
            ),
            (0.002, 0.008, 1.0)),
        spawn_opponents=True,
        terminal_conditions=[TimeoutCondition(fps * 30),
            NoTouchTimeoutCondition(fps * 15), GoalScoredCondition()],
        obs_builder=CustomObs(),
        state_setter=RandomState(ball_rand_speed=True, cars_rand_speed=
            True),
        action_parser=CustomActionParser()
    )

    env = SB3MultipleInstanceEnv(get_match, num_instances, wait_time
        =50)
    env = VecCheckNan(env) # Wrapper pentru debugging
    env = VecMonitor(env)  # Wrapper pentru monitorizarea statisticilor
    env = VecNormalize(env, norm_obs=True, gamma=gamma) # Wrapper
        pentru normalizarea recompenselor si a observatiilor
```

2.2.1 Funcție de recompensă

Funcția de recompensă este esențială pentru configurarea mediului de învățare. Pentru modelarea funcției, este urmărit un echilibru între următoarele tipuri de recompense:

- Recompense sporadice vs recompense frecvente
- Recompense zero-sum vs recompense positive-sum

Pentru a alinia scopul agentului cu scopul dorit, este necesar ca funcția de recompensă să maximizeze probabilitatea de a ajunge într-o stare terminală "câștigătoare". În Rocket League, echipa câștigătoare este cea care a înscris mai multe goluri la finalul meciului, deci recompensa principală a agentului ar putea fi oferită în funcție de diferența dintre golurile înscrise și cele primite.

Cu toate acestea, dacă agentul primește recompense exclusiv pentru înscrierea/primirea unui gol, procesul de învățare poate deveni lent și ineficient. Din cauza faptului că agentul învață din propria experiență în mediul simulat, mulți pași de învățare vor fi irosiți dacă nu se înscrie niciun gol în episodul respectiv. Ținând cont că agentul ia decizii aleatorii la început, probabilitatea de a înscrie un gol sau chiar de a atinge mingea într-un episod este mică. Astfel, apare nevoia de a oferi recompense mai frecvente, pentru a spori viteza de învățare, în raport cu numărul de pași de antrenare. Deși agentul primește astfel semnale mai frecvente de la mediu pentru ajustarea performanței, există pericolul ca aceste scopuri "instrumentale" să conducă spre un policy ce nu maximizează scopul terminal (câștigarea meciului).

Pentru a alinia agentul cu scopul de a câștiga meciul, recompensa și penalizarea maximă sunt acordate pentru un gol înscris/primit. Întrucât episodul se încheie imediat ce se marchează un gol, agentul primește un semnal mai puternic în legatură cu stările și acțiunile dinaintea unui gol. Scopul acestui design este ca agentul să învețe mai ușor conexiunile dintre evenimentele importante și cauzele acestora. În plus, agentul este recompensat pentru următoarele evenimente: atingerea mingii, șut pe cadrul porții, salvarea șutului oponentului, distrugerea adversarului, atingerea mingii, acumularea de boost.

Pe lângă recompensele oferite după un eveniment anume, agentul primește 2 recompense pentru fiecare pas din mediul de învățare. Scopul acestora este de a spori eficiența de învățare, semnalând agentului un feedback continuu pozitiv/negativ. Recompensele continue sunt:

- `VelocityPlayerToBallReward` - proiecția scalară a vitezei agentului asupra diferenței de poziție dintre el și minge.

- `VelocityBallToGoalReward` - proiecția scalară a vitezei mingii asupra diferenței de poziție dintre minge și poarta oponentului.
- `VelocityReward` - viteza liniară a agentului

Întrucât agentul este antrenat prin *self-play*, sunt implementate în mod deliberat recompense ce pot fi maximizate simultan atât de către agent, cât și de către oponentul său. Aceste tipuri de recompense se numesc *positive-sum* și contribuie în special în procesul de monitorizare al agentului. Recompensele de acest tip sunt: `VelocityReward`, `VelocityPlayerToBallReward`, `Save`, `Shot`, `Demolition`, `Touch`, `Boost Pickup`.

În caz contrar, dacă recompensa primită de agent ar fi în totalitate de tip *zero-sum*, monitorizarea progresului prin intermediul graficului de recompensă medie ar fi ineficientă. Recompensa medie per episod ar însuma toate recompensele primite de agent, în ambele echipe, fiind 0. Considerând acești factori, recompensa agentului este implementată conform **Tabelului 2.1**:

Recompensă	**Valoare / Interval**	**Coeficient**
Goal scored	1000.0	1.0
Goal Conceded	-1000.0	1.0
Save	100.0	1.0
Shot	50.0	1.0
Demolition	20.0	1.0
Touch	10	1.0
Boost pickup	0.5	1.0
VelocityPlayerToBallReward	[-1,1]	0.0002 - 0.002
VelocityBallToGoalReward	[-1,1]	0.0008 - 0.008
VelocityReward	[-1,1]	0.0001 - 0.002

Tabel 2.1: Funcția de recompensă

2.2.2 Constructor de observație

Agentul interpretează mediul de antrenare prin observația primită la fiecare pas. Dacă în cazul problemelor de Computer Vision, observația poate fi o imagine captată de agent, în cazul de față observația este modelată prin selectarea informațiilor disponibile la fiecare pas. Informațiile complete despre joc includ:

- starea mașinilor (poziție, direcție, viteză, nivel de boost etc.)
- starea mingii (poziție, direcție, viteză)
- starea boost-urilor (activ, inactiv)

- elemente statice (poziții pentru porți, boost-uri)
- ultima acțiune executată de agent

Pentru modelarea observației, este urmarită includerea a cât mai multe informații relevante despre starea jocului, fără a construi o observație atât de complexă încât să îngreuneze procesul de învățare. Clasa implementată este `CustomObs`, în **Fragmentele de cod 2.3** și **2.4**, inspirată din clasa `AdvancedObs`, disponibilă in RLGym [24].

Fragment de cod 2.3: "Constructor de observație"

```
class CustomObs(ObsBuilder):
# Informatii despre minge, boosturi, porti, actiunea precedenta
    POS_STD, ANG_STD = 2300, math.pi # Termeni de normalizare
    BLUE_GOAL = common_values.BLUE_GOAL_BACK \ POS_STD
    ORANGE_GOAL = common_values.ORANGE_GOAL_BACK \ POS_STD

    def build_obs(self, player: PlayerData, state: GameState,
        previous_action: np.ndarray) -> Any:
        # Observatia este reprezentata simetric pentru ambele echipe
        if player.team_num == common_values.ORANGE_TEAM:
            inverted = True
            ball = state.inverted_ball
            pads = state.inverted_boost_pads
            goals_position = self.ORANGE_GOAL + self.BLUE_GOAL
        else:
            inverted = False
            ball = state.ball
            pads = state.boost_pads
            goals_position = self.BLUE_GOAL + self.ORANGE_GOAL

        obs = [ball.position / self.POS_STD,
               ball.linear_velocity / self.POS_STD,
               ball.angular_velocity / self.ANG_STD,
               previous_action,
               pads,
               np.array(goals_position)]
```

Fragment de cod 2.4: ”Constructor de observație - continuare”

```
# Observatii legate de coechipieri, inamici
    player_car = self._add_player_to_obs(obs, player, ball, inverted)
    allies, enemies = [], []
    # Sunt adaugati si ceilalti jucatori in observatie
    for other in state.players:
        if other.car_id == player.car_id:
            continue
        if other.team_num == player.team_num:
            team_obs = allies
        else:
            team_obs = enemies

        other_car = self._add_player_to_obs(team_obs, other, ball,
            inverted)
        team_obs.extend([
            # Distanta dintre pozitia oponentului si a agentului
            (other_car.position - player_car.position) / self.POS_STD,
            # Diferenta de viteza liniara (oponent, agent)
            (other_car.linear_velocity - player_car.linear_velocity) /
                self.POS_STD
        ])

    obs.extend(allies)
    obs.extend(enemies)
    return np.concatenate(obs)
```

Metoda ajutătoare adaugă informațiile despre un jucător în observație. Implementarea acesteia este aceeași cu cea folosită în RLGym [24]:

Fragment de cod 2.5: "Metoda ajutătoare - Adăugarea unui agent în observație

```
def _add_player_to_obs(self, obs: List, player: PlayerData, ball:
    PhysicsObject, inverted: bool):
    # Reprezentare simetrica
    if inverted:
        player_car = player.inverted_car_data
    else:
        player_car = player.car_data

    rel_pos = ball.position - player_car.position
    rel_vel = ball.linear_velocity - player_car.linear_velocity

# Alte observatii adaugate
    obs.extend([
        rel_pos / self.POS_STD,
        rel_vel / self.POS_STD,
        player_car.position / self.POS_STD,
        player_car.forward(),
        player_car.up(),
        player_car.linear_velocity / self.POS_STD,
        player_car.angular_velocity / self.ANG_STD,
        [player.boost_amount,
        int(player.on_ground),
        int(player.has_flip),
        int(player.is_demoed)]
        ])

        return player_car
```

Extensia implementată în această lucrare față de `AdvancedObs` este adăugarea de informații statice ce conțin pozițiile celor 2 porți. Prin acest model de observație se urmărește ca agentul să aibă o percepție mai clară asupra mediului, fiind capabil să asocieze diferite elemente de observație pentru a "explica" o recompensă primită. A fost considerată adă-

ugarea acestor informații, întrucât modificau în mod neglijabil complexitatea observației, pentru un potențial beneficiu obținut în antrenare. Pornind de la aceeași idee, adăugarea poziției relative a agentului față de propria poartă și față de poarta adversă ar putea fi de asemenea benefice. Acestea nu au fost implementate în lucrarea curentă, întrucât ideea de implementare a apărut în cursul procesului de antrenare, iar orice modificare a clasei `ObservationBuilder` implică pierderea progresului de antrenare.

2.2.3 Parser de acțiuni

Clasa `RLGym.ActionParser` permite modificarea spațiului standard de acțiuni. Astfel, agentul "alege" la fiecare pas o acțiune în formatul standard definit de Action Parser, iar framework-ul `RLGym` primește de la Action Parser o acțiune în format standard.

Formatul standard în care framework-ul RLGym primește o acțiune este un vector de lungime 8. Fiecare element al vectorului corespunde unui mijloc de control al mașinii, după cum urmează:

- Accelerație - număr real în intervalul [-1, 1]
- Direcție - număr real în intervalul [-1, 1]
- Girație - număr real în intervalul [-1, 1]
- Tangaj - număr real în intervalul [-1, 1]
- Ruliu - număr real în intervalul [-1, 1]
- Săritură - boolean
- Boost - boolean
- Frână de mână - boolean

Pentru antrenarea agentului a fost implementată clasa `CustomActionParser`. Aceasta facilitează procesul de învățare prin discretizarea acțiunilor continue, dar și prin eliminarea acțiunilor redundante sau contraproductive. În procesul de discretizare, este urmărit echilibrul între simplificarea deciziei agentului și păstrarea unei rezoluții adecvate pentru acțiuni. De exemplu, discretizarea prea drastică a spațiului de acțiuni poate rezulta într-un agent care este incapabil să execute acțiuni care necesită o combinație precisă de mișcări. Totuși, faptul că agentul poate executa 15 acțiuni pe secundă permite un grad ridicat de discretizare.

Ideea principală de implementare este inspirată din ActionParser-ul utilizat pentru antrenarea agentului *Necto* [25]. Asemănător parser-ului implementat pentru `Necto`, predefinim o listă ce conține toate acțiunile posibile, prin metoda statică `_get_all_possible_actions()`. Astfel, agentul alege la fiecare pas un singur număr întreg în intervalul $[0, n-1]$, unde n reprezintă lungimea listei de acțiuni.

Modificările principale față de Action Parser-ul implementat pentru Necto sunt creșterea preciziei pentru controlul direcției și eliminarea acțiunii în care frâna de mână este inactivă (în aer).

Precizia pentru controlul direcției a fost majorată prin modificarea numărului de acțiuni posibile pentru a modifica direcția mașinii. În loc de a discretiza intervalul în 3 acțiuni posibile (-1, 0, 1), au fost adăugate acțiunile -0.5 și +0.5. Astfel, agentul nu este constrâns să modifice direcția în stânga complet, dreapta complet sau drept înainte, ci poate alege și ajustări intermediare, anume: dreapta parțială (+0.5), stânga parțială (-0.5). Prin această modificare, se urmărește o acuratețe sporită a schimbării direcției când mașina se află pe sol.

Frâna de mână este menținută mereu activă pentru acțiunile aeriene. Motivele pentru care este implementată această modificare sunt următoarele:

a) Ineficiența acțiunii în aer. Cât timp mașina este în totalitate în aer, activarea/dezactivarea frânei de mână nu influențează controlul mașinii.

b) Utilitatea activării frânei de mână pentru aterizări. În mod contraintuitiv, frâna de mână este un control esențial pentru a asigura aterizări line în Rocket League [26]. Din acest motiv, a fost considerat că menținerea activă a frânei de mână pentru toate acțiunile aeriene oferă un beneficiu în vasta majoritate a cazurilor. Există o situație specifică în care dezactivarea frânei de mână pentru aterizare ar putea fi acțiunea cea mai bună, însă a fost ignorat acest caz. De exemplu, agentul se poate afla în situația unei aterizări iminente în imediata proximitate a mingii, în direcția propriei porți. În acest caz, agentul ar putea evita un ”autogol” printr-o aterizare mai fermă, urmărind să-și reducă rapid viteza după impactul cu solul. În scopul antrenării agentului, s-a considerat că probabilitatea unei asemenea situații este neglijabilă.

Implementarea clasei `CustomActionParser` este ilustrată în **Fragmentele de cod 2.6**, **2.7** și **2.8**.

Fragment de cod 2.6: ”Biblioteci utilizate”

```
from typing import Any
import gym.spaces
import numpy as np
from gym.spaces import Discrete
from rlgym.utils.action_parsers import ActionParser
from rlgym.utils.gamestates import GameState
```

Fragment de cod 2.7: ”CustomActionParser”

```
class CustomActionParser(ActionParser):
    def __init__(self):
        super().__init__()
        self._available_actions = self._get_all_possible_actions()

    @staticmethod # Este creata o lista cu actiunile posibile
    def _get_all_possible_actions():
        # definitia se regaseste in urmatorul fragment de cod

    def get_action_space(self) -> gym.spaces.Space:
        return Discrete(len(self._available_actions))

    def parse_actions(self, actions: Any, state: GameState) -> np.
        ndarray:
        parsed_actions = []
        # este returnata actiunea corespunzatoare indexului primit
        for action in actions:
            parsed_actions.append(self._available_actions[action])
        return np.asarray(parsed_actions)
```

Fragment de cod 2.8: "Funcție ajutătoare pentru ActionParser"

```
@staticmethod
def _get_all_possible_actions():
    actions = []
    # Actiuni pe sol
    for throttle in (-1, 0, 1):
        for steer in (-1, -0.5, 0, 0.5, 1):
            for boost in (0, 1):
                for handbrake in (0, 1):
                    if boost == 1 and throttle != 1:
                        continue
                    actions.append([throttle or boost, steer, 0, steer,
                        0, 0, boost, handbrake])
    # Actiuni aeriene
    for pitch in (-1, 0, 1):
        for yaw in (-1, 0, 1):
            for roll in (-1, 0, 1):
                for jump in (0, 1):
                    for boost in (0, 1):
                        if jump == 1 and yaw != 0:  # Se poate executa
                            side-flip cu air-roll
                            continue
                        if pitch == roll == jump == 0:  # Exista deja
                            in actiuni pe sol
                            continue
                        # Frana de mana este mereu utila in "aer",
                            pentru aterizari
                        handbrake = 1
                        actions.append([boost, yaw, pitch, yaw, roll,
                            jump, boost, handbrake])

    actions = np.array(actions)
    return actions
```

2.2.4 Setter de stare

State setter-ul este obiectul ce configurează starea inițială a fiecărui episod. Pentru a obține o performanță cât mai bună în meciuri de tip 1v1, este important să oferim agentului stări diverse pentru a experimenta și a învăța din cât mai multe situații. Cu toate acestea, starile inițiale în care se regăsește agentul trebuie să fie realiste și similare cu ipostaze în care s-ar putea regăsi în timpul unui meci obișnuit. Adițional, starile au fost alese astfel încât să corespundă nivelului de abilități al agentului. De exemplu, ar putea fi selectate stări în care agentul se află în poziții dificile, pe perete, tavan sau în aer, însă aceste stări ar fi îngreunat considerabil procesul de învățare. Agentul ar fi fost astfel nevoit să încerce multe acțiuni pentru a obține un progres într-o situație mult prea complexă pentru abilitățile sale, ceea ce ar consuma ineficient timp de antrenare.

Din acest motiv, StateSetter-ul configurat în mediu este concentrat pe îmbunătățirea jocului "la sol" al agentului, o componentă esențială în modul 1v1, mai puțin complexă decât controlul aerian. Obiectul utilizat pentru resetarea stării este `RandomState()`, din biblioteca RLGym, ilustrat în **Fragmentul de cod 2.9**.

Fragment de cod 2.9: "State Setter"

```
from rlgym.utils.state_setters import RandomState

def get_match():  # Functia de configurare a mediului de antrenare
    return Match(
        ...,
        state_setter=RandomState(ball_rand_speed=True, cars_rand_speed=
            True)
    )
```

Clasa `RandomState` plasează agenții și mingea într-o poziție aleatoare, pe sol. În cazul agentului antrenat, obiectul este inițializat cu cele 2 flag-uri `ball_rand_speed` și `cars_rand_speed` active. Astfel, sunt întâmpinate frecvent stări inițiale dinamice și poziții diverse, urmărindu-se îmbunătățirea abilității agentului de a generaliza comportamentul într-un meci real. Un alt flag posibil pentru diversificarea stărilor inițiale poate plasa agentul atât în aer, cât și pe sol. Decizia de a nu plasa agentul în astfel de situații a fost luată din cauza performanței relativ scăzute a agentului. Din acest motiv, este preferată exploatarea jocului pe sol în detrimentul explorării jocului aerian, până ce agentul atinge un prag suficient de competență.

2.2.5 Condiție terminală

Condiția terminală este reprezentată prin setul tuturor stărilor terminale. Starea terminală este ultima stare dintr-un episod, pentru care se primește recompensa finală. Pentru a impune condițiile terminale, acestea pot fi incluse în lista trimisă către parametrul `terminal_conditions` în funcția de configurare a mediului.

Fragment de cod 2.10: "Condițiile terminale"

```
frame_skip = 8  # Numar de tick-uri pentru care se executa o actiune
ticks_per_second = 120 # Numar de tickuri rulate pe secunda
fps = 120 // frame_skip # Actiuni executate pe secunda

def get_match():
    return Match(
        ...
        terminal_conditions=[TimeoutCondition(fps * 30),
                             NoTouchTimeoutCondition(fps * 15),
                             GoalScoredCondition()]
    )
```

Condiția principală de terminare a episodului este `TimeoutCondition(seconds)` și reprezintă durata maximă de timp a unui episod. În funcție de aceasta, un set de date de experiență poate conține mai multe episoade scurte sau mai puține episoade lungi. În cazul în care se optează pentru un număr mare de episoade antrenate, agentul se va regăsi mai frecvent în stări similare cu `StateSetter`-ul configurat. Dezavantajul îl reprezintă durata scurtă a episodului, care nu permite agentului să exploreze atât de bine stări îndepărtate de starea inițială sau consecințele acțiunilor pe termen lung. De asemenea, varianța dintre recompense pentru această abordare tinde să fie crescută din cauza explorării frecvente a stării inițiale, care asigură diversitatea de experiențe. În caz contrar, agentul explorează mai departe de starea inițială, însă este posibil ca estimările de valori pentru acțiuni să nu mai fie atât de precise. De exemplu, o acțiune luată în prezent are un impact neglijabil asupra recompensei totale primite după un episod de 1 minut. Această abordare permite agentului să găsească acțiuni ce pot avea consecințe întârziate, însă recompensele întârziate sunt penalizate prin factorul de discount *gamma*. Un beneficiu adițional al acestei abordări îl reprezintă varianța scăzută a recompensei obținute pe episod, întrucât acțiunile agentului au mai mult impact asupra mediului de învățare. Acest fapt este util din punct de vedere practic pentru monitorizarea graficelor de performanță.

Intuiția utilizată pentru selectarea duratei maxime a episodului este că numărul de secunde maxime pentru un episod ar trebui să fie similar cu durata unei ”secvențe” mai lungi de joc dintr-un meci de Rocket League. Din acest motiv, timer-ul este selectat la 30 de secunde.

A doua condiție terminală este `NoTouchTimeoutCondition(seconds)` și aceasta încheie un episod în cazul în care mingea nu a fost atinsă timp de 15 minute de niciun agent. Această condiție este impusă pentru a penaliza comportamentul pasiv și pentru a încuraja agentul să atingă mingea. De asemenea, neatingerea mingii pentru 15 secunde poate semnala o stare prea dificilă pentru abilitățile agentului, fapt ce poate fi monitorizat și corectat prin reconfigurarea state setter-ului.

Ultima condiție terminală este `GoalScoredCondition`. Încheierea episodului după un gol înscris sau primit este motivată de modelarea episodului ca pe un ”mini-meci”. Astfel, obiectivul de a înscris corespunde obiectivului de a câștiga un meci, stare din care agentul primește recompensa finală. Acest model se aseamănă cu lucrări precum (ceva exemplu PPO Atari) în care episodul poate fi considerat ”pierzător” sau ”câștigător”. Un beneficiu suplimentar al acestei stări terminale este semnalul clar trimis către agent pentru stările mai apropiate de un gol marcat. Întrucât agentul primește recompensa finală maximă/-minimă, recompensa viitoare primită pentru stările de dinaintea unui gol este apropiată de recompensa primită pentru gol. În caz contrar, dacă episodul nu s-ar încheia după un gol, semnalul primit de agent ar deveni mai zgomotos pentru că este posibil ca acesta să se afle într-o stare dezavantajoasă după primirea recompensei de gol marcat.

Activarea unei singure condiții terminale determină încheierea episodului. Condiția terminală `GoalScoredCondition` a fost eliminată din configuratiile mai recente de antrenare. Decizia este motivată de faptul că încheierea prematură a unui episod reprezintă în sine o penalizare (recompensa totală posibilă este limitată) și poate oferi semnale contradictorii agentului. Aditional, încheierea episodului pentru goluri marcate îngreunează monitorizarea progresului prin graficul de recompensă adunată pe episod din cauza faptului că episoadele în care se marchează devreme obțin o sumă de recompensă sub medie.

2.3 Antrenare

Antrenarea este procesul care are loc după fiecare etapă de colectare a experiențelor în mediul de învățare. Pentru antrenarea agentului, este implementat un model care implică actualizarea directă a policy-ului prin maximizarea unui obiectiv legat de datele colectate.

Odată ce etapa de antrenare este executată, datele colectate sunt șterse și se generează un set nou de date folosind strategia actualizată.

2.3.1 Algoritm utilizat

Algoritmul utilizat pentru antrenarea agentului este **Proximal Policy Optimization (PPO-Clip)**. PPO face parte din familia de algoritmi de tip "Policy Gradient" și a fost introdus ca o îmbunătățire a algoritmului Trust Region Policy Optimization (TRPO). Concret, PPO introduce o funcție nouă de obiectiv, prin care este simplificată actualizarea policy-ului. Astfel, devine posibilă antrenarea mai multor epoci prin mini-batch gradient ascent, ceea ce îmbunătățește eficiența algoritmului în comparație cu TRPO.

Scopul principal al algoritmilor de tip "Policy Gradient" este de a crește probabilitățile acțiunilor care conduc la un randament mai mare, și de a reduce probabilitățile acțiunilor care conduc la un randament inferior, până când se ajunge la o strategie optimă [27].

Motivul principal pentru folosirea PPO-Clip îl reprezintă succesul demonstrat în antrenarea agenților în diverse medii de învățare. Agenți precum Seer [21] sau cei testați în lucrarea originală PPO [22] au reușit să obțină performanțe considerabile. Un beneficiu adițional al antrenării cu PPO-Clip este gradul de control al magnitudinii unei actualizări de policy. În comparație cu TRPO, funcția de *Clip* este mai simplă și mai eficientă computațional decât ajustarea țintei pentru *divergența KL*.

În ceea ce privește punctele slabe, algoritmul necesită un set larg de date colectate prin experiență pentru a îmbunătăți policy-ul, în special pentru mediile de învățare complexe. Fiecare set de date este folosit pentru un singur pas de antrenare, în comparație cu alți algoritmi de Reinforcement Learning (DQN), care salvează datele colectate într-un "replay buffer", pe baza căruia se efectuează mai multe iterații de antrenare [7]. Din această cauză, colectarea de experiență poate consuma o durată lungă de timp, agenți precum Seer fiind antrenați pentru mai mult de un miliard de pași [21].

Pașii algoritmului PPO-Clip sunt ilustrați în cele ce urmează [27]:

Algorithm 1 PPO-Clip

1: Inițializează aleator policy-ul inițial θ_0, funcția de evaluare initială ϕ_0.
2: **for** $k = 0, 1, 2, ...$ **do**
3: Colectează set de episoade $\mathcal{D}_k = \{\tau_i\}$, urmărind policy-ul $\pi_k = \pi(\theta_k)$.
4: Calculează *Rewards-to-go* $\hat{R}_t$.
5: Calculează estimările de avantaj, $\hat{A}_t$, folosind funcția de evaluare curentă V_{ϕ_k}.
6: Antrenează policy-ul prin maximizarea obiectivului PPO-Clip:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

de obicei prin stochastic gradient ascent, cu optimizatorul Adam.
7: Antrenează funcția de evaluare prin regresie pe eroarea medie la pătrat:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

de obicei prin Stochastic Gradient Ascent, cu optimizatorul Adam.
8: **end for**

Pentru implementarea algoritmului este utilizată biblioteca **StableBaselines3** (SB3). StableBaselines3 conține implementări de calitate pentru mai mulți algoritmi de Reinforcement Learning, care sunt bazate pe PyTorch. StableBaselines3 a fost integrată, întrucât oferă următoarele funcționalități:

- implementare pentru PPO
- API pentru configurarea parametrilor modelului
- API pentru configurarea rețelelor neurale antrenate
- suport pentru medii de învățare vectorizate (instanțe concurente de Rocket League)
- suport pentru normalizarea recompensei și a observației
- suport pentru salvarea și încărcarea modelului
- suport pentru evaluarea modelului

2.3.2 Funcția de obiectiv

Pentru actualizarea policy-ului, este utilizată următoarea formulă:

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} [L(s, a, \theta_k, \theta)]$$

L este funcția de pierdere (loss function) și se calculează după formula:

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \quad g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right)$$

g este funcția de tăiere (clip function) și este definită astfel:

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A, & \text{if } A \geq 0 \\ (1-\epsilon)A, & \text{if } A < 0. \end{cases}$$

2.3.3 Rețele neurale

Algoritmul PPO antrenează două rețele neurale. Ambele rețele primesc o observație a mediului ca strat de intrare. Rețeaua de policy returnează o acțiune selectată din distribuția de probabilitate a acțiunilor disponibile agentului. Rețeaua de evaluare returnează o estimare a recompenselor viitoare, urmând policy-ul curent [27].

Ambele rețele sunt reprezentate printr-un Multi-Layer Perceptron (MLP), feed-forward cu layere complet conectate. Layer-ul de intrare este comun pentru rețele și este reprezentat de observația primită de către agent, de lungime 113. Rețelele conțin 2 layere ascunse, fiecare conținând 192 de noduri. Pentru layer-ul de ieșire, rețeaua de policy returnează un singur index, în intervalul [0, 105], reprezentând una dintre acțiunile predefinite în ActionParser. Rețeaua de evaluare returnează un număr real, ce reprezintă "valoarea" stării s, primite ca input.

Pentru activarea neuronilor este utilizată funcția *Tanh* sau *Tangentă Hiperbolică*. Această funcție este utilizată frecvent în probleme de Deep Reinforcement Learning. Tangenta Hiperbolică primește un număr real și returnează un rezultat în intervalul (-1, 1). Funcția este definită astfel [28]:

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.4}$$

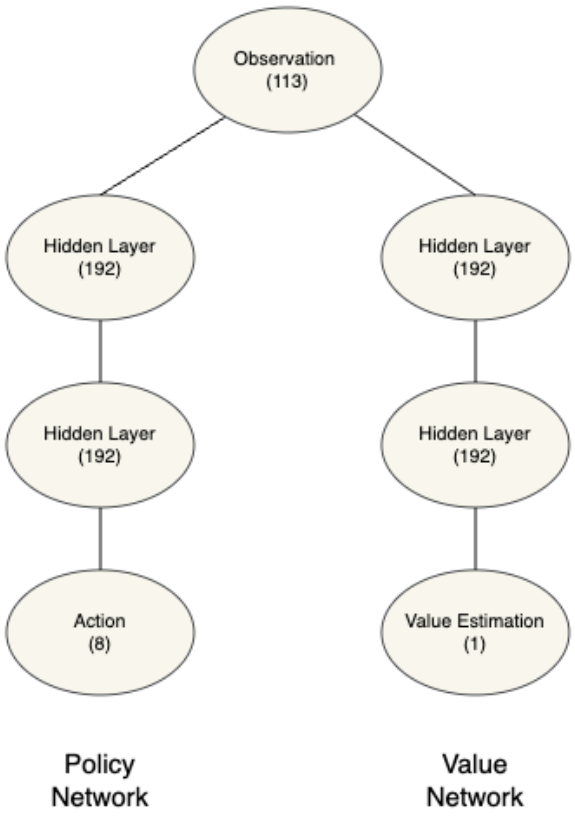

Figura 2.2: Arhitectura rețelelor neurale.

2.3.4 Hiperparametri

Pentru ca algoritmul de antrenare să obțină o performanță cât mai bună, este necesar ca parametrii acestuia să fie ajustați corespunzător. Hiperparametrii configurați pentru antrenarea agentului sunt [29]:

- **learning_rate**: Rata de învățare determină dimensiunea pasului de gradient descent în direcția gradientului. Alegerea unui learning_rate prea mic încetinește convergența către optim, pe când un learning_rate prea mare poate duce la instabilitate în progresul de învățare sau chiar la divergența în funcția de optimizare.
- **n_steps**: Numărul de pași pe care algoritmul îi parcurge in etapa de colectare de exeperienta (per agent). Acesta controlează frecventa cu care actualizăm policy-ul.
- **batch_size**: Numărul de experiențe selectate din setul de date colectate pentru fiecare actualizare a policy-ului.
- **n_epochs**: Numărul de epoci de actualizare a policy-ului prin gradient descent, pentru un set de experiențe colectate.
- **ent_coef**: Acesta este coeficientul de entropie, care determină nivelul de randomizare al policy-ului. Un coeficient mare determină agentul să exploreze mai mult

mediul.

- **vf_coef**: Acesta este coeficientul care echilibrează importanța dintre funcția de evaluare (vf) și policy. Acest coeficient controlează măsura în care algoritmul optimizează policy-ul vs. funcția de evaluare. În extremități, valoarea 0 corespunde unui focus total pe funcția de policy, în timp ce valoarea 1 corespunde în totalitate optimizării funcției de evaluare.

- **gae_lambda**: Factorul de reducere a avantajului estimat generalizat (GAE). Acesta controlează cât de multă importanță se acordă recompenselor în viitorul apropiat, comparativ cu cele din viitorul îndepărtat.

- **clip_range**: Acesta este punctul de tăiere folosit in funcția de clip. Acest parametru controlează proximitatea dintre policy-ul actualizat și cel anterior.

Pentru alegerea potrivită a intervalelor hiperparametrilor, au fost studiate lucrări menționate anterior, precum PPO Atari [22], Seer v4 [21]. Adițional, au fost considerate limitările temporale și computaționale ale proiectului. Odată ce au fost stabilite intervalele comune pentru hiperparametri, acestea au fost introduse într-un algoritm de optimizare, cu ajutorul Optuna.

2.3.5 Optimizarea hiperparametrilor

Optimizarea hiperparametrilor PPO este esențială pentru a maximiza progresul agentului. Optimizarea parametrilor poate fi efectuată manual, observând metricile de performanță pentru diferite configurații de antrenare, sau în mod automat, prin algoritmi de optimizare. În cadrul lucrării, parametrii au fost ajustați atât manual, cât și printr-un algoritm de optimizare.

Printre metodele clasice de optimizare automată a hiperparametrilor se numără: Grid Search, Random Search, Bayesian Optimization. În forma standard, acești algoritmi tind să necesite resurse computaționale complexe sau să necesite rularea completă a configurațiilor de hiperparametri pentru a obține o concluzie despre experimentul efectuat [30].

Pentru optimizarea automată este preferată o soluție performantă, rapidă, ușor de configurat și de implementat, care să consume resurse minime. Aceste standarde sunt comune în optimizarea hiperparametrilor în Machine Learning, întrucât timpul consumat pentru optimizare ar putea fi alocat pentru antrenarea agentului.

In această lucrare, ajustarea automată a parametrilor este efectuată folosind optimizatorul de tip blackbox, *Optuna*. Optimizarea automată a fost implementată cu ajutorul

acestei biblioteci datorită eficienței de experimentare și a algoritmilor performanți pe care îi oferă "out of the box". Optuna este o bibliotecă de Python ce permite utilizatorului o suită complexă de funcționalități, printre care se numără:

- definirea setului de parametri de optimizat
- definirea intervalelor în care aceștia pot lua valori și a pasului de explorare
- definirea unui obiectiv de optimizat
- configurarea experimentelor de optimizare

În plus, funcția de optimizare poate preciza o strategie de oprire prematură a experimentului realizat, prin care Optuna verifică dacă progresul obținut de configurația actuală merită explorat în continuare, ținând cont de durata pentru care a rulat experimentul. În cazul în care experimentul este considerat eșuat după un anumit număr de epoci de optimizare, acesta este oprit și este selectată o nouă configurație de parametri [31].

Intervalele inițiale în care poate fi explorat fiecare parametru au fost alese în funcție de valorile întâlnite în lucrări precum lucrarea originala PPO [27]. Aditional, s-a ținut cont și de constrângerea de timp și de resurse computationale alocate pentru rularea optimizării, dar și a antrenării propriu-zise. Astfel, se explică opțiunea de a limita destul de mult numărul de layere ascunse și numărul de noduri pentru fiecare layer, deși taskul de a învăța Rocket League este complex. Demo-ul susținut de către Crissman Loomis este utilizat ca sursă de inspirație pentru implementarea optimizării cu Optuna [32].

Fragment de cod 2.11: "Obiectivul de optimizare - Configurarea parametrilor"

```
1    def objective(trial):
2        # Sunt trimise intervale de cautare pentru fiecare parametru
3        learning_rate = trial.suggest_float('learning_rate', 1e-5, 5e
             -4, step=2e-5)
4        steps = trial.suggest_int("steps", min_steps, max_steps, step=
             n_steps_step)
5        batch_size = trial.suggest_int("batch", min_batch_size,
             max_batch_size, step=batch_size_step)
6
7        n_epochs = trial.suggest_int('n_epochs', 1, 31, step=5)
8        vf_coef = trial.suggest_float('vf_coef', 0.5, 1., step=0.1)
9        ent_coef = trial.suggest_float('ent_coef', 0., 0.2, step=0.05)
```

```
10          clip_range = trial.suggest_float('clip_range', 0.1, 0.4, step
                =0.05)
11          gae_lambda = trial.suggest_float('gae_lambda', 0.2, 1.0, step
                =0.2)
12
13          layer_size = trial.suggest_int("neurons", 64, 256, step=64)
14          layers_num = trial.suggest_int("layers", 1, 3)
15
16          # Sunt optimizate numarul de layere si de noduri pentru
                retelele neurale
17          policy_kwargs = dict(
18              activation_fn=Tanh,
19              net_arch=dict(pi=[layer_size] * layers_num, vf=[layer_size]
                    * layers_num)
20          )
21
22          # Sunt optimizati hiperparametrii pentru PPO
23          model = PPO(
24              "MlpPolicy",
25              env,
26              n_epochs=n_epochs,
27              policy_kwargs=policy_kwargs,
28              learning_rate=learning_rate,
29              ent_coef=ent_coef,
30              vf_coef=vf_coef,
31              gamma=0.999,
32              verbose=3,
33              batch_size=batch_size,
34              n_steps=steps,
35              clip_range=clip_range,
36              gae_lambda=gae_lambda,
37              tensorboard_log="./rl_tensorboard_log",
38              device="auto"
39          )
```

Fragment de cod 2.12: ”Obiectiv de optimizare - Experimentare și oprire prematură

```
        training_steps_num = 30_000_000  # Numarul maxim de pasi al
            unui experiment
        training_epochs = 3
        mean_reward, std_reward = 0.0, 0.0

        # Bucla de antrenare
        # Antrenarea este impartita in 3 etape, astfel este posibila
            oprirea prematura dupa 10 sau dupa 20 de milioane de epoci
        for epoch in range(training_epochs):
            model.learn(training_steps_num // 3, tb_log_name="1
                s_config_1", reset_num_timesteps=False)

            # Este evaluata configuratia dupa fiecare 1/3 din durata
                experimentului
            mean_reward, std_reward = evaluate_policy(model, model.
                get_env(), n_eval_episodes=5000)

            # Oprire prematura daca progresul este prea mic pentru
                epoca curenta
            trial.report(mean_reward, epoch)
            if trial.should_prune():
                raise optuna.exceptions.TrialPruned()

        print(f'Mean reward: {mean_reward}, STD: {std_reward}')
        return mean_reward
```

Setul cel mai bun de parametri a fost utilizat pentru prima configurație de antrenare a modelului. Ajustări manuale au fost efectuate ulterior asupra hiperparametrilor pentru a îmbunătăți progresul agentului în procesul de învățare. Parametrii care au rămas nemodificați până la finalul antrenamentului sunt: numărul de layere ale rețelelor neurale (2), numărul de neuroni al fiecărui layer ascuns (192) și numărul de epoci de antrenare al policy-ului (31). Cea mai bună configurație este ilustrată în **Tabelul 2.2**, iar statisticile experimentelor sunt ilustrate în **Tabelul 2.3**.

Parametru	Valoare
learning_rate	0.00037
n_steps	93750
batch	75000
n_epochs	31
vf_coef	1.0
ent_coef	0.05
clip_range	0.4
gae_lambda	0.8
layers_size	192
layers_num	2

Tabel 2.2: Configurația cea mai performantă

Statistica	Valoare
Numărul experimentelor complete	35
Numărul experimentelor oprite prematur	65
Cel mai bun experiment	12/100
Rewardul mediu/episod maxim obținut	12.49

Tabel 2.3: Rezultatul experimentelor

2.3.6 Configurații de antrenare

Pentru a facilita monitorizarea progresului, a fost utilizată biblioteca `Tensorboard`. Tensorboard este un software de vizualizare ce permite: plotarea statisticilor primite prin mediul de învățare, comparații între diferite configurații, diverse moduri de vizualizare a graficelor etc. Tensorboard poate fi integrat ușor direct prin intermediul StableBaselines3. De asemenea, pentru salvarea configurației curente a modelului, este apelat un callback o dată la 5 milioane de pași de antrenare.

Fragment de cod 2.13: "Monitorizare statistici și salvare model"

```
from stable_baselines3 import PPO
from stable_baselines3.common.callbacks import CheckpointCallback
from stable_baselines3.common.vec_env import VecMonitor

env = VecMonitor(env) # Necesar wrapper-ul de monitorizare pentru mediu vectorizat
model = PPO(..., env, tensorboard_log="./rl_tensorboard_log") # folder pentru statistici

# Policy-ul este salvat o data la 5 milioane de pasi
callback = CheckpointCallback(round(5_000_000 / env.num_envs),
```

```
                                        save_path=f"./models/{Constants.
                                            CONFIG_NAME.value}",
                                        name_prefix="rl_model")

# Sunt setati callback-ul si numele fisierului in care se salveaza
    statistici
model.learn(total_timesteps=Constants.TRAINING_INTERVAL.value,
            callback=callback,
            tb_log_name=Constants.CONFIG_NAME.value)
```

Configurațiile mediului și ale algoritmului de învățare au fost ajustate pe parcursul antrenării. Arhitectura rețelelor neurale, constructorul de observație, parser-ul de acțiuni și condițiile terminale au rămas nemodificate pe întreaga durată a antrenării. Pentru găsirea unei configurații potrivite, au fost testate 4 configurații care nu au fost utilizate în antrenarea finală. Aceste configurații au fost implementate pentru a explora progresul agentului cu diverse funcții de recompensă, cu diferiți parametri de învățare etc. Din păcate, modelul obținut prin acest antrenament a fost dificil de testat, din cauza normalizării efectuate cu ajutorul wrapper-ului `VecNormalize` din StableBaselines3. Acest wrapper modifică parametrii de normalizare pentru recompensă și pentru observație prin intermediul mediei și varianței observate în distribuția fiecărei componente a observației. Întrucât parametrii de normalizare nu sunt salvați automat odată cu modelul, constructorul de observație rămâne dificil de normalizat pentru a evalua inferența modelului. Din acest motiv, a fost reîncepută antrenarea modelului de la 0, cu o configurație care a produs rezultate similare într-un interval mult mai scurt de timp. În continuare, vor fi prezentate cele 2 etape ale antrenării și configurațiile explorate în cadrul acestora.

Prima etapă de antrenare constă în 4 configurații care au fost antrenate pentru un număr total de aproximativ 1.3 miliarde de pași. Configurațiile sunt numerotate de la A la D, în ordinea în care au fost antrenate. Fiecare configurație începe de unde se încheie antrenarea pentru configurația precedentă, cu excepția configurației A, care începe de la 0. Agentul progresează în fiecare configurație de antrenare, însă există și un "zgomot" omniprezent pentru această etapă de antrenare. Configurația A reprezintă chiar configurația obținută prin optimizarea automată cu Optuna, urmând ca parametrii să fie ajustați pe parcurs. Statisticile pentru recompensa medie acumulată de agent în prima etapă de antrenare se regăsesc în **Figura 2.3**.

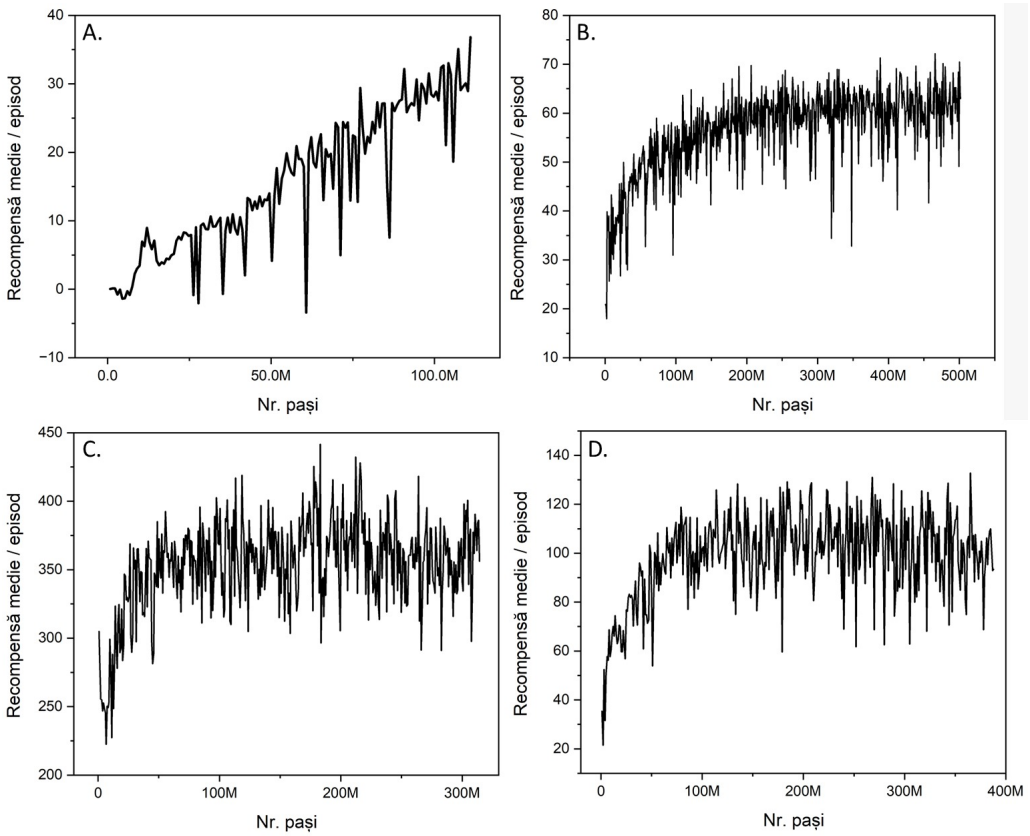

Figura 2.3: Etapa 1 de antrenare: Recompensa medie obținută

Etapa a doua de antrenare este cea care a fost evaluată pentru a stabili performanța agentului. Aceasta este ultima configurație antrenată și este bazată parțial pe configuratia obținută prin optimizarea automată cu Optuna. Hiperparametrii sunt ilustrați în **Tabelul 2.4**.

Parametru	**Valoare**
learning_rate	3.7e-4
ent_coef	0.01
vf_coef	1
gamma	0.999
verbose	3
batch_size	50.000
n_steps	37.500
clip_range	0.2
gae_lambda	0.8

Tabel 2.4: Parametri PPO configurația finală

Intervalul de clip și coeficientul de entropie au fost reduse pentru a spori stabilitatea învățării. Mărimea lotului de experiențe, pe care se optimizează experiența, a fost redusă pentru a accelera procesul de învățare. Numărul total de pași de experiență pentru care este rulat un policy a rămas neschimbat, însă parametrul `n_steps` a fost modificat din cauza schimbării numărului de instanțe rulate simultan. Inițial, au fost rulate 4 instanțe

în mod simultan, ulterior fiind posibilă rularea a 10 instanțe de Rocket League simultan, prin folosirea de resurse computaționale suplimentare. Statisticile obținute în aceasta etapă de antrenare se regăsesc în figurile următoare.

În **Figura 2.4** se poate observa progresul recompensei medii/episod. Această configurație este eficientă pentru maximizarea recompensei, întrucât progresul este constant și nu există căderi bruște sau zgomot excesiv, precum în cazul configurațiilor anterioare.

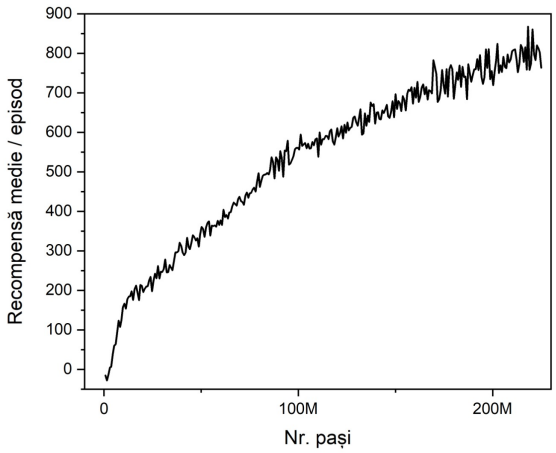

Figura 2.4: Recompensă medie obținută

În **Figura 2.5** se poate observa durata medie a episoadelor. Statisticile sunt satisfăcătoare, întrucât agentul converge spre durata maximă a episodului. Astfel se poate deduce că agentul reușește să atingă mingea în maximum 15 secunde în fiecare episod, întrucât nu mai este activată condiția terminală `NoTouchTimeoutCondition`, care ar termina prematur episodul.

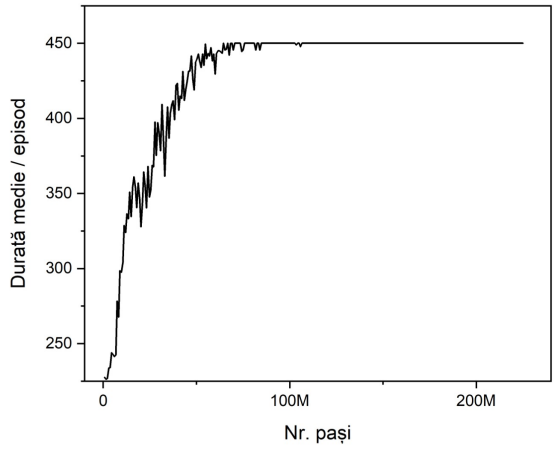

Figura 2.5: Durata medie a episoadelor

Figura 2.6 ilustrează acuratețea de predicție a funcției de evaluare. Se poate observa că predicția devine din ce în ce mai precisă pe parcursul antrenării, acuratețea fiind constant aproximativ 95%.

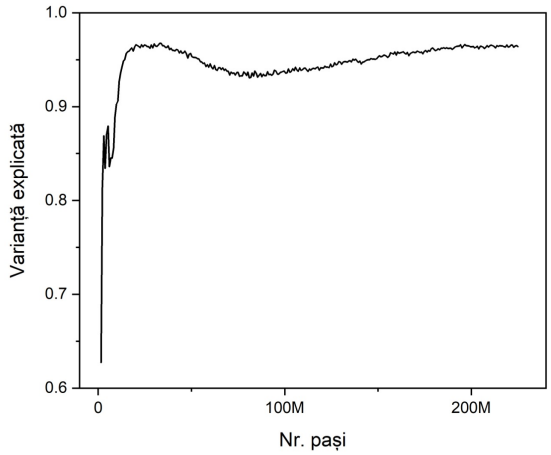

Figura 2.6: Varianța explicată

În **Figura 2.7** este prezentată fracția tăiată din fiecare pas de actualizare a policy-ului. Cu cât fracția tăiată este mai mare, cu atât parametrul de ”clip” inhibă mai mult actualizarea strategiei. Graficul acesta indică de obicei gradul de divergentă dintre policy-uri, plot-ul fiind similar cu cel al divergenței KL, în cazul algoritmului TRPO [33].

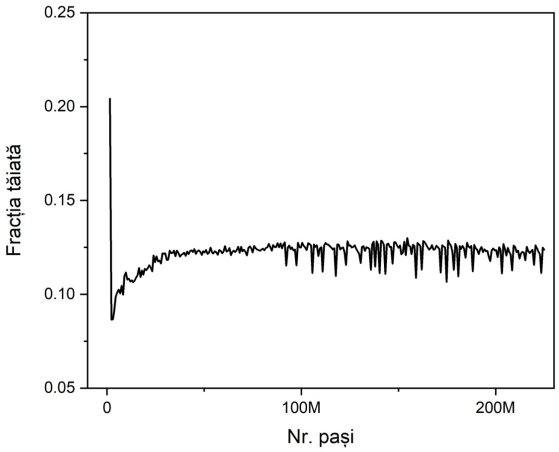

Figura 2.7: Fracția tăiată

Adițional față de această configurație, Fram a fost antrenat încă 20 de milioane de pași pentru a obține un surplus de performanță. Această ultimă configurație de antrenare este identică cu cea precedentă, cu excepția coeficientului de entropie, care a fost redus

la 0. Această modificare are scopul de a încuraja exploatarea abilităților deja învățate, în detrimentul explorării mediului.

2.4 Evaluare

Pentru a compara agentul cu alți agenți și a putea concura împotriva lor, a fost utilizată interfața `RLBot`. Aceasta dispune de un wrapper în care poate fi incarcat botul. In wrapper este apelata metoda de predicție, prin care agentul execută acțiuni in funcție de observația curentă. Această metodă este ilustrată in **Fragmentul de cod** . Constructorul de observație și parserul de acțiuni sunt aceleași ca cele utilizate pentru antrenarea modelului. Din interfața grafică se pot încărca agenți, crea agenți noi sau pot fi utilizați agenți deja existenți. Aceasta este ilustrată în **Figura 2.8**.

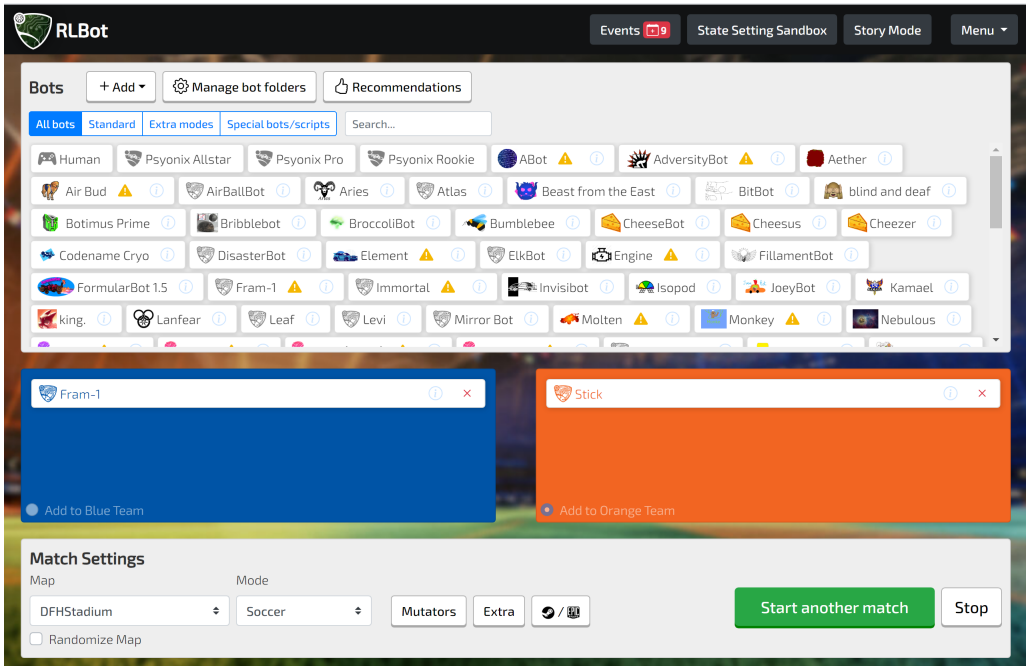

Figura 2.8: Interfața RLBot

Estimarea rank-ului maxim atins de agent este o problemă dificilă întrucât roboților nu le este permis să participe în competiții online alături de jucători umani. Din acest motiv, singura opțiune disponibilă pentru estimarea abilității agentului este de a juca împotriva altor roboți. Ulterior, folosind rezultatele obținute împotriva altor roboți, se poate aproxima un interval de rank-uri în care s-ar putea regăsi agentul, dacă ar participa împotriva oamenilor.

Pentru a obține un clasament care conține abilitățile altor roboți din comunitate, a fost utilizată lucrarea *Seer: Reinforcement Learning in Rocket League* [21], în cadrul

căreia autorul a organizat meciuri între mai mulți agenți robotici. Pentru a obține un rating pentru fiecare agent, autorul a utilizat o metodologie bazată pe sistemul *Trueskill* pe parcursul a 281 de meciuri între agenții selectați.

TrueSkill este un sistem de clasament, dezvoltat de Microsoft. În comparație cu sistemul clasic *Elo*, TrueSkill poate fi utilizat și pentru jocuri cu mai mult de 2 jucători. Sistemul TrueSkill reprezintă abilitățile unui jucător ca o distribuție normală *N*. Astfel, abilitățile jucătorului sunt exprimate prin media distribuției (μ) și varianța (σ). Media reprezintă abilitățile jucătorului, iar varianța reprezintă gradul de incertitudine al rating-ului. În final, rating-ul jucătorului este calculat după formula [34]:

$$R = \mu - 3 \times \sigma \tag{2.5}$$

Pentru evaluarea agentului, acesta a concurat împotriva agenților din clasament în meciuri de 1v1. Pentru a eficientiza procesul, agentul a fost evaluat împotriva roboților mai slabi chiar înainte de a încheia procesul de antrenare. Din acest motiv, scorurile împotriva agenților înfrânți tind să fie conservatoare în raport cu performanța maximă obținută de agent. Evaluarea agentului împotriva unui alt robot constă într-o serie de meciuri de 1v1, tip "Best of 5". În cazul în care unul dintre agenți nu câștigă seria cu 3-0, seria devine "Best of 7" pentru a ajunge la un rezultat mai concludent. Clasamentul este ilustrat in **Tabelul 2.5**.

Agent robot	**TrueSkill**
Necto	42.52 ± 3.29
Self-driving car	40.69 ± 2.20
Wildfire	36.44 ± 2.22
Diablo	35.77 ± 2.17
Botimus Prime	34.89 ± 2.24
Bubo	33.98 ± 2.18
Beast from the East	32.35 ± 2.26
ReliefBot	32.18 ± 2.27
Kamael	31.93 ± 2.22
rashBot	30.34 ± 2.30
AdversityBot	30.29 ± 2.29
PenguinBot	29.77 ± 2.26
Bribblebot	29.30 ± 2.21
Stick	28.14 ± 2.24
BroccoliBot	27.33 ± 2.24
FormularBot 1.5	27.09 ± 2.20
Atlas	26.97 ± 2.25
Leaf	25.68 ± 2.26
Zoomelette	25.45 ± 2.30
Allstar	24.29 ± 2.31
ABot	24.27 ± 2.21
DisasterBot	23.68 ± 2.25
Pro	22.94 ± 2.22
ElkBot	22.35 ± 2.29
Lanfear	21.79 ± 2.27
Air Bud	20.67 ± 2.32
SkyBot	20.60 ± 2.28
FillamentBot	20.39 ± 2.29
St. Peter	19.74 ± 2.28
Codename Cryo	18.19 ± 2.28
NomBot v1.0	17.30 ± 2.30
VirxEB	16.98 ± 2.26
Rookie	16.33 ± 2.31
RocketNoodles	11.88 ± 2.31

Tabel 2.5: Clasament agenți [21]

Media de Trueskill a clasamentului este aproximativ 27.5. Dintre cei 34 de agenți testați, doar 14 dintre aceștia (41%) depășesc aceasta medie.

Din rezultatele obținute, Fram se situează în intervalul de TrueSkill [27, 30], comparativ cu liga de referință. Astfel, Fram depășește rating-ul mediu al ligii de referință, fiind situat în top 40% din roboți. Stabilirea unui rank în raport cu jucătorii umani este dificilă din cauza modului de joc încă nedezvoltat complet al agentului, însă aceasta poate fi estimată în proximitatea rank-ului Silver 3 - Gold 1, fiind potențial mai bun de-

cât 5-10 % din jucătorii umani [18]. Rezultatele finale obținute se regăsesc în **Tabelul 2.6**.

Oponent	**TrueSkill**	**Scor Serie (Fram vs Op.)**	**Versiune Fram (iterație)**
Rookie	16.33	3-0	190/250
Pro	22.94	3-0	190/250
Allstar	24.29	4-1	190/250
Zoomelette	25.45	3-0	215/250
Leaf	25.68	4-1	220/250
Atlas	26.97	3-0	250/250
FormularBot 1.5	27.09	4-1	220/250
BroccoliBot	27.33	3-0	250/250
Stick	28.14	1-4	250/250
BribbleBot	29.30	3-0	250/250
PenguinBot	29.77	3-0	250/250
rashBot	30.34	1-4	250/250
Kamael	31.93	0-3	250/250

Tabel 2.6: Rezultate împotriva altor roboți

2.5 Tehnologii auxiliare

În afara de tehnologiile deja menționate, au fost utilizate și alte programe software pentru dezvoltarea proiectului. Programele au fost folosite pentru controlul versiunii și pentru gestionarea mediilor virtuale în Python.

Git și GitHub au fost utilizate pentru a controla versiunile proiectului și pentru a putea lucra la proiecte de pe mai multe dispozitive. Întrucât antrenarea agentului este un proces care necesită resurse computaționale considerabile, dezvoltarea proiectului pe același sistem nu este fezabilă în timp ce agentul este antrenat. Din acest motiv, proiectul a fost ”pushed” într-un repository remote pentru a putea fi clonat și dezvoltat în paralel. În plus, GitHub Desktop a fost programul utilizat local pentru gestionarea modificărilor, datorită interfeței cuprinzătoare și sugestive. Funcționalități precum ”stashing”, ”staging”, ”commit”, ”push” au fost facilitate prin utilizarea acestei interfețe grafice. Interfata Github este ilustrată în **Figura 2.9**.

Anaconda a fost utilizat pentru gestionarea mediilor virtuale în Python. Întrucât proiectul necesită integrarea de biblioteci externe, păstrarea versiunilor compatibile pentru acestea este dificilă fără crearea de medii virtuale. În plus, utilizarea wrapper-ului de RLBot pentru testarea performanței agentului depinde de biblioteci și versiuni diferite față de biblioteca RLGym. Din acest motiv, au fost utilizate două medii virtuale diferite: unul

pentru dezvoltarea robotului cu RLGym și unul pentru evaluarea prin RLBot. Interfața navigatorului Anaconda este ilustrată in **Figura 2.10**.

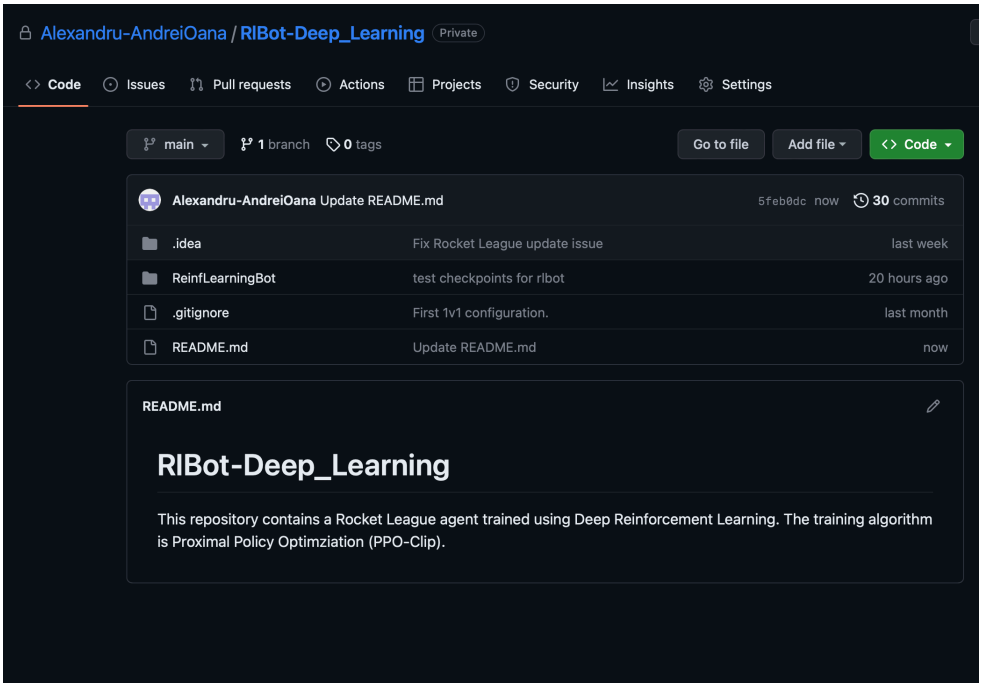

Figura 2.9: Github remote repository

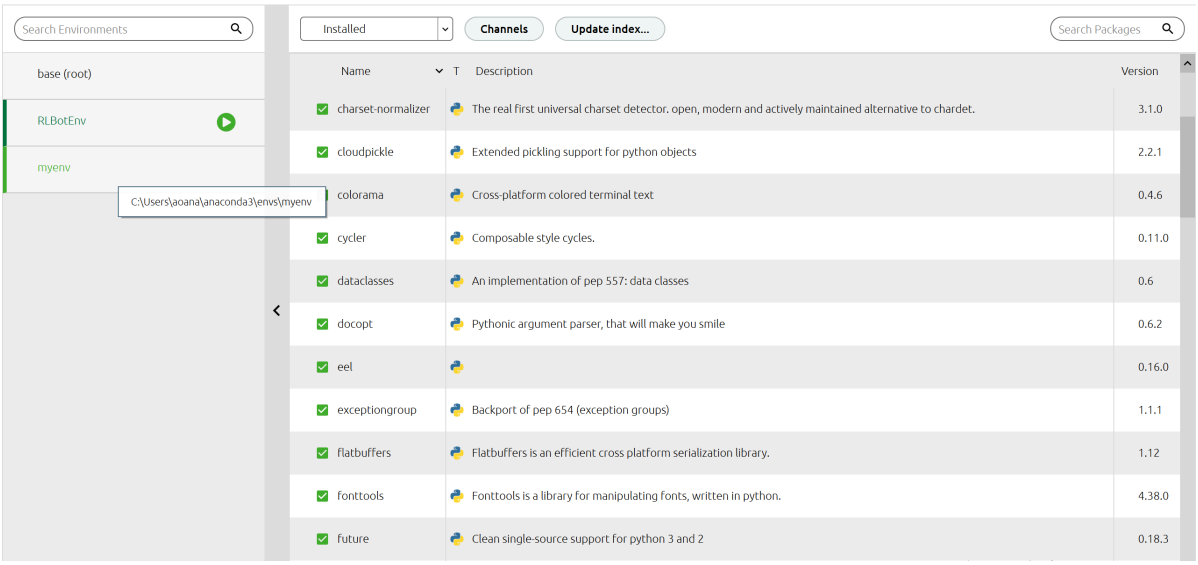

Figura 2.10: Anaconda Navigator

Capitolul 3

Concluzie

În încheierea lucrării, am demonstrat că metodele de Reinforcement Learning pot fi utilizate chiar și pentru antrenarea într-un mediu complex, precum Rocket League. Agentul dezvoltat este un jucător competitiv în comparație cu alți roboți din comunitate, reușind să învingă mai mult de jumătate dintre roboții considerați ca referință. Deși nu a reușit să dezvolte abilități complexe și nu este capabil să fie competitiv cu majoritatea agenților umani, Fram poate fi un adversar potrivit pentru un jucător începător. Cu toate acestea, agentul poate depăși limita abilităților atinse în această lucrare. Pentru a îmbunătăți performanțele acestuia, pot fi modificați parametrii de învățare, mediul de antrenare sau chiar metodologia de dezvoltare. În plus, agenții antrenați prin PPO devin competitivi după mulți pași de antrenare, fapt ce presupune alocare suplimentară de resurse computaționale și de timp de antrenare.

3.1 Perspective de îmbunătățire

Pentru a îmbunătăți performanța modelului în viitor, un potențial punct de pornire poate fi configurarea mediului de antrenament. Agentul tinde să dezvolte un stil de joc la sol, din cauza faptului că majoritatea stărilor în care se regăsește conduc spre acest stil. Din această cauză, agentul nu reușește să învețe strategii eficiente pentru navigarea pe pereți, în special la înălțimi mai mari. Pentru a remedia această problemă, agentul poate fi pus în stări inițiale specifice, care conduc mingea spre perete sau în care poate fi chiar el pe perete. În plus, pot fi luate în considerare recompense negative pentru consumarea de ”resurse”. De exemplu, agentul utilizează în mod excesiv săritura pentru a schimba direcția, astfel neînvățând să modifice direcția prin viraje.

Un punct suplimentar de îmbunătățire a metologiei de lucru poate fi modul de monitorizare al performanței. În această lucrare, performanța a fost corelată cu recompensa

medie acumulată pe episod. Luând în considerare faptul că agentul este antrenat strict prin self-play, metoda de monitorizare nu are în vedere îmbunătățirile de performanță decât prin măsurare recompenselor "positive-sum". Astfel, modelarea funcției de recompensă trebuie să includă o cantitate semnificativă de astfel de recompense. Alternativa este ca progresul să fie evaluat prin meciuri sporadice împotriva unor versiuni anterioare sau împotriva altor agenți cu rating cunoscut, pentru a testa că agentul progresează. Dezavantajul acestei metode este că implică o investiție suplimentară de timp și resurse computaționale.

Adițional, pot fi considerate modificări la nivel de arhitectură sau de algoritm de învățare, însă aceste îmbunătățiri necesită documentare suplimentară.

Bibliografie

[1] Wikipedia, *Reinforcement Learning*, Accesat: 14.06.2023, URL: https://en.wikipedia.org/wiki/Reinforcement_learning.

[2] Shweta Bhatt, *Reinforcement Learning 101*, Accesat: 14.06.2023, URL: https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292.

[3] D. Silver, A. Huang, C. Maddison et al., „Mastering the game of Go with deep neural networks and tree search", în *Nature* 529 (2016), pp. 484–489, URL: https://doi.org/10.1038/nature16961.

[4] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski și Susan Zhang, *Dota 2 with Large Scale Deep Reinforcement Learning*, arXiv:1912.06680 [cs.LG], 2019, URL: https://arxiv.org/abs/1912.06680.

[5] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan și Demis Hassabis, *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, 2017, arXiv: 1712.01815 [cs.AI].

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra și Martin Riedmiller, „Playing Atari with Deep Reinforcement

Learning", în *arXiv preprint arXiv:1312.5602* (2013), NIPS Deep Learning Workshop 2013, URL: https://arxiv.org/abs/1312.5602.

[7] OpenAI, *Introduction to RL: Kinds of RL Algorithms*, Accesat: 17.06.2023, URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

[8] OpenAI, *Spinning Up in Deep RL: Key Concepts in RL*, Accesat: 12.06.2023, URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html.

[9] Wikipedia, *Rocket League*, Accesat: 14.06.2023, URL: https://en.wikipedia.org/wiki/Rocket_League.

[10] SquishyMuffinz, *HOW TO MANAGE YOUR BOOST LIKE A PRO PLAYER (TUTORIAL) | THE BEST TIPS FOR IMPROVING IN ROCKET LEAGUE*, Accesat: 14.06.2023, URL: https://www.youtube.com/watch?v=eK3DLp-Yjwc&ab_channel=SquishyMuffinz.

[11] Ellis Lane, *Rocket League: Kick-off strategy guide*, Accesat: 18.06.2023, URL: https://upcomer.com/rocket-league-kick-off-strategy-guide.

[12] Epic Games, *Rocket League Car Hitboxes*, Accesat: 14.06.2023, URL: https://www.epicgames.com/help/en-US/rocket-league-c5719357623323/basics-c7261959845275/rocket-league-car-hitboxes-a5720112690459.

[13] Jack Marsh, *Rocket League Hitboxes: All Car Body Types, Shapes, And Sizes*, Accesat: 18.06.2023, URL: https://www.ggrecon.com/guides/rocket-league-hitboxes/.

[14] Fandom.com, *Demolitions*, Accesat: 14.06.2023, URL: https://rocketleague.fandom.com/wiki/Demolition.

[15] Wikipedia, *Stockfish (chess)*, Accesat: 18.06.2023, URL: https://en.wikipedia.org/wiki/Stockfish_(chess).

[16] Wikipedia, *List of chess players by peak FIDE rating*, Accesat: 18.06.2023, URL: https://en.wikipedia.org/wiki/List_of_chess_players_by_peak_FIDE_rating.

[17] Wikipedia, *Pluribus (poker bot)*, Accesat: 18.06.2023, URL: https://en.wikipedia.org/wiki/Pluribus_(poker_bot).

[18] Jack Marsh, *Rocket League rank distribution Season 10*, Accesat: 18.06.2023, URL: https://www.ggrecon.com/guides/rocket-league-rank-distribution/.

[19] Lethamyr, *This bot is better than 99 percent of the players, can I beat 3 of them?*, Accesat: 18.06.2023, URL: https://www.youtube.com/watch?v=AyAiJFnRFCk&ab_channel=Lethamyr.

[20] King Ranny, *SSL 1V1 PLAYER VS NEXTO | GROUND GAME ONLY*, Accesat: 18.06.2023, URL: https://www.youtube.com/results?search_query=king+ranny+ground.

[21] Neville Walo, „Seer: Reinforcement Learning in Rocket League", Master's thesis, ETH Zurich, Aug. 2022.

[22] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford și Oleg Klimov, „Proximal Policy Optimization Algorithms", în *arXiv preprint arXiv:1707.06347* (2017).

[23] RLGym, *The Rocket League Gym: Introduction*, Accesat: 12.06.2023, URL: https://rlgym.org/docs-page.html#introduction.

[24] Emery Lucas, *RLGym*, Accesat: 12.06.2023, URL: https://github.com/lucas-emery/rocket-league-gym/blob/main/rlgym/utils/obs_builders/advanced_obs.py.

[25] Rolv-Arild, *Necto*, Accesat: 12.06.2023, URL: https://github.com/Rolv-Arild/Necto/blob/master/training/parser.py.

[26] Beebs: Dignitas, *The Versatility of Powersliding in Rocket League*, Accesat: 18.06.2023, URL: https://dignitas.gg/articles/the-versatility-of-powersliding-in-rocket-league.

[27] OpenAI, *Proximal Policy Optimization*, Accesat: 18.06.2023, URL: https://spinningup.openai.com/en/latest/algorithms/ppo.html.

[28] Jason Brownlee, *How to Choose an Activation Function for Deep Learning*, Accesat: 18.06.2023, URL: https://machinelearningmastery.com/.

[29] Stable-Baselines3, *PPO: Parameters*, Accesat: 18.06.2023, URL: https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html.

[30] Ian Goodfellow, Yoshua Bengio și Aaron Courville, *Deep Learning*, MIT Press, 2016.

[31] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta și Masanori Koyama, „Optuna: A Next-generation Hyperparameter Optimization Framework", în *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[32] Pytorch, *Auto-Tuning Hyperparameters with Optuna and PyTorch*, Accesat: 12.06.2023, URL: https://www.youtube.com/watch?v=P6NwZVl8ttc.

[33] AurelianTactics, *Understanding PPO Plots in TensorBoard*, Accesat: 18.06.2023, URL: https://medium.com/aureliantactics/understanding-ppo-plots-in-tensorboard-cbc3199b9ba2.

[34] Wikipedia, *TrueSkill*, Accesat: 18.06.2023, URL: https://en.wikipedia.org/wiki/TrueSkill.