

UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA MATEMATICĂ-INFORMATICĂ



Lucrare de licență

Simularea unui ecosistem utilizând învățarea automată

Absolvent

Comșa Roberto-Marian

Coordonator științific

Conf. Dr. Cidota Marina

București, Februarie 2021

Cuprins

Rezumat	4
1 Introducere	5
1.1 Istoric	6
1.2 Tipuri de învățare automată	8
1.3 Învățare ranforsată	9
1.3.1 Introducere	9
1.3.2 Discipline intersectate	10
1.3.3 Exemple de aplicații ale învățării ranforsate	10
2 Învățare ranforsată - Fundamentele teoretice	12
2.1 Caracteristici cheie	12
2.2 Cadrul formal	13
2.2.1 Proprietatea Markov	14
2.2.2 Diferite categorii de politici	15
2.2.3 Recompensă redusă și funcția obiectiv	16
2.2.4 Funcțiile de valoare a stării și a acțiunii	16
2.2.5 Componente necesare pentru învățarea unei politici	17
2.2.6 <i>On-policy</i> vs. <i>Off-policy</i>	18
2.2.7 Exploatare vs. Explorare	18
2.3 Tehnici de tip <i>Policy Gradient</i>	19
2.3.1 <i>Stochastic policy gradient</i>	19
2.3.2 <i>Natural Policy Gradient</i>	20

2.3.3	Optimizarea regiunii de încredere	21
2.3.4	<i>Proximal policy optimization (PPO)</i>	22
2.3.5	Metode actor-critic	23
2.3.6	<i>Soft Actor-Critic (SAC)</i>	24
3	Descrierea metodelor de antrenare și de evaluare	26
3.1	Structura generală a unui agent ML-Agents	26
3.1.1	Scriptare	26
3.1.2	Alte tipuri de observații	28
3.1.3	Parametri de comportament	30
3.1.4	Fișierul de configurare a antrenamentului	31
3.1.5	Antrenarea și evaluarea unui agent	35
3.2	Agent carnivor	36
3.2.1	Pregătirea modelului	36
3.2.2	Antrenare și evaluare	39
3.3	Agent erbivor	43
3.3.1	Pregătirea modelului	43
3.3.2	Antrenare și evaluare	47
3.4	Agent aerian de sprijin	49
3.4.1	Pregătirea modelului	49
3.4.2	<i>Imitation learning (IL)</i>	52
3.4.3	Antrenare și evaluare	55
4	Utilizarea aplicației	57
4.1	Meniul principal	58
4.2	Meniul de editare al parametrilor	58
4.3	Amplasarea agenților	59
4.4	Acțiuni în timpul simulării	60
5	Tehnologii utilizate	61
5.1	Unity3D	61

5.2	C# (CSharp)	61
5.3	Microsoft Visual Studio	62
5.4	ML-Agents	62
5.5	Anaconda	62
5.6	TensorFlow	62
5.7	GitHub Desktop	63
5.8	Overleaf	63
6	Concluzii și direcții viitoare de dezvoltare	64

Rezumat

Atât industria învățării automate cât și industria jocurilor sunt în curs de expansiune. Interesul în aceste două direcții și în aplicabilitatea inteligenței artificiale în jocuri a servit ca motivație în alegerea subiectului. În lucrarea de față sunt prezentate noțiuni teoretice ale învățării ranforsate, în direcția algoritmilor din familia metodelor *policy gradient* precum PPO (*Proximal Policy Optimization*) și SAC (*Soft-Actor-Critic*). Antrenamentele agenților din interiorul ecosistemului au fost realizate prin utilizarea motorului de joc Unity3D și a setului de instrumente ML-Agents, combinație ce permite antrenarea prin învățare ranforsată într-un mediu controlat. Modelele captează observații din interiorul spațiilor de antrenare (numerice sau de tip *raycast*) și oferă un vector de acțiuni (discrete sau continue) folosite pentru a controla componenta fizică a agentului. Agenții rezultați pot fi folosiți drept caractere non-jucător pentru a aduce la viață o scenă cu care un utilizator uman interacționează.

Capitolul 1

Introducere

Termenul de învățare automată se referă la detectarea mecanică a tiparelor semnificative din date. În ultimele decenii a devenit un instrument comun în rezolvarea sarcinilor ce necesită extragerea de informații din seturi mari de date, fiind adesea mai eficientă decât omul în soluționarea acestor probleme. De exemplu, un algoritm predictiv va crea un model predictiv, care va oferi anticipări bazate pe cunoștințele obținute din datele cu care a fost antrenat [21].

O caracteristică comună aplicațiilor de învățare automată este că, în contrast cu utilizarea tradițională a computerelor pentru a rezolva probleme, din cauza complexității tiparelor ce trebuie detectate în seturile de date, programatorilor umani le este foarte dificil să furnizeze o descriere explicită, bine detaliată a modului în care ar trebui executate astfel de sarcini. Luând exemplu omul ca ființă inteligentă, multe din aptitudinile noastre sunt dobândite sau rafinate prin învățare din experiențele trăite, în mod egal instrumentele de învățare automată sunt preocupate de a dota programele cu posibilitatea de a învăța (prin experiență) și de a se adapta dobândind astfel un oarecare nivel de "inteligentă" [21].

1.1 Istoric

Preliminarii

Multe concepte cheie ale învățării automate sunt derivate din teoria probabilității și statistica ale căror rădăcini datează încă din secolul al XVIII-lea. De exemplu, în 1763, Thomas Bayes a elaborat o teoremă matematică pentru probabilități care rămâne un concept central în unele abordări moderne ale învățării automate. Numeroși cercetători și oameni de știință au construit bazele computerelor moderne ce sunt indispensabile domeniului de învățare automată. Printre aceștia îi menționăm pe germanul Gottfried Leibniz care a conceput sistemul de coduri binare în anul 1679 și pe George Boole ce a creat în anul 1847 o formă de algebră în care toate valorile pot fi reduse la “adevărat” sau “fals”. Ideea de inteligență artificială a fost prezentată publicului larg pentru prima dată în anul 1927 o dată cu difuzarea în cinematografe a filmului *Metropolis*, peliculă sci-fi regizată de Fritz Lang. Acțiunea setată în Berlin, anul 2026 introduce ideea de mașină gânditoare prin caracterul False Maria, primul robot inteligent descris în cinema. În 1936, inspirat de metodele prin care urmărim procese specifice în rezolvarea sarcinilor, Alan Turing, logician și criptanalist englez, a teoretizat modul în care o mașină ar putea descifra și executa un set de instrucțiuni. Demonstrațiile publicate au fost și sunt considerate piatra de temelie a informaticii [3].

Drumul de la teorie la practică

Neurofiziologul Warren McCulloch și matematicianul Walter Pitts au publicat în anul 1943 o lucrare despre neuroni și modul în care funcționează. Pentru a ilustra teoria au modelat o rețea neuronală utilizând circuite electrice. Munca lor a fost esențială evoluției domeniului, conceptul fiind aplicat ulterior și de către alți oameni de știință. În 1952, Arthur Samuel, pionier al învățării automate a creat primul program care putea învăța în timp ce rula. Programul ce utiliza noțiuni noi precum algoritmul minimax, tăiere alfa-beta și funcție de scor a ajutat un computer deținut de IBM să devină mai bun la jocul de dame. Frank Rosenblatt a proiectat în anul 1958 primul model de rețea

neurală artificială, numit Perceptron. În anul următor Bernard Widrow și Marcian Hoff au creat primul model de rețea neuronală aplicat în rezolvarea unei probleme reale. Dezvoltat la Universitatea din Stanford, MADELINE folosește un filtru adaptiv pentru eliminarea ecourilor din liniile telefonice fiind utilizată și în prezent. Timp de două decenii domeniul nu a văzut progrese însemnate, până în anul 1985 când Terry Sejnowski și Charles Rosenberg au creat rețeaua neuronală NETtalk care a învățat într-o săptămână să pronunțe corect aproximativ 20.000 de cuvinte. Douăsprezece ani mai târziu computerul deținut de IBM, numit Deep Blue a câștigat o partidă de șah în fața campionului mondial Garry Kasparov dovedind că datorită puterii mai ridicate de calcul, cu ajutorul învățării automate mașinile pot deveni mai bune decât omul în rezolvarea anumitor sarcini. Un alt exemplu în acest sens este prototipul stației de lucru inteligentă dezvoltat în 1999 la Universitatea din Chicago ce avea ca scop identificarea pacienților bolnavi de cancer. După verificarea a 22.000 de mamografii computerul a depistat pacienții afectați având o acuratețe mai mare cu 52% față de radiologi [3, 19].

Învățarea automată în zilele noastre

De la începutul secolului XXI multe companii au realizat că învățarea automată va spori potențialul de calcul, astfel modelele de învățare automată s-au mutat din laboratoare în viața noastră cu aplicații în diverse domenii. GoogleBrain, rețea neuronală dezvoltată la Google în anul 2012 a învățat să recunoască oameni și pisici în videoclipuri de pe YouTube fără a ști explicit cum să le caracterizeze având o acuratețe de 74.8% în detectarea felinei și 81.7% în detectarea fețelor. Doi ani mai târziu chatbot-ul Eugene Goostman a reușit să treacă testul Turing. Elaborat de Alan Turing în anul 1950, testul impune ca un computer să păcălească un interlocutor uman, astfel încât acesta să nu realizeze dacă vorbește cu un om sau cu un calculator. Alan considera că o mașină care trece testul prezintă un nivel de inteligență uman. Eugene a convins 33% din jurații umani că este un tânăr ucrainian. În anul 2015 programul AlphaGo a fost primul program care a reușit să câștige o partidă de Go, considerat cel mai dificil joc de masă, împotriva unui jucător profesionist. Cu această victorie computerele au învins oponenți umani de top în fiecare

joc de masă clasic. North Face a devenit în anul 2016 primul distribuitor de haine ce folosește funcția de procesare a limbajului natural a computerului Watson deținut de IBM într-o aplicație mobilă. Expertul personal în cumpărături ajută clienții să găsească produsele dorite prin conversație, asemenea unui consultant de vânzări uman. Pe lângă aceste modele impresionante, lumea din jurul nostru este acaparată de tehnologii bazate pe învățare automată precum asistenții personali virtuali prezenți în mobilele noastre, serviciile mediilor de socializare ca de exemplu sugestiile de prietenie sau recunoașterea facială, funcția de filtrare a email-urilor, sistemele de căutare folosite în numeroase aplicații și chiar recomandarea de produse prin reclame [3, 19].

1.2 Tipuri de învățare automată

Învățarea reprezintă un câmp foarte larg, prin urmare domeniul învățării automate s-a împărțit în mai multe subclase pentru a soluționa diferite sarcini de învățare. Când ne referim la paradigmele standard, în mod tradițional avem trei tipuri fundamentale de învățare automată.

Învățare supervizată

Primul este învățarea supervizată, un tip de învățare în care atât datele de intrare cât și datele de ieșire dorite sunt furnizate. Datele sunt etichetate, oferind o bază de învățare pentru prelucrarea viitoare a datelor. Această paradigmă constă într-o variabilă dependentă numită țintă care trebuie să fie prevăzută dintr-un anumit set de variabile independente. Prin utilizarea acestor variabile este generată o funcție de mapare a intrărilor în ieșirile dorite. Procesul de antrenare continuă până când modelul obține nivelul de precizie dorit (sau maxim) pe datele de instruire [18].

Învățare nesupervizată

Cel de-al doilea tip este învățarea nesupervizată ce presupune antrenarea unui model folosind informații care în acest caz nu sunt etichetate, permițând algoritmului să acționeze fără a-i fi oferite orientări. Ideea principală ce stă la baza acestei paradigme este expunerea modelului la volume mari de date variate, astfel încât acesta să învețe și să deducă din informațiile primite caracteristicile care fac punctele din date mai mult sau mai puțin asemănătoare între ele [18].

1.3 Învățare ranforsată

În acest subcapitol este descris într-un mod mai general cel de-al treilea tip fundamental de învățare automată. Având în vedere că agenții din proiectul prezentat în această lucrare sunt antrenați prin învățare ranforsată, această paradigmă are un capitol dedicat unde este expusă amănunțit.

1.3.1 Introducere

Învățarea ranforsată este caracterizată de un agent care interacționează în mod continuu și învață dintr-un mediu înconjurător stocastic. Ne putem imagina spre exemplu un robot biped care are sarcina de a învăța să se deplaseze de la un punct A la un diferit punct B. El încearcă inițial în mod întâmplător diferite moduri de a-și mișca picioarele și învață atât din mișcările de succes cât și din căderile sale pentru a găsi în cele din urmă cel mai eficient mod de a merge în direcția punctului B. Învățarea ranforsată este o ramură a inteligenței artificiale care formalizează metoda de învățare prin încercare și eroare [1].

Este în esență știința de a lua decizii secvențiale. Cum ar trebui robotul să-și miște membrele inferioare pentru a învăța în cele din urmă să meargă și să ajungă cât mai repede la punctul B? Mai general, cum ar trebui să interacționeze agentul cu mediul înconjurător, ce acțiuni ar trebui să ia acum astfel încât să poată afla mai multe despre mediu și să aibe mai mult succes în viitor [1].

1.3.2 Discipline intersectate

Învățarea ranforsată se află la intersecția mai multor discipline științifice, și anume:

- Control optim (Inginerie)
- Programare dinamică (Cercetare operațională)
- Sisteme de recompense (Neuro-știință)
- Condiționare clasică / operantă (Psihologie)

În toate aceste domenii diferite există o ramură care încearcă să studieze aceeași problemă ca învățarea ranforsată și anume problema modului de a lua decizii secvențiale optime. În inginerie este problema găsirii controlului optim iar în cercetarea operațională este studiată în cadrul programării dinamice. Principiile algoritmice din spatele învățării ranforsate au ca motivație fenomenele naturale ce stau la baza luării deciziilor umane, în cuvinte simple ”recompensele oferă o întărire pozitivă unei acțiuni”: acest fenomen este studiat în psihologie ca și condiționare iar în neuroștiință ca sisteme de recompense [1].

1.3.3 Exemple de aplicații ale învățării ranforsate

- **Învățarea jocurilor:** Cele mai cunoscute succese ale învățării ranforsate sunt în jucarea jocurilor. AlphaZero dezvoltat de Google Deepmind atinge nivel super-uman și învinge campioni mondiali la jocuri complexe precum șah, shogi și Go. Aceeași echipă a construit un sistem de învățare ranforsată care a învățat să joace de la zero o suită de jocuri de la Atari, primind ca observații doar pixelii ecranului și recompensă bazată pe scor, fara vreo indicație în direcția regulilor pentru vreunul dintre jocuri [1].
- **Chatbots:** O altă aplicație populară sunt roboții de conversație sau asistenții personali inteligenți precum Siri, Google Now, Cortana și Alexa. Acești agenți au în comun faptul că încearcă să facă o conversație cu un utilizator uman. Un astfel

de robot primește semnale încurajatoare dacă oferă răspunsuri pertinente sau semnale negative atunci când nu satisface cu răspunsurile sale și se folosește de acest feedback pentru a învăța prin încercare și eroare [1].

- **Sănătate:** O aplicație ușor diferită a învățării ranforsate, dar importantă este în domeniul medical. Putem alege ca exemplu planificarea tratamentului medical, unde problema constă în a învăța o secvență de tratamente pentru un pacient pe baza reacțiilor la tratamentele anterioare și starea actuală a pacientului. În acest caz, încercările sunt foarte scumpe și trebuie să fie efectuate cu atenție pentru a obține o învățare cât mai eficientă [1].

Subiectul abordat în lucrarea de față este simularea unui ecosistem în motorul de joc Unity3D prin aplicarea învățării ranforsate pentru antrenarea de agenți inteligenți. Lucrarea continuă cu noțiuni teoretice ale învățării ranforsate și ale algoritmilor utilizați. Apoi sunt prezentate atât structura generală a unui agent creat prin setul de instrumente ML-agents cât și structura agenților dezvoltați și tehnicile de antrenare și de evaluare folosite. De asemenea, este prezentat un scurt ghid de utilizare al aplicației. În finalul lucrării sunt listate tehnologiile folosite și sunt prezentate concluziile și direcțiile viitoare de dezvoltare propuse.

Capitolul 2

Învățare ranforsată - Fundamentele teoretice

În acest capitol sunt prezentate atât detaliile tehnice care stau la baza învățării ranforsate cât și teoria din spatele metodelor folosite pentru antrenarea agenților prezentați în această lucrare.

2.1 Caracteristici cheie

Mai jos sunt listate câteva caracteristici ce diferențiază învățarea ranforsată de alte paradigme de optimizare și de alte metode de învățare automată precum învățarea supravegheată [1]:

- **Lipsa unui supervizor:** Una dintre caracteristicile principale ale învățării ranforsate este că nu există nici un supervizor, nici o etichetă care să prezinte cea mai bună acțiune de întreprins, doar semnale de recompensă pentru a impune unele acțiuni mai mult decât altele.
- **Decizii secvențiale:** În învățarea ranforsată timpul contează cu adevărat deoarece ”secvența” în care deciziile sunt alese va decide calea agentului spre încheierea unei sarcini și, prin urmare, rezultatul final.
- **Feedback întârziat:** O altă distincție majoră este că feedback-ul este adesea

întarziat în sensul că efectul acțiunii curente s-ar putea să nu fie complet vizibil instantaneu, dar poate afecta puternic semnalul de recompensă câțiva pași mai târziu. În exemplul robotului, o mișcare agresivă a piciorului poate arăta bine la momentul actual deoarece ajută robotul să se deplaseze mai rapid în direcția dorită, dar o succesiune de pași mai târziu putem realiza că acea mișcare a făcut ca robotul să cadă.

- **Acțiunile afectează observațiile:** Să zicem ca robotul din exemplele anterioare primește ca observație distanța dintre cap și podea, evident că această valoare este modificată de modul în care agentul se deplasează. Deci putem trage concluzia că observațiile pe care un agent le face în timpul procesului de învățare nu sunt întocmai independente, de fapt ele sunt o funcție a propriilor acțiuni ale agentului, pe care acesta le decide pe baza observațiilor sale anterioare.

2.2 Cadrul formal

Problema generală de învățare ranforsată este formalizată ca un proces de control stocastic de timp discret, unde agentul interacționează cu mediul său în următoarea manieră: agentul începe procesul de învățare într-o anumită stare $s_0 \in S$ (spațiul stărilor) prin colectarea unei observații inițiale $\omega_0 \in \Omega$ (spațiul observațiilor). La fiecare pas în timp t agentul trebuie să întreprindă o acțiune $a_t \in A$. După cum este ilustrat și în figura 2.1 aceasta urmează trei consecințe:

- (a) agentul primește o recompensă $r_t \in R$
- (b) starea actuală trece la starea $s_{t+1} \in S$
- (c) agentul obține noua observație $\omega_{t+1} \in \Omega$

Acest cadru de control a fost propus pentru prima dată de către Richard Bellman în anul 1957 și apoi extins la învățare de către Andrew G. Barto în anul 1983 [4].

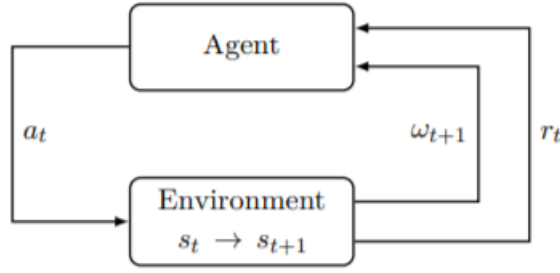


Figura 2.1: Schemă interacțiune agent-mediu. Preluat din [4].

2.2.1 Proprietatea Markov

În cadrul învățării ranforsate, agentul ia deciziile sale ca funcția unui semnal din ambient numit starea mediului. În particular definim formal o proprietate a mediilor și semnalelor de stare ale acestora de interes special numită proprietatea Markov [4].

Definiția 2.1. *Un proces de control stocastic de timp discret este Markovian (adică are proprietatea Markov) dacă:*

$$(i) \ P(\omega_{t+1}|\omega_t, a_t) = P(\omega_{t+1}|\omega_t, a_t, \dots, \omega_0, a_0)$$

$$(ii) \ P(r_t|\omega_t, a_t) = P(r_t|\omega_t, a_t, \dots, \omega_0, a_0)$$

Mai specific, proprietatea Markov se referă la faptul că viitorul procesului depinde doar de observația curentă iar agentul nu are nici un interes privind întregul istoric al observațiilor. Un proces decizional Markov (sau MDP) este un proces de control stocastic de timp discret definit după cum urmează [4]:

Definiția 2.2. *Un proces decizional Markov este tuplul de cinci elemente $\langle S, A, T, R, \gamma \rangle$ unde:*

- S este spațiul stărilor,
- A este spațiul acțiunilor,
- $T : S \times A \times S \rightarrow [0, 1]$ este funcția de tranziție (set de probabilități de tranziție condiționate între stări),

- $R : S \times A \times S \rightarrow \hat{R}$ este funcția de recompensă, unde \hat{R} este un set continuu de compensații posibile.
- $\gamma \in [0, 1]$ este factorul de actualizare (sau factorul de reducere).

Sistemul este complet perceptibil într-un MDP (ilustrat în figura 2.2), ceea ce înseamnă că observația este aceeași cu starea mediului: $\omega_t = s_t$. La fiecare pas în timp t , probabilitatea de trecere către starea s_{t+1} este oferită de către funcția de tranziție $T(s_t, a_t, s_{t+1})$ și recompensa r_t este dată de o funcție de recompensă delimitată $R(s_t, a_t, s_{t+1}) \in \hat{R}$ [4].

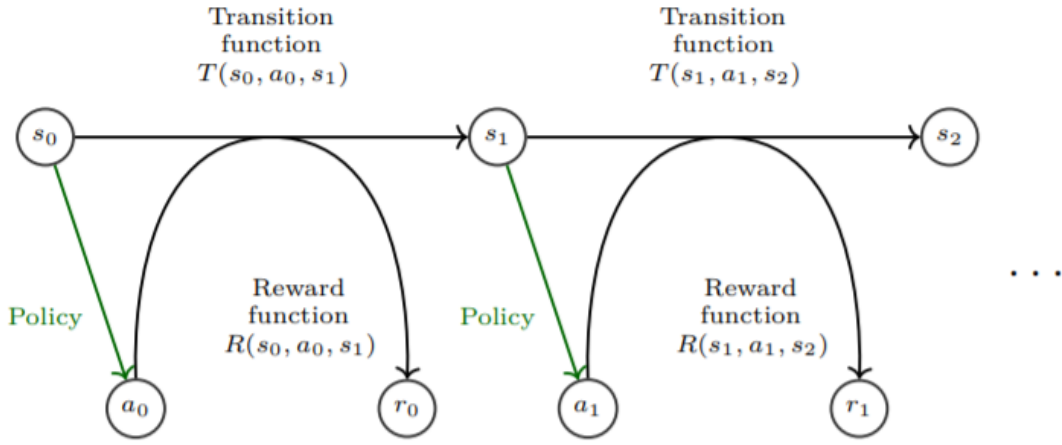


Figura 2.2: Ilustrarea unui MDP. La fiecare pas agentul ia o acțiune care îi schimbă starea în mediu și îi oferă (eventual) o recompensă r_t . Preluat din [4].

2.2.2 Diferite categorii de politici

Politica (notată cu π) caracterizează comportamentul agentului, modul în care acesta selectează acțiunile. Politicile pot fi clasificate sub criteriul de a fi staționare sau nestaționare. O politică staționară nu depinde de pasul în timp, însemnând că agentul va lua aceeași decizie ori de câte ori sunt îndeplinite anumite condiții, poate fi probalistică (nedeterministă) ceea ce implică faptul că probabilitatea de a alege o acțiune rămâne aceeași chiar dacă agentul ia decizii diferite. Pe când cea din urmă depinde de pasul în timp și este utilă în contextul orizontului finit în care recompensele cumulative pe care agentul caută să le optimizeze sunt limitate la un număr finit de pași în viitor [4].

Politicile pot fi clasificate sub un al doilea criteriu, ele sunt fie deterministe fie stocastice:

- În cazul determinist politica este descrisă de: $\pi(s) : S \rightarrow A$
- În cazul stocastic politica este descrisă de: $\pi(s, a) : S \times A \rightarrow [0, 1]$ unde $\pi(s, a)$ denotă probabilitatea ca acțiunea a să fie aleasă în starea s .

2.2.3 Recompensă redusă și funcția obiectiv

Pentru acțiunile făcute în mediul de antrenare agentul primește o recompensă r_t . Recompensa cumulativă poate fi scrisă sub forma unei sume. Adăugând factorul de reducere γ facem ca viitoarele recompense să fie mai puțin importante și transformăm suma într-una finită. Recompensa redusă poate fi definită ca

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k}. \quad (2.1)$$

Scopul învățării ranforsate este de a maximiza randamentul redus preconizat. Având în vedere o politică π funcția obiectiv poate fi formalizată ca

$$J = \mathbb{E}_{r_i, s_i \sim E, a_i \sim \pi} [G_1] \quad (2.2)$$

unde starea s_i este prelevată din mediul E și acțiunile a_i sunt prelevate din politica π [13].

2.2.4 Funcțiile de valoare a stării și a acțiunii

Funcția de valoare a stării este definită ca suma preconizată a recompenselor viitoare care urmează politica π , adică

$$V^\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \quad (2.3)$$

și poate fi exprimată recursiv conform ecuației lui Bellman ca

$$V^\pi(s) = \mathbb{E}_\pi[r_t + \gamma V^\pi(s_{t+1}) | S_t = s] \quad (2.4)$$

Funcția de valoare a acțiunii poate fi descompusă în mod similar începând cu returul

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \quad (2.5)$$

până la forma recursivă obținută cu ecuația lui Bellman [13]:

$$Q^\pi(s, a) = \mathbb{E}_\pi[r_t + \gamma Q^\pi(s_{t+1}, a_{t+1}) | S_t = s, A_t = a] \quad (2.6)$$

2.2.5 Componente necesare pentru învățarea unei politici

Un agent de învățare automată include una sau mai multe din următoarele componente:

- O reprezentare a unei funcții de valoare care oferă o predicție a cât de bună este fiecare stare sau fiecare pereche stare-acțiune,
- o reprezentare directă a politicii $\pi(s)$ sau $\pi(s, a)$, sau
- un model al mediului (estimări ale funcției de tranziție și de recompensă).

Primele două componente sunt legate de ceea ce se numește învățare ranforsată fără model (*model-free*), unde agentul poate învăța o politică în mod direct folosind algoritmi precum *Q-Learning* sau metode *policy gradient*. A se nota că agenții prezentați în această lucrare sunt antrenați prin algoritmi *model-free*. Când se folosește ultima componentă, algoritmul este denumit învățare ranforsată bazată pe model (*model-based*), unde agentul încearcă să înțeleagă lumea prin capturarea funcțiilor de tranziție și de recompensă cu ajutorul cărora își formează o referință și poate planifica în consecință [4].

2.2.6 *On-policy vs. Off-policy*

Algoritmii principali propuși antrenării prin setul de instrumente ML-Agents sunt PPO (*Proximal Policy Optimization*) și SAC (*Soft Actor Critic*).

Un algoritm de politică (*on-policy*) precum PPO colectează un număr de probe pe baza cărora învață să-și îmbunătățească politica și o actualizează în consecință, crescând probabilitatea de a întreprinde acțiuni recompensate și scăzând probabilitatea celor care nu sunt recompensatoare. Majoritatea algoritmilor moderni de politică, cum ar fi PPO învață și o formă a unei funcții de evaluare (vezi 2.1.3 și 2.1.4) pentru estimarea seriei recompenselor cu ajutorul căreia se antrenează mai stabil [15].

Algoritmii în afara politicii (*off-policy*) precum SAC lucrează în mod diferit. Presupunând că mediul are o dinamică fixă și o funcție de recompensare, există o relație optimă între efectuarea unei anumite acțiuni într-o anumită stare și obținerea unei recompense cumulative. În loc să învețe cât de bună este politica actuală, acești algoritmi învață o funcție optimă de evaluare în toate politicile. Problema este mai dificilă decât în cazul anterior deoarece această funcție poate fi foarte complexă, însă deoarece este globală se pot folosi eșantioanele colectate de la începutul timpului pentru învățarea acestui evaluator [15].

2.2.7 *Exploatare vs. Explorare*

O problemă fundamentală a învățării ranforsate este dilema exploatării și a explorării. Pentru ca agentul să ia decizii optime sau aproape optime, trebuie să exploreze spațiul cu scopul de a aduna mai multe informații pentru optimizarea politicii actuale. O politică ce are o rată ridicată de exploatare va avea probleme pentru a converge spre un comportament optim deoarece poate abuza anumite strategii pentru maximizarea recompensei conducând la un comportament "nenatural". În schimb o politică care explorează prea mult va atinge cu greu îmbunătățiri ale comportamentului deoarece agentul va acționa în mare parte aleatoriu. O orientare rațională este de a avea o strategie de explorare cu o rată mare inițială dar care se reduce în timp pentru a vedea îmbunătățiri în cadrul politicii.

2.3 Tehnici de tip *Policy Gradient*

Această secțiune este concentrată pe o anumită familie de algoritmi de învățare ranforsată care utilizează metode *policy gradient* ce optimizează un obiectiv de performanță (de obicei recompensa cumulată așteptată) prin găsirea unei politici bune.

2.3.1 *Stochastic policy gradient*

Returul așteptat al unei politici stocastice π începând de la o stare dată s_0 poate fi scris ca:

$$V^\pi(s_0) = \int_S p^\pi(s) \int_A \pi(s, a) R'(s, a) da ds \quad (2.7)$$

unde $R'(s, a) = \int_{s' \in S} T(s, a, s') R(s, a, s')$ și $p^\pi(s)$ este distribuția redusă a stărilor definită ca

$$p^\pi(s) = \sum_{t=0}^{\infty} \gamma^t Pr\{s_t = s | s_0, t\}. \quad (2.8)$$

Pentru o politică diferențiabilă π_w , rezultatul fundamental care stă la baza acestor algoritmi este teorema gradientului de politică [13]:

$$\nabla_w V^{\pi_w}(s_0) = \int_S p^{\pi_w}(s) \nabla_w \pi_w(s, a) Q^{\pi_w}(s, a) da ds \quad (2.9)$$

Acest rezultat ne permite să adaptăm parametrii politicii din experiență și este interesant în mod particular deoarece gradientul politicii nu depinde de gradientul distribuției stărilor. Cel mai simplu mod de a obține estimatorul gradientului de politică (adică $\nabla_w V^{\pi_w}(s_0)$) este de a folosi o funcție scor estimator gradient [4].

Trucul de raport al probabilității poate fi exploatat după cum urmează pentru a obține o metodă generală de estimare a gradientilor:

$$\begin{aligned}
 \nabla_w \pi_w(s, a) &= \pi_w(s, a) \frac{\nabla_w \pi_w(s, a)}{\pi_w(s, a)} \\
 &= \pi_w(s, a) \nabla_w \log(\pi_w(s, a))
 \end{aligned} \tag{2.10}$$

Având în vedere ecuația 2.10, rezultă că

$$\nabla_w V^{\pi_w}(s_0) = \mathbb{E}_{s \sim p^{\pi_w}, a \sim \pi_w} [\nabla_w (\log \pi_w(s, a)) Q^{\pi_w}(s, a)] \tag{2.11}$$

Metodele *policy gradient* trebuie să includă o evaluare a politicii urmată de o îmbunătățire a politicii. Pe de o parte evaluarea politicii estimează Q^{π_w} . Pe de altă parte îmbunătățirea politicii face un pas în gradient pentru a optimiza politica $\pi_w(s, a)$ cu respect față de estimarea funcției de valoare. Pentru a preveni politica să devină deterministă, este comună adăugarea unui regulator de entropie la gradient ce asigură că politica continuă să fie explorată [4].

O remarcă importantă este că în ecuația 2.11, funcția de valoare Q^{π_w} poate fi înlocuită cu o funcție avantaj A^{π_w} . În timp ce $Q^{\pi_w}(s, a)$ rezumă performanța fiecărei acțiuni pentru o anumită stare conform politicii π_w , funcția de avantaj $A^{\pi_w}(s, a)$ oferă o măsură de comparație pentru fiecare acțiune la returul așteptat pentru starea s , dat de $V^{\pi_w}(s)$. Funcția de avantaj $A^{\pi_w}(s, a) = Q^{\pi_w}(s, a) - V^{\pi_w}(s)$ oferă în general magnitudini mai mici decât $Q^{\pi_w}(s, a)$, lucru ce ajută la reducerea varianței estimatorului de gradient $\nabla_w V^{\pi_w}(s_0)$ în etapa de îmbunătățire a politicii, fără a modifica așteptările. Cu alte cuvinte, funcția de valoare $V^{\pi_w}(s)$ poate fi văzută ca o variație de bază sau de control pentru estimatorul de gradient [4].

2.3.2 *Natural Policy Gradient*

Această metodă este inspirată de ideea gradientilor naturali pentru actualizarea politicii și folosește cea mai abruptă direcție dată de metrica informației Fisher, care folosește

varietatea funcției obiectiv. În cea mai simplă formă de ascensiune a celei mai abrupte direcții pentru o funcție obiectiv $J(w)$, actualizarea are forma $\Delta w \propto \nabla_w J(w)$. Cu alte cuvinte, actualizarea urmează direcția care maximizează $(J(w) - J(w + \Delta w))$ sub o constrângere asupra $\|\Delta w\|_2$. În cazul ipotezei în care constrângerea de pe Δw este definită cu o metrică diferită de L_2 , soluția de ordin prim a problemei de optimizare constrânsă are de obicei forma $\Delta w \propto B^{-1} \nabla_w J(w)$ unde B este o matrice $n_w \times n_w$. În gradienti naturali, norma utilizează metrica informației Fisher, dată de o aproximare patratică locală la divergența (Kullback-Leibler) $D_{KL}(\pi^w || \pi^{w+\Delta w})$. Ascensiunea în gradient natural pentru îmbunătățirea politicii π_w este dată de

$$\Delta w \propto F_w^{-1} \nabla_w V^{\pi_w}(\cdot), \quad (2.12)$$

unde F_w este matricea de informații Fisher dată de

$$F_w = \mathbb{E}_{\pi_w} [\nabla_w \log \pi_w(s, \cdot) (\nabla_w \log \pi_w(s, \cdot))^T]. \quad (2.13)$$

Metodele *policy gradient* ce urmăresc $\nabla_w V^{\pi_w}(\cdot)$ sunt adesea lente pentru că sunt predispușe blocării în platouri locale. În schimb, gradientii naturali nu urmăresc direcția cea mai abruptă obișnuită din spațiul parametrilor, ci cea mai abruptă direcție în raport cu metrica Fisher [4].

O problema a gradientilor naturali este că în cazul rețelelor neuronale cu număr mare de parametri, este de obicei impracticabil pentru a calcula, inversa și stoca în memorie matricea de informații Fisher, motiv pentru care această tehnică este de obicei neutilizată pentru învățare ranforsată profundă. Cu toate acestea au fost descoperite alternative inspirate de această idee, una despre care vom discuta în secțiunea următoare [4].

2.3.3 Optimizarea regiunii de încredere

Ca o modificare a metodei gradientului natural, metodele de optimizare a politicii bazate pe o regiune de încredere vizează îmbunătățirea politicii într-un mod controlat. Aceste

metode de optimizare a politicilor bazate pe constrângeri se axează pe restricționarea modificărilor aduse unei politici utilizând divergența Kullback Leibler (KL) între distribuțiile acțiunii. Prin limitarea dimensiunii actualizării politicii, metodele regiunii de încredere leagă modificările aduse în distribuția stărilor, asigurând îmbunătățiri ale politicii [4].

Trust Region Policy Optimization (TRPO) folosește actualizări constrânse și estimarea funcției de avantaj pentru efectuarea actualizării, rezultând astfel optimizarea reformulată prin

$$\max_{\Delta w} \mathbb{E}_{s \sim p^{\pi_w}, a \sim \pi_w} \left[\frac{\pi_w + \Delta w(s, a)}{\pi_w(s, a)} A^{\pi_w}(s, a) \right] \quad (2.14)$$

supusă relației: $\mathbb{E} D_{KL}(\pi_w(s, \cdot) || \pi_w + \Delta w(s, \cdot)) \leq \delta$, unde $\delta \in \mathbb{R}$ este un hiperparametru. Deși TRPO este un algoritm foarte puternic, acesta suferă de o problemă semnificativă și anume constrângerea, care adaugă costuri suplimentare problemei de optimizare. Metoda următoare propune includerea constrângerii direct în obiectivul de optimizare [4].

2.3.4 *Proximal policy optimization (PPO)*

Calea spre succes în învățarea prin ranforsare nu este una evidentă, algoritmi sunt greu de depanat și necesită în general eforturi substanțiale pentru a obține rezultate bune. PPO atinge un echilibru între ușurința de implementare, complexitatea probelor și ușurința de reglare, încercând să calculeze o actualizare la fiecare pas care să minimizeze funcția de cost, asigurând în același timp că abaterea de la politica anterioară este relativ mică. Algoritmul care a devenit de încredere și la OpenAI este una din propunerile de bază ale setului ML-Agents și a fost folosit la antrenarea agenților din această lucrare.

Proximal policy optimization este o variantă a algoritmului TRPO, care formulează constrângerea ca o penalizare sau ca un obiectiv de tăiere în loc să utilizeze constrângerea KL. Spre deosebire de TRPO, PPO are în vedere modificarea funcției obiectiv pentru a penaliza modificări ale politicii care mută $r_t(w) = \frac{\pi_w + \Delta w(s, a)}{\pi_w(s, a)}$ departe de 1. Obiectivul de

tăiere pe care PPO îl maximizează este dat de

$$\mathbb{E}_{s \sim p^{\pi_w}, a \sim \pi_w} [\min(r_t(w)A^{\pi_w}(s, a), \text{clip}(r_t(w), 1 - \epsilon, 1 + \epsilon)A^{\pi_w}(s, a))] \quad (2.15)$$

unde $\epsilon \in \mathbb{R}$ este un hiperparametru. Această funcție obiectiv conservă raportul de probabilitate pentru a constrânge modificările aduse lui r_t în intervalul $[1 - \epsilon, 1 + \epsilon]$ [4].

2.3.5 Metode actor-critic

Arhitectura actor-critic folosește două structuri pentru a optimiza returul așteptat și anume actorul ce folosește gradientele derivate din teorema gradientului de politică (vezi ecuația 2.9) și ajustează parametrii w ai politicii și criticul parametrizat cu θ ce estimează funcția de valoare aproximativă pentru politica curentă π : $Q(s, a; \theta) \approx Q^\pi(s, a)$ [4].

Criticul

Dintr-un set de tupluri $\langle s, a, r, s' \rangle$, posibil preluate dintr-o reluare din memorie, cea mai simplă abordare în afara politicii de estimare a criticilor este de a folosi un algoritm de bootstrapping (precum $TD(0)$) unde, la fiecare iterație, valoarea curentă $Q(s, a; \theta)$ este actualizată către o valoare țintă:

$$Y_k^Q = r + \gamma Q(s', a = \pi(s'); \theta) \quad (2.16)$$

Această abordare are avantajul de a fi simplă, dar nu este eficientă d.p.d.v. al calculului întrucât folosește o tehnică pură de bootstrapping care este predispusă la instabilități și are o propagare lentă a recompensei înapoi în timp [4].

Actorul

Din ecuația 2.11 gradientul *off-policy* în faza pentru îmbunătățirea politicii pentru cazul stocastic este dat ca:

$$\nabla_w V^{\pi_w}(s_0) = \mathbb{E}_{s \sim p^{\pi_\beta}, a \sim \pi_\beta} [\nabla_\theta (\log \pi_w(s, a)) Q^{\pi_w}(s, a)] \quad (2.17)$$

unde β este o politică de comportament în general diferită de π . Această utilizare se comportă în mod corect în practică, dar utilizează un estimator de gradient de politică părtinitor ceea ce face dificilă analiza convergenței sale [4].

O alternativă este de a combina eșantioane în afara politicii și de politică pentru a obține un echilibru folosind eficiența d.p.d.v. al probelor în cazul metodelor în afara politicii, cât și stabilitatea estimărilor gradientului de politică. De exemplu, Q-Prop folosește un estimator de gradient de politică (on-policy) Monte Carlo, în timp ce reduce varianța estimatorului de gradient prin utilizarea unui critic în afara politicii (off-policy) [4].

2.3.6 *Soft Actor-Critic* (SAC)

Cel de-al doilea algoritm de bază propus pentru antrenarea agenților cu setul de instrumente ML-Agents este SAC care optimizează o politică stocastică într-un mod în afara politicii (*off-policy*), formând o punte între optimizarea politicii stocastice și abordările în stil DDPG (*Deep Deterministic Policy Gradient*) [11].

O caracteristică centrală a SAC este regularizarea entropiei. Politica este instruită pentru a maximiza un compromis între returul așteptat și entropie, o măsură a aleatorului în politică. Această regularizare are o legătură strânsă cu compromisul explorare-exploatare: creșterea entropiei are ca rezultat o explorare mai mare, care poate accelera învățarea ulterior, de asemenea poate împiedica convergența prematură a politicii la un optim local prost [11].

SAC învață simultan o politică π_θ și două funcții Q_{ϕ_1}, Q_{ϕ_2} , de valoare învățate în mod similar cu *TD3* [11]:

Similarități:

1. Ambele funcții Q sunt învățate cu minimizarea MSBE, regresând la o singură țintă partajată.
2. Ținta partajată este calculată utilizând rețelele Q țintă, obținute prin aplicarea mediei Polyak parametrilor rețelei Q pe parcursul antrenamentului.
3. Ținta partajată folosește trucul dublu Q tăiat. (Clipped double Q trick).

Diferențe:

1. Tinta include, de asemenea, un termen care provine din utilizarea regularizării entropiei de către SAC.
2. Acțiunile următoarei stări utilizate în țintă provin din politica actuală în loc de o politică țintă.
3. TD3 pregătește o politică deterministă, pe când SAC formează o politică stocastică.

Capitolul 3

Descrierea metodelor de antrenare și de evaluare

3.1 Structura generală a unui agent ML-Agents

Un agent este o entitate care primește observații ale mediului pe baza cărora decide și execută cea mai bună cale de acțiune (conform unei politici) în mediul respectiv. A se nota că acest cadru inițial este prezentat pentru versiunea 0.14.0 a setului de instrumente ML-Agents.

3.1.1 Scriptare

Fiecare agent trebuie să conțină un script ce extinde (prin moștenire) clasa de bază *Agent* și include implementarea următoarelor metode:

Agent.InitializeAgent()

Este apelată o singură dată atunci când agentul este activat. Folosită pentru *cache* de componente sau inițializarea anumitor parametri (e.g. componenta fizică *rigidbody*)

Agent.CollectObservations()

Această metodă este utilizată pentru a oferi ca observații aspecte ale mediului care sunt

numerice și non-vizuale fiind apelată la fiecare pas în care agentul cere o decizie. Pentru adăugarea a noi observații în vectorul de observații implementarea acestei metode folosește metoda *AddVectorObs()* ce oferă o serie de supraîncărcări pentru adăugarea tipurilor comune de date la vectorul de observații precum numere întregi sau reale, *boolean*, *Vector2*, *Vector3*, *Quaternion*, etc... [10].

Agent.AgentAction(float[] vectorAction)

O acțiune este o instrucțiune din politică pe care agentul o desfășoară. Atât algoritmul de instruire cât și politica nu știu nimic despre ceea ce înseamnă acțiunile în sine. Algoritmul de antrenament pur și simplu încearcă valori diferite pentru lista de acțiuni *vectorAction* și observă efectul asupra recompenselor acumulate în timp pe mai multe episoade de antrenament. Există două tipuri de acțiuni pe care un agent le poate utiliza [10]:

- **Discrete:** În acest caz parametrul *vectorAction* este un tablou care conține indici formați din ramuri. O ramură ia câte o valoare întreagă diferită pentru fiecare posibilitate a unei acțiuni.
- **Continue:** Aici parametrul *vectorAction* este un vector de valori reale unde pentru fiecare indice există o semnificație atribuită pe care agentul învață să o controleze.

Agent.Heuristic()

Când tipul de comportament este setat la *Heuristic Only* în parametrii de comportament ai agentului, acesta va folosi metoda *Heuristic()* pentru a genera acțiunile agentului, mai exact pentru a scrie *vectorAction* prin input de la un utilizator uman. Este utilă pentru testarea logicilor din interiorul scenei [10].

Recompense

În învățarea ranforsată, recompensa este un semnal că agentul a făcut ceva corect (sau greșit). Algoritmii furnizați optimizează alegerile pe care agentul le face astfel încât acesta să câștige cea mai mare recompensă cumulativă în timp. Recompensarea agentilor se face prin următoarele metode [10]:

- ***Agent.AddReward(float value)***: Adaugă la recompensa pasului (implicit a episodului) valoarea reală furnizată.
- ***Agent.SetReward(float value)***: Înlocuiește recompensa pasului curent al agentului cu valoarea furnizată și actualizează recompensa episodului în consecință.

3.1.2 Alte tipuri de observații

Observații vizuale

Observațiile vizuale utilizează texturi redată direct de la una sau mai multe camere dintr-o scenă. Politica vectorizează apoi texturile într-un tensor 3D care poate fi alimentat într-o rețea neuronală convoluțională (CNN).

Agenții care utilizează observații vizuale pot surprinde stări de complexitate arbitrară și sunt utili atunci când este dificilă exprimarea stării într-un mod numeric. Cu toate acestea ei sunt de obicei mai puțin eficienți și mai greu de antrenat [10].

Observații de tip *raycast*

Raycast-urile sunt un sistem alternativ pentru a oferi agentului observații bazate pe mediul fizic. În timpul observațiilor, mai multe raze (sau sfere precum în figura 3.1) sunt aruncate în lumea fizică, iar obiectele lovite determină vectorul de observare care este produs [10].

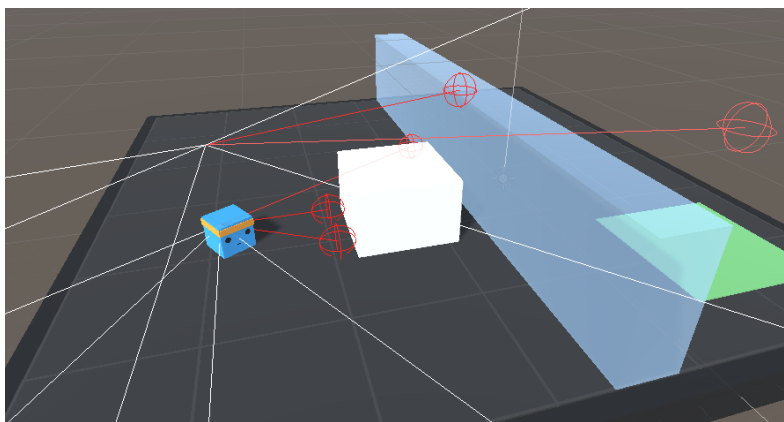


Figura 3.1: Exemplu de observații *raycast* [10]

Implementarea unui astfel de sistem se face prin adăugarea unui *RayPerceptionSensorComponent3D* (figura 3.2) la *gameObject*-ul agentului.

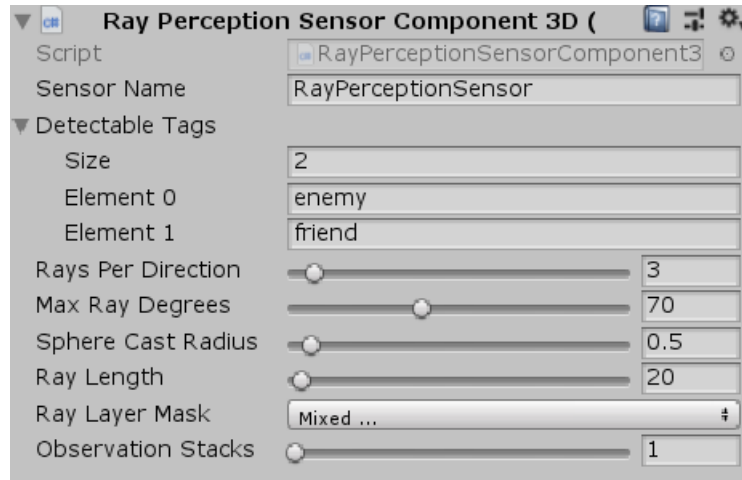


Figura 3.2: (Captură de ecran) Componentă *raycast* în interiorul editorului

Această componentă conține mai multe setări [10]:

- **Detectable Tags:** O listă de șiruri de caractere corespunzătoare tipurilor de obiecte între care agentul ar trebui să fie în măsură să facă distincție.
- **Rays Per Direction:** Determină numărul de raze care sunt aruncate în mediul fizic. O rază este întotdeauna aruncată înainte iar valoarea acestui parametru dictează câte raze sunt aruncate în stânga și în dreapta.
- **Max Ray Degrees:** Unghiul (în grade) pentru razele exterioare. 90° corespund cu stânga și cu dreapta agentului.
- **Sphere Cast Radius:** Mărimea sferei aruncate în mediul fizic. Dacă este setat la 0, se vor utiliza raze în loc de sfere.
- **Ray Length:** Lungimea razelor/sferelor aruncate în mediul fizic.
- **Observation Stacks:** Numărul de rezultate anterioare adăugate la observația curentă.

Un astfel de sistem alimentează rețeaua neuronală (modelul) cu un vector a cărui mărime este egală cu: $(\text{Observation Stacks}) \times (1 + 2 \times \text{Rays Per Direction}) \times (\text{Detectable tags size} + 2)$ [10].

3.1.3 Parametri de comportament

Aplicarea unui script ce moștenește clasa *Agent* unui *GameObject*, adaugă în mod automat componenta script *Behavior Parameters*

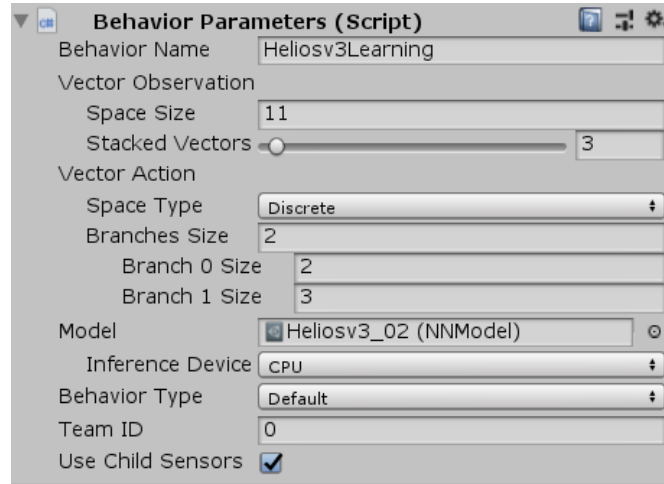


Figura 3.3: (Captură de ecran) Componentă *Behavior Parameters* în interiorul editorului

Această componentă dictează ce politică va forma agentul prin următoarele setări [10]:

- **Behavior Name:** Identificatorul comportamentului. Agenții cu același comportament vor învăța aceeași politică.
- **Vector Observation:** Informații ale vectorului de observații oferit prin metoda *Agent.CollectObservations()*
 1. **Space size:** Lungimea vectorului
 2. **Stacked vectors:** Numărul de observații vectoriale anterioare adăugate la observația curentă. Mărimea efectivă a observației vectoriale transmise politicii este $Space\ Size \times Stacked\ Vectors$. A se nota că acest parametru este independent față de *Observation Stacks* din *RayPerceptionSensorComponent3D*.
- **Vector Action:** Informații ale vectorului de acțiuni oferit de politică funcției *Agent.AgentAction()* sau creat de un utilizator prin *Agent.Heuristic()*.
 1. **Space Type:** Specifică tipul vectorului de acțiuni (discret sau continuu).

2. **Space Size:** (Caz continuu) Reprezintă lungimea vectorului de acțiuni.
 3. **Branches Size:** (Caz discret) Reprezintă lungimea tabloului *Branches*. Se specifică apoi numărul posibilităților pentru fiecare ramură de acțiune prin *Branch i Size*.
- **Model:** Modelul rețelei neuronale utilizat pentru inferență (Obținut după antrenament).
 - **Inference Device:** Dacă folosim procesorul central (CPU) sau procesorul grafic (GPU) pentru a rula modelul în timpul inferenței.
 - **Behavior Type:** Stabilește tipul de comportament.
 1. **Inference Only:** Agentul va efectua întotdeauna procesul de inferență.
 2. **Heuristic Only:** Agentul va folosi întotdeauna metoda *Agent.Heuristic()*.
 3. **Default:** Agentul se va antrena dacă este conectat la un antrenor Python, altfel, va efectua procesul de inferență dacă are un model atașat. Dacă nu are un model atașat va folosi metoda *Agent.Heuristic()*.
 - **Team ID:** Folosit pentru a defini echipa în cazul *self-play*.
 - **Use Child Sensors:** Dacă să folosească sau nu componentele senzor atașate la copii (*GameObjects*) agentului.

3.1.4 Fișierul de configurare a antrenamentului

Setul de instrumente ML-Agents propune configurarea parametrilor prin utilizarea unui fișier YAML (*YAML Ain't Markup Language*). Vom discuta atât despre parametrii comuni ai algoritmilor principali PPO și SAC, cât și despre cei specifici fiecăruia.


```

default-common:
  trainer: ppo
  summary_freq: 5000
  time_horizon: 64
  max_steps: 500000
  learning_rate: 3e-4
  buffer_size: 10240 # implicit PPO | 50000 implicit SAC
  batch_size: 1024
  learning_rate_schedule: linear # PPO | constant SAC
  hidden_units: 128
  num_layers: 2
  normalize: false
  vis_encode_type: simple

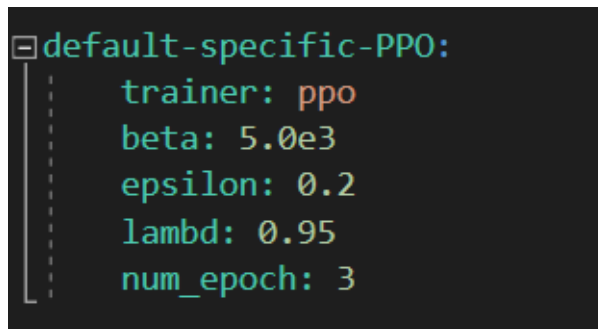
```

Figura 3.4: (Captură de ecran) Parametri de configurare ai antrenamentului comuni pentru PPO și SAC cu valorile implicite.

Configurarea *default-common* din figura 3.4 prezintă următorii parametri [10]:

- ***trainer***: Tipul de antrenor utilizat.
- ***summary_freq***: Numărul de experiențe care trebuie colectate înainte de generarea și afisarea statisticilor de antrenare în TensorBoard.
- ***time_horizon***: Câți pași de experiență să fie colectați pentru fiecare agent înainte de a fi adăugați în *buffer-ul* de experiență.
- ***max_steps***: Numărul total de pași (observații colectate și acțiuni întreprinse) care trebuie făcuți în mediu (sau în toate mediile dacă se utilizează mai multe în paralel) înainte de a încheia procesul de antrenare.
- ***learning_rate***: Rata de învățare inițială pentru coborârea în gradient.
- ***buffer_size***:
 1. ***PPO***: Numărul de experiențe de colectat înainte de actualizarea modelului de politică.
 2. ***SAC***: Dimensiunea maximă a bufferului de experiență. În general de ordinul miilor mai mare decât lungimea episoadelor astfel încât SAC să poată învăța atât din experiențe vechi, cât și din experiențe noi.

- ***batch_size***: Numărul de experiențe în fiecare iterație a coborârii în gradient. Valoarea ar trebui să fie întotdeauna divizor pentru *buffer_size*.
- ***learning_rate_schedule***: Determină modul în care rata de învățare se schimbă în timp.
- ***hidden_units***: Numărul de unități din straturile ascunse ale rețelei neuronale
- ***num_layers***: Numărul de straturi ascunse din rețeaua neuronală.
- ***normalize***: Dacă se aplică normalizarea intrării vectorului de observații.
- ***vis_econder_type***: Tipul codicatorului pentru codarea observațiilor vizuale.



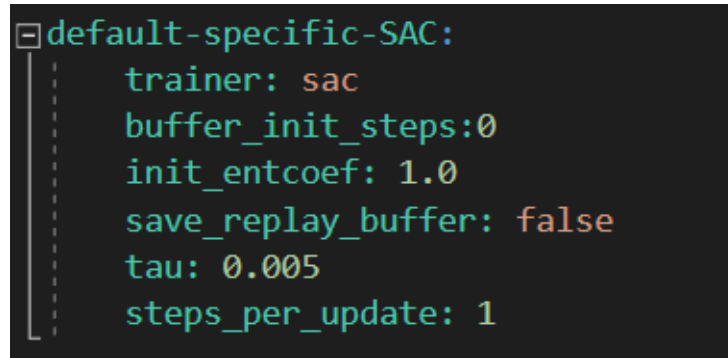
```
default-specific-PPO:  
  trainer: ppo  
  beta: 5.0e3  
  epsilon: 0.2  
  lambd: 0.95  
  num_epoch: 3
```

Figura 3.5: (Captură de ecran) Parametri de configurare ai antrenamentului specifici PPO cu valorile implicite.

Configurarea *default-specific-PPO* din figura 3.5 prezintă următorii parametri [10]:

- ***beta***: Puterea regularizării entropiei, ceea ce face ca politica să fie ”mai aleatorie”. Acest lucru asigură că agenții explorează corect spațiul de acțiune în timpul antrenamentului.
- ***epsilon***: Influențează cât de rapid poate evolua politica în timpul instruirii. O setare mică a acestei valori va duce la actualizări mai stabile, dar va încetini și procesul de antrenare.
- ***lambd***: Parametrul de regularizare utilizat la calcularea estimării avantajului generalizat.

- ***num_epoch***: Numărul de treceri prin bufferul de experiență ce trebuie efectuate atunci când se desfășoară optimizarea coborârii în gradient.

A screenshot of a code editor with a dark background. It shows a configuration block for 'default-specific-SAC' with several parameters: 'trainer' set to 'sac', 'buffer_init_steps' set to 0, 'init_entcoef' set to 1.0, 'save_replay_buffer' set to false, 'tau' set to 0.005, and 'steps_per_update' set to 1. The code is color-coded with green for identifiers, orange for strings, and white for numbers and punctuation.

```
default-specific-SAC:  
  trainer: sac  
  buffer_init_steps: 0  
  init_entcoef: 1.0  
  save_replay_buffer: false  
  tau: 0.005  
  steps_per_update: 1
```

Figura 3.6: (Captură de ecran) Parametri de configurare ai antrenamentului specifici SAC cu valorile implicite

Configurarea *default-specific-SAC* din figura 3.6 prezintă următorii parametri [10]:

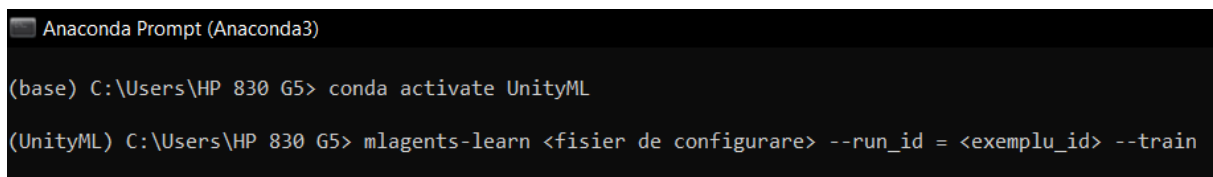
- ***buffer_init_steps***: Numărul de experiențe de colectat în *buffer* înainte de actualizarea modelului de politică.
- ***init_entcoef***: Cât de mult ar trebui să exploreze agentul la începutul antrenamentului.
- ***save_replay_buffer***: Dacă se salvează și se încarcă *buffer-ul* de redare a experienței, precum și modelul, atunci când se încheie și reîncepe antrenamentul.
- ***tau***: Cât de agresiv se actualizează rețeaua țintă Q utilizată pentru estimarea valorii prin *bootstrapping* în SAC.
- ***steps_per_update***: Raportul mediu dintre pașii (acțiunile) agenților și actualizările făcute de politica agentului.

3.1.5 Antrenarea și evaluarea unui agent

Antrenare

Setul de instrumente ML-Agents efectuează instruirea utilizând un proces de antrenare extern Python. În timpul instruirii, acest proces extern comunică cu academia din interiorul motorului de joc pentru a genera un bloc de experiențe al agenților. Aceste experiențe devin setul de antrenare pentru o rețea neuronală utilizată pentru a optimiza politica agentului [10].

Odată ce mediul de învățare (inclusiv agentul) a fost creat și este pregătit pentru instruire, următorul pas este inițierea unui antrenament.



```
Anaconda Prompt (Anaconda3)
(base) C:\Users\HP 830 G5> conda activate UnityML
(UnityML) C:\Users\HP 830 G5> mlagents-learn <fișier de configurare> --run_id = <exemplu_id> --train
```

Figura 3.7: (Captură de ecran) Comenzi necesare pentru inițierea unui antrenament

Prin comanda ***conda activate UnityML*** este activat mediul Python (*UnityML*) în care sunt instalate *Python 3.6*, *TensforFlow 1.7.1* și pachetul Python *mlagents*, ce expune comanda ***mlagents-learn*** care este unicul punct de intrare pentru toate tipurile de antrenare propuse de ML-Agents.

A doua comandă din figura 3.7 este comanda de bază pentru inițierea unui antrenament unde:

- **<fișier de configurare>**: Este calea către fișierul (YAML) de configurare a algoritmului de antrenare.
- **<exemplu_id>**: Este un nume unic utilizat pentru a identifica rezultatele antrenamentelor.

Evaluare

În timpul unei sesiuni de antrenament, programul de antrenament salvează și tipărește actualizări la intervale regulate (specificate de opțiunea *summary_freq* din fișierul de configurare). Statisticile salvate sunt grupate în funcție de valoarea *run-id* (pentru a vizualiza statisticile este necesară atribuirea unui id unic fiecărui model antrenat) [10].

Aceste statistici pot fi urmărite folosind TensorBoard în timpul antrenamentului sau după, executând comanda: ***tensorboard - -logdir=<calea directorului summaries>*** și deschiderea adresei URL: *https://<localhost>:6006*

3.2 Agent carnivor

Primul agent dezvoltat utilizează un model de învățare ranforsată antrenat prin PPO pentru a soluționa sarcina de căutare terestră a unei ținte (pradă).

3.2.1 Pregătirea modelului

În această subsecțiune sunt prezentate alegerile inițiale în ceea ce privește structura modelului de căutare terestră. Modificările aduse pentru optimizarea acestuia sunt prezentate în subsecțiunea următoare.

Acțiuni

Acțiunile pe care agentul le poate lua sunt legate strict de mișcarea în mediul înconjurător și sunt de tip discret, structurate pe două ramuri după cum urmează:

Ramura 1 - <i>forwardValue</i>	Ramura 2 - <i>turnValue</i>
0 → Stă pe loc	-1 → Se rotește la stânga
1 → Înaintează	0 → Nu se rotește
	1 → Se rotește la dreapta

Tabela 3.1: Acțiunile modelului de căutare terestră

forwardValue și *turnValue* sunt apoi alimentate (în fiecare cadru) unor metode ce controlează componenta fizică a agentului pentru aplicarea mișcării.

Observații

Atunci când spunem componenta părinte ne referim la *gameObject*-ul de care este atașat *prefab*-ul agentului (în cazul de față, mediul de antrenare).

1. Numerice:

- Poziția locală (raportată la componenta părinte) normalizată, un vector de 3 valori reale.
- Direcția "înainte" normalizată a agentului, un vector de 3 valori reale.

2. *Raycast* (figura 3.8):

- *Detectable Tags*: 2 (Prada și limitele spațiului de antrenare).
- *Rays Per Direction*: 4
- *Max Ray Degrees*: 77
- *Sphere Cast Radius*: 0.25
- *Ray Length*: 50 unități.

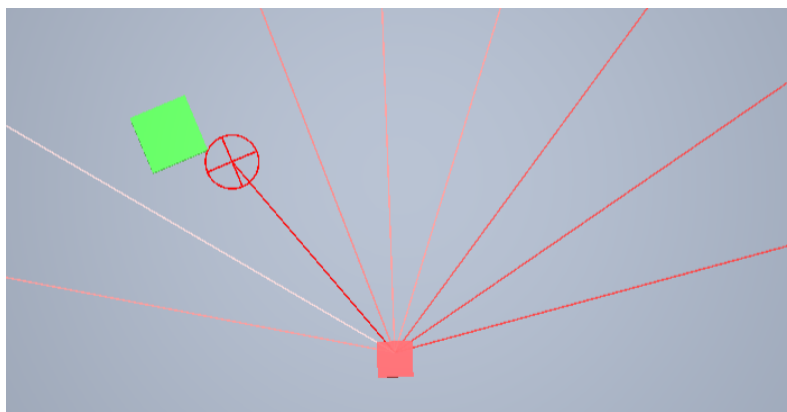


Figura 3.8: (Captură de ecran) Sistemul (inițial) de raze al modelului de căutare terestră (în modul previzualizare scenă)

Atât observațiile numerice cât și observațiile de tip *raycast* sunt stivuite câte trei, deci avem un vector de observații numerice egal cu $3 \times (3+4+3) = 30$ și un vector de observații

raycast a cărui mărime este egală cu (conform formulei de la sfârșitul subsecțiunii 3.1.2):

$$3 \times (1 + 2 \times 4) \times (2 + 2) = 108$$

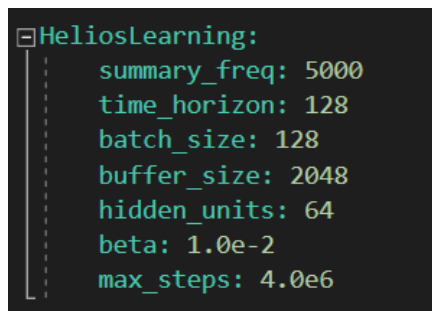
Strutura antrenamentului, recompense și penalizări

Înainte de a prezenta modul în care este recompensat sau penalizat agentul, trebuie discutată structura antrenamentului.

Atât agentul cât și un *gameObject* static ce reprezintă prada sunt instanțiați aleatoriu în scenă. Sarcina agentului este de a intra în contact cu prada (aceasta va fi reinstanciată aleatoriu) de 6 ori, pentru a încheia cu succes un episod de antrenare. Episodul se încheie automat dacă sarcina nu a fost îndeplinită după un număr de pași prestabiliți. Recompensarea agentului se realizează în următorul mod:

- Agentul primește +0.2 pentru fiecare contact cu prada;
- este penalizat cu $-0.01/s$ pentru a încuraja acțiunea;
- atunci când intră în contact cu limitele spațiului de antrenare recompensa este setată la -1 și episodul se încheie.

Parametrizare



```

HeliosLearning:
  summary_freq: 5000
  time_horizon: 128
  batch_size: 128
  buffer_size: 2048
  hidden_units: 64
  beta: 1.0e-2
  max_steps: 4.0e6
  
```

Figura 3.9: (Captură de ecran) Setările (inițiale) pentru parametrii modelului de căutare terestră.

Parametrii care nu sunt suprascriși în configurarea *HeliosLearning* afișată în figura 3.9, vor avea valorile implicite prezentate în subsecțiunea 3.1.4.

În această configurare inițială a fost înjumătățită valoarea parametrului *summary_freq* pentru o afișare mai frecventă a statisticilor în Tensorboard. De asemenea, a fost ales un *batch_size* mai mic decât cel inițial și un *buffer_size* de 16 ori mai mare decât *batch_size* (recomandat să-i fie multiplu). Numărul unităților din straturile ascunse a fost redus la 64 iar pentru a asigura explorarea spațiului acțiunilor, valoarea parametrului *beta* a fost crescută. Numărul maxim de pași alocați instruirii a fost setat la 4.000.000.

3.2.2 Antrenare și evaluare

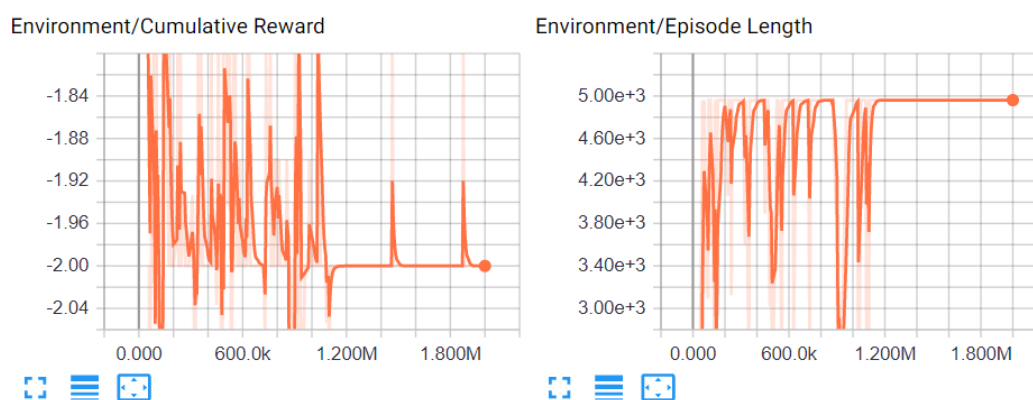


Figura 3.10: (Captură de ecran) Statisticile (mediului) pentru primul model de căutare terestră afișate în Tensorboard.

După verificarea statisticilor de antrenare (figura 3.10) concluzia trasă a fost că versiunea inițială a avut o performanță slabă, cu o sesiune de instruire instabilă. Agentul nu a putut învăța un comportament în care evită limitele scenei și nu a fost nici îndeajuns de stimulat să ”prindă” constant prada.

Îmbunătățiri aduse

În urma câtorva experimente s-a decis întărirea parametrilor de configurare: *batch_size* mărit la 2048 și *buffer_size* de 10 ori mai mare (20480). În legătură cu setările rețelei neuronale, un strat ascuns a fost adăugat și numărul unităților din straturile ascunse a fost setat la 256.

Modificările aduse la parametri au fost complementate de întăriri ale agentului:

- **Observații numerice:** Au fost adăugate observații legate de cea mai apropiată țintă:

1. Distanța normalizată până la cea mai apropiată țintă, un vector de 3 valori.
2. Poziția locală (raportată la componenta părinte) normalizată a țintei, un vector de 3 valori.

- **Observații *raycast*:** *Sphere Cast Radius* de la 0.25 \rightarrow 0.6.

De asemenea, pentru a mări șansele agentului de a intra în contact cu prada au fost adăugate 5 ținte (6 în total). A se ține cont că recompensa maximă pe care un agent o poate atinge într-un episod de antrenare este de 1.2 (0.2×6 ținte), asta dacă nu este luată în calcul penalizarea de existență ce încurajează acțiunea. Recompensa cumulată și durata unui episod variază datorită instanțierii aleatorii a țintelor.

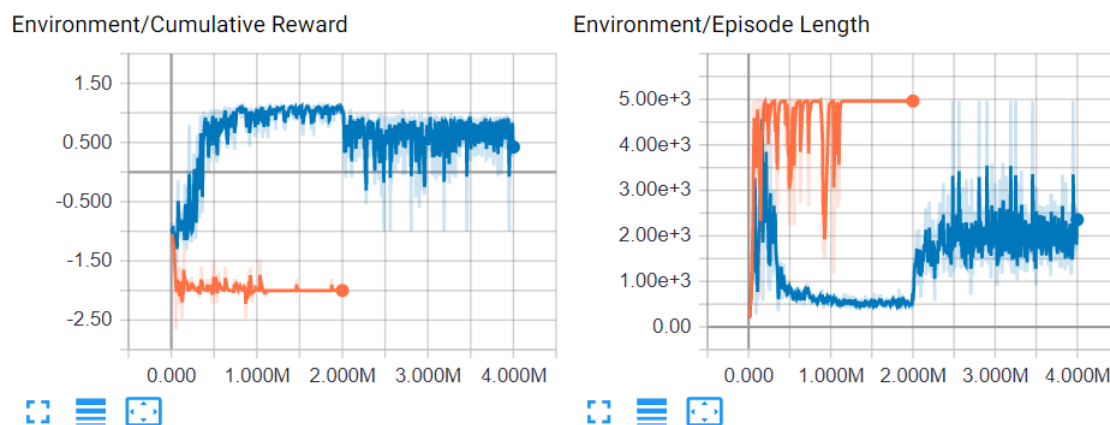


Figura 3.11: (Captură de ecran) Statisticile (mediului) pentru modelul inițial (portocaliu) și versiunea îmbunătățită (albastru) a modelului de căutare terestră afișate în Tensorboard.

Comparând statisticile de antrenare pentru cele două versiuni (figura 3.11) se observă o îmbunătățire a modelului. În prima fază de antrenare (până la 2 milioane de pași) modelul se stabilizează rapid și în jurul a 600.000 de pași agentul începe să pastreze recompensa cumulată între 0.8-1.1 și durata unui episod între 400-600 de pași.

În a doua fază (de la 2 la 4 milioane de pași), după ce agentul a învățat esența comportamentului dorit, au fost eliminate țintele suplimentare. În mod natural această modificare

crește durata episodului și implicit scade recompensa cumulativă deoarece în medie, durează mai mult ca agentul să atingă 6 ținte.

Se poate observa totuși în grafic că există unele episoade problematice, în care dacă agentul este la o distanță prea mare de țintă nu reușește să o mai depisteze, ceea ce înseamnă ca modelul depinde prea mult de observațiile *raycast* pentru a îndeplini sarcina. Astfel au fost aduse noi modificări agentului:

- **Observații numerice:** Poziția locală a țintei a fost eliminată în favoarea a:
 1. Direcția normalizată în care se află ținta, un vector de 3 valori,
 2. Un produs scalar între direcția înainte proprie și direcția în care se află ținta, o valoare reală cuprinsă între -1 (ținta este exact în spatele agentului) și 1 (ținta este în fața agentului)
- **Observații raycast:** Sistemul anterior a fost înlocuit cu unul format din două seturi:
 1. Primul set conține 3 *Rays Per Direction* (7 în total) ce depistează limitele spațiului, cu un *Max Ray Degrees* larg de 80 și *Sphere Cast Radius* de 0.75.
 2. Al doilea este inspirat de modul în care prădătorii au vederea și conține 2 *Rays Per Direction* (5 în total) ce depistează doar prada, cu un *Max Ray Degrees* ascuțit de 35 de grade și *Sphere Cast Radius* de 0.60.
- **Modificări aduse la penalizare:** Penalizarea de încurajare a acțiunii fixă de $-0.01/s$ a fost înlocuită cu o penalizare bazată pe distanța dintre agent și cea mai apropiată țintă cu valori cuprinse în $(-0.01, -0.001)$. În acest mod agentul este încurajat atât să ia acțiuni cât și să se apropie de țintă.

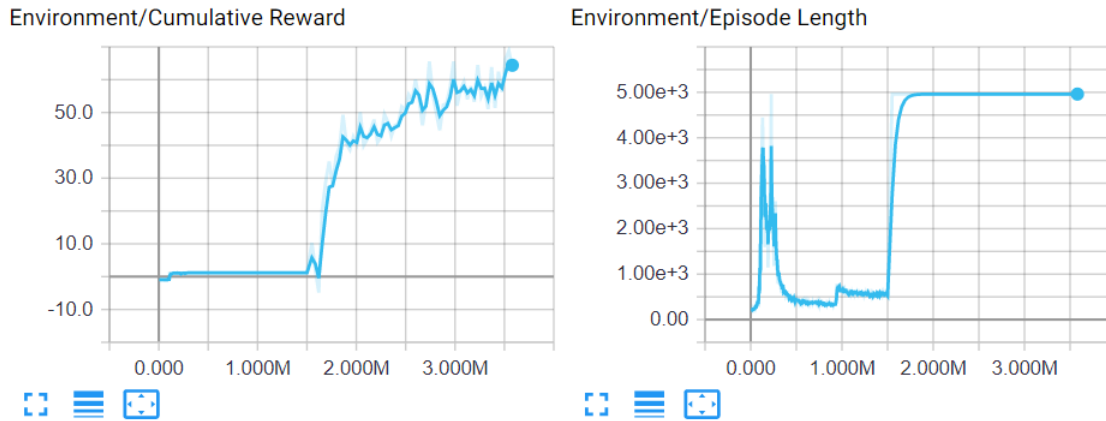


Figura 3.12: (Captură de ecran) Statisticile (mediului) pentru versiunea finală a modelului de căutare terestră afișate în Tensorboard.

După cum se poate observa în figura 3.12, modelul final a fost antrenat în mai multe etape. Până la 930.000 de pași a fost folosită setarea inițială cu 6 ținte și agentul s-a descurcat similar modelului anterior. În schimb, în setarea cu o singură țintă (De la 930.00 pași până la 1.500.000) această versiune a performat mult mai bine.

Problema a revenit când agentul a fost mutat într-o scenă mult mai mare (de scara mediului final, în care se rulează simulari). Dar a fost rezolvată rapid cu modificări la modul în care este recompensat/penalizat agentul:

- Agentul primește $+0.5$ pentru fiecare contact cu ținta,
- este penalizat cu -0.5 pentru fiecare contact cu un obiect limită (marginea spațiului sau obiecte statice din scenă) și
- pe lângă penalizarea continuă bazată pe distanța față de țintă, agentul primește o nouă recompensă/penalizare formată din $(0.1 \times x \in [-1, 1])/s$, unde x e bazat pe produsul vectorial al direcțiilor oferit ca observație, astfel încurajăm agentul să se uite la țintă.

În setarea finală (de la 1.500.000 pași), pe o scenă de scară mai mare și cu un singur erbivor, episodul nu se mai încheie când agentul atinge 6 ținte, astfel durata episodului este maximă (5000 de pași). Odată ce agentul învață să depisteze direcția în care se află ținta și să meargă înspre ea recompensa cumulativă crește semnificativ.

3.3 Agent erbivor

Cel de-al doilea agent creat are ca sarcină găsirea unui partener și evitarea agentului carnivor. În mod inițial a fost încercată utilizarea primului model, cu modificări aduse la observațiile de tip *raycast* (pentru o depistare 360° a pradatorului într-o vecinătate) însă, acțiunile modelului de căutare limitează modul în care agentul erbivor îl poate evita pe cel carnivor în timp ce se deplasează spre o țintă.

3.3.1 Pregătirea modelului

O soluție pe luată în considerare a fost atașarea a două modele la acest agent, unul care să rezolve sarcina de căutare și deplasare spre partener (modelul utilizat și la agentul carnivor) și unul care să rezolve problema evitării prădătorului. Mai departe vom discuta despre dezvoltarea celui din urmă.

Acțiuni

Pentru acest model defensiv a fost aleasă utilizarea unui vector de acțiuni continue $vectorAction = \{x \in [-1, 1], y \in [0, 1], z \in [-1, 1], w \in [0, 1]\}$, prin intermediul căruia agentul decide direcția în care sare ($jumpDirection = \{x, y, z\}$) și forța săriturii ($jumpForce = w$).

Observații

Observațiile pe care modelul le primește sunt un sistem *raycast* (figura 3.13) format din două seturi de raze (unul frontal și unul posterior) cu următoarele setări:

- *Detectable Tags*: 2 (Prădătorul și limitele spațiului)
- *Rays Per Direction*: 7
- *Max Ray Degrees*: 85
- *Sphere Cast Radius* 1.5
- *Ray Length*: 20

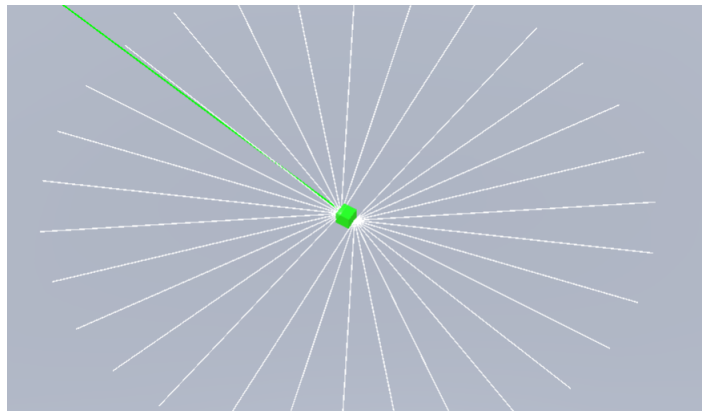


Figura 3.13: (Captură de ecran) Sistemul de raze al modelului defensiv (în modul previzualizare scenă)

Observațiile nu sunt stivuite deci mărimea finală a vectorului de observații *raycast* este (conform formulei de la sfârșitul subsecțiunii 3.1.2): 2 (numărul de seturi de raze) $\times 1 \times (1 + 2 \times 7) \times (2 + 2) = 120$.

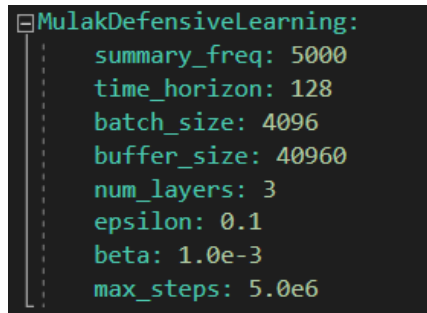
Structura antrenamentului, recompense și penalizări

Fiecare mediu de antrenare conține câte doi agenți: agentul carnivor, cu tipul de comportament (Parametrul *Behaviour Type* din componenta *Behaviour Parameters*) setat pe inferență, adică agentul folosește modelul furnizat dar nu se poate antrena, și agentul erbivor prin care este antrenat modelul defensiv.

Ambii agenți sunt instanțiați aleatoriu la începutul fiecărui episod sau atunci când agentul erbivor intră în contact cu prădătorul sau cu limitele spațiului de antrenare (sarcina agentului erbivor este strict de a evita aceste obiecte). Cu structura antrenamentului clarificată putem vorbi de modul în care este recompensat agentul:

- Agentul este penalizat cu -0.2 pentru fiecare contact cu prada sau cu limitele spațiului de antrenare;
- Primește o răsplată mică pentru existență calculată în funcție de *maxStep*, astfel încât agentul să atingă o recompensă maximă cumulată de 1 (dacă nu este prins de agentul carnivor sau nu atinge limitele spațiului).

Parametrizare



```
MulakDefensiveLearning:
  summary_freq: 5000
  time_horizon: 128
  batch_size: 4096
  buffer_size: 40960
  num_layers: 3
  epsilon: 0.1
  beta: 1.0e-3
  max_steps: 5.0e6
```

Figura 3.14: (Captură de ecran) Setările (inițiale) pentru parametrii modelului defensiv.

Pentru configurarea inițială a parametrilor modelului defensiv (figura 3.14) au fost crescute valorile *batch_size* și *buffer_size*, deoarece modelul utilizează acțiuni continue. Un strat ascuns a fost adăugat rețelei neuronale și valoarea parametrului *epsilon* a fost redusă pentru actualizări mai stabile (și mai lente) ale politicii. Valoarea parametrului *beta* a fost scăzută pentru a reduce factorul de explorare al politicii.

Rezultate

Au fost aduse diferite modificări precum modul în care este recompensat agentul, observațiile pe care le primește, felul în care deciziile agentului sunt aplicate asupra componentei fizice și reglarea parametrilor de configurare ai antrenamentului. Cu toate acestea un comportament dorit nu a fost obținut, concluzionând că *design*-ul nu este unul tocmai optim.

Un nou *design*

Considerând în continuare că problema de evitare a pradătorului trebuie rezolvată prin schimbarea modului în care agentul se deplasează a fost creat un singur model ce folosește un nou sistem de manevrabilitate și conceptele de căutare ale primului model.

Acțiuni

Noul sistem de mișcare propus este bazat pe un set de acțiuni discrete care facilitează modul în care agentul erbivor poate evita prădătorul, structurat pe două ramuri după cum urmează:

Ramura 1 - <i>dashDirectionIndex</i>	Ramura 2 - <i>turnValue</i>
0 → Stă pe loc	-1 → Se rotește la stânga
1 → Alunecă înainte	0 → Nu se rotește
2 → Alunecă înapoi	1 → Se rotește la dreapta
3 → Alunecă la stânga	
4 → Alunecă la dreapta	

Tabela 3.2: Acțiunile modelului folosit pentru agentul erbivor

Valoarea *dashDirectionIndex* dictează direcția în care se va îndrepta agentul după aplicarea unei forțe componente fizice. (Forța este aplicată o dată la *cooldownValue* secunde chiar dacă agentul alege această valoare în fiecare cadru). *turnValue* este aplicată în fiecare cadru și dictează rotația agentului.

Observații

Pentru observațiile de tip *raycast* a fost utilizat sistemul folosit pentru modelul defensiv anterior (figura 3.13). Ca și observații numerice au fost adăugate următoarele:

- Viteza normalizată a componente fizice, un vector de 3 valori reale
- Poziția locală (raportată la componenta părinte) normalizată a agentului și a țintei, 2 vectori de câte 3 valori reale
- Distanța normalizată până la țintă, 1 valoare reală
- Un produs scalar între direcția în care se află ținta și direcția înainte proprie, 1 valoare reală

Structura antrenamentului, recompense și penalizări

Structura antrenamentului este similară versiunii anterioare, însă primele antrenamente (de testare) vor fi rulate fără un carnivor în scenă. Agentul este recompensat după cum urmează:

- Este penalizat cu -0.5 pentru contactul cu limitele spațiului (sau cu prada pentru viitoarele sesiuni);
- Este penalizat cu $x \in (-0.1, -0.01)/s$, unde x este bazat pe distanța dintre agent și țintă
- Primește o recompensă/penalizare $y \in (-0.1, 0.1)/s$, unde y este bazat pe valoarea produsului vectorial dintre direcția înainte și direcția în care se află ținta.

Parametrizare

Considerând atât sarcina cât și deciziile pe care le poate face modelul similare cu agentul carnivor, s-a decis utilizarea ultimei configurații folosite pentru antrenarea modelului de căutare terestră prezentat în subcapitolul anterior.

3.3.2 Antrenare și evaluare

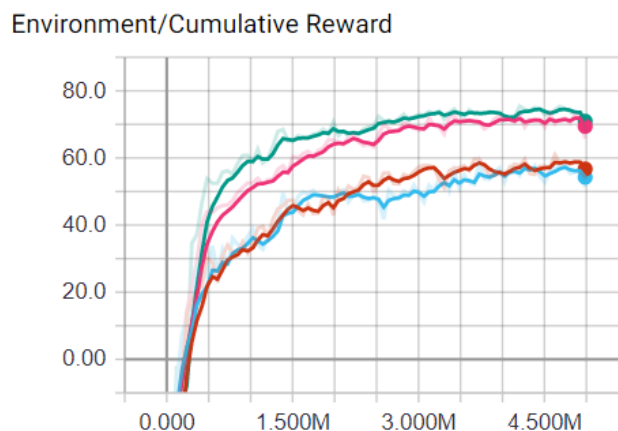


Figura 3.15: (Captură de ecran) Statisticile recompensei cumulate pentru primele versiuni antrenate.

În continuare vor fi prezentate modificările aduse fiecărei versiuni, folosind drept identificator al variantelor culoarea aferentă din graficul afișat în figura 3.15. Varianta inițială (Maro) prezintă performanțe similare cu varianta a cărei grafic este albastru, ce folosește acțiuni stivuite câte trei (stivuirea acțiunilor nu ajută modelul așa că a fost eliminată).

Pentru a întări comportamentul agentului de a se uita la țintă au fost adăugate pentru versiunea roz ca observații numerice direcția normalizată spre țintă și direcția normalizată înainte proprie (doi vectori a câte 3 valori reale). O ultimă problemă a fost că agentul se îndrepta uneori înapoi, în loc să se îndrepte spre țintă. Reconsiderând nevoia agentului de a se deplasa în spate pentru a evita agentul prădător (este mai eficient și suficient să evite prădătorul prin mișcări laterale) a fost eliminată această acțiune din ramura ce dictează direcția de mișcare, îmbunătățind recompensa cumulată (verde). .

În figura 3.16 sunt prezentate rezultatele obținute după adăugarea prădătorului și a obstacolelor în scenă, ce scad în mod natural scorul cumulat deoarece durează mai mult timp ca agentul să ajungă la țintă. Graficul gri reprezintă rezultatele celui mai optim model anterior (verde) în setarea nouă. Modificările aduse mediului au făcut ca valoarea entropiei să scadă mai brusc. Pentru modelul portocaliu, parametrul *beta* a fost redus, îmbunătățind performanța și politica obținute.

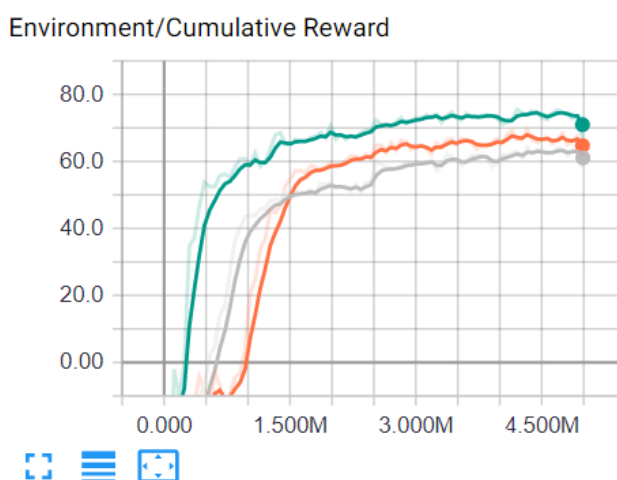


Figura 3.16: (Captură de ecran) Statisticile recompensei cumulate pentru versiunile antrenate cu obstacole și prădător.

3.4 Agent aerian de sprijin

Agentul aerian de sprijin folosește în esență mecanismele de căutare a unei ținte utilizate anterior dar adaugă complexitate modelului prin spațiul acțiunilor. Rolul acestui agent este de a culege fructe din copaci pentru a le oferi apoi agentului erbivor.

3.4.1 Pregătirea modelului

Pentru acest model au fost efectuate mai multe antrenamente și experimente pentru a îmbunătăți structura modelului. În această subsecțiune va fi prezentată structura celei mai optime variante dezvoltate.

Acțiuni

Output-ul pe care îl oferă rețeaua neuronală (acțiunile modelului) este format dintr-un vector (*vectorAction*) ce conține cinci componente:

- **Index 0:** Mișcare pe axa x, pozitiv → dreapta, negativ → stânga,
- **Index 1:** Mișcare pe axa y, pozitiv → sus, negativ → jos,
- **Index 2:** Mișcare pe axa z, pozitiv → înainte, negativ → înapoi,
- **Index 3:** Înclinare (rotație pe axa x), pozitiv → în sus, negativ → în jos,
- **Index 4:** Girație (rotație pe axa y), pozitiv → spre dreapta, negativ → spre stânga.

A se nota că fiind un vector de acțiuni continue (valori reale) agentul are control asupra forțelor pe care le aplică asupra componentei fizice.

Observații

Atunci când spunem componenta părinte ne referim la *gameObject*-ul de care este atașat *prefab*-ul agentului (în cazul de față, mediul de antrenare).

1. Numerice:

- Rotația locală (raportată la componenta părinte) normalizată, un *Quaternion* de 4 valori reale.
- Poziția locală proprie normalizată, un vector de 3 valori reale.
- Poziția locală (normalizată) a celei mai apropiate ținte, un vector de 3 valori reale.
- Distanța normalizată până la țintă, o valoare reală.
- Direcția "înainte" normalizată a ciocului, un vector de 3 valori reale (a).
- Direcția de la cioc la țintă normalizată, un vector de 3 valori reale (b).
- Un produs vectorial între (a) și (b), ce indică agentului dacă este cu fața sau cu spatele la țintă (o valoare reală).
- Un $\text{int } x \in \{0, 1\}$ ce indică agentului dacă a cules deja un fruct sau nu.

2. *Raycast* (figura 3.17): Sistemul *raycast* este conceput din:

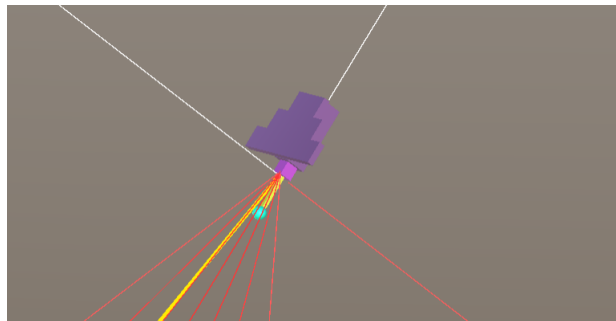


Figura 3.17: (Captură de ecran) Sistemul (complet) de raze al modelului aerian (în modul previzualizare scenă).

- 3 raze singulare de lungime 60, care detectează limitele spațiului și pământul (în jos, în spate și deasupra agentului) cu următoarele setări:
 - *Rays Per Direction* = 0
 - *Max Ray Degrees* = 0
 - *Sphere Cast Radius* = 0
 - *Ray Length* = 60

- Un set de raze frontal, care detectează limitele spațiului, pământul și fructele din copaci cu următoarele setări:
 - *Rays Per Direction* = 3
 - *Max Ray Degrees* = 40
 - *Sphere Cast Radius* = 0.2
 - *Ray Length* = 60

Structura antrenamentului, recompense și penalizări

În timpul antrenamentelor sarcina agentului este de a culege fructe cu ciocul. Agentul este instanțiat în una din opt poziții posibile și cu rotația aleasă aleatoriu pentru a evita *overfitting*-ul.

Spațiul acțiunilor este mai complex decât în cazurile anterioare, fiind astfel necesară folosirea unui plan de învățare (*Curriculum learning*) în care sarcina se îngreunează treptat. Acest plan se bazează pe modificarea succesivă a patru variabile ce definesc dificultatea antrenamentului:

- ***Distance_required (float)***: Distanța la care trebuie să se aproprie agentul de fruct pentru a îndeplini sarcina (0 înseamnă ca trebuie să atingă fructul cu ciocul).
- ***New_pos_value (float)***: Această valoare dictează la câte unități distanță este instanțiat agentul față de centrul scenei.
- ***Random_fruit_position (float)***: Setează mărimea proximității în care poate fi aleasă (aleatoriu) o poziție pentru plantă.
- ***Random_fruit_height (float)***: Componenta ce dictează înălțimea poziției la care este plasat fructul este aleasă pe baza acestei variabile ($transform.position.y \in [-Random_fruit_height, Random_fruit_height]$).

Recompensarea agentului este realizată astfel:

- Agentul primește +1 de fiecare dată când culege livrează un fruct

- În cazul contactului cu pământul sau limitele spațiului de antrenare este penalizat cu -1 și resetat.
- Recompensat/Penalizat cu $\frac{x}{maxStep}$ în fiecare pas în care agentul ia o decizie, unde $x \in [-1, 1]$ și este bazat pe direcția ciocului față de țintă.
- Penalizat cu $\frac{y}{maxStep}$ în fiecare pas în care agentul ia o decizie, unde $y \in [0, 1]$ și este bazat pe distanța până la țintă.

3.4.2 *Imitation learning* (IL)

Precum a fost menționat și mai sus, spațiul complex al acțiunilor (și secvența lungă de acțiuni pe care agentul trebuie să le întreprindă pentru soluționarea sarcinii) crește exponențial numărul experiențelor necesare pentru antrenare. Această subsecțiune prezintă o soluție eficientă (din punct de vedere al probelor) pentru a învăța agentul structura generală a antrenamentului.

Înregistrarea demonstrațiilor

Pentru a putea folosi IL (utilizând setul de instrumente ML-Agents) este necesară înregistrarea unei demonstrații ce conține comportamentul dorit al agentului. Demonstrațiile cuprind informații legate de observațiile, acțiunile și recompensele pe care le primește agentul în timpul sesiunii de înregistrare [9].

Pentru a înregistra o demonstrație este necesară atașarea componentei *Demonstration Recorder* (figura 3.18) la *gameObject*-ul agentului.



Figura 3.18: (Captură de ecran) Componentă *Demonstration Recorder* în interiorul editorului.

După setarea tipului de comportament (*Behaviour Type*) la *Heuristic Only* (agentul este controlat prin *input* de la un utilizator uman) în componenta *Behaviour Parameters* și bifarea variabilei *Record* (din componenta *Demonstration Recorder*) editorul este pregătit pentru înregistrarea demonstrațiilor. Următorul pas este apăsarea butonului *Play* din interiorul editorului, ce permite utilizatorului să controleze agentul și să înregistreze o demonstrație [9].

Tehnicile propuse de ML-Agents pentru IL

Pachetul ML-Agents expune două funcții ce permit agentului să învețe din demonstrații. A se ține cont că în cele mai multe cazuri este recomandată combinarea celor două [9].

1. *Generative Adversarial Imitation Learning* (GAIL)

GAIL este un algoritm IL ce folosește o abordare concurențială. În acest *framework*, o rețea neuronală secundară numită *discriminator*, este învățată să distingă dacă *input*-ul și *output*-ul sunt din demonstrație sau produse de agent. Acest discriminator examinează o nouă observație/acțiune și oferă o recompensă bazată pe similaritatea acestora cu demonstrațiile furnizate [9].

Parametrii de configurare pentru GAIL se adaugă la configurarea modelului ca și semnale de recompensă (*reward_signals*) și sunt prezentați în figura 3.19 cu valorile de bază implicite.

```
default-GAIL:
  reward_signals:
    gail:
      strength: 1
      demo_path: <calea către demonstrație>
      gamma: 0.99
      encoding_size: 64
      use_actions: false
```

Figura 3.19: (Captură de ecran) Parametri de configurare pentru GAIL cu valorile implicite.

Configurarea *default-GAIL* conține următorii parametri [10]:

- ***strength***: Factorul de multiplicare a recompensei brute oferită de discriminator.
- ***demo_path***: Calea către fișierul demonstrației.
- ***gamma***: Factorul de reducere al recompenselor viitoare.
- ***encoding_size***: Mărimea stratului ascuns folosit de discriminator.
- ***use_actions***: Determină dacă discriminatorul folosește și acțiunile.

2. *Behavioral Cloning* (BC)

BC antrenează rețeaua neuronală a agentului pentru a imita exact acțiunile prezentate într-un set de demonstrații. *Behavioral Cloning* tinde să funcționeze cel mai bine atunci când există o mulțime de demonstrații sau împreună cu GAIL și / sau o recompensă *extrinsic* [9].

Parametrii de configurare pentru BC se adaugă la configurarea modelului și sunt prezentați în figura 3.20 cu valorile de bază implicite.

```
default-BC:
  behavioral_cloning:
    strength: 1
    demo_path: <calea către demonstrație>
    steps: 0
    batch_size: # batch size-ul antrenorului (PPO/SAC)
    num_epoch: 3
    samples_per_update: 0
```

Figura 3.20: (Captură de ecran) Parametri de configurare pentru BC cu valorile implicite.

Configurarea *default-BC* conține următorii parametri [10]:

- ***strength***: Rata de învățare a imitației în raport cu rata de învățare a antrenorului (PPO/SAC). Corespunde cu cât de puternic este influențată politica de BC.
- ***demo_path***: Calea către fișierul demonstrației.

- ***steps***: Numărul de pași limită pentru clonarea comportamentului.
- ***batch_size***: Numărul de experiențe demonstrative utilizate pentru o iterație a unei actualizări de coborâre în gradient.
- ***num_epoch***: Numărul de treceri prin *buffer*-ul de experiență în timpul coborârii în gradient.
- ***samples_per_update***: Numărul maxim de probe utilizate în timpul fiecărei actualizări de imitație.

3.4.3 Antrenare și evaluare

Valorile variabilelor ce dictează dificultatea antrenamentului (prezentate în finalul subcapitolului 3.4.1) pentru prima lecție sunt:

- *Distance_required*: 15
- *New_pos_value*: 30
- *Random_fruit_position*: 0
- *Random_fruit_height*: 0

După antrenări inițiale ale algoritmilor PPO/SAC nu au fost obținute rezultate satisfăcătoare, cel puțin, nu într-o sesiune de 3 milioane de pași. Pentru a eficientiza antrenamentul (din punct de vedere al probelor și a timpului), au fost folosite concomitent învățarea ranforșată (RL) și învățarea prin imitare (IL).

Algoritmii de antrenare folosesc inițial IL pentru a învăța structura sarcinii pe care trebuie să o rezolve iar apoi încearcă să depășească performanțele atinse în demonstrația oferită. Rezultatele obținute în prima lecție sunt afișate în figura 3.21:

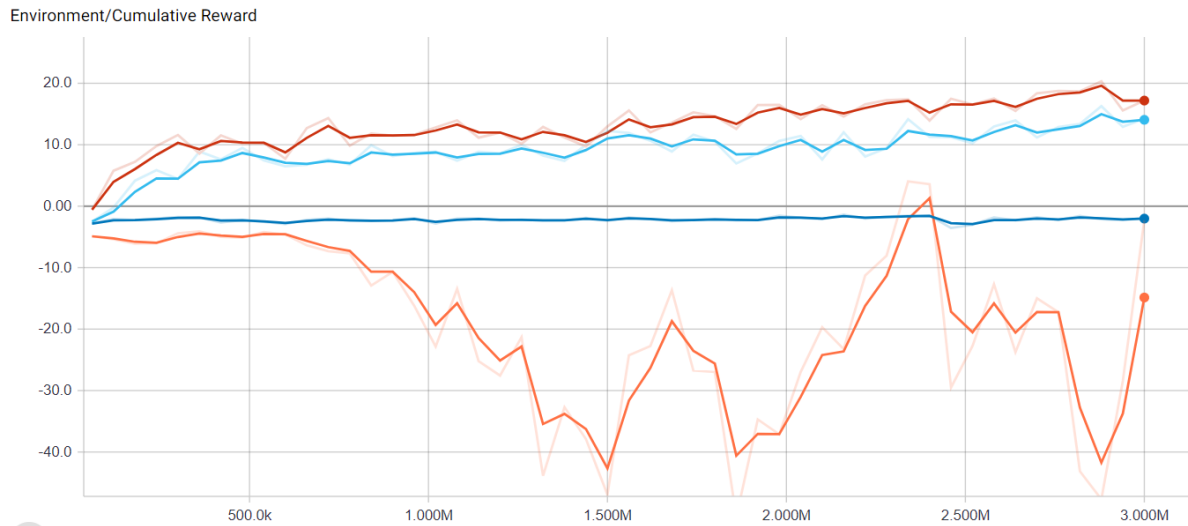


Figura 3.21: (Captură de ecran) Statisticile recompensei cumulative pentru modelele agentului aerian în prima lecție a planului de învățare afișate în Tensorboard.

Legendă figura 3.21:

- Portocaliu: SAC
- Albastru închis: PPO
- Albastru deschis: SAC (GAIL+BC)
- Maro: PPO (GAIL+BC)

Se observă imediat diferența de performanță, de vreme ce fără IL modelul nu poate crea (în 3 milioane de pași) o politică de zbor înspre țintă. Faptul că nu mai oferă acțiuni aleatorii până la descoperirea unei secvențe care să rezolve sarcina oferă un mare avantaj (inițial) versiunilor ce au folosit și învățare prin imitare (pe lângă RL).

Modelele ce au folosit GAIL și BC sunt antrenate în continuare în lecții de câte 4 milioane de pași, cu modificări aduse la variabilele ce definesc dificultatea antrenamentului până la obținerea unui comportament dorit.

Capitolul 4

Utilizarea aplicației

În acest capitol este prezentat un scurt ghid de utilizare al aplicației. Rularea fișierului executabil (.EXE) *MLAgents Ecosystem* deschide un meniu de configurare al setărilor generale (figura 4.1). Selectarea butonului *Play!* pornește aplicația.

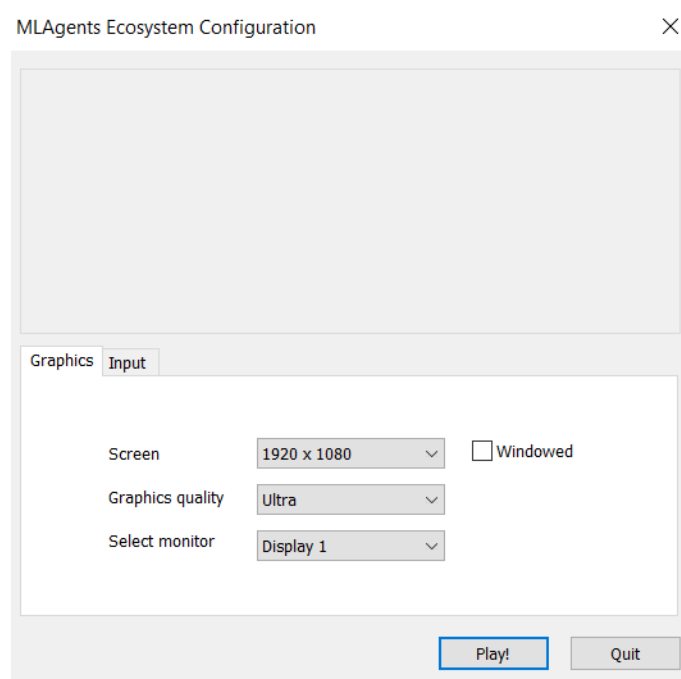


Figura 4.1: (Captură de ecran) Meniu de configurare al setărilor generale.

4.1 Meniul principal

Aplicația se deschide în meniul principal, unde utilizatorul poate selecta (prin folosirea *mouse*-ului) una dintre următoarele două opțiuni:

- ***Prepare simulation***: Pentru a merge mai departe în zona de editare a parametrilor;
- ***Quit***: Pentru a părăsi aplicația.

4.2 Meniul de editare al parametrilor

În meniul de editare al parametrilor (figura 4.2) utilizatorul poate schimba setările agenților prin input de tip *slider*. De asemenea, poate selecta butonul *Back* pentru a reveni la meniul principal sau butonul *Place* pentru a începe procesul de amplasare al agenților. În continuare este prezentată lista parametrilor.



Figura 4.2: (Captură de ecran) Meniul de editare al parametrilor.

- ***Move Speed***: Viteza de mișcare a agentului (*Helios* & *Phaoris*).
- ***Rotation speed***: Viteza de rotație (pe axa Y) a agenților tereștri.

- ***Search Proximity***: Distanța (în unități) în care agentul poate depista o țintă.
- ***Starving Interval***: Limita (în secunde) de timp pe care agenții o au la dispoziție pentru a se hrăni (se resetează când agentul mănâncă).
- ***Dash Force***: Forța pe care agentul erbivor o aplică asupra componentei fizice.
- ***Dash Cooldown***: Intervalul de timp (în secunde) dintre aplicările forței *DashForce* asupra componentei fizice (agentul erbivor).
- ***Mate Proximity***: Distanța necesară (în unități) la care un agent erbivor trebuie să se apropie de un partener compatibil pentru a se multiplica.
- ***Maximum Mulak Agents***: Limita agenților erbivori ce pot fi instanțiați (de utilizator sau prin multiplicare).
- ***Y-Axis Rotation Speed***: Viteza de rotire pe axa Y (girație) a agentului aerian.
- ***X-Axis Rotation Speed***: Viteza de rotire pe axa X (înclinare) a agentului aerian.
- ***Delivery Distance***: Distanța necesară (în unități) la care un agent aerian trebuie să se apropie de un agent erbivor pentru a da drumul la fruct.

4.3 Amplasarea agenților

Amplasarea agenților se realizează în scena în care sunt rulate simulările. Utilizatorul controlează o cameră *first person view* prin *input* de la *mouse*, cu care dictează rotația camerei și *input* de la tastatură prin care schimbă poziția camerei după cum urmează:

- **Tasta W**: Înainte
- **Tasta S**: Înapoi
- **Tasta A**: La stânga
- **Tasta D**: La dreapta

- **Tasta Q:** Urcă
- **Tasta E:** Coboară



Figura 4.3: (Captură de ecran) *Actionbar* (element G.U.I.)

Tastele afișate în *actionbar*-ul din scena principală în modul de amplasare al agenților prezintă următoarele roluri:

- **Tastele numerice:** Selectează în funcție de valoare unul dintre agenți, pentru a fi amplasat în scenă.
- **Tasta P:** Dă start simulării.
- **Tasta T:** Deschide un meniu de editare al parametrilor.

Odată ce un agent a fost selectat, acesta poate fi amplasat prin *click* dreapta. Este posibilă anularea selecției prin *click* stânga.

4.4 Acțiuni în timpul simulării

Pe lângă controlarea camerei (pentru a vizualiza liber scena) prin sistemul prezentat în subsecțiunea anterioară, utilizatorul poate folosi următoarele taste:

- ***Escape*:** Pune pauză simulării.
- **R:** Activează/dezactivează razele ce indică ținta fiecărui agent.

Capitolul 5

Tehnologii utilizate

5.1 Unity3D

Mediile de antrenare și de rulare a simulărilor au fost create cu ajutorul motorului de joc dezvoltat de Unity Technologies, anunțat și lansat pentru prima dată în iunie 2005. Poate fi folosit pentru a crea jocuri bidimensionale, tridimensionale, aplicații de realitate virtuală, realitate augmentată, diferite simulări și a fost adoptat chiar și de industrii din afara jocurilor video, cum ar fi producția cinematografică, arhitectura, ingineria și construcțiile [14].

5.2 C# (CSharp)

C# este un limbaj de programare general și modern dezvoltat de Microsoft sub inițiativa .Net. Funcțiile precum colectarea automată a gunoiului, interfețele și faptul că este un limbaj de programare orientat pe obiecte fac din C# un limbaj popular pentru dezvoltarea jocurilor folosit și în interiorul motorului de joc Unity3D. Cu ajutorul acestui limbaj au fost sriptate comportamentul agenților, a obiectelor din scene și a diferiților manageri de joc [7].

5.3 Microsoft Visual Studio

Ca și mediu de dezvoltare a fost utilizat Microsoft Visual Studio datorită compatibilității ridicate cu Unity3D și a diferitelor funcții de utilitate precum corectarea greșelilor, completarea inteligentă și depanarea eficientă a codului [8].

5.4 ML-Agents

Unity Machine Learning Agents (ML-Agents) este un set de instrumente *open – source* ce permite scenelor din interiorul motorului de joc Unity3D să servească drept medii de antrenare pentru agenți inteligenți. ML-Agents oferă implementări (bazate pe PyTorch) ale algoritmilor de ultimă generație și o interfață de programare a aplicației (API) Python simplă de utilizat pentru a instrui agenții folosind învățare ranforsată, învățare prin imitație, neuroevoluția sau chiar metode personalizate [10].

5.5 Anaconda

Anaconda este o distribuție a limbajelor de programare Python și R pentru calcul științific (știința datelor, aplicații de învățare automată, prelucrare de date la scară largă, analiză predictivă, etc.), care are ca scop simplificarea gestionării și implementării pachetelor [5]. Deși există și alte modalități de a instala Python, Anaconda a fost considerată cea mai optimă alegere pentru a gestiona, a instala și a lucra în medii Python.

5.6 TensorFlow

TensorFlow este o bibliotecă Python pentru calcul numeric rapid creată și lansată de Google, necesară antrenării agenților și care a fost implementată în mediul (Python) UnityML din Anaconda. Statisticile salvate de setul de instrumente ML-Agents în timpul sesiunilor de învățare pot fi vizualizate prin intermediul unei utilități TensorFlow numite *TensorBoard* [16].

5.7 GitHub Desktop

Ca și unealtă de control a versiunii a fost utilizată GitHub desktop, instrument ce permite interacționarea cu GitHub direct de pe desktop într-un mod foarte intuitiv datorită interfeței grafice pentru utilizator (GUI) minimalistă și explicită [6].

5.8 Overleaf

Lucrarea de față a fost redactată în Overleaf, editor LaTeX utilizat pentru scrierea, editarea și publicarea documentelor științifice. Decizia a fost luată pe baza avantajelor aduse de această platformă: nu necesită instalări, o sumedenie de tutoriale, numeroase ghiduri și template-uri, controlul versiunii și bineînțeles salvarea automată în cloud a documentelor [12].

Capitolul 6

Concluzii și direcții viitoare de dezvoltare

În această lucrare au fost prezentate detalii generale despre învățarea automată și noțiuni teoretice despre învățarea ranforsată, în direcția metodelor *Policy gradient*, precum PPO și SAC folosite pentru antrenarea agenților utilizați. Apoi au fost aduse în discuție setul de instrumente ML-Agents și tehnicile pe care le propune pentru antrenarea agenților. (modul în care *input-ul* este furnizat rețelei neuronale, felul în care *output-ul* este convertit în acțiuni, recompensarea agenților și metodele de evaluare ale antrenamentelor).

Este important de notat puterea motorului de joc Unity3D care este optim cu privire la simularea sesiunilor de antrenare pentru agenți de învățare ranforsată datorită motorului fizic (*physics engine*) și a motorului de redare (*rendering engine*) implementate. Această abordare este excelentă datorită flexibilității pe care o expune, în sensul că modificarea diferiților parametri sau componente ale antrenării se realizează cu ușurință. De asemenea, faptul că mediul este controlat reduce resursele necesare antrenării. Putem lua ca exemplu antrenarea unui robot care trebuie să învețe o politică de mișcare cât mai eficientă a articulațiilor pentru deplasarea în spațiu. În lumea reală dezvoltarea unui astfel de robot și modificarea sa între sesiunile de antrenare prezintă un efort substanțial, pe când în Unity3D, pot fi antrenați și modificați simultan n astfel de roboți.

Gama de agenți ce pot fi antrenați cu setul de instrumente ML-Agents este largă, de la reprezentări ale unor agenți reali, până la agenți utilizați în industria jocurilor precum un caracter non-jucător inamic inteligent sau chiar un agent care joacă din perspectiva utilizatorului pentru a testa jocul (sau pentru a obține performanțe super-umane). În cazul agenților antrenați în lucrarea de față, aceștia pot fi considerați viețuitoare dintr-un ecosistem, caractere non-jucător (NPC) ce pot fi folosite pentru a aduce la viață o scenă cu care un jucător uman interacționează.

Directii viitoare de dezvoltare

O extensie intuitivă este adăugarea a noi viețuitoare în ecosistem și eventual utilizarea observațiilor vizuale. Cel mai mare minus al aplicației este însă lipsa animațiilor, ce se poate rezolva în mod clasic prin crearea și animarea unor modele 3D sau ca soluție mai tehnică dar mai potrivită pentru agenți, pot fi create animații procedurale. O abordare și mai interesantă care susține utilizarea învățării reinforcement, este antrenarea unor modele ce primesc ca input un vector direcție furnizat de agenții de căutare deja dezvoltați, și observații legate de articulațiile unui manechin (precum viteză, poziție, rotație, etc...) cu ajutorul cărora învață o politică de deplasare în spațiu.

Bibliografie

- [1] Shipra Agrawal. Lecture 1: Introduction to reinforcement learning, 2019. <https://ieor8100.github.io/rl/docs/Lecture%201%20-MDP.pdf>.
- [2] Acorn Bringer. Simplistic low poly nature - prefab-uri utilize în scena principală. <https://assetstore.unity.com/packages/3d/environments/simplistic-low-poly-nature-93894>.
- [3] Google Cloud. A history of machine learning. <https://cloud.withgoogle.com/build/data-analytics/explore-history-machine-learning/>.
- [4] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560*, 2018.
- [5] Anaconda Inc. Anaconda. <https://www.anaconda.com/>.
- [6] GitHub Inc. Github desktop. <https://desktop.github.com/>.
- [7] Microsoft. C#. <https://docs.microsoft.com/en-us/dotnet/csharp/>.
- [8] Microsoft. Visual studio. <https://visualstudio.microsoft.com/>.
- [9] ML-Agents. Unity ml-agents (v0.14.0) imitation learning. <https://github.com/Unity-Technologies/ml-agents/blob/release-0.14.0/docs/Training-Imitation-Learning.md>.
- [10] ML-Agents. Unity ml-agents (v0.14.0) toolkit documentation. <https://github.com/Unity-Technologies/ml-agents/tree/release-0.14.0/docs>.

- [11] OpenAI. Documentation on soft actor critic. <https://spinningup.openai.com/en/latest/algorithms/sac.html>.
- [12] Overleaf. <https://www.overleaf.com/>.
- [13] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press, 2011.
- [14] Unity Technologies. Unity3d game engine. <https://unity.com/>.
- [15] Ervin Teng. Training your agents 7 times faster with ml-agents, 2019. <https://blogs.unity3d.com/2019/11/11/training-your-agents-7-times-faster-with-ml-agents/>.
- [16] TensorFlow. <https://www.tensorflow.org/>.
- [17] Font zone 108. Potta one - font utilizat în aplicație. <https://fonts.google.com/specimen/Potta+One>.
- [18] Judith Hurwitz și Daniel Kirsch. *Machine Learning For Dummies®*, IBM Limited Edition. John Wiley & Sons, Inc., 2018.
- [19] Hugo Mayo și Hashan Punchihewa și Julie Emile și Jack Morrison. History of machine learning, 2018. <https://www.doc.ic.ac.uk/~jce317/history-machine-learning.html#top>.
- [20] Shai Shalev-Shwartz și Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.
- [21] Alex Smola și S.V.N. Vishwanathan. *Introduction to Machine Learning*. Cambridge University Press, 2008.