

Antrenarea agenților autonomi în jocuri video folosind Reinforcement Learning

LUCRARE DE MASTER

Absolvent: **Pop Gabriel-Bogdan**

Coordonator **PROF. DR. ING. Florin Ioan Oniga**
științific:

2025

DECAN,
Prof.dr.ing. Vlad MUREȘAN

DIRECTOR DEPARTAMENT,
Prof. dr. ing. Rodica POTOLEA

Student masterand: **Pop Gabriel Bogdan**

Antrenarea agenților autonomi în jocuri video folosind Reinforcement Learning

1. **Enunțul temei:** *Studiul metodei de reinforcement learning in cadrul jocurilor video si al realizarii de agenti autonomi.*
2. **Conținutul lucrării:** *Introducere, Obiectivele proiectului, Studiul bibliografic/Stadiul actual al domeniului, Proiectare de detaliu si implementare, Testare si Validare, Concluzii, Bibliografie*
3. **Locul documentării:** Universitatea Tehnică din Cluj-Napoca, Departamentul Calculatoare
4. **Consultanți:** Prof. Dr. Ing. Florian Ioan Oniga
5. **Data emiterii temei:** noiembrie 2024
6. **Data predării:** 11 iulie 2025

Student masterand: Pop Gabriel Bogdan 

Coordonator științific: Prof. Dr. Ing. Florin Ioan

**Declarație pe proprie răspundere privind
autenticitatea lucrării de disertație**

Subsemnatul(a), Pop Gabriel - Bogdan legitimat(ă) cu CI seria MM nr. 960456 CNP 5000922245022 , autorul lucrării Antrenarea agenților autonomi în jocuri video folosind Reinforcement Learning elaborată în vederea susținerii examenului de finalizare a studiilor de disertație la Facultatea de Automatică și Calculatoare, specializarea TIE din cadrul Universității Tehnice din Cluj-Napoca, sesiunea Iulie a anului universitar 2024-2025, declar pe proprie răspundere că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate în textul lucrării și în bibliografie.

Declar că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de disertație.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, *anularea examenului de disertație*.

Sunt de acord ca, pe tot parcursul vieții, în cazul în care este necesar și se va dori verificarea autenticității lucrării mele să fiu identificat și verificat în baza datelor declarate de mine.

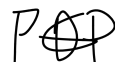
Data

11.07.2025

Nume, Prenume

Pop Gabriel - Bogdan

Semnătura



Cuprins

Capitolul 1. Introducere	1
1.1 Context.....	1
1.2 Motivație	2
Capitolul 2. Obiectivele proiectului.....	3
2.1 Specificarea problemei.....	3
2.2 Obiectivul principal	3
2.3 Obiectivele secundare	4
Capitolul 3. Studiu bibliografic/Stadiul actual al domeniului.....	5
3.1 Reinforcement Learning – concepte și aplicații în gaming	5
3.2 Implementări clasice: Snake, Breakout, CartPole.....	6
3.3 Aplicații RL în medii vizuale 2D – Google Dino	6
3.4 RL în medii 3D și FPS – ViZDoom, DoomRNN, ViZDoom+PPO	7
3.5 Metode existente	8
3.6 Noțiuni de bază din RL: agent, mediu, policy, reward	13
3.7 Algoritmi: Q-learning, Deep Q-Network (DQN)	13
3.8 Arhitecturi de rețele neuronale folosite.....	14
3.9 Tehnici auxiliare: experience replay, epsilon decay	16
3.10 Frameworkuri și biblioteci utilizate	17
3.11 Cerințele sistemului	31
Capitolul 4. Proiectare de detaliu și implementare	32
4.1 Snake Game	32
4.2 Google Dino.....	35
4.3 ViZDoom	39
Capitolul 5. Testare și validare.....	46
5.1 Metodologia de testare	46
5.2 Măsurători și rezultate pe fiecare joc	47
5.3 Probleme și interpretarea rezultatelor	49
5.4 Comparatie între rezultatele obținute și cele din literatura de specialitate	51
Capitolul 6. Concluzii	52
6.1 Concluzia proiectului	52

6.2 Analiza critică a rezultatelor obținute	52
6.3 Dezvoltări ulterioare	53
Capitolul 7. Bibliografie	54

Capitolul 1. Introducere

1.1 Context

Inteligența artificială a devenit în ultimii ani o componentă esențială în dezvoltarea tehnologiilor interactive, iar industria jocurilor video este unul dintre domeniile unde aceasta și-a demonstrat impactul concret. Dintre tehnicile moderne de învățare automată, Reinforcement Learning (RL) se remarcă prin capacitatea sa de a antrena agenți care pot învăța comportamente complexe prin interacțiune directă cu mediul. Acest tip de abordare se potrivește perfect jocurilor video, unde agentul trebuie să ia decizii secvențiale și să reacționeze în funcție de feedback, fără a avea acces la reguli explicite. În cadrul acestei lucrări, am explorat RL ca mijloc de a construi agenți care învață autonom să joace jocuri 2D și 3D, pornind de la date brute precum imagini de ecran.

Jocurile video oferă un cadru ideal pentru testarea și validarea algoritmilor RL, întrucât acestea simulează medii controlate dar variate, în care un agent poate lua decizii, învăța din greșeli și evalua performanța pe baza unor metrici clare. De la jocuri simple precum Snake sau CartPole, până la simulări complexe 3D precum ViZDoom, domeniul RL permite explorarea unor concepte fundamentale precum învățarea secvențială, explorarea versus exploatarea și optimizarea politicilor de acțiune.

În același timp, progresul hardware (în special în utilizarea plăcilor grafice pentru antrenarea rețelelor neuronale) și dezvoltarea unor biblioteci precum TensorFlow, PyTorch sau Stable-Baselines3 au permis cercetătorilor și dezvoltatorilor să experimenteze și să implementeze soluții de RL într-un mod mai accesibil și mai eficient.

Această lucrare de disertație își propune să exploreze aplicabilitatea reinforcement learning în antrenarea agenților care pot juca automat jocuri video. Scopul este de a analiza performanța acestor agenți în medii de dificultate diferită: de la un joc clasic 2D (Snake), la un joc vizual cu obstacole dinamice (Google Dino), și până la un mediu 3D complex și parțial observabil (ViZDoom).

În cadrul proiectului, accentul nu a fost pus doar pe demonstrarea aplicabilității RL în jocuri video, ci și pe documentarea riguroasă a provocărilor tehnice întâlnite: alegerea arhitecturii rețelei neuronale potrivite fiecărui scenariu, definirea corectă a funcției de recompensă pentru a evita comportamentele indezirabile, și evaluarea performanței agentului prin experimente controlate. Aceste aspecte au fost analizate în detaliu pentru fiecare dintre jocurile selectate.

Pe lângă valoarea tehnică a cercetării, am investigat și posibile aplicații comerciale ale agenților antrenați prin RL. Una dintre direcțiile analizate constă în utilizarea acestor agenți pentru obținerea automată de resurse virtuale – o practică deja întâlnită în unele jocuri multiplayer online. Prin simularea unor ferme de „boti” autonomi, capabili să obțină monede sau obiecte rare, am evaluat potențialul economic al acestui tip de sistem, în special în contextul piețelor digitale precum Steam Market sau platformele bazate pe NFT.

1.2 Motivație

Motivația personală care a stat la baza acestui proiect este strâns legată de pasiunea de lungă durată pentru domeniul jocurilor video. În calitate de gamer dedicat, ideea de a crea agenți capabili să învețe și să joace automat jocuri care, în mod normal, necesită reflexe, strategie și experiență, a reprezentat dintotdeauna o provocare. De-a lungul anilor, ideea de a construi un „bot” care să poată învăța singur să joace ca un om, sau chiar mai bine, a fost o idee la care m-am tot gândit.

Alegerea tehnicii de reinforcement learning vine natural în acest context, fiind una dintre puținele metode de inteligență artificială care nu se bazează pe date etichetate în prealabil, ci pe explorarea activă a mediului și adaptarea continuă în funcție de feedback-ul primit. Această paradigmă este extrem de apropiată de modul în care învață și un jucător uman prin încercări repetate, prin eșecuri și prin succes. Prin urmare, aplicarea RL în jocuri video nu este doar o problemă interesantă din punct de vedere tehnic, ci și o oglindă a procesului natural de învățare.

Pentru a construi un traseu coerent și scalabil în procesul de cercetare, am ales să aplic metodele RL asupra unor jocuri de dificultate progresivă: de la Snake, un joc cu reguli simple și mediu complet observabil, la Google Dino, care introduce reacții rapide la stimuli vizuali și o dinamică mai accelerată, până la ViZDoom, un mediu 3D complex și parțial observabil, cu factori precum navigația, precizia în luptă și luarea deciziilor în condiții de incertitudine. Această progresie a permis o analiză mai detaliată a limitelor și potențialului învățării prin întărire, în funcție de complexitatea mediului.

Pe lângă dimensiunea personală, proiectul oferă și o oportunitate reală de analiză a potențialului economic al inteligenței artificiale în gaming. Într-un ecosistem digital în care resursele virtuale pot fi tranzacționate, antrenarea unor agenți capabili să joace eficient și să „farmeze” iteme de valoare deschide posibilitatea unor noi modele de afacere. Astfel, proiectul devine nu doar o explorare tehnologică, ci și un exercițiu aplicat în identificarea și valorificarea oportunităților economice din industria jocurilor video.

Capitolul 2. Obiectivele proiectului

2.1 Specificarea problemei

În contextul actual al inteligenței artificiale aplicate, una dintre cele mai importante provocări este dezvoltarea de agenți autonomi care pot învăța comportamente eficiente într-un mediu necunoscut, fără asistență externă și fără un set de date predefinit. Această problemă devine și mai complexă atunci când mediile sunt dinamice, parțial observabile sau prezintă un nivel ridicat de incertitudine, așa cum este cazul în majoritatea jocurilor video reale.

Jocurile video reprezintă o platformă ideală pentru testarea acestor agenți, deoarece îmbină interacțiunea în timp real cu luarea deciziilor, planificarea pe termen scurt și lung, precum și adaptarea la situații noi. Totuși, dezvoltarea unui agent care să poată performa bine într-un astfel de mediu ridică mai multe provocări:

- Cum poate un agent să „înțeleagă” starea curentă a jocului din date brute (de exemplu, poziția șarpelui, obstacolele sau imaginea pe ecran)?
- Cum poate decide acțiunea optimă, având în vedere un spațiu de acțiuni redus dar cu consecințe semnificative?
- Cum învață să evite deciziile greșite (care duc la „moarte” în joc) și să favorizeze comportamentele care maximizează scorul sau supraviețuirea?

În plus, apar dificultăți tehnice legate de stabilitatea algoritmilor de învățare, alegerea funcției de recompensă și definirea corectă a stării agentului. Aceste provocări devin din ce în ce mai accentuate odată cu creșterea complexității jocului, de la jocuri deterministe și complet observabile precum Snake, până la simulări 3D precum ViZDoom, unde agentul primește doar imagini parțiale din mediu și trebuie să învețe atât navigația cât și tactici de luptă.

2.2 Obiectivul principal

Obiectivul principal al acestei lucrări de disertație este dezvoltarea unui sistem bazat pe Reinforcement Learning capabil să antreneze agenți care pot juca automat jocuri video de diferite tipuri, fără a avea acces la reguli explicite, scripturi sau exemple predefinite de joc.

Mai precis, se urmărește implementarea unui agent RL generic, capabil să interacționeze cu un mediu de joc, să-și extragă singur informațiile relevante din starea curentă și să își îmbunătățească performanța prin experiență acumulată. Agentul trebuie să învețe să ia decizii care maximizează o funcție de recompensă definită în funcție de obiectivele jocului (de exemplu, supraviețuire, scor, evitarea coliziunilor etc.).

Pentru a valida acest obiectiv, proiectul va fi aplicat pe trei categorii de jocuri video, cu un grad de dificultate progresiv:

- Snake Game : mediu complet observabil, logică simplă, acțiuni discrete
- Google Dino : mediu vizual 2D, cu obstacole dinamice, input vizual și reacții rapide
- ViZDoom : joc FPS 3D, mediu complex și parțial observabil, acțiuni combinate și luare de decizii strategice

Scopul este de a demonstra că metodele de reinforcement learning pot fi adaptate și aplicate atât în medii simple, cât și în cele complexe, și că un agent poate învăța să joace doar prin interacțiune directă cu mediul, fără acces la reguli explicite. În plus, proiectul urmărește să evidențieze bune practici de proiectare a rețelelor neuronale și a funcțiilor de recompensă, precum și limitările întâlnite în fiecare caz.

2.3 Obiectivele secundare

Pentru a sprijini atingerea obiectivului principal, au fost definite o serie de obiective secundare, care vizează aspecte esențiale ale procesului de proiectare, implementare și validare a agenților inteligenți antrenați prin învățare prin întărire:

- Implementarea unui agent RL pentru jocul Snake, folosind tehnici de tip Deep Q-Learning (DQN), unde starea este reprezentată simbolic (poziții, direcții, pericole) și acțiunile sunt discrete. Acesta va servi drept punct de plecare conceptual și de testare a funcționalității de bază.
- Adaptarea agentului pentru jocul Google Dino, cu focus pe prelucrarea datelor vizuale (captura de ecran), detectarea obstacolelor și deciziile rapide (salt, ghemuire, nimic). În acest caz, agentul trebuie să învețe să interpreteze date vizuale și să reacționeze corespunzător într-un mediu cu obstacole dinamice.
- Dezvoltarea unor agenți capabili să joace scenarii complexe în ViZDoom, precum `basic.wad`, `defend_the_center.wad` și `deadly_corridor.wad`. Aceste scenarii testează capacitatea agentului de a naviga, a reacționa în condiții de pericol și a învăța politici eficiente de luptă într-un mediu 3D cu vizibilitate parțială.
- Configurarea și testarea arhitecturilor de rețele neuronale adaptate fiecărui joc, folosind rețele complet conectate (fully connected) în cazul Snake, rețele convoluționale (CNN) pentru date vizuale în Dino și ViZDoom, și antrenarea acestora folosind biblioteci specializate precum PyTorch și Stable-Baselines3.
- Definirea funcțiilor de recompensă pentru fiecare joc în parte, astfel încât agentul să fie încurajat să adopte comportamente dorite (supraviețuire, scor ridicat, evitarea pericolelor), evitând în același timp recompense greșite sau comportamente nedorite.
- Evaluarea performanței agenților în funcție de scoruri medii, epoci de convergență, stabilitatea învățării și robustețea comportamentului. Acest obiectiv implică generarea de grafice și interpretarea datelor pentru a trage concluzii relevante.
- Explorarea posibilităților de aplicare economică, prin analizarea unui model de afacere bazat pe utilizarea acestor agenți pentru obținerea de iteme sau resurse în jocuri, care pot fi ulterior comercializate în piețele digitale de profil.

Prin atingerea acestor obiective secundare, proiectul își propune să ofere o privire de ansamblu completă asupra procesului de construire a unor agenți RL funcționali, demonstrând aplicabilitatea practică a acestei tehnologii atât în domeniul tehnic, cât și în cel economic.

Capitolul 3. Studiu bibliografic/Stadiul actual al domeniului

3.1 Reinforcement Learning – concepte și aplicații în gaming

Reinforcement Learning – RL este o ramură a învățării automate în care un agent autonom învață să ia decizii prin interacțiune directă cu un mediu, primind feedback sub forma unui sistem de recompense. Scopul agentului este să învețe o politică optimă o strategie de acțiune care maximizează recompensa cumulativă pe termen lung.

Modelul formal folosit în RL se bazează pe procesul de decizie markovian (Markov Decision Process – MDP), definit prin următoarele elemente:

- Agentul, entitatea care ia decizii;
- Mediul, cu care agentul interacționează;
- Starea (state), reprezentarea situației curente din joc;
- Acțiunea (action), alegerea făcută de agent;
- Recompensa (reward), semnal numeric primit după fiecare acțiune;
- Politica (policy), funcția de decizie a agentului.

RL se deosebește de alte forme de învățare automată (cum ar fi învățarea supervizată sau nesupervizată) prin faptul că nu are nevoie de un set de date etichetat, ci învață „din mers”, experimentând și ajustându-și comportamentul în funcție de consecințele acțiunilor sale.

În ultimii ani, Reinforcement Learning a devenit una dintre cele mai populare metode în domeniul inteligenței artificiale aplicate în jocuri. Datorită naturii secvențiale și interactive a jocurilor video, RL oferă un cadru ideal pentru antrenarea agenților care pot învăța prin încercare și eroare. Exemple notabile includ:

- Deep Q-Network (DQN), algoritmul introdus de DeepMind, care a demonstrat că un agent poate învăța să joace jocuri Atari direct din imagini brute, depășind performanțele jucătorilor umani în mai multe titluri clasice precum Breakout sau Space Invaders.
- AlphaGo și AlphaZero, sisteme dezvoltate de DeepMind care au aplicat RL pentru a învăța să joace Go, șah și shogi la nivel super-uman, pornind de la zero, fără cunoștințe preprogramate.
- OpenAI Five, un agent antrenat să joace Dota 2 împotriva echipelor umane profesionale, bazat pe politici multi-agent și învățare profundă pe scară largă.

În contextul jocurilor 2D și 3D, aplicarea RL necesită rezolvarea unor provocări suplimentare:

- spații de stare și de acțiuni mari;
- observabilitate parțială (agentul nu vede tot mediul);
- recompense rare sau întârziate;
- antrenament lent în medii vizual complexe.

Totuși, tehnologiile moderne precum rețelele convoluționale (CNN), librăriile de simulare (OpenAI Gym, ViZDoom), și algoritmi avansați (PPO, A3C, DDPG) au dus la o accelerare semnificativă a cercetării în acest domeniu.

Prin urmare, Reinforcement Learning nu este doar o tehnică promițătoare din punct de vedere academic, ci și una extrem de aplicabilă în industrie, în special în domenii precum automatizarea, robotică, simulări, finanțe sau după cum este demonstrat în această lucrare gaming.

3.2 Implementări clasice: Snake, Breakout, CartPole

Pentru cercetătorii din domeniul RL, jocurile clasice precum Snake, Breakout și CartPole au reprezentat de-a lungul timpului puncte de plecare esențiale în validarea și calibrarea algoritmilor de control inteligent.

Jocul Snake este considerat un exemplu ideal pentru testarea conceptelor de bază din RL, datorită regulilor simple, spațiului de acțiuni redus și feedback-ului constant. Agentul trebuie să navigheze într-o arenă pentru a colecta mâncare și a evita coliziunile cu pereții sau propriul corp. Starea poate fi reprezentată simbolic (vecinătatea imediată, direcția curentă, poziția hranei), iar acțiunile sunt de obicei discrete: înainte, stânga, dreapta. Astfel, algoritmi precum DQN sau Q-learning pot fi testați într-un mediu determinist și complet observabil, oferind o bază solidă pentru înțelegerea comportamentului unui agent RL.

Breakout este un joc clasic dezvoltat de Atari, utilizat extensiv în lucrările DeepMind pentru a valida performanța rețelelor neuronale convoluționale în RL. Agentul controlează o paletă care trebuie să respingă o minge și să distrugă blocurile din partea superioară a ecranului. Jocul introduce dinamica fizică (reflectarea mingii), recompense întârziate (blocurile se distrug după un timp) și o stare vizuală completă reprezentată prin cadre brute. Este un exemplu ideal pentru antrenarea agenților folosind intrări vizuale, necesitând rețele CNN pentru extragerea automată a caracteristicilor din imagine.

CartPole este unul dintre cele mai simple și cunoscute exemple de control din RL, disponibil în cadrul OpenAI Gym. Sarcina constă în menținerea unui stâlp echilibrat pe un cărucior mobil, agentul având doar două acțiuni disponibile: deplasarea la stânga sau la dreapta. Deși simplu din punct de vedere vizual, CartPole ilustrează perfect noțiunea de control secvențial în timp real, și este utilizat pentru testarea rapidă a algoritmilor precum DQN, SARSA sau policy gradient. Recompensa este primită la fiecare pas în care stâlpul rămâne vertical, iar episodul se încheie în momentul în care acesta cade.

Aceste exemple clasice stau la baza multor cercetări și dezvoltări ulterioare în domeniul RL, fiind utilizate pentru:

- verificarea funcționării corecte a rețelelor și a algoritmilor;
- testarea diverselor funcții de recompensă;
- compararea performanțelor între metode (model-based vs model-free, value-based vs policy-based);
- analizarea timpilor de convergență și a stabilității învățării.

Proiectul prezentat în această lucrare pornește de la aceste exemple și avansează către scenarii mai complexe, menținând însă legătura conceptuală cu principiile de bază testate și validate în aceste jocuri.

3.3 Aplicații RL în medii vizuale 2D – Google Dino

Google Dino, cunoscut și sub numele de Chrome Dino, este un joc 2D integrat în browserul Google Chrome, activat automat atunci când nu există conexiune la internet. Jucătorul controlează un dinozaur care trebuie să sară peste cactuși și să evite obstacole (inclusiv păsări zburătoare) într-un mediu care devine treptat mai rapid și mai dificil. Deși simplu ca design, jocul devine un caz interesant de studiu pentru aplicațiile RL în medii vizuale 2D cu dinamică crescută.

Spre deosebire de jocurile simbolice precum Snake sau CartPole, Google Dino presupune prelucrarea informației vizuale – agentul nu are acces direct la poziția

obstacolelor sau la viteză, ci trebuie să „vadă” și să înțeleagă situația dintr-o imagine (frame) capturată de pe ecran. Acest lucru introduce provocări suplimentare:

- extracția caracteristicilor din imagine (necesită CNN);
- timp de reacție redus, întrucât obstacolele apar brusc și trebuie evitate instantaneu;
- lipsa unei funcții de recompensă evidente, fiind necesar un reward shaping corect (ex. +1 pentru fiecare cadru supraviețuit, -100 la coliziune).

În literatura de specialitate, probleme similare sunt abordate prin folosirea de rețele convoluționale pentru interpretarea imaginilor și algoritmi de tip DQN sau PPO pentru alegerea acțiunilor. Cadrele capturate sunt convertite în grayscale, redimensionate și normalizate înainte de a fi introduse în rețea. Această tehnică este eficientă și în cazul jocului Dino, unde o arhitectură CNN bine calibrată poate învăța comportamente precum:

- salt automat la apariția unui obstacol;
- ghemuire la întâlnirea păsărilor;
- inactivitate atunci când nu este necesară acțiunea.

Dificultatea crește progresiv pe măsură ce viteza jocului se intensifică, ceea ce face ca agentul să învețe nu doar reguli statice, ci și adaptabilitate în fața unor condiții dinamice. De asemenea, lipsa unei interfețe programabile directe (API) a jocului obligă dezvoltatorul să utilizeze metode de interacțiune indirectă, cum ar fi capturi de ecran și comenzi automate simulate (ex: pydirectinput), ceea ce adaugă un strat de complexitate tehnică.

În concluzie, Google Dino oferă un echilibru interesant între simplitate vizuală și provocare algoritmică, fiind un exemplu relevant de aplicare a RL în medii 2D vizuale, fără suport nativ pentru integrarea AI. Astfel, el servește ca punte între jocurile simbolice și cele 3D, cum este cazul ViZDoom.

3.4 RL în medii 3D și FPS – ViZDoom, DoomRNN, ViZDoom+PPO

Pe măsură ce Reinforcement Learning a progresat, cercetarea s-a extins dincolo de jocurile simple 2D și a intrat în zona simulărilor 3D și a jocurilor de tip First Person Shooter (FPS), care implică nu doar navigație și luare de decizii, ci și percepție vizuală, viteză de reacție și strategie. Unul dintre cele mai utilizate medii de testare în această direcție este ViZDoom o interfață AI dezvoltată peste motorul original Doom, care permite controlul unui agent prin comenzi și accesul la date vizuale în timp real.

ViZDoom este folosit în numeroase lucrări științifice datorită:

- posibilității de configurare a propriilor scenarii;
- controlului detaliat asupra randării (ex: doar imagini RGB sau adăugarea de hartă de adâncime);
- rulării rapide chiar și pe configurații hardware modeste;
- integrării directe cu Python și biblioteci precum Gym, Stable-Baselines, PyTorch.

Un exemplu avansat din literatură este DoomRNN, un sistem creat de OpenAI care folosește un model de lume generativ (world model) pentru a învăța dinamica jocului într-un spațiu latent. În loc să antreneze agentul direct în jocul complet, DoomRNN construiește o rețea neuronală recurentă (RNN) care simulează mediul, iar agentul este antrenat în această simulare. Această abordare reduce semnificativ timpul de antrenament, dar necesită modelare avansată și o înțelegere profundă a dinamicii mediului.

O altă direcție cercetată în mod frecvent este utilizarea algoritmilor de tip policy gradient, cum ar fi Proximal Policy Optimization (PPO), în combinație cu ViZDoom. Spre deosebire de DQN, care învață o funcție de valoare și derivă politica din aceasta, PPO învață direct politica (distribuția acțiunilor) și este mult mai stabil în medii complexe. PPO s-a dovedit eficient în scenarii precum:

- defend_the_center, unde agentul trebuie să supraviețuiască împotriva mai multor inamici;
- health_gathering, unde agentul explorează pentru a găsi medkit-uri;
- deadly_corridor, un test clasic de navigație + luptă.

ViZDoom combinat cu PPO oferă rezultate competitive, dar necesită:

- rețele convoluționale profunde pentru extragerea caracteristicilor vizuale;
- tuning atent al hiperparametrilor (pasul de învățare, gamma, clip range etc.);
- shaping al funcției de recompensă pentru a evita stagnarea în comportamente pasive.

Aplicarea RL în medii 3D precum ViZDoom aduce cu sine dificultăți semnificative:

- observabilitate parțială: agentul nu are vedere de ansamblu;
- acțiuni complexe: combinarea direcției, deplasării și tragerii;
- explorare dificilă: uneori recompensele sunt rare sau ascunse;
- cost computațional: antrenamentul durează mult mai mult decât în jocurile 2D.

Cu toate acestea, ViZDoom rămâne un etalon în testarea agenților RL capabili să acționeze în medii apropiate de realitate, iar proiectul prezentat în această lucrare se aliniază acestor cercetări prin implementarea și testarea mai multor scenarii din acest motor.

3.5 Metode existente

3.5.1 ViZDoom – „A Doom-based AI Research Platform for Visual Reinforcement Learning (IEEE CIG 2016, W. Jaśkowski et al.)”

Această lucrare este prima prezentare oficială a platformei ViZDoom în comunitatea științifică internațională, „propunând-o ca un mediu standardizat pentru testarea și dezvoltarea agenților inteligenți în medii vizuale 3D, cu perspectivă first-person”. Autorii identifică nevoia acută de a trece de la jocuri 2D (ex: Atari) la simulări mai apropiate de realitate, în care percepția vizuală este singura sursă de informație.

Motivația creării ViZDoom: „Jocurile video au fost utilizate pe scară largă ca medii de testare în RL, însă platformele clasice (ex. Atari 2600) au limitări evidente”:

- medii 2D, non-realiste;
- control complet și nerealist al stării;
- lipsa unei perspective naturale (3rd person);
- lipsa dinamicii spațiale și a navigației reale.

Pentru a depăși aceste limitări, autorii aleg să construiască ViZDoom peste motorul open-source ZDoom. Această alegere este justificată prin:

- rulare rapidă (până la 7000 FPS);
- cerințe de sistem reduse;
- suport pentru scripting și scenarii personalizate;
- acces la screen buffer și depth buffer;
- suport pentru moduri multiplayer;
- acces direct din C++, Python și Java.

Structura tehnică și capabilitățile platformei: ViZDoom permite rularea în patru moduri:

- synchronous player: agentul decide, jocul așteaptă;
- asynchronous player: jocul rulează, agentul reacționează în timp real;
- spectator modes: agentul observă un jucător uman;
- multiplayer: agenți vs agenți sau agenți vs oameni.

Cele mai utile pentru cercetare sunt modurile sincrone, întrucât permit rularea deterministă, pas-cu-pas, controlând complet fluxul de învățare.

Platforma oferă și:

- off-screen rendering, utilă pentru rulare pe servere fără interfață grafică;
- frame skipping, esențială pentru accelerarea învățării fără a pierde semnificația secvenței;
- custom reward signals;
- acces la variabile interne (ex: health, ammo);
- scenarii editabile vizual, cu unelte precum Doom Builder 2 și SLADE3.



[Figura 3.1 Doom](#)

Experimente efectuate:

Scenariul 1 – Move & Shoot

Agentul este plasat într-o cameră unde un monstru staționar apare pe perețele opus.

Acțiuni posibile: stânga, dreapta, foc. Scor:

- +101 pentru omor;
- -5 pentru ratat;
- -1 per acțiune.

Rețea folosită:

- 2 straturi convoluționale (7x7, 4x4);
- max pooling + ReLU;
- 1 strat complet conectat (800 neuroni);
- 8 neuroni la ieșire (acțiuni posibile).

Rezultate:

- frame skip ideal între 4–10;
- învățare mai rapidă și mai stabilă cu skipcount 4+;
- comportamente coerente și similare celor umane (poziționare + țintire).

Scenariul 2 – Health Gathering

Agentul este plasat într-un labirint toxic unde trebuie să colecteze medkits și să evite otrava. Acțiuni: deplasare față/spate, rotire stânga/dreapta. Rețea:

- 3 straturi convoluționale (7x7, 5x5, 3x3);
- 1024 neuroni fully connected;
- shaping reward: +100 (medkit), -100 (fiolă).

Scor final: max 2100, scor mediu agent \approx 1300, cu comportamente naturale (căutare, evitare, corectare traseu). Uneori apăreau comportamente oscilatorii (ezitare direcțională).

Contribuții importante:

- demonstrează fezabilitatea RL pur vizual în FPS-uri 3D;
- stabilește ViZDoom ca standard academic pentru studii RL în medii first-person;
- arată cum complexitatea acțiunilor și a observației poate fi gestionată cu CNN-uri relativ simple;
- pune bazele multor alte cercetări ulterioare în ViZDoom.

Relevanță pentru lucrarea actuală: „Această lucrare este punctul de referință principal pentru orice proiect bazat pe ViZDoom. Scenariile, rețelele și metoda de învățare sunt direct comparabile cu implementările tale. De asemenea, ea oferă justificarea teoretică pentru utilizarea frame skip, shaping reward, și DQN ca soluții viabile în scenarii precum basic.wad și deadly_corridor.wad.”

3.5.2 ViZDoom: „A Doom-based AI Research Platform for Visual Reinforcement Learning (IEEE CIG 2016)”

Această lucrare semnată de Michael Kempka și colaboratorii de la Poznań University of Technology este una dintre cele mai importante publicații care fundamentează „ViZDoom ca platformă de cercetare în învățarea prin întărire vizuală. Autorii introduc ViZDoom ca o alternativă realistă și eficientă la mediile 2D din seria Atari, permițând dezvoltarea agenților care pot învăța direct din fluxuri vizuale brute, în medii 3D și din perspectivă first-person.”

Scopul platformei ViZDoom: „Lucrarea subliniază limitările mediilor existente precum Atari 2600 reprezentări 2D, control perfect al stării și o perspectivă ne-naturală (3rd person)”. ViZDoom este creat pentru a acoperi aceste lacun e, oferind:

- control complet asupra motorului jocului (poate fi oprit între acțiuni);
- scenarii complet personalizabile;
- posibilitatea de a accesa bufferul de adâncime (depth buffer);
- randare off-screen pentru execuție pe servere sau în fundal;
- FPS constant (până la 7000 cadre/secundă).

Platforma permite agenților să interacționeze cu jocul exclusiv prin observații vizuale (imagini RGB), imitând percepția umană și eliminând accesul la date „nedrepte” precum coordonate exacte sau hărți.

Experimente efectuate:

1. Scenariul de bază – Move and Shoot

Agentul este plasat într-o cameră în care un inamic static apare aleatoriu pe perețele opus. Acțiunile posibile: stânga, dreapta și foc. Recompense:

- +101 pentru omor;
- -5 pentru glonț ratat;
- -1 pentru fiecare acțiune.

Modelul folosit este o rețea convoluțională (CNN) cu două straturi convoluționale (7x7 și 4x4), un strat complet conectat cu 800 de neuroni și o ieșire de 8 acțiuni posibile (combinații de stânga/dreapta/foc). Se folosește Deep Q-Learning cu:

- replay buffer;
- ϵ -greedy policy;
- decay progresiv;
- batch size de 40;
- imagine RGB de 60x45 drept input.

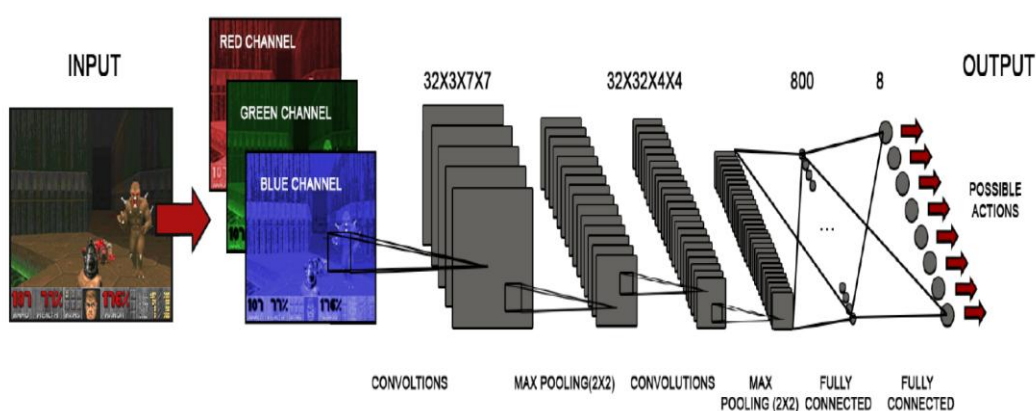


Figura 3.2 Reteaua convoluțională

Rezultate:

- Cel mai eficient interval de frame skip: 4–10 cadre (balance între performanță și timp de training);
- Rețelele cu frame skip mare au învățat mai repede, cu mai puține comportamente iraționale (idle, mers în direcția greșită);
- Agentul a învățat să doboare inamicul cu o singură lovitură după ~600.000 de pași de antrenament.

2. Scenariul Maze – Health Gathering

Agentul este plasat într-un labirint cu podea acidă, pierzând viață constant. Obiectivul este să supraviețuiască prin colectarea de medkits și evitarea fiolelor otrăvite.

Setări:

- imagine RGB de 120x45;
- 4 stări memorate;
- rețea cu 3 straturi convoluționale și 1024 unități fully connected;
- shaping reward: +100 la medkit, -100 la fiolă;
- scorul maxim teoretic: 2100 puncte (1 punct pe tick, episod de 2100 ticks);
- skipcount = 10;
- training: 1.000.000 de pași, 29 ore.

Agentul a învățat să supraviețuiască eficient, să evite pereții și să navigheze în labirint, dar avea comportamente ezitante (rotații repetate), ceea ce sugerează nevoia de fine-tuning mai avansat.

Contribuții importante pentru lucrarea actuală:

- Demonstrează că RL vizual este fezabil chiar și în medii 3D semi-realiste.

- Introduce ideea de shaping reward și influența majoră a frame-skipping asupra performanței și eficienței antrenamentului.
- Confirmă că o arhitectură CNN relativ simplă poate genera comportamente realiste și funcționale în jocuri FPS.

Relevanță pentru proiectul propriu: „Acest articol este unul dintre fundamentele teoretice și tehnice pentru implementarea RL în ViZDoom. Metodele prezentate aici sunt similare cu cele din scenariile proprii (basic, defend_center, deadly_corridor). În plus, abordarea modulară și analiza detaliată a parametrilor sunt direct aplicabile în validarea și justificarea alegerilor făcute în proiectul tău.”

3.5.3 „Training an Agent for FPS Doom Game using Visual Reinforcement Learning and VizDoom (IJACSA, 2017)”

Lucrarea propusă de Khan urmărește antrenarea unui agent RL în medii de tip first-person shooter (FPS), utilizând platforma ViZDoom și informații vizuale brute preluate direct din cadrul jocului. „Obiectivul principal al cercetării este demonstrarea faptului că un agent bazat pe Deep Q-Learning poate depăși performanțele jucătorilor umani și ale agenților încorporați în joc, folosind exclusiv imagini RGB capturate de pe ecran.”

Arhitectura agentului și configurarea rețelei neuronale: „Pentru antrenarea agentului, autorii utilizează o rețea neuronală convoluțională (CNN) alcătuită din două straturi convoluționale (cu filtre de dimensiuni 7x7 și 4x4), urmate de pooling și un strat complet conectat cu 800 de unități Leaky ReLU. Ieșirea finală conține 8 neuroni care corespund celor 8 combinații posibile de acțiuni (stânga, dreapta, foc). Este aplicată o strategie ϵ -greedy cu decădere liniară, iar agentul este antrenat folosind Q-learning, SGD și experience replay. Parametrii principali utilizați sunt: $\gamma = 0.99$, $\alpha = 0.00025$, și batch size = 64.”

Scenariul testat: „Experimentul este realizat pe un scenariu de bază („basic scenario”), în care agentul este poziționat într-o cameră dreptunghiulară, iar un monstru staționar apare pe peretele opus. Agentul poate doar să se deplaseze lateral (stânga/dreapta) și să tragă. Obiectivul este eliminarea rapidă a monstrului pentru a maximiza scorul (+101 la lovitură, -5 pentru ratare, -1 pentru fiecare acțiune inutilă).”

Rezultate experimentale: „Agentul este antrenat timp de peste 20 de epoci, acumulând un total de 23.239 de pași de învățare. În fiecare epocă, agentul este testat pe 100 de episoade pentru a evalua progresul. În primele epoci, scorurile medii sunt negative sau aproape de zero, dar pe măsură ce învățarea avansează, agentul obține scoruri constante și stabile (medie de ~79 puncte). Agentul învață să tragă precis, să economisească acțiuni și să atingă poziția optimă pentru lovitură cu un singur foc.”

Observații și contribuții relevante:

- Arhitectura propusă diferă semnificativ de lucrările anterioare, fiind mai eficientă în scenariile testate.
- Agenții au fost capabili să învețe comportamente similare celor umane, fără a avea acces la date abstracte precum poziția inamicilor sau harta.
- Lucrarea susține ideea că platforma ViZDoom este potrivită pentru testarea metodelor RL vizuale în medii semi-realiste și 3D.
- Timpul total de antrenament a fost de aproximativ 30 de minute pe o placă NVIDIA GTX 1080, demonstrând fezabilitatea experimentelor pe hardware de uz general.

Relevanță pentru proiectul propriu: „Această lucrare validează metodologia aplicată în proiectul curent, care utilizează tot ViZDoom și un setup asemănător (DQN, CNN, reward shaping). În plus, modelul propus de autori — cu acțiuni limitate și recompense

clare — este direct comparabil cu unul dintre scenariile tale (basic.wad), oferind un cadru de referință excelent pentru calibrarea performanței agentului propriu.”

3.6 Noțiuni de bază din RL: agent, mediu, policy, reward

Reinforcement Learning – RL este o paradigmă a învățării automate în care un agent autonom învață să ia decizii optime prin interacțiune directă cu un mediu. Agentul explorează acest mediu și primește un feedback numeric (recompensă) pentru fiecare acțiune întreprinsă, scopul fiind maximizarea recompensei cumulate pe termen lung.

Modelul formal utilizat pentru descrierea procesului de reinforcement learning este cunoscut ca Markov Decision Process (MDP) și este definit prin următoarele componente fundamentale:

- **Agentul:** entitatea decizională care efectuează acțiuni în mediu. În contextul jocurilor video, agentul este un bot care controlează personajul jucătorului.
- **Mediul:** reprezintă lumea în care agentul operează. Acesta include regulile jocului, starea curentă, obiectele, inamicii și orice alt element care influențează deciziile agentului.
- **Starea (state):** este o reprezentare (numerică sau vizuală) a situației curente în care se află agentul. În jocuri, aceasta poate fi compusă din poziția personajului, distanța față de obstacole, imaginea RGB capturată de pe ecran etc.
- **Acțiunea (action):** reprezintă alegerea făcută de agent într-o stare dată. Acțiunile sunt specifice jocului și pot include mișcări direcționale (stânga, dreapta, sărit), atac, ghemuire, sau chiar inacțiunea.
- **Recompensa (reward):** este semnalul numeric primit de agent ca urmare a unei acțiuni. Poate fi pozitivă (pentru acțiuni dorite, ex: eliminarea unui inamic) sau negativă (pentru greșeli sau comportamente indezirabile, ex: coliziune, moarte).
- **Politica (policy):** este strategia agentului de luare a deciziilor, adică o funcție π care mapează stări la acțiuni: $\pi(s) = a$. Aceasta poate fi deterministă (întotdeauna aceeași acțiune într-o stare) sau stocastică (alegere probabilistică).
- **Funcția de valoare (value function):** estimează cât de bună este o anumită stare (sau pereche stare-acțiune) în ceea ce privește recompensa viitoare așteptată. Această funcție este adesea învățată prin metode iterative.
- **Discount factor (γ):** un coeficient între 0 și 1 care stabilește cât de important este viitorul în comparație cu prezentul. Valori apropiate de 1 încurajează planificarea pe termen lung.
- **Episod și politică optimă:** un episod este o secvență de stări și acțiuni care începe de la o stare inițială și se termină când se ajunge într-o stare terminală (ex: game over). Politica optimă este acea strategie care maximizează recompensa totală așteptată pe parcursul episodului.

3.7 Algoritmi: Q-learning, Deep Q-Network (DQN)

Pentru ca un agent de reinforcement learning să-și îmbunătățească performanța în timp, este necesar un algoritm care să-i permită estimarea și optimizarea politicii sale de acțiune. Unul dintre cei mai utilizați algoritmi în această direcție este Q-learning, care stă la baza versiunilor mai avansate precum Deep Q-Network (DQN).

Q-learning este un algoritm de învățare off-policy care învață o funcție de valoare notată $Q(s, a)$ – adică valoarea așteptată a unei acțiuni a într-o stare s . Scopul

algoritmului este de a învăța o funcție Q cât mai apropiată de valoarea optimă Q^* , folosind o formulă de actualizare derivată din ecuația Bellman:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \cdot a' \max_{s'} Q(s',a') - Q(s,a)]$$

unde:

- α este rata de învățare (learning rate),
- γ este factorul de discount,
- r este recompensa primită,
- s' este starea rezultată în urma acțiunii a ,
- a' sunt acțiunile posibile din noua stare.

Q-learning este eficient în medii discrete, dar are dificultăți atunci când spațiile de stare și acțiune sunt foarte mari sau continue, așa cum se întâmplă în cazul jocurilor video cu observație vizuală.

Pentru a extinde Q-learning la probleme cu stări de mare dimensiune (ex: imagini), a fost introdus Deep Q-Network. În această variantă, funcția Q este aproximată printr-o rețea neuronală profundă (deep neural network), care primește ca intrare o stare și returnează valorile Q pentru toate acțiunile posibile.

Elementele cheie ale algoritmului DQN sunt:

- Rețea neuronală: înlocuiește tabelul Q cu un model parametric.
- Replay Buffer (Memorie de experiență): în loc să actualizeze rețeaua pe fiecare tranziție consecutivă, agentul stochează tranzițiile (s, a, r, s') într-o memorie și antrenează rețeaua pe mini-batch-uri eșantionate aleatoriu. Astfel, se rupe corelația temporală și se stabilizează învățarea.
- Target Network: o copie a rețelei Q este menținută separat și actualizată periodic. Această rețea este utilizată pentru a calcula valoarea țintă din ecuația Bellman, reducând astfel oscilațiile în procesul de învățare.
- Politica ϵ -greedy: agentul selectează cu probabilitate ϵ o acțiune aleatoare (explorare), iar cu probabilitate $1 - \epsilon$ acțiunea cu cea mai mare valoare Q (exploatare). Valoarea ϵ scade progresiv în timpul antrenamentului.
- Funcția de pierdere (loss function): diferența între valoarea Q estimată și valoarea-țintă calculată. Antrenamentul rețelei constă în minimizarea acestei erori.

DQN a fost folosit cu succes în jocuri Atari, obținând performanțe comparabile sau superioare jucătorilor umani. În contextul proiectului de față, DQN este utilizat în scenariile Snake și ViZDoom, fiind adaptat în funcție de specificul fiecărui joc:

- pentru jocuri simbolice (ex: Snake), intrările sunt vectori numerici;
- pentru jocuri vizuale (ex: Dino, ViZDoom), intrările sunt cadre de imagine (frame-uri), iar rețelele includ straturi convoluționale (CNN).

3.8 Arhitecturi de rețele neuronale folosite

În cadrul reinforcement learning cu funcții de valoare aproximative, alegerea arhitecturii rețelei neuronale joacă un rol esențial în performanța agentului. Rețelele trebuie să fie suficient de expresive pentru a învăța comportamente optime, dar și eficiente computațional, pentru a permite antrenamente rapide și stabile.

În proiectul de față, au fost utilizate două tipuri de rețele principale:

- rețele complet conectate (fully connected), folosite în jocuri cu stare simbolică;
- rețele convoluționale (CNN), folosite în jocuri cu observație vizuală.

Snake – Rețea Fully Connected:

Pentru jocul Snake, starea este reprezentată printr-un vector numeric care codifică direcția șarpelui, poziția mâncării, pericolele din vecinătate și starea direcțională curentă. Astfel, nu este necesară procesarea imaginilor, iar rețeaua neuronală poate fi de tip fully connected (densă):

- Input layer: 14 neuroni (dimensiunea vectorului de stare);
- Hidden layer: 256 neuroni, activare ReLU;
- Output layer: 3 neuroni (acțiuni posibile: înainte, stânga, dreapta).

Această arhitectură simplă este suficientă pentru un joc complet observabil și determinist ca Snake, permițând învățarea rapidă și eficientă a unei politici optime.

Dino Game – Rețea Convoluțională (CNN):

În cazul jocului Google Dino, observația este bazată pe imagini capturate de pe ecran, procesate ca imagini grayscale. Pentru a extrage automat caracteristici vizuale relevante (poziția cactușilor, păsări, teren), se folosește o rețea convoluțională:

- Input: imagine grayscale redimensionată la (100x83), reshape în tensor (1, 83, 100);
- Conv layer 1: 32 filtre 8x8, activare ReLU;
- Conv layer 2: 64 filtre 4x4, activare ReLU;
- Flatten;
- Fully connected layer: 256 neuroni;
- Output layer: 3 neuroni (acțiuni: jump, duck, no-op).

Această rețea este capabilă să recunoască rapid obstacolele în apropierea dinozaurului și să ia decizii în timp real, chiar și la viteze ridicate ale jocului.

ViZDoom – Rețele CNN extinse:

Pentru jocurile 3D din ViZDoom, observațiile sunt și mai complexe: agentul primește imagini RGB din perspectivă first-person, cu rezoluție scăzută (ex. 60x45). Se folosesc rețele convoluționale mai profunde, deoarece:

- agentul trebuie să înțeleagă geometria spațiului;
- este nevoie de recunoaștere a poziției inamicilor și obiectelor;
- input-ul este zgomotos și parțial observabil.

Arhitectura folosită în scenariul basic.wad:

- Input: imagine RGB 60x45x3;
- Conv layer 1: 32 filtre 7x7, max pooling, ReLU;
- Conv layer 2: 32 filtre 4x4, max pooling, ReLU;
- Fully connected: 800 neuroni Leaky ReLU;
- Output layer: 8 neuroni (combinații posibile ale acțiunilor: stânga, dreapta, foc).

Pentru scenariile mai complexe (health_gathering, deadly_corridor), se adaugă:

- un al treilea strat convoluțional;
- codificare suplimentară pentru variabile numerice (ex: health, ammo);
- input extins cu memorii temporale (ultimele 4 cadre consecutive).

Aceste rețele sunt antrenate cu algoritmi precum DQN, RMSprop sau Adam, folosind replay memory, ϵ -greedy policy și, în unele cazuri, shaping reward.

3.9 Tehnici auxiliare: experience replay, epsilon decay

În implementările moderne ale reinforcement learning, utilizarea rețelelor neuronale profunde introduce instabilități majore în procesul de antrenament. Pentru a contracara aceste instabilități și a îmbunătăți eficiența antrenării, se aplică o serie de tehnici auxiliare esențiale. Cele mai frecvent utilizate în cadrul acestui proiect sunt experience replay și epsilon decay.

Experience Replay (Memorie de experiență)

Experience replay este o tehnică introdusă pentru a elimina corelația temporală dintre tranzițiile observate consecutiv în timpul jocului. În mod normal, dacă agentul învață pe fiecare tranziție în ordinea în care apare, rețeaua neuronală poate ajunge să se „suprapotrivească” pe secvențe scurte și să nu generalizeze.

Funcționare:

- Fiecare tranziție de tip (stare, acțiune, recompensă, stare următoare) este salvată într-o memorie (buffer) de dimensiune fixă.
- La fiecare pas de antrenament, se selectează un mini-batch aleatoriu din buffer.
- Se actualizează rețeaua pe baza acestor tranziții amestecate, ceea ce duce la o învățare mai stabilă și eficientă.

Avantaje:

- Reduce varianța actualizărilor gradientului;
- Permite reutilizarea tranzițiilor valoroase;
- Crește stabilitatea procesului de învățare.

În toate scenariile din proiect (Snake, Dino, ViZDoom), a fost utilizat experience replay cu o capacitate între 10.000 și 100.000 de tranziții și batch size-uri între 32 și 64. Epsilon Decay (Strategie de explorare controlată)

O provocare fundamentală în RL este echilibrul între explorare (încercarea de acțiuni noi pentru a descoperi strategii mai bune) și exploatare (folosirea celor mai bune acțiuni deja cunoscute). Pentru a gestiona acest echilibru, se folosește o strategie ϵ -greedy, în care:

- Cu probabilitate ϵ , agentul alege o acțiune aleatorie (explorare);
- Cu probabilitate $1 - \epsilon$, agentul alege acțiunea cu cea mai mare valoare Q (exploatare).

Epsilon decay înseamnă că valoarea ϵ scade gradual în timp, astfel încât agentul explorează mai mult la începutul antrenamentului, dar tinde spre comportamente exploatare pe măsură ce învață.

Exemplu concret de decay:

- ϵ inițial = 1.0 (explorare totală);
- ϵ final = 0.1;
- scădere liniară pe parcursul a 100.000 – 200.000 de pași de antrenament.

Această tehnică este crucială pentru a evita „blocarea” agentului în politici suboptimale descoperite prematur. În proiectul propriu, toate jocurile folosesc ϵ decay, cu parametri adaptați la durată și complexitatea fiecărui mediu.

3.10 Frameworkuri și biblioteci utilizate

3.10.1 Python:

Este un limbaj de programare de nivel înalt, interpretat, dinamic și multi-paradigmă, care a devenit una dintre cele mai populare alegeri în domeniul inteligenței artificiale, învățării automate și științei datelor. Creat de Guido van Rossum și lansat în 1991, Python este cunoscut pentru sintaxa sa clară și concisă, care favorizează lizibilitatea codului și o curba de învățare accesibilă chiar și pentru începători. Spre deosebire de alte limbaje de programare, Python încurajează dezvoltarea rapidă și prototiparea prin stilul său expresiv și orientat pe obiecte.

Unul dintre cele mai importante aspecte ale limbajului este comunitatea activă și vastul ecosistem de biblioteci disponibile. Python este susținut de o infrastructură matură care acoperă toate domeniile importante ale programării: rețele neuronale, vizualizare, procesare a datelor, dezvoltare web, simulări, automatizări și chiar crearea de jocuri. Prin extensii precum NumPy, PyTorch, TensorFlow sau OpenCV, Python a devenit limbajul principal utilizat în cercetarea și industria AI.



Figura 4.1 Python

Datorită acestor caracteristici, Python este preferat atât în mediul academic, cât și în companii de top precum Google, Facebook, Amazon și Microsoft. El este folosit în proiecte de anvergură din robotică, procesarea limbajului natural (NLP), recunoaștere facială, predicții economice și, relevant pentru această lucrare, învățarea prin întărire aplicată în jocuri video.

3.10.2 Python în aplicația noastră:

În cadrul proiectului de față, Python reprezintă pilonul central al întregii aplicații. Toate modulele, fie că sunt legate de logica jocului, antrenarea agenților sau interfațarea cu medii externe (ex: ViZDoom), sunt implementate în acest limbaj. Utilizarea Python a permis o dezvoltare modulară, clară și rapidă, lucru esențial pentru un proiect de cercetare cu o componentă practică puternică.

Un prim exemplu de utilizare este în implementarea logicii de joc pentru Snake și Google Dino. Folosind doar câteva sute de linii de cod, a fost posibilă crearea unor motoare de joc complet funcționale, în care agentul primește în mod constant feedback de la mediu, efectuează acțiuni și înregistrează tranzițiile. Datorită simplității limbajului, logica de coliziune, randarea vizuală și funcțiile de recompensă au fost implementate într-un mod intuitiv și ușor de ajustat pe parcursul testelor.

În ceea ce privește integrarea cu rețelele neuronale, Python excelează prin interoperabilitatea cu biblioteci precum PyTorch. Arhitecturile rețelelor sunt definite într-un mod declarativ și flexibil, permițând ajustarea rapidă a numărului de neuroni, tipului de activare sau structurii stratificate. Python oferă, de asemenea, suport nativ pentru GPU prin interfețele cu CUDA, facilitând antrenamente rapide chiar și pe hardware modest.

Python a fost esențial și în colectarea și gestionarea datelor necesare pentru experience replay. Prin intermediul structurii deque sau a listelor Python native, tranzițiile (state, action, reward, next_state) au fost memorate și accesate eficient în timpul antrenamentului. De asemenea, fișierele de configurare, salvarea modelelor și managementul sesiunilor au fost realizate folosind module Python precum os, pickle, json sau argparse.

Pentru partea de vizualizare și debugging, Python a oferit instrumente precum matplotlib pentru desenarea curbelor de învățare, scorurilor per episod și diverși alți indicatori. Acest aspect a fost crucial în etapa de validare, unde au fost comparate performanțele între jocuri și s-a verificat stabilitatea antrenamentului.

Nu în ultimul rând, Python a permis integrarea fluidă cu biblioteca ViZDoom, oferind acces complet la motorul jocului și la toate resursele vizuale, variabilele interne și comenzile necesare controlului agentului. Interfața Python a ViZDoom a fost stabilă, documentată și ușor de utilizat, făcând posibilă comunicarea în timp real între agent și mediu.

În concluzie, Python nu este doar limbajul de implementare, ci reprezintă un liant între toate componentele proiectului: joc, agent, mediu, algoritm, antrenament și testare. Alegerea sa a contribuit semnificativ la succesul și coerența tehnică a proiectului.

3.10.3 Numpy:

NumPy (Numerical Python) este o bibliotecă fundamentală pentru programarea științifică în Python, specializată în manipularea eficientă a datelor sub formă de matrici (array-uri multidimensionale) și în efectuarea de calcule matematice de mare viteză. Creată inițial de Travis Oliphant în 2005 ca o unificare a mai multor biblioteci de matematică existente (Numeric și numarray), NumPy a devenit rapid un standard industrial și academic pentru lucrul cu date numerice în Python.

În esență, NumPy oferă un nou tip de obiect – ndarray – care permite stocarea eficientă a datelor în formă tabelară, împreună cu un set de funcții optimizate pentru manipularea acestora: operații vectoriale, produse scalare, algebră liniară, funcții statistice, generatoare de numere aleatoare, transformate Fourier etc. De asemenea, biblioteca oferă suport nativ pentru broadcasting, ceea ce permite efectuarea de operații între array-uri de dimensiuni diferite într-un mod automat și performant.



Figura 4.2 Numpy

Un alt avantaj major este faptul că funcțiile din NumPy sunt implementate în C și Fortran, ceea ce înseamnă că rularea codului este extrem de rapidă, comparabilă cu limbaje compilate, dar cu simplitatea și flexibilitatea Python. Acest lucru este crucial în aplicații precum rețele neuronale, simulări sau procesarea de imagini, unde volumele de date sunt mari și viteza este un factor important.

Datorită acestor caracteristici, NumPy este biblioteca pe care se bazează majoritatea celorlalte frameworkuri de AI, cum ar fi TensorFlow, PyTorch, scikit-learn sau OpenCV. Ea stă la baza ecosistemului Python pentru știința datelor și este utilizată pe scară largă în domenii precum robotică, analiză financiară, fizică computațională și, bineînțeles, învățarea automată.

3.10.4 NumPy în aplicația noastră:

În cadrul proiectului de disertație, NumPy a jucat un rol esențial în manipularea și procesarea eficientă a datelor utilizate în toate etapele dezvoltării agenților de învățare. În mod particular, NumPy a fost folosit pentru reprezentarea stărilor agentului, pentru manipularea imaginilor capturate din joc și pentru operații matematice implicate în calculul funcțiilor de pierdere și actualizarea rețelei neuronale.

În cazul jocului Snake, starea agentului este exprimată sub formă de vector de caracteristici (ex. poziție relativă față de mâncare, pericole în jur, direcția curentă etc.). Acești vectori sunt manipulați constant în timpul antrenamentului, fiind convertiți în array-uri NumPy, normalizați și transmiși către rețelele neurale. Operații precum `np.argmax`, `np.random.choice` sau `np.dot` sunt folosite frecvent pentru selecția acțiunii, explorare stocastică și propagarea valorilor.

În cazul jocului Google Dino, imaginile capturate din fereastra de joc sunt prelucrate ca matrice (array-uri bidimensionale), convertite în tonuri de gri, redimensionate și stocate sub formă de tensori. Aici, NumPy este folosit împreună cu OpenCV pentru manipularea directă a pixelilor, iar apoi array-urile sunt trecute în PyTorch ca tensori, cu funcția `torch.from_numpy()`.

În jocurile ViZDoom, unde datele vizuale sunt RGB, NumPy este utilizat pentru extragerea canalelor de culoare, normalizarea pixelilor și construirea unor mini-batch-uri de imagini consecutive (ex: stacking a 4 cadre pentru input temporal). Dimensiunile array-urilor sunt manipulate cu funcții precum `np.reshape`, `np.transpose` și `np.concatenate`, în scopul compatibilizării cu cerințele CNN-urilor.

O altă utilizare majoră a NumPy apare în mecanismul de experience replay. Fiecare tranziție (stare, acțiune, recompensă, stare nouă) este salvată ca un tuplu de array-uri, iar selecția mini-batch-urilor pentru antrenament se face prin indexare vectorială (`np.random.randint`, `np.array([...])`). Această abordare permite antrenarea eficientă a rețelei pe date amestecate și necorelate temporal, esențial pentru stabilitatea algoritmilor DQN.

De asemenea, NumPy a fost utilizat în etapa de analiză a performanței: calcularea scorului mediu per episod, deviației standard, maximizarea/minimizarea valorilor, sau generarea de date pentru graficul de convergență. Astfel, biblioteca a contribuit activ nu doar la partea de antrenament, ci și la validarea și interpretarea rezultatelor.

Prin toate aceste utilizări, NumPy a acționat ca un liant între toate componentele matematice și de date ale aplicației, oferind performanță, simplitate și fiabilitate. A fost indispensabil în manipularea eficientă a datelor și în funcționarea corectă a algoritmilor de învățare folosiți în proiect.

3.10.5 Pygame:

Pygame este o bibliotecă gratuită și open-source pentru dezvoltarea de jocuri 2D și aplicații multimedia în limbajul Python. A fost creată inițial în anul 2000 de către Pete Shinnars și este construită pe baza bibliotecii SDL (Simple DirectMedia Layer), o interfață C foarte performantă folosită în dezvoltarea de jocuri și aplicații grafice interactive. Pygame oferă un set extins de module care permit controlul complet al graficii, sunetului, evenimentelor de la tastatură sau mouse, coliziunilor, animației și afișării de text.



[Figura 4.3 Pygame](#)

Scopul principal al bibliotecii este de a oferi un cadru simplu și accesibil pentru dezvoltarea de jocuri sau simulări educaționale, fără a necesita cunoștințe avansate de programare orientată pe obiecte sau de grafică OpenGL. Ea este compatibilă cu toate sistemele de operare majore (Windows, Linux, macOS) și poate fi integrată ușor în proiecte de cercetare, experimente didactice sau chiar produse comerciale 2D.

Un avantaj important al Pygame constă în controlul granular asupra fiecărui pixel al ecranului, ceea ce o face utilă nu doar pentru afișare vizuală, ci și pentru capturarea și procesarea cadrelor de joc. Această caracteristică o transformă într-o alegere excelentă pentru testarea algoritmilor de inteligență artificială în jocuri simple, precum Flappy Bird, Tetris, Snake sau Google Dino, oferind în același timp performanță acceptabilă și ușurință în utilizare.

3.10.6 Pygame în aplicația noastră:

În cadrul proiectului de disertație, biblioteca Pygame a fost utilizată în mod extensiv pentru implementarea și controlul jocului Google Dino. Deoarece jocul original, integrat în browserul Google Chrome, nu oferă o interfață de programare (API) accesibilă, a fost necesară dezvoltarea unei versiuni proprii a jocului care să fie complet controlabilă de un agent RL. Pygame a fost alegerea ideală pentru acest scop datorită flexibilității și a suportului nativ pentru desenarea și actualizarea în timp real a obiectelor grafice.

În această implementare, Pygame a permis:

- Definirea obiectelor de joc precum dinozaurul, cactușii, păsările, solul și cerul;
- Gestionarea coliziunilor, detectând momentul în care personajul agent atinge un obstacol;
- Controlul logicii de joc, cum ar fi generarea obstacolelor, creșterea vitezei în timp, sau resetarea episodului în caz de eșec;
- Capturarea imaginilor (frame-urilor) pentru a fi utilizate drept stare de intrare în rețeaua neuronală a agentului.

Unul dintre cele mai importante roluri ale Pygame în acest proiect a fost acela de a oferi o sursă constantă de date vizuale pentru agentul RL. La fiecare pas de joc, imaginea generată de Pygame este capturată cu funcții dedicate (`pygame.surfarray.array3d()`), convertită în grayscale, redimensionată și stocată sub formă de matrice NumPy. Aceste cadre reprezintă stările de intrare în rețeaua neuronală convoluțională a agentului care decide ce acțiune să întreprindă: să sară, să se ghemuiască sau să continue.

Pygame a fost, de asemenea, utilă în procesul de debugging vizual și evaluare. Prin afișarea scorului, a numărului de episoade și a vitezei de rulare, a fost posibilă urmărirea performanței agentului în timp real și intervenția asupra parametrilor de antrenament în caz de stagnare. Mai mult decât atât, Pygame a permis simularea unor sesiuni lungi de joc într-un mod complet autonom, în care agentul învață doar pe baza observațiilor vizuale, fără alte date explicite despre joc.

Un alt aspect notabil este faptul că Pygame a fost folosit și în modul „spectator”, în care agentul RL antrenat era testat fără a mai înregistra experiență nouă, ci doar pentru a demonstra comportamentul învățat. Acest mod a fost esențial pentru validarea performanței și realizarea graficelor de convergență a scorului.

În concluzie, Pygame a fost coloana vertebrală a componentei vizuale și logice a jocului Dino, făcând posibilă integrarea ușoară a învățării prin întărire într-un mediu controlabil și reproductibil. Alegerea acestei biblioteci a oferit echilibrul optim între simplitate, funcționalitate și flexibilitate necesară într-un proiect educațional cu aplicabilitate practică.

3.10.7 PyTorch:

PyTorch este un framework open-source pentru învățare profundă (deep learning), dezvoltat și întreținut de Meta AI Research (fosta echipă Facebook AI Research – FAIR), fiind lansat oficial în 2016. De atunci, a devenit rapid unul dintre cele mai populare instrumente pentru dezvoltarea și antrenarea rețelelor neuronale, în special în mediul academic. Popularitatea PyTorch a crescut exponențial datorită flexibilității sale, simplității în utilizare și performanței ridicate în aplicații practice de inteligență artificială.



[Figura 4.4 PyTorch](#)

Unul dintre principiile fundamentale ale PyTorch este oferirea unui model de programare imperativ și dinamic, în contrast cu modelul declarativ adoptat anterior de

TensorFlow 1.x. Acest lucru înseamnă că grafurile computaționale sunt construite „în timp real”, în momentul execuției codului, nu în prealabil. Această abordare le oferă dezvoltatorilor un control mult mai mare asupra fluxului logic și facilitează procesul de debugging, făcând codul mai intuitiv și mai apropiat de programarea Python obișnuită.

PyTorch oferă suport nativ pentru accelerare pe GPU prin CUDA, ceea ce permite rularea și antrenarea rețelelor neuronale în paralel, la viteză mare. Acesta include componente modulare pentru toate etapele unui proiect AI: definirea rețelelor (`torch.nn`), optimizare (`torch.optim`), manipularea tensorilor (`torch.Tensor`), autodiferențiere (`torch.autograd`) și multe altele. De asemenea, oferă compatibilitate completă cu NumPy, permițând conversia directă între tipurile de date.

Comparativ cu alte frameworkuri de învățare automată, cum ar fi TensorFlow, Keras sau MXNet, PyTorch s-a impus în special în mediile de cercetare, fiind alegerea preferată în publicații academice de top (NeurIPS, ICLR, ICML). Motivele acestei preferințe includ:

- simplitatea în definirea și modificarea arhitecturilor rețelei;
- suportul complet pentru grafuri dinamice;
- documentație extensivă și comunitate activă;
- integrarea ușoară cu instrumente externe precum Hugging Face, OpenAI Gym sau ViZDoom.

În prezent, PyTorch este folosit nu doar în laboratoare de cercetare, ci și în producție, de către companii precum Meta, Microsoft, Tesla, NVIDIA, Amazon și multe altele. Aplicațiile sale variază de la viziune computerizată (CV), procesarea limbajului natural (NLP) și recunoaștere vocală, până la robotică, medicină și – relevant pentru acest proiect – învățarea prin întărire aplicată în jocuri video.

Prin combinația sa de putere, flexibilitate și eleganță în programare, PyTorch reprezintă o alegere strategică ideală pentru dezvoltarea agenților inteligenți, permițând o tranziție naturală de la prototipuri rapide la implementări robuste și eficiente.

3.10.8 Fundamente tehnice și matematice în PyTorch:

La baza funcționării frameworkului PyTorch se află câteva concepte fundamentale, care susțin atât arhitectura internă a sistemului, cât și procesele matematice din spatele învățării automate. Acestea includ: tensori, grafuri computaționale dinamice, autograd (autodiferențiere) și algoritmi de optimizare.

Un tensor este o generalizare a conceptului de scalari, vectori și matrici. Matematic, un tensor este un obiect multidimensional care poate conține date numerice și permite efectuarea de operații algebrice asupra acestora. În PyTorch, tensorii sunt implementați prin clasa `torch.Tensor` și funcționează analog cu array-urile din NumPy, dar cu suport nativ pentru GPU.

Tip Tensor	Exemplar
Scalar (0-D)	<code>torch.tensor(5.0)</code>
Vector (1-D)	<code>torch.tensor([1.0, 2.0])</code>
Matrice (2-D)	<code>torch.tensor([[1, 2], [3, 4]])</code>
Tensor 3-D+	image RGB: <code>[3 x H x W]</code>

Aceste structuri de date sunt esențiale în învățarea profundă, deoarece ele stochează intrările, greutatea, ieșirile și toate calculele intermediare într-o rețea neuronală.

Unul dintre cele mai inovatoare aspecte ale PyTorch este sistemul său de grafe computaționale dinamice. Spre deosebire de TensorFlow 1.x, unde graful de operații este definit static înainte de execuție, în PyTorch graful este construit dinamic adică pas cu pas, în timpul rulării efective a codului.

Acest lucru este realizat prin sistemul autograd, care se ocupă de calculul automat al derivatelor. În timpul propagării directe (forward pass), PyTorch construiește un arbore de operații. Când se apelează `loss.backward()`, autograd parcurge graful în sens invers și calculează derivata parțială a funcției de pierdere față de fiecare parametru – adică execută algoritmul backpropagation.

Matematic, pentru o funcție de pierdere $L(w)$, învățarea presupune actualizarea parametrilor w cu regula:

$$w \leftarrow w - \alpha \cdot \partial w / \partial L$$

unde:

- α este rata de învățare (learning rate),
- $\partial w / \partial L$ este gradientul derivat automat de autograd.

În PyTorch, acest proces este automatizat, dar complet transparent. Fiecare tensor cu `requires_grad=True` devine parte din graful computațional, iar gradientul său este stocat în `tensor.grad`.

Pentru a actualiza greutatea rețelei în funcție de gradient, PyTorch oferă o gamă de optimizatori în modulul `torch.optim`. În cadrul acestui proiect au fost utilizați:

- SGD (Stochastic Gradient Descent)
Algoritm de bază, actualizează fiecare parametru după regula gradientului negativ. Deși simplu, este lent în convergență și sensibil la platouri.
- Adam (Adaptive Moment Estimation)
Combină avantajele SGD cu moment și adaptarea din RMSProp. Folosește o medie mobilă a gradientului și a pătratului gradientului pentru actualizări adaptative ale fiecărui parametru. Adam este folosit pe scară largă în rețele neuronale convoluționale (CNN), fiind optim atât pentru stabilitate, cât și pentru viteza de antrenare.

3.10.9 Rețele neuronale în PyTorch:

Una dintre cele mai importante facilități oferite de PyTorch este modulul său dedicat pentru definirea și antrenarea rețelelor neuronale, cunoscut sub denumirea de `torch.nn`. Acesta oferă o arhitectură modulară și extensibilă care permite construirea ușoară a rețelelor de tip feed-forward, convoluționale sau recurente, prin utilizarea unor componente standardizate precum straturi liniare, funcții de activare, normalizări și regularizări.

Definirea unei rețele neuronale în PyTorch presupune modelarea acesteia sub forma unei clase ce derivă dintr-o clasă de bază predefinită. Arhitectura este declarată în constructorul clasei, unde sunt inițializate straturile dorite, iar logica propagării înainte (forward propagation) este descrisă într-o funcție dedicată. Această abordare permite reutilizarea codului, flexibilitate în ajustarea arhitecturii și o claritate crescută în procesul de definire a modelelor.

În cadrul acestui proiect, au fost utilizate două tipuri principale de rețele:

- Rețele complet conectate (fully connected): specifice jocurilor în care starea agentului poate fi descrisă printr-un vector numeric (ex: jocul Snake). Aceste

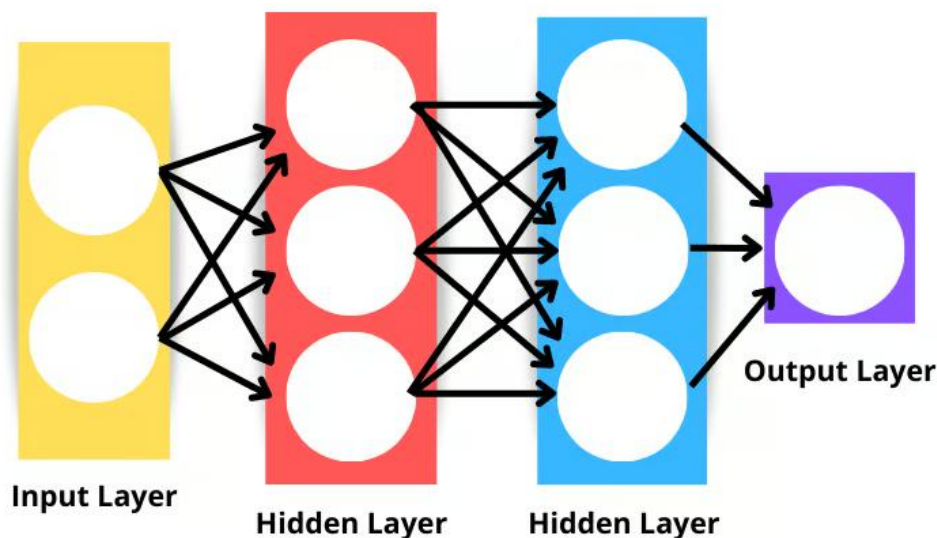
rețele conțin un strat de intrare, unul sau mai multe straturi ascunse dense și un strat de ieșire care generează predicția asupra acțiunii optime.

- Rețele convoluționale (CNN): utilizate în jocuri cu observații vizuale (ex: Dino sau ViZDoom). Acestea includ unul sau mai multe straturi convoluționale, eventual urmate de straturi de pooling și complet conectate, fiind special concepute pentru a extrage caracteristici relevante din imagini, cum ar fi poziția obstacolelor sau prezența inamicilor.

PyTorch oferă o suită completă de funcții de activare esențiale pentru introducerea non-liniarității în rețea. Cele mai utilizate în acest proiect au fost:

- ReLU (Rectified Linear Unit) – funcție de activare standard în rețele moderne, datorită comportamentului său simplu și eficient;
- Leaky ReLU – o variantă îmbunătățită a ReLU, care permite un gradient mic și pentru valori negative;
- Softmax – utilizată în stratul final al rețelelor atunci când este necesară o ieșire de tip distribuție de probabilități, mai ales în cazul clasificării acțiunilor.

Un alt element fundamental în antrenarea rețelelor neuronale este alegerea funcției de pierdere (loss function), care cuantifică eroarea dintre predicțiile rețelei și valorile așteptate. În acest proiect a fost folosită funcția de eroare pătratică medie (Mean Squared Error MSE), adecvată în contextul algoritmului DQN, unde se încearcă aproximarea valorilor Q pentru fiecare acțiune într-o stare dată. Această funcție penalizează mai sever deviațiile mari, încurajând astfel predicții mai stabile.



[Figura 4.5 Rețea în Pytorch](#)

Prin aceste componente, PyTorch permite definirea unor rețele neuronale complexe într-un mod intuitiv, menținând în același timp un nivel ridicat de control asupra fluxului de date și al calculelor matematice. Structura modulară a rețelelor permite ajustarea rapidă a arhitecturii (număr de neuroni, adâncime, tipuri de straturi), ceea ce este esențial în etapa de experimentare și optimizare a agenților inteligenți.

3.10.10 Utilizarea PyTorch în cadrul aplicației noastre:

În cadrul acestui proiect de disertație, PyTorch a reprezentat componenta centrală în dezvoltarea, antrenarea și evaluarea agenților inteligenți pentru toate cele trei jocuri studiate: Snake, Google Dino și ViZDoom. Utilizarea acestui framework a permis crearea de rețele neuronale personalizate, definirea logicii de antrenament, optimizarea parametrilor și salvarea modelelor într-un mod eficient și reproductibil.

Snake:

În cazul jocului Snake, s-a utilizat o rețea neuronală complet conectată, definită în PyTorch prin clasa QNetwork. Starea agentului este un vector numeric format din 14 elemente care descriu: direcția actuală, pericolele din jur, distanța până la mâncare etc. Rețeaua este formată din:

- un strat de intrare cu 14 neuroni,
- un strat ascuns cu 256 de neuroni și activare ReLU,
- un strat de ieșire cu 3 neuroni corespunzători acțiunilor (stânga, dreapta, înainte).

Modelul a fost antrenat cu MSELoss și optimizat cu Adam, folosind tranziții extrase din experience replay. Rețeaua a fost salvată și reutilizată în sesiuni de testare pentru evaluarea performanței.

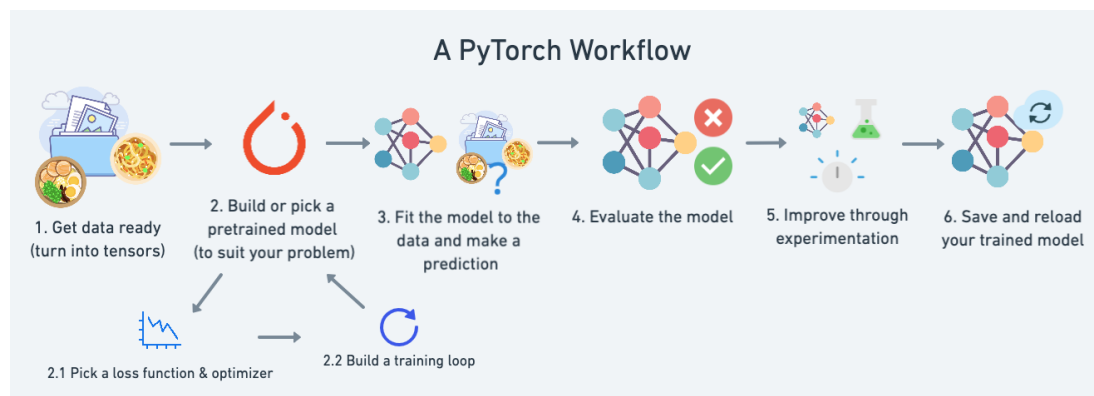


Figura 4.6 Workflow of a PyTorch Model

Google Dino:

În cazul Google Dino, agentul trebuie să proceseze date vizuale. La fiecare pas, imaginea de pe ecran este capturată, convertită în grayscale, redimensionată și transformată în tensor PyTorch. Rețeaua folosită este de tip CNN și conține:

- două straturi convoluționale (32 și 64 filtre),
- un strat fully connected de 256 neuroni,
- un strat de ieșire cu 3 neuroni (sări, ghemuire, nimic).

Propagarea directă (forward) primește imaginea ca tensor 4D [batch, channel, height, width] și returnează predicția pentru acțiunea optimă. PyTorch a fost esențial în definirea convoluțiilor, activărilor ReLU și funcției de pierdere. S-a folosit tot MSELoss, deoarece este implementat un agent de tip DQN.

De asemenea, PyTorch a permis:

- aplicarea de gradient clipping pentru a evita explozia gradientului;
- salvarea modelului după fiecare 1000 de episoade;
- vizualizarea în tensorboard a valorilor Q în timp (opțional).

ViZDoom:

În cazul jocului ViZDoom, PyTorch a fost utilizat pentru dezvoltarea celor mai complexe rețele din proiect. Inputul este o imagine RGB de dimensiune redusă (ex.

60x45), procesată printr-o rețea CNN cu 2 sau 3 straturi convoluționale, urmate de fully connected și strat de ieșire. În funcție de scenariu:

- basic.wad – rețea cu 2 straturi conv, 800 neuroni hidden, 8 acțiuni posibile;
- defend_the_center.wad – rețea cu 3 straturi conv, input extins cu 4 cadre consecutive;
- deadly_corridor.wad – input + variabile interne (health, ammo), concatenate în forward.

În toate cazurile, autograd a fost folosit pentru backpropagation, iar optimizatorul Adam pentru actualizarea greutăților. De asemenea:

- în training loop s-au calculat valorile-țintă folosind Bellman equation;
- funcția `loss.backward()` a declanșat autodiferențierea;
- optimizatorul a aplicat `step()` după fiecare batch.

Pe lângă arhitectura rețelelor și training loop, PyTorch a mai fost folosit pentru:

- Gestionarea pe GPU: prin `.to(device)`, rețelele și datele au fost mutate pe GPU dacă era disponibil;
- Salvarea și încărcarea modelelor

În concluzie, PyTorch a fost un element esențial în implementarea acestui proiect. El a oferit toate uneltele necesare pentru a trece rapid de la prototipuri simple la rețele complexe de tip DQN în jocuri 3D, cu suport pentru analiză matematică riguroasă, execuție pe GPU și integrare cu alte biblioteci. Fără PyTorch, dezvoltarea acestui proiect ar fi fost mult mai lentă, mai rigidă și mai predispusă la erori.

3.10.11 OpenCV:

OpenCV (Open Source Computer Vision Library) este o bibliotecă open-source dedicată procesării imaginilor și viziunii computerizate. A fost dezvoltată inițial de către Intel în anul 2000 și este în prezent întreținută de o comunitate largă de dezvoltatori. Biblioteca este scrisă în limbajele C și C++, dar oferă interfețe pentru Python, Java și alte limbaje, fiind una dintre cele mai utilizate soluții în aplicații de analiză vizuală, inteligență artificială și robotică.

OpenCV conține peste 2500 de funcții optimizate care acoperă o gamă largă de sarcini specifice viziunii computerizate, cum ar fi:

- procesarea imaginilor (filtrare, transformări geometrice, conversii de culoare);
- detectarea marginilor, contururilor și trăsăturilor;
- recunoaștere facială, detecție de obiecte și mișcare;
- calibrare de cameră și analiză 3D;
- citire, scriere și manipulare de fișiere video și foto.

Un avantaj major al OpenCV este eficiența sa computațională: multe dintre funcțiile sale sunt scrise în C++ și optimizează utilizarea resurselor de procesare, putând rula în timp real chiar și pe dispozitive embedded sau mobile. Acest aspect face biblioteca ideală pentru aplicații care necesită viteză mare de prelucrare a imaginilor, cum ar fi sisteme autonome, supraveghere video sau, în cazul acestui proiect, interacțiunea vizuală cu jocuri video.

Datorită versatilității sale, OpenCV este utilizată pe scară largă atât în cercetare, cât și în industrie, în domenii precum automotive (mașini autonome), securitate (detecție de fețe sau numere de înmatriculare), realitate augmentată, control de calitate industrial și robotică.

3.10.12 OpenCV în aplicația noastră:

În cadrul acestui proiect, OpenCV a fost utilizată ca un instrument auxiliar esențial pentru procesarea imaginilor capturate din medii vizuale complexe, în special în cazul jocurilor Google Dino și ViZDoom. Deoarece agenții de tip reinforcement learning antrenați în aceste jocuri nu primesc date simbolice (cum ar fi poziția exactă a obstacolelor sau a inamicilor), ci doar observații sub formă de imagini brute, prelucrarea acestor imagini este o etapă critică înainte de a fi transmise rețelei neuronale.

OpenCV a fost folosit în mai multe scopuri concrete în aplicația noastră:

- Grayscale: pentru a reduce dimensiunea datelor și a elimina canalele de culoare irelevante, cadrele capturate din joc au fost convertite din format RGB în grayscale. Acest pas reduce complexitatea rețelei și accelerează procesul de învățare, menținând în același timp informațiile vizuale esențiale (forme, obstacole, poziția pe ecran etc.).
- Redimensionare a imaginilor: rețelele convoluționale utilizate în proiect necesită un input cu dimensiuni fixe. Prin urmare, cadrele au fost redimensionate (ex. la 100x83 pixeli) pentru a standardiza intrarea în rețea și a reduce costul computațional.
- Normalizarea valorilor pixelilor: pixelii imaginii, inițial cu valori între 0 și 255, au fost normalizați între 0 și 1, o practică standard în rețelele neuronale pentru a stabili procesul de antrenament și a îmbunătăți convergența.
- Transformarea imaginilor în array-uri numerice: OpenCV a fost folosit pentru a converti cadrele în matrici NumPy, care ulterior au fost convertite în tensori PyTorch, ce pot fi procesați direct de rețelele convoluționale.
- Debug vizual și afișare în timp real: în timpul testării agentului, OpenCV a permis afișarea în timp real a stării curente, a predicțiilor rețelei și a scorului, oferind un feedback vizual clar asupra comportamentului învățat. Acest lucru a fost crucial pentru observarea eventualelor probleme în logica agentului sau în procesul de preprocesare.

Un exemplu relevant este antrenarea agentului pentru Google Dino. Deoarece jocul rulează într-un mediu grafic 2D fără acces programatic la informații interne (API), cadrele au fost capturate direct de pe ecran, prelucrate cu OpenCV și transmise modelului CNN. Fără o prelucrare eficientă a imaginii, agentul nu ar fi fost capabil să înțeleagă apariția obstacolelor sau momentul optim pentru a sări sau a se ghemui.

De asemenea, în cazul ViZDoom, unde mediul este 3D și imaginile sunt mai complexe, OpenCV a fost folosit pentru vizualizarea cadrelor de input și pentru diagnosticarea cazurilor în care agentul învață comportamente incorecte (ex: ignorarea inamicilor, rătăcirea în mediu). Prin afișarea live a secvenței de cadre, cercetătorul a putut ajusta funcția de recompensă sau parametrii modelului în funcție de comportamentul observat.

În concluzie, OpenCV a fost un instrument-cheie în faza de preprocesare a datelor vizuale și a asigurat o punte eficientă între mediul grafic al jocului și structura numerică necesară rețelelor neuronale. Fără această etapă de conversie și ajustare a datelor vizuale, învățarea automatizată a agentului nu ar fi fost posibilă într-un mod robust și scalabil.

3.10.13 ViZDoom:

ViZDoom este o platformă de cercetare în inteligență artificială specializată pentru Reinforcement learning în medii vizuale 3D. Aceasta este construită peste motorul open-source al celebrului joc Doom (versiunea ZDoom) și oferă o interfață programabilă ce permite controlul complet al unui agent virtual într-un mediu FPS (first-person shooter). ViZDoom a fost creat și este întreținut de Universitatea Tehnologică din Poznań, Polonia, fiind lansat oficial în 2016.

Principalul obiectiv al ViZDoom este să faciliteze testarea algoritmilor de învățare automată într-un mediu realist, dar controlabil, unde agentul trebuie să ia decizii bazate exclusiv pe informații vizuale (cadre RGB sau hărți de adâncime), similare percepției umane. Platforma nu oferă acces la informații abstracte, cum ar fi pozițiile exacte ale inamicilor sau coordonatele hărții, ceea ce obligă agentul să învețe din observarea mediului.

Caracteristicile esențiale ale ViZDoom includ:

- perspectivă subiectivă (first-person), cu observabilitate parțială;
- posibilitatea de a crea și personaliza scenarii, cu hărți, obiective și reguli definite de utilizator;
- control complet al vitezei de execuție, inclusiv rulare în mod asincron sau sincronizat cu agentul;
- acces la date vizuale și variabile interne (ex: sănătate, muniție, scor);
- execuție rapidă (până la 7000 de cadre pe secundă în modul fără randare grafică);
- suport pentru platforme multiple (Windows, Linux) și integrare directă în limbajul Python.

ViZDoom a fost utilizat în numeroase lucrări științifice, inclusiv în competiții internaționale precum Visual Doom AI Competition, și este considerat un mediu standard pentru testarea agenților de învățare vizuală în contexte complexe, care necesită percepție, luare de decizii, navigație și strategie de luptă.

3.10.14 ViZDoom în aplicația noastră:

În cadrul acestui proiect, ViZDoom a reprezentat platforma centrală pentru dezvoltarea și validarea agenților în medii complexe 3D, mult mai apropiate de jocurile comerciale decât Snake sau Google Dino. Agentul are acces exclusiv la imagini RGB din perspectiva unui jucător, iar sarcina sa este să învețe comportamente eficiente – cum ar fi orientarea, deplasarea, atacul și supraviețuirea – într-un mediu necunoscut, dinamic și parțial observabil.

Au fost abordate și antrenate trei scenarii ViZDoom, fiecare cu niveluri diferite de dificultate:

- `basic.wad` – un scenariu simplu, în care agentul trebuie să se orienteze stânga/dreapta și să elimine un inamic apărut într-o cameră fixă;
- `defend_the_center.wad` – un scenariu de tip „arena” unde inamicii vin din toate direcțiile, iar agentul trebuie să se rotească, să tragă și să supraviețuiască cât mai mult timp;
- `deadly_corridor.wad` – un nivel liniar, complex, unde agentul trebuie să avanseze printr-un coridor periculos, eliminând inamici și evitând proiectile.

Pentru fiecare scenariu, ViZDoom a oferit:

- generarea cadrelor de joc ca imagini RGB de dimensiune redusă (ex. 60x45 pixeli);

- interfață de control a acțiunilor agentului, sub formă de vector binar (ex. [turn_left, move_forward, shoot]);
- recompense configurabile, adaptate la obiectivele scenariului (ex. +100 la uciderea unui inamic, -5 pentru glonț ratat);
- posibilitatea de rulare în mod sincron, astfel încât agentul să ia o decizie pe fiecare cadru.

În cadrul procesului de antrenament, ViZDoom a fost integrat cu rețeaua neuronală implementată în PyTorch. Cadrele generate de ViZDoom au fost preluate, convertite în grayscale redimensionate, normalizate și transformate în tensori. Acestea au fost transmise ca input către un model convoluțional (CNN), care a returnat predicții pentru valorile Q asociate fiecărei acțiuni posibile. Alegerea acțiunii s-a făcut fie prin politică ϵ -greedy, fie prin maximizarea valorii Q.

Funcția de recompensă pentru fiecare scenariu a fost ajustată în funcție de comportamentul observat al agentului. ViZDoom a permis definirea unor recompense personalizate, ceea ce a fost crucial pentru a stimula comportamente dorite (precizie, orientare corectă, economie de muniție) și a penaliza acțiuni inutile (rotații în gol, stagnare).

Un alt avantaj major a fost capacitatea ViZDoom de a rula în mod off-screen, fără interfață grafică, ceea ce a permis accelerarea antrenamentului și rularea de mii de episoade în fundal. Acest lucru a fost esențial pentru convergența algoritmilor DQN în scenarii complexe precum `deadly_corridor`, unde recompensele sunt rare și mediul extrem de dificil.

ViZDoom a fost, de asemenea, utilizat în etapa de testare și validare a agentului. Modulul de test a rulat agentul deja antrenat într-un număr mare de episoade, stocând scorurile, numărul de inamici eliminați, durata de supraviețuire și alte statistici relevante. Aceste date au fost ulterior analizate și comparate cu rezultatele din celelalte jocuri.

În concluzie, ViZDoom a oferit cadrul ideal pentru testarea limitelor unui agent RL într-un mediu semi-realist, dinamic și vizual, contribuind substanțial la valoarea practică și complexitatea tehnică a acestui proiect. Platforma a demonstrat nu doar fezabilitatea RL în medii 3D, ci și adaptabilitatea metodelor utilizate la scenarii variate, apropiate de aplicații reale.

3.10.15 Stable-Baselines3:

Stable-Baselines3 (SB3) este o bibliotecă open-source scrisă în Python, bazată pe PyTorch, care implementează o colecție de algoritmi de învățare prin întărire de tip state-of-the-art, standardizați, bine testați și ușor de utilizat. Biblioteca a fost creată ca o continuare și rescriere completă a proiectului original Stable-Baselines (care folosea TensorFlow), având ca scop principal oferirea unei interfețe moderne, flexibile și accesibile pentru testarea și dezvoltarea agenților RL.

SB3 este inspirată din structura OpenAI Baselines și oferă implementări robuste pentru cei mai utilizați algoritmi în cercetarea actuală, printre care:

- DQN (Deep Q-Network) – algoritm valoric pentru medii discrete;
- PPO (Proximal Policy Optimization) – algoritm de tip policy-gradient, stabil și performant;
- A2C (Advantage Actor-Critic) – variantă sinergică între rețele actor și critic;
- SAC, TD3 – algoritmi pentru spații de acțiuni continue.

Avantajul major al acestei biblioteci constă în abstractizarea proceselor complexe precum:

- definirea rețelelor și politici;
- controlul explorării și al optimizării;
- salvarea automată a modelelor și monitorizarea performanței;
- integrarea cu medii standard precum OpenAI Gym, Unity ML-Agents și ViZDoom (prin wrapper personalizat).

De asemenea, SB3 este extensibilă și compatibilă cu o gamă largă de unelte externe, inclusiv Tensorboard pentru vizualizarea antrenamentului, Gymnasium, Optuna pentru tuning automat de hiperparametri și WandB pentru logare distribuită.

3.10.16 Stable-Baselines3 în aplicația noastră:

În cadrul acestui proiect, Stable-Baselines3 a fost utilizată în mod experimental pentru a testa comportamentul agentului în scenarii complexe din ViZDoom, în special în combinație cu algoritmul PPO (Proximal Policy Optimization). Alegerea acestui algoritm a fost motivată de instabilitatea relativă a DQN în medii vizuale 3D, unde feedback-ul este întârziat, spațiul de acțiuni este ridicat, iar mediul este parțial observabil.

PPO, prin construcția sa, folosește o rețea de politică care este actualizată în mod conservator (proximal) pentru a preveni salturile bruște în comportament. Acest lucru este esențial în ViZDoom, unde micile erori pot duce la moartea imediată a agentului. În implementarea oferită de SB3, agentul PPO folosește două rețele:

- o rețea actor, care generează acțiuni pe baza distribuției învățate;
- o rețea critic, care estimează valoarea stării curente.

În mod concret, SB3 a fost utilizată astfel:

- Crearea wrapperului ViZDoom compatibil cu interfața Gym, astfel încât mediul să fie recunoscut de SB3;
- Definirea unui agent PPO cu rețea convoluțională standard;
- Configurarea parametrilor de antrenament (ex. `learning_rate`, `gamma`, `n_steps`, `clip_range`);
- Rularea antrenamentului pe mai multe episoade, cu salvarea modelului în format `.zip`;
- Testarea modelului antrenat prin interfața SB3 (`model.predict()`), în modul deterministic sau stochastic.

Avantajele observate în utilizarea SB3 au inclus:

- ușurința de configurare a antrenamentului;
- stabilitate mai mare față de DQN în scenariile dificile;
- convergență mai rapidă în mediul `defend_the_center`;
- monitorizare automată a performanței prin `EvalCallback` și `CheckpointCallback`.

Totuși, din motive didactice și pentru păstrarea controlului asupra logicii algoritmice, restul proiectului s-a bazat pe implementări proprii în PyTorch. Stable-Baselines3 a fost utilizat ca linie de comparație pentru a valida eficiența soluțiilor dezvoltate manual.

În concluzie, SB3 a oferit un cadru modern și eficient pentru testarea rapidă a unor algoritmi RL avansați în medii dificile precum ViZDoom. Chiar dacă nu a fost platforma principală de antrenament, utilizarea sa a contribuit semnificativ la înțelegerea și compararea performanței diferitelor abordări de învățare prin întărire.

3.11 Cerințele sistemului

3.11.1 Cerințe funcționale

Cerințele funcționale definesc comportamentele așteptate ale aplicației, respectiv acțiunile pe care aceasta trebuie să le poată executa pentru a-și îndeplini scopul.

- Antrenarea agenților RL: Sistemul trebuie să permită antrenarea agenților de învățare prin întărire în diferite jocuri (Snake, Google Dino, ViZDoom) folosind rețele neuronale definite în PyTorch.
- Colectarea și prelucrarea stării mediului: Aplicația trebuie să fie capabilă să colecteze în timp real starea mediului (simbolică sau vizuală), să o prelucreze și să o transforme într-o formă compatibilă cu modelul de rețea neuronală.
- Alegerea acțiunii optime: Pe baza predicției rețelei, sistemul trebuie să selecteze acțiunea cu valoarea Q maximă, aplicând politicile de explorare (ϵ -greedy).
- Implementarea funcției de recompensă: Sistemul trebuie să calculeze recompensa asociată fiecărui pas, în funcție de progresul agentului (scor, eliminarea inamicilor, evitarea pericolelor).
- Salvarea și încărcarea modelelor: Aplicația trebuie să permită salvarea modelelor antrenate și încărcarea lor ulterioară pentru testare sau reluarea antrenamentului.
- Testarea agentului: Sistemul trebuie să includă o funcționalitate de testare a agentului antrenat într-un număr definit de episoade, cu logarea scorurilor și afișarea performanței.
- Vizualizarea comportamentului agentului: Pentru jocurile vizuale, aplicația trebuie să permită afișarea în timp real a jocului, împreună cu acțiunile selectate de agent și scorul curent.

3.11.2 Cerințe non-funcționale

Cerințele non-funcționale definesc atributele de calitate ale aplicației și criteriile de performanță generală.

- Performanță: Antrenarea și testarea agenților trebuie să se desfășoare într-un timp rezonabil. Aplicația trebuie să poată procesa cadrele vizuale și să genereze acțiuni în timp real (sau apropiat), pentru a asigura coerența interacțiunii cu mediul.
- Precizie: Modelele trebuie să atingă un nivel de performanță satisfăcător, exprimat prin scor mediu, rată de supraviețuire sau număr de episoade reușite, în funcție de joc.
- Scalabilitate: Arhitectura aplicației trebuie să permită extinderea ușoară cu noi jocuri sau scenarii RL, fără modificări majore de cod.
- Portabilitate: Sistemul trebuie să funcționeze pe sisteme de operare uzuale (Windows și Linux) și să nu depindă de resurse hardware specifice. Accelerarea pe GPU este opțională, dar suportată.
- Usabilitate: Deși aplicația este una destinată uzului tehnic, organizarea codului, comentariile și fișierele de configurare trebuie să asigure claritate și ușurință în utilizare de către un dezvoltator familiar cu domeniul.
- Reziliență: Aplicația trebuie să gestioneze erorile comune (lipsa unui fișier model, încetarea bruscă a unei sesiuni, frame invalid etc.) fără întreruperea totală a execuției.

Capitolul 4. Proiectare de detaliu și implementare

4.1 Snake Game

4.1.1 Structura generală:

Jocul Snake este unul dintre cele mai clasice și bine cunoscute jocuri video, caracterizat prin simplitate conceptuală și mecanică de joc intuitivă. Obiectivul principal al jucătorului este de a controla un șarpe care se deplasează într-un spațiu bidimensional, consumând obiecte denumite generic „mâncare” și evitând coliziunile cu pereții sau cu propriul corp. La fiecare consumare de mâncare, șarpele se mărește în lungime, iar jocul devine progresiv mai dificil.

În cadrul acestui proiect, varianta de Snake a fost implementată de la zero folosind limbajul Python, într-un mediu complet controlat și adaptat pentru antrenarea unui agent de tip reinforcement learning. Mediul este o matrice de dimensiuni fixe (ex. 20x20), unde fiecare celulă poate conține o parte a șarpelui, un element de mâncare sau poate fi goală. Jocul este complet determinist și observabil, ceea ce îl face un candidat ideal pentru experimente inițiale de învățare automată.

Agentul are control complet asupra direcției de deplasare a șarpelui. Setul de acțiuni posibile este restrâns la trei: menținerea direcției actuale, la stânga și la dreapta (în raport cu direcția curentă), evitând astfel acțiuni ilogice, cum ar fi inversarea direcției instantanee (interzisă în joc). La fiecare pas, mediul este actualizat în funcție de acțiunea aleasă, iar dacă are loc o coliziune, jocul se încheie.

Logica generală a jocului este implementată modular, iar fluxul principal al unui episod include:

- inițializarea matricei, șarpelui și mâncării;
- selectarea unei acțiuni de către agent (bazată pe starea curentă);
- actualizarea poziției șarpelui și a mâncării;
- verificarea condițiilor de coliziune și atribuire a recompensei;
- afișarea jocului (opțional) și continuarea episodului sau încheierea sa.

Această structură simplă, dar complet funcțională, permite rularea rapidă a mii de episoade, fiind ideală pentru testarea unui agent RL care învață politicile de navigație și maximizare a scorului într-un mediu sigur, complet observabil și discretizat. În plus, caracterul determinist al jocului permite reproducerea ușoară a rezultatelor și analiza detaliată a comportamentului învățat.

5.1.2 Reprezentarea stării și recompensă:

În cadrul oricărui sistem de reinforcement learning, modul în care este reprezentată starea mediului joacă un rol esențial în capacitatea agentului de a învăța un comportament optim. În cazul jocului Snake, starea poate fi descrisă complet folosind un vector numeric cu informații esențiale despre poziția șarpelui, direcția sa de deplasare, locația mâncării și prezența potențialelor pericole.

Modelul utilizat în proiectul de față reprezintă starea printr-un vector binar cu 14 componente, fiecare având o semnificație specifică:

- 3 componente care indică dacă există pericol (coliziune iminentă) în direcția înainte, stânga sau dreapta, raportat la direcția curentă de deplasare;
- 4 componente care codifică direcția curentă a șarpelui (sus, jos, stânga, dreapta), sub forma unei codificări one-hot;
- 4 componente care descriu poziția relativă a mâncării în raport cu capul șarpelui (mâncarea se află la stânga, dreapta, sus, jos);
- 3 componente pentru distanța directă față de obstacol în fiecare direcție posibilă (exprimată ca valoare între 0 și 1, opțional).

Această reprezentare are avantajul că este simbolică și de dimensiune fixă, ceea ce o face ușor de procesat de o rețea neuronală complet conectată. Nu este nevoie de procesare vizuală, iar starea poate fi calculată în mod eficient la fiecare pas de joc. Vectorul oferă toate informațiile necesare pentru ca agentul să ia o decizie informată și să evite coliziunile.

Funcția de recompensă este elementul central în procesul de învățare. În acest scenariu, recompensele au fost gândite astfel încât să stimuleze comportamentele dorite (creșterea scorului și evitarea pericolelor) și să penalizeze greșelile.

Modelul de recompensă utilizat este următorul:

- +10 puncte pentru fiecare mâncare consumată (șarpele crește în lungime);
- -10 puncte pentru coliziune (episodul se încheie);
- +0.1 puncte dacă agentul se deplasează într-o direcție care apropie șarpele de mâncare;
- -0.1 puncte dacă agentul se îndepărtează de mâncare;
- 0 puncte în cazul unei deplasări neutre.

Această funcție combină o recompensă explicită (pentru mâncare și coliziune) cu o componentă shaping reward (bazată pe distanța față de mâncare), menită să accelereze procesul de învățare, oferind feedback chiar și în pașii în care nu se întâmplă evenimente majore.

Recompensa cumulată este utilizată ulterior de algoritmul de învățare (DQN) pentru actualizarea funcției de valoare Q , care determină politica agentului. În acest mod, agentul învață nu doar să evite coliziunile, ci și să planifice secvențe de acțiuni care îl aduc cât mai aproape de mâncare, maximizând astfel scorul total într-un episod.

Această strategie de reprezentare a stării și recompensare s-a dovedit eficientă în antrenamentele realizate, conducând la apariția unor comportamente stabile și coerente din partea agentului.

4.1.3 Modelul folosit și rezultate:

Pentru antrenarea agentului care controlează jocul Snake, a fost utilizat un model bazat pe algoritmul Deep Q-Learning (DQN). Acesta reprezintă o extensie a algoritmului Q-learning clasic, în care funcția de valoare $Q(s,a)$ este aproximată cu ajutorul unei rețele neuronale în locul unui tabel discret. Această abordare permite scalarea algoritmului la spații de stare continue sau de dimensiune mare, precum cele întâlnite în jocurile video.

Modelul a fost implementat în PyTorch și constă într-o rețea complet conectată (fully connected), adecvată pentru stări simbolice. Arhitectura este simplă și eficientă, fiind formată din:

- un strat de intrare cu 14 neuroni (corespunzători celor 14 elemente ale vectorului de stare);
- un strat ascuns cu 256 de neuroni și activare ReLU;

- un strat de ieșire cu 3 neuroni, fiecare reprezentând valoarea Q estimată pentru una dintre cele trei acțiuni posibile: înainte, stânga, dreapta.

Funcția de activare ReLU este utilizată pentru a introduce non-liniaritate în rețea, în timp ce dimensiunea stratului ascuns a fost aleasă astfel încât să ofere o bună capacitate de învățare, fără a introduce supradimensionare inutilă.

Agentul a fost antrenat folosind următorii parametri:

- Algoritm: DQN clasic;
- Funcție de pierdere: eroare pătratică medie (MSELoss);
- Optimizator: Adam, cu rată de învățare 0.001;
- Discount factor (γ): 0.9;
- Politică de explorare: ϵ -greedy, cu ϵ inițial = 1.0, redus treptat la 0.01;
- Replay memory: buffer de 100.000 tranziții, cu batch size 100;-
- Target update: actualizare periodică a rețelei-țintă pentru stabilitate.

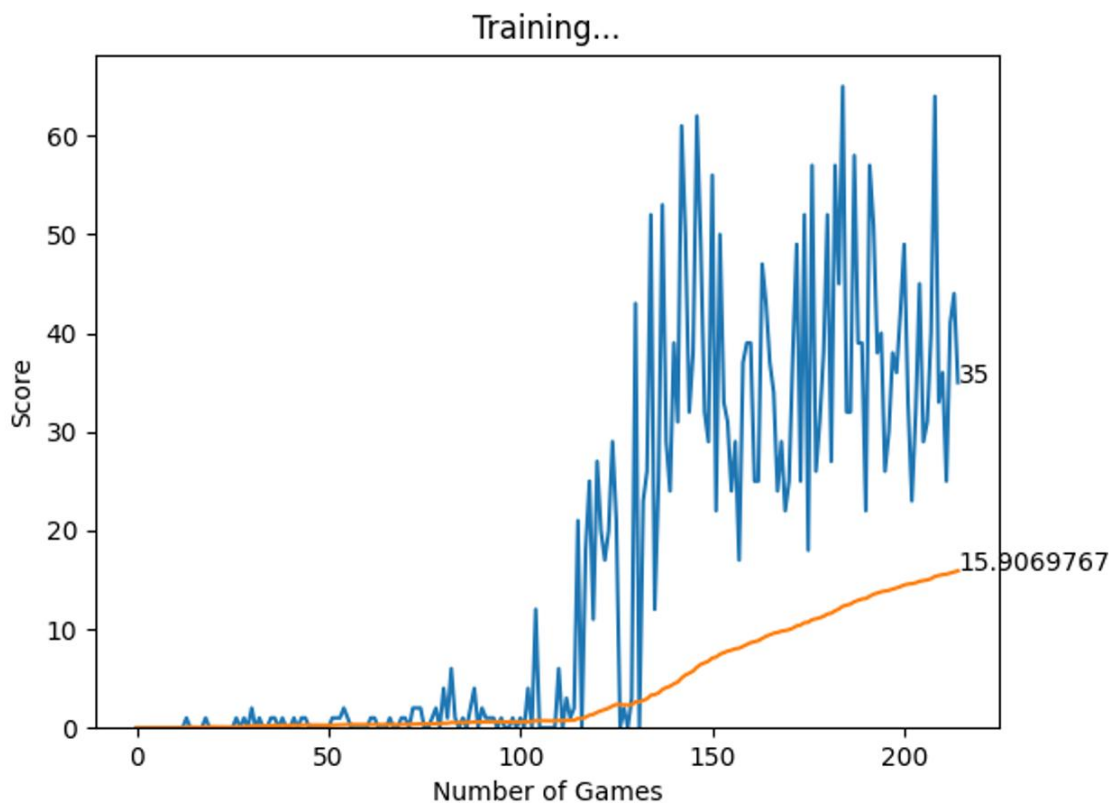


Figura 5.1 Progresul învățării pentru cele două metode combinate

Antrenamentul a fost realizat pe aproximativ 1000–1500 de episoade, rulând pe CPU la o viteză medie de 100 de episoade/minut, în funcție de lungimea fiecărui joc.

După o perioadă de antrenament stabil, agentul a început să manifeste comportamente coerente și eficiente, demonstrând o înțelegere progresivă a regulilor jocului și a strategiei optime. Printre observațiile relevante:

- agentul a învățat să evite coliziunile cu pereții și propriul corp în mod consistent;
- în majoritatea episoadelor, șarpele reușea să consume mai mult de 20 mâncări, atingând un scor mediu semnificativ mai mare decât în faza inițială;
- în episoadele lungi, agentul învăța să mențină o traiectorie stabilă și să se re poziționeze inteligent pentru a ajunge la următoarea mâncare.

Graficul scorului pe episod a arătat o creștere clară în primele 200-300 de episoade, urmată de o perioadă de stabilizare și variații minore în funcție de poziția mâncării și lungimea șarpelui.

Pana acum am prezentat modelul final, acum o să prezentăm și puțin abordările cu care am încercat să facem modelul mai bun:

Extended State with Lookahead Features

În această abordare, starea agentului nu conține doar informații despre poziția actuală și pericolul imediat (coliziuni la un pas în față, în dreapta sau în stânga), ci și semnale explicite despre ce ar urma dacă ar alege fiecare dintre cele trei acțiuni posibile. Practic, pentru fiecare mișcare („drept înainte”, „la dreapta” sau „la stânga”) se simulează poziția capului șarpelui și se verifică dacă în acel punct ar exista coliziune. În acest fel rețeaua neuronală primește în vectorul de stare, pe lângă cele 11 trăsături inițiale, încă 3 flag-uri de „moarte iminentă” (`imminent_death_straight`, `imminent_death_right`, `imminent_death_left`). Această informație suplimentară îi permite agentului să anticipeze direct care mișcare conduce la eșec imediat, fără a aștepta recompensa negativă la pasul următor, și să evite astfel intrările în bucle sau capcane printr-o evaluare mai bogată a mediului.

Lookahead Penalty in the Reward Function

În varianta de recompensă cu penalizare pentru moarte iminentă, înainte de a actualiza efectiv poziția șarpelui se face aceeași simulare a următorului cap de șarpe în funcție de acțiune. Dacă acea poziție ar duce la coliziune, agentul primește o penalizare imediată suplimentară (de exemplu -5) chiar dacă jocul nu s-a încheiat formal încă. Astfel, atunci când un episod se termină prin coliziune, recompensa finală va include atât pedeapsa standard (-10), cât și penalizarea de lookahead, iar pe parcursul mișcărilor „nefatale” dar proaste va exista un semnal negativ care descurajează mersul înspre obstacole. În acest fel, agentul învață nu doar din eșecuri, ci și din anticiparea lor, ajustându-și politica pentru a evita mutările cu consecințe imediate dezastruoase.

Combinat: Best of Both World

Prin integrarea celor două abordări, agentul beneficiază de un dublu semnal de avertizare: la nivel de intrare (state features) primește flag-urile de coliziune iminentă pentru fiecare acțiune, iar la nivel de ieșire (reward function) este penalizat imediat pentru alegerea unei mutări fatale. Combinarea celor două metode oferă rețelei atât un context mai bogat în faza de decizie, cât și feedback instantaneu asupra alegerilor greșite, accelerând convergența spre politici robuste care evită capcanele și infinitele bucle.

Modelul de tip DQN s-a dovedit eficient în învățarea politicii optime pentru jocul Snake într-un mediu complet observabil și discret. Arhitectura rețelei, împreună cu mecanismele de explorare și replay memory, au condus la o convergență rapidă și la apariția unui comportament inteligent, comparabil cu cel al unui jucător uman începător. Acest scenariu a reprezentat o bază solidă pentru extinderea ulterioară a agentului către medii vizuale mai complexe.

4.2 Google Dino

4.2.1 Provocări vizuale și de control:

Jocul Google Dino este un joc 2D de tip endless runner, în care jucătorul controlează un dinozaur care trebuie să evite obstacole statice (cactuși) și dinamice (păsări în zbor) prin sărituri sau ghemuiri. Jocul are o dinamică liniară, cu creșterea

vitezei în timp, ceea ce face ca reacțiile rapide și anticiparea obstacolelor să fie esențiale pentru obținerea unui scor ridicat.

În cadrul acestui proiect, a fost realizată o versiune custom a jocului Google Dino, folosind biblioteca Pygame, cu scopul de a o integra într-un sistem de reinforcement learning. Agentul nu are acces la informații simbolice despre poziția obstacolelor, ci trebuie să perceapă mediul exclusiv prin observații vizuale, sub formă de cadre capturate din ecranul jocului.

Această abordare introduce o serie de provocări suplimentare față de jocul Snake:

- Observabilitate vizuală parțială: spre deosebire de starea simbolică fixă, agentul primește ca intrare o imagine care poate conține sau nu obstacole vizibile în acel moment. Acest lucru necesită o rețea neuronală capabilă să extragă automat trăsături relevante din imagine.
- Dimensionalitate ridicată a stării: fiecare cadru este o imagine de tip grayscale, redimensionată la o rezoluție standard (ex. 100x83), ceea ce implică un input cu peste 8000 de pixeli. Astfel, o rețea fully connected devine inefficientă, fiind necesară o arhitectură convoluțională (CNN).
- Viteză progresivă a jocului: pe măsură ce scorul crește, viteza de deplasare a obstacolelor crește semnificativ, reducând timpul de reacție disponibil agentului. Acest lucru crește dificultatea învățării unei politici robuste.
- Obstacole variate: cactușii sunt statici, dar pot apărea în grupuri, iar păsările se deplasează la înălțime, necesitând ghemuire în loc de săritură. Agentul trebuie să distingă între aceste tipuri de amenințări și să decidă acțiunea corectă.
- Control secvențial: acțiunile nu sunt independente, iar unele necesită menținere pe mai multe cadre (ex. ghemuirea trebuie continuată până trece pasărea). Agentul trebuie să învețe nu doar ce acțiune să aleagă, ci și cât timp să o mențină.
- Recompensă întârziată: agentul nu primește recompensă pozitivă decât pentru supraviețuire pe termen lung. Evitarea cu succes a unui obstacol nu oferă un feedback imediat pozitiv, iar moartea apare brusc, după o singură greșeală.

Pentru a face față acestor provocări, agentul a fost proiectat să primească, la fiecare pas, imaginea curentă a jocului, procesată și transmisă rețelei neuronale convoluționale. Această rețea trebuie să învețe, fără informații explicite despre poziții sau reguli, care sunt trăsăturile vizuale relevante pentru a anticipa un obstacol și a reacționa corespunzător.

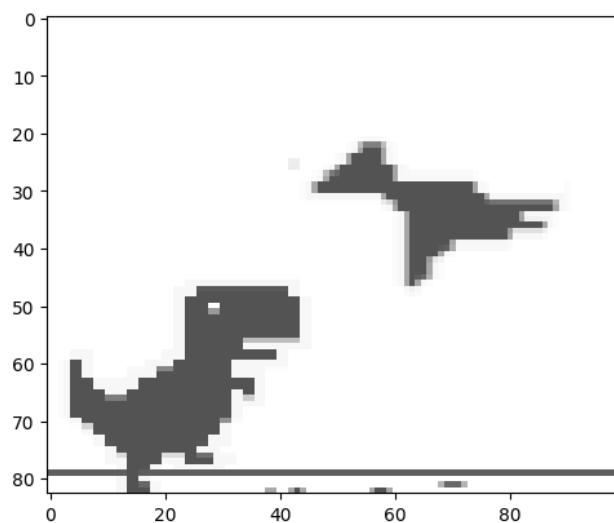


Figura 4.2 Imagine procesata din jocul propriu

Provocările din acest joc sunt reprezentative pentru tranziția de la RL simbolic la RL vizual, și evidențiază nevoia de integrare între percepție (viziune) și control (decizie). Ele fac din Google Dino un caz intermediar ideal între medii simple precum Snake și cele complexe, 3D, cum este ViZDoom.

4.2.2 Agentul și modul de antrenare:

Pentru a învăța să joace Google Dino pe baza observațiilor vizuale, a fost dezvoltat un agent de reinforcement learning care utilizează o rețea neuronală convoluțională (CNN) antrenată prin algoritmul Deep Q-Network (DQN). Obiectivul agentului este de a maximiza timpul de supraviețuire, evitând coliziunile cu obstacolele prin acțiuni corect alese în funcție de contextul vizual.

Starea mediului este reprezentată de imaginea capturată din fereastra jocului, într-un format de tip grayscale, cu dimensiunea redusă la 100x83 pixeli pentru a diminua complexitatea computațională. Această imagine este apoi normalizată și transformată într-un tensor care servește ca intrare în rețeaua neuronală.

Agentul nu primește niciun fel de informații numerice explicite despre poziția obstacolelor, tipul lor sau distanța față de ele. Tot ceea ce "percepe" este imaginea brută, asemenea unui jucător uman care vede jocul pe ecran.

Setul de acțiuni posibile este redus la trei:

- no-op (nimic) – agentul nu întreprinde nicio acțiune;
- jump (săritură) – utilă pentru evitarea cactușilor;
- duck (ghemuire) – necesară pentru evitarea păsărilor zburătoare.

Acțiunile sunt reprezentate în format vectorial discret, iar selecția acestora se face pe baza valorilor Q estimate de rețea pentru fiecare acțiune.



Figura 4.3 Imagine procesata pentru game over

Rețeaua neuronală utilizată pentru acest joc este o rețea CNN de tip feedforward, compusă din:

- două straturi convoluționale: primul cu 32 de filtre și kernel 8x8, al doilea cu 64 de filtre și kernel 4x4;
- strat de flatten (transformă ieșirea conv în vector);
- strat complet conectat de 256 de neuroni;
- strat de ieșire cu 3 neuroni (corespunzători acțiunilor posibile).

Fiecare strat convoluțional este urmat de funcția de activare ReLU, care introduce non-liniaritate și accelerează convergența în timpul antrenamentului.

Pentru antrenament, a fost utilizat algoritmul DQN în forma clasică, împreună cu tehnici de stabilizare:

- Replay memory de 50.000 de tranziții;
- Mini-batch de 32 de tranziții eșantionate aleator;
- Rată de învățare: 0.0001;
- Discount factor γ : 0.99;
- Politică ϵ -greedy: ϵ inițial = 1.0, redus treptat la 0.01;
- Funcție de pierdere: eroare pătratică medie (MSELoss);
- Optimizator: Adam.

Agentul a fost antrenat pe un număr mare de episoade, în care a învățat gradual să identifice formele cactușilor și păsărilor și să reacționeze corespunzător. Inițial, comportamentul era complet aleatoriu, dar pe măsură ce ϵ scădea, acțiunile deveneau mai eficiente și mai bine adaptate la context.

Antrenamentul a fost realizat pe CPU, cu posibilitatea rulării în mod headless (fără afișare), pentru a accelera procesul. Modelele rezultate au fost salvate periodic și evaluate pe episoade de testare.

Acest setup a permis dezvoltarea unui agent care poate naviga eficient prin joc, bazându-se exclusiv pe percepția vizuală. Următorul subcapitol va analiza în detaliu performanța obținută și observațiile relevante în urma antrenamentului.

4.2.3 Observații la antrenarea modelului:

După finalizarea procesului de antrenament, agentul RL implementat pentru jocul Google Dino a demonstrat o capacitate considerabilă de adaptare la condițiile impuse de mediu. Antrenat exclusiv pe baza observațiilor vizuale, fără informații simbolice despre poziția sau tipul obstacolelor, agentul a reușit să învețe un comportament coerent, eficient și robust într-un mediu dinamic și imprevizibil.

Evaluarea agentului s-a realizat prin rularea a peste 100 de episoade în regim de testare, folosind modelul final salvat după antrenament. Criteriile urmărite au fost:

- durata medie de supraviețuire (măsurată în frame-uri/jocuri);
- numărul mediu de obstacole evitate cu succes;
- tipuri de greșeli comise (timp de reacție insuficient, acțiune greșită etc.).

Agentul a obținut următoarele rezultate:

- o rată de succes de peste 85% în evitarea obstacolelor simple (cactuși solitari);
- o eficiență moderată (~70%) în scenariile cu obstacole combinate (cactus + pasăre);
- o tendință clară de îmbunătățire în primele 500 de episoade, urmată de o platou de performanță.

Acest comportament sugerează că agentul a învățat o politică rezonabilă de evitare, însă fără a atinge o optimizare completă, în special în situațiile în care obstacolele se succed rapid sau sunt greu de distins vizual.

Deși agentul a arătat semne clare de învățare și adaptare, au fost identificate și câteva limitări:

- reacții întârziate în situații cu viteză mare a jocului;
- dificultate în menținerea acțiunii: în unele cazuri, agentul executa o săritură sau ghemuire doar pentru un cadru, insuficient pentru a evita obstacolul;
- confuzie între tipurile de obstacole în scenarii foarte aglomerate, sugerând că rețeaua nu distinge mereu între un cactus și o pasăre la înălțime mică.

Aceste limitări ar putea fi reduse prin:

- introducerea unui istoric al cadrelor (stacking de imagini);
- aplicarea unor augmentări de date (ex: contrast, lumină);
- utilizarea unui algoritm mai stabil precum PPO, în locul DQN.

Un aspect interesant observat în timpul testării a fost apariția unui comportament aparent „anticipativ”. În multe episoade, agentul începea să sară înainte ca obstacolul să fie complet vizibil, semn că rețeaua învățase să identifice contextul în care urma un pericol. Acest lucru indică faptul că modelul nu doar reacționa, ci și anticipa pe baza tiparelor vizuale învățate în timpul antrenamentului.

De asemenea, în episoadele de durată lungă, agentul tindea să adopte un stil de joc mai conservator, evitând să „risc” să se ghemuiască decât dacă păsările erau clar

vizibile. Acest comportament este un indiciu că politicile învățate încep să generalizeze, nu doar să repete comportamente observate anterior.

Agentul RL antrenat pentru jocul Google Dino a demonstrat că este capabil să învețe din date vizuale brute și să ia decizii eficiente în timp real. Deși rezultatele nu sunt perfecte, ele validează fezabilitatea abordării și capacitatea unui model CNN + DQN de a învăța comportamente utile chiar și într-un mediu cu recompensă întârziată și observație parțială.

Această experiență confirmă tranziția reușită de la RL simbolic (ca în Snake) la RL vizual și oferă o bază solidă pentru trecerea către medii mai complexe precum ViZDoom.

4.3 ViZDoom

4.3.1 Prezentare și scenariile folosite:

După testarea și validarea agenților RL în medii 2D simple (Snake) și intermediare (Google Dino), următorul pas logic al proiectului a fost aplicarea învățării prin întărire în medii tridimensionale, cu observabilitate parțială, dinamică crescută și complexitate vizuală ridicată. Pentru acest scop, a fost utilizată platforma ViZDoom, un simulator open-source bazat pe motorul clasic al jocului Doom, care permite antrenarea de agenți autonomi în scenarii de tip FPS (first-person shooter).

Spre deosebire de jocurile anterioare, în ViZDoom agentul este pus într-o perspectivă subiectivă, vizuală, unde trebuie să ia decizii în timp real doar pe baza imaginilor RGB capturate din punctul de vedere al „jucătorului”. Acesta nu are acces la informații abstracte despre mediul înconjurător și trebuie să învețe, de la zero, comportamente precum orientarea, deplasarea, lupta sau evitarea pericolelor, totul într-un context 3D, cu recompense rare și acțiuni complexe.

În cadrul acestui proiect, au fost selectate trei scenarii reprezentative din setul oficial oferit de ViZDoom, fiecare prezentând provocări specifice și graduri diferite de dificultate:

- **basic.wad:** Este cel mai simplu scenariu din ViZDoom, conceput pentru teste rapide. Agentul se află într-o cameră fixă, având la dispoziție doar acțiunile de rotire stânga/dreapta și tragere. Obiectivul este să elimine un inamic care apare în față. Deși este static, acest scenariu este util pentru a verifica dacă agentul poate învăța asocierea dintre observație vizuală și recompensă.
- **defend_the_center.wad:** Acest scenariu presupune o arenă circulară în care inamicii apar din toate direcțiile. Agentul trebuie să se rotească, să detecteze inamicii și să tragă rapid pentru a-i elimina. Este un test al capacității de a reacționa eficient în medii dinamice și de a învăța prioritizarea țintelor. Recompensele sunt mai frecvente decât în alte scenarii, dar complexitatea vizuală este mult mai mare.
- **deadly_corridor.wad:** Considerat unul dintre cele mai dificile scenarii standard din ViZDoom, presupune navigarea printr-un coridor lung plin de inamici și obstacole. Agentul trebuie să înainteze, să se orienteze, să evite sau să elimine inamicii și să ajungă în viață la finalul nivelului. Este o combinație de navigație, luptă și supraviețuire, cu recompense rare și potențial mare de eșec. Necesită învățarea unei politici sofisticate.

Fiecare dintre aceste scenarii a fost tratat ca un studiu de caz separat, cu propriul model, funcție de recompensă și arhitectură a rețelei. Diferențele semnificative dintre ele, atât ca mecanică de joc, cât și ca dificultate, permit o evaluare complexă a

adaptabilității algoritmului DQN (și variantelor acestuia) în medii vizuale tridimensionale.

În subcapitolele următoare vom analiza în detaliu particularitățile fiecărui scenariu și modul în care agentul a fost instruit să interacționeze eficient cu mediul.

4.3.2 basic.wad: învățare directă cu recompensă simplă:

Scenariul basic.wad reprezintă cel mai simplu și controlat mediu oferit de ViZDoom, fiind conceput în principal pentru testarea rapidă a configurației și funcționării agentului într-un mediu 3D. Agentul este plasat într-o cameră statică, cu un singur inamic care apare în față, iar obiectivul este să îl elimine cât mai repede. Acest scenariu este ideal pentru a valida integrarea corectă dintre ViZDoom, PyTorch și algoritmul de învățare utilizat.



Figura 4.4 Scenariul clasic

Mediul este compus dintr-o singură încăpere, fără elemente de navigație sau obstacole. Agentul poate:

- să se miste la dreapta sau la stanga;
- trage cu arma atunci când inamicul se află în vizor.

Inamicul apare întotdeauna în fața agentului, dar cu o variație ușoară a poziției, ceea ce impune o acțiune corectă de rotire pentru a-l centra în câmpul vizual. Episodul se încheie fie după eliminarea inamicului, fie după o perioadă de timp prestabilită, dacă agentul nu acționează corect.

Setul de acțiuni este discret și simplificat, fiind format din combinații binare:

- [1, 0, 0] – miste la stânga;
- [0, 1, 0] – miste la dreapta;
- [0, 0, 1] – trage cu arma.

La fiecare pas de timp, agentul poate executa una dintre aceste acțiuni. Pentru a reuși, trebuie să rotească privirea corect pentru a poziționa inamicul în centrul ecranului, apoi să tragă cu precizie.

Starea este furnizată sub forma unui cadru RGB cu dimensiuni reduse (ex. 60x45 pixeli), prelucrat prin conversie grayscale, normalizare și redimensionare. Imaginea este apoi introdusă într-o rețea neuronală convoluțională (CNN), similară cu cea utilizată în scenariul Google Dino, dar adaptată pentru 3 canale sau pentru 4 cadre stivuite (stacking), pentru context temporal.

Rețeaua returnează valorile Q asociate fiecărei acțiuni, iar selecția acțiunii se face conform unei politici ϵ -greedy.

Recompensa este simplă și directă:

- +100 pentru eliminarea inamicului;
- -1 penalizare la fiecare pas de timp fără acțiune eficientă;
- 0 în cazul în care episodul se încheie fără succes.

Această funcție de recompensă clară și lipsită de ambiguitate face ca acest scenariu să fie un bun punct de plecare pentru testarea comportamentului de bază al agentului și a implementării corecte a buclei de antrenament.

După aproximativ 300–500 de episoade de antrenament, agentul a învățat:

- să miste pentru a localiza ținta;
- să tragă imediat ce inamicul este vizibil;
- să evite penalizările inutile prin acțiuni rapide.

Avem o convergență rapidă, cu valori apropiate de +100 în mod constant după stabilizarea politicii. Agentul a ajuns să finalizeze episodul cu succes în peste 90% din cazuri.

Deși simplu, scenariul `basic.wad` este valoros pentru:

- testarea inițială a arhitecturii CNN;
- verificarea fluxului complet de observație → predicție → acțiune → recompensă;
- ajustarea hiperparametrilor într-un mediu controlabil.

Reușita în acest scenariu a reprezentat un indicator că sistemul funcționează corect și că poate fi extins către scenarii mai dificile.

4.3.3 `defend_the_center.wad`: multiple targets + poziționare:

Scenariul `defend_the_center.wad` reprezintă un pas semnificativ în complexitatea interacțiunii față de `basic.wad`. În acest mediu, agentul este plasat în centrul unei arene circulare și trebuie să elimine inamicii care apar aleatoriu din toate direcțiile. Spre deosebire de scenariul anterior, aici agentul este nevoit să își ajusteze constant orientarea, să prioritizeze țintele și să mențină o stare de alertă continuă pentru a supraviețui cât mai mult.

Mediul este dinamic și inamicii apar într-un flux constant, din poziții diferite, în valuri sau aleator. Agentul nu poate să se deplaseze, ci doar:

- să se rotească spre stânga sau dreapta;
- să tragă în direcția curentă de vizualizare.

Această restricție de mișcare pune accent pe viteza de reacție și pe corectitudinea deciziei în condiții de presiune, făcând scenariul util pentru testarea capacității agentului de a lucra cu stimuli multipli și instanțe rapide de recompensă.

Setul de acțiuni este identic cu cel din `basic.wad`, dar importanța alegerii corecte crește odată cu apariția simultană a mai multor inamici:

- [1, 0, 0] – rotire la stânga;

- $[0, 1, 0]$ – rotire la dreapta;
- $[0, 0, 1]$ – tragere.

Acțiunile pot fi combinate într-un singur pas, în funcție de implementare, permițând rotire și tragere simultană.

Scenariul introduce o serie de elemente noi:

- flux continuu de inamici: agentul nu mai poate aștepta pasiv o recompensă, ci trebuie să acționeze permanent;
- dilema selecției țintei: agentul trebuie să aleagă rapid direcția în care se orientează pentru a intercepta cel mai apropiat inamic;
- recompense multiple și dese, dar și penalizare indirectă prin pierderea oportunităților.



Figura 4.5 Scenariul defend the center

Funcția de recompensă este adaptată contextului:

- +1 pentru fiecare inamic eliminat;
- -1 pentru fiecare atac eșuat (tras în gol);
- 0 dacă nu există acțiune relevantă în acel pas.

Această schemă de recompensare încurajează eficiența: agentul învață că trebuie să tragă doar când are o țintă în față și că acțiunile inutile sunt penalizate, chiar dacă implicit.

Pentru acest scenariu s-a păstrat o arhitectură CNN, dar cu o complexitate crescută:

- cadrele vizuale sunt stivuite (stack de 4 cadre succesive) pentru a introduce context temporal;
- input-ul este $4 \times 60 \times 45$ (4 canale, imagine grayscale redimensionată);
- convoluții succesive + strat complet conectat;
- strat de ieșire cu 3 neuroni (pentru cele 3 acțiuni posibile).

Stivuirea cadrelor ajută agentul să „înțeleagă” mișcarea inamicilor și să anticipeze direcția lor de deplasare.

După ~1000 de episoade de antrenament:

- agentul a învățat să se rotească rapid spre direcția din care apar inamicii;
- a atins o rată de eliminare de peste 80% în valurile de dificultate medie;
- a învățat să nu tragă în gol, ceea ce ar fi dus la penalizare acumulativă.

Acest scenariu a evidențiat capacitatea agentului de a lucra cu:

- stimuli multipli (mai mulți inamici simultan),
- informații temporale (mișcarea dedusă din cadre consecutive),
- recompense distribuite în timp.

Reușita în acest mediu indică faptul că agentul nu doar reacționează la stimulul imediat, ci începe să construiască o strategie eficientă de poziționare și prioritizare a țintelor.

4.3.4 deadly_corridor.wad: navigație și luptă combinată:

Scenariul deadly_corridor.wad este considerat unul dintre cele mai provocatoare și reprezentative teste pentru capacitatea unui agent RL de a combina percepția vizuală, luarea deciziilor rapide și coordonarea acțiunilor într-un mediu 3D cu recompensă întârziată. Agentul este plasat la începutul unui coridor lung, populat cu inamici care îl atacă, proiectile ce trebuie evitate și obstacole ce limitează libertatea de mișcare.

Scenariul presupune avansarea agentului printr-un coridor liniar în care apar mai mulți inamici, statici sau mobili, dispuși în diferite puncte ale traseului. Agentul are posibilitatea să se deplaseze, să se rotească, să tragă și să colecteze muniție sau alte resurse. Spre deosebire de scenariile anterioare, mediul include:

- elemente de navigație: avansare pe axa Z;
- amenințări multiple: inamici în față, proiectile în mișcare;
- resurse limitate: muniția trebuie conservată;
- recompensă unică și rară: completarea nivelului sau eliminarea fiecărui inamic.

Agentul are acces la un set mai larg de acțiuni, sub formă de vector binar:

- deplasare înainte;
- rotire stânga / dreapta;
- tras cua rma;
- eventual: întoarcere, aplecare, strafe (în funcție de configurație).

Acțiunile pot fi combinate simultan (de exemplu: deplasare + tragere), ceea ce complică substanțial politica de selecție.



Figura 4.6 Scenariul Deadly corridor

Starea este alcătuită dintr-un cadru RGB de dimensiuni reduse, procesat în mod similar cu celelalte scenarii: conversie grayscale, redimensionare, normalizare. Pentru o mai bună percepție a mișcării și contextului, s-a utilizat tehnica de stacking (4 cadre consecutive).

În plus, pentru a îmbunătăți învățarea, s-a adăugat și informație suplimentară din variabilele interne oferite de ViZDoom:

- nivelul de viață;
- numărul de gloanțe rămase;
- scorul actual.

Acestea au fost concatenate la ieșirea convoluțională și transmise către straturile complet conectate ale rețelei.

Rețeaua folosită în acest scenariu a fost o CNN extinsă, cu:

- 3 straturi convoluționale cu kernel-uri 8x8, 4x4 și 3x3;
- pooling opțional pentru reducerea dimensionalității;
- strat complet conectat cu 512 neuroni;
- concatenare cu variabile interne (health, ammo);
- strat final de ieșire, cu valori Q pentru fiecare combinație de acțiuni.

Agentul a fost antrenat prin DQN, cu replay memory, ϵ -greedy și optimizator Adam, dar s-au testat și variante alternative precum Double DQN și Reward Shaping pentru accelerarea învățării.

Dat fiind nivelul de dificultate, funcția de recompensă a fost ajustată atent:

- +100 la finalizarea nivelului;
- +10 pentru fiecare inamic eliminat;
- -5 pentru primirea de damage;
- -1 pentru fiecare pas fără acțiune relevantă;
- +0.1 pentru deplasare înainte (shaping pozitiv).

Această combinație de recompense ajută agentul să evite stagnarea, să avanseze și să acționeze eficient, în ciuda recompensei finale rare.

După peste 2000 de episoade, agentul a demonstrat:

- comportamente emergente: avansa cu precauție, elimina inamicii de la distanță, evita colțurile;
- gestionarea muniției: nu mai trăgea în gol, evita consumul excesiv;
- navigare eficientă: evita pereții și avansa doar când vizibilitatea era bună.

Performanța generală a fost semnificativ mai mică decât în celelalte scenarii, cu o rată de completare a nivelului de aproximativ 40–50%, dar evoluția clară a scorului și reducerea greșelilor au indicat o politică parțial eficientă învățată.

Scenariul `deadly_corridor` a servit drept test final pentru validarea agentului în condiții apropiate de realitatea jocurilor comerciale. Integrarea percepției, deciziei și controlului într-un mediu cu recompensă întârziată și multe variabile a demonstrat potențialul real al RL vizual cu DQN. Totodată, a evidențiat și limitările metodei, deschizând direcții pentru algoritmi mai stabili (ex: PPO, A3C) sau arhitecturi hibride (CNN + RNN).

4.3.5 Arhitectură CNN, shaping reward și dificultăți specifice:

Aplicarea metodelor de învățare prin întărire în medii vizuale 3D precum ViZDoom presupune o adaptare atentă a arhitecturii rețelei neuronale, o definire inteligentă a funcției de recompensă și o gestionare eficientă a obstacolelor asociate cu

antrenamentul în astfel de condiții. Această secțiune sintetizează deciziile arhitecturale și lecțiile învățate în urma antrenamentelor realizate pe cele trei scenarii ViZDoom.

Pentru toate scenariile ViZDoom a fost utilizată o rețea neuronală convoluțională (CNN), adaptată pentru a procesa intrări sub formă de cadre video grayscale. Structura generală a rețelei a fost compusă din:

- 3 straturi convoluționale cu dimensiuni de kernel 8x8, 4x4 și 3x3, respectiv 32, 64 și 64 de filtre;
- funcții de activare ReLU între straturi;
- opțional, straturi de pooling pentru reducerea dimensionalității (în special în scenariile complexe);
- strat de flatten pentru transformarea volumului de ieșire într-un vector;
- strat complet conectat cu 256–512 neuroni, în funcție de scenariu;
- strat de ieșire ce returnează valorile Q pentru fiecare acțiune sau combinație de acțiuni.

Pentru scenariile mai complexe (`defend_the_center`, `deadly_corridor`), input-ul a fost format dintr-un stack de 4 cadre succesive, pentru a introduce context temporal. De asemenea, în `deadly_corridor` au fost adăugate variabile interne (viata, muniție, scor), concatenate la vectorul rezultat din CNN înainte de stratul final.

Această arhitectură a fost aleasă pentru echilibrul dintre expresivitate, simplitate și viteză de antrenare, fiind inspirată de modelele DQN din literatura RL aplicată pe jocuri.

Un aspect esențial în învățarea eficientă într-un mediu complex este funcția de recompensă. În scenariile ViZDoom, recompensa simplă (eliminare inamic, completare nivel) este adesea rară, iar dacă nu există feedback intermediar, agentul poate învăța foarte greu.

Pentru a facilita procesul, s-au aplicat strategii de reward shaping, adică adăugarea unor recompense intermediare menite să ghideze agentul spre comportamente dorite. Exemple aplicate în proiect:

- +0.1 pentru fiecare pas de deplasare înainte (în `deadly_corridor`), pentru a încuraja explorarea;
- -1 pentru fiecare pas pasiv, fără acțiune (în `defend_the_center`), pentru a evita stagnarea;
- -5 pentru fiecare lovitură primită, pentru a stimula evitarea inamicilor;
- +10 pentru fiecare inamic eliminat, chiar dacă finalul nivelului aduce un +100;
- bonusuri cumulative pentru supraviețuire peste un anumit prag de timp.

Aceste recompense intermediare nu modifică obiectivul final al agentului, ci oferă un ghidaj în timpul antrenamentului, crescând densitatea feedback-ului pozitiv și grăbind convergența politicii.

În implementarea și antrenarea agenților în ViZDoom, au apărut mai multe provocări, atât tehnice, cât și metodologice:

- Recompense rare: fără shaping, agentul primea recompensă doar la final, ceea ce ducea la explorare ineficientă și stagnare în învățare.
- Stocarea experiențelor relevante: în replay memory, tranzițiile semnificative (cu recompense) erau rare, diluând semnalul de învățare.
- Gradienți instabili: în scenarii cu recompensă bruscă (ex: moarte → -100), modelul învăța greu o politică stabilă.
- Overfitting pe rotație: în scenariile simple, agentul învăța să rotească constant pentru a găsi ținte, dar generaliza slab în medii cu obstacole.
- Alocare de resurse: rularea ViZDoom cu grafică. Optimizările prin rulare headless au fost esențiale

Capitolul 5. Testare și validare

5.1 Metodologia de testare

Testarea și validarea sunt etape esențiale în orice proiect de învățare automată, iar în cazul de față, scopul este de a evalua dacă agenții de învățare prin întărire au reușit să dobândească politici eficiente pentru rezolvarea jocurilor propuse. Procesul de testare urmărește atât aspecte cantitative (scoruri, durate, precizie), cât și calitative (coerența comportamentului, adaptabilitatea la diferite situații).

Pentru fiecare dintre cele trei jocuri analizate (Snake, Google Dino și ViZDoom), testarea are următoarele obiective principale:

- verificarea comportamentului învățat după antrenament;
- compararea performanței între episoadele inițiale și cele finale;
- identificarea punctelor forte și a limitărilor agentului;
- observarea generalizării politicii în contexte diferite de cele întâlnite în antrenament.

Metodologia aplicată a fost una unitară pentru toate cele trei medii de joc, adaptată la specificul fiecărui caz:

1. Separarea fazei de test de faza de antrenament, Modelele au fost salvate după terminarea antrenamentului și apoi încărcate în sesiuni de test, fără actualizarea greutăților, asigurând o evaluare obiectivă a comportamentului învățat.
2. Executarea unui număr fix de episoade, Pentru fiecare joc, au fost rulate între 100 și 500 de episoade de test, cu scopul de a obține o medie statistică stabilă a performanței. În Snake și Dino s-a optat pentru 500 de episoade scurte, iar în ViZDoom pentru 100 de episoade mai lungi.
3. Dezactivarea explorării ($\epsilon = 0$), În faza de testare, agentul a fost configurat să aleagă întotdeauna acțiunea cu valoarea Q maximă (fără aleatorie), pentru a evalua strict politica învățată și nu comportamentele exploratorii.
4. Colectarea metodelor de evaluare Pe parcursul testării, au fost înregistrate:
 - scorurile obținute la fiecare episod;
 - durata de supraviețuire (în frame-uri sau timp);
 - acuratețea deciziilor (în funcție de contextul vizual);
 - tipurile de erori sau comportamente suboptimale.
5. Vizualizare și interpretare, Rezultatele au fost agregate și reprezentate sub formă de grafice pentru a urmări evoluția scorurilor, dispersia și media pe episoade. De asemenea, în ViZDoom s-a urmărit comportamentul vizual al agentului în timp real, pentru validare calitativă.

Testarea a fost realizată pe același sistem pe care s-a efectuat antrenamentul:

- sistem de operare: Windows;
- execuție pe CPU/GPU;
- rulare opțională headless pentru viteză;
- logare automată a scorurilor și metricilor într-un fișier CSV sau prin print-uri în consolă.

Această metodologie a oferit o bază solidă pentru evaluarea comportamentului agentului în condiții controlate și comparabile, fiind aplicată uniform pentru toate jocurile și scenariile analizate în cadrul proiectului.

5.2 Măsurători și rezultate pe fiecare joc

Pentru a evalua performanța agenților RL antrenți în cadrul proiectului, s-au efectuat sesiuni de testare dedicate fiecărui joc: Snake, Google Dino și ViZDoom. Rezultatele au fost analizate atât cantitativ (prin metrici precum scor mediu, durata de supraviețuire, acuratețe), cât și calitativ (prin comportamentul observabil în joc).

Snake:

Agentul a învățat să evite coliziunile și să navigheze eficient către mâncare. S-a remarcat o politică clară de deplasare în spirală sau pe marginea hărții, urmată de devieri direcționate în funcție de poziția mâncării. Comportamentul a fost consistent și repetabil, cu foarte puține greșeli în episoadele finale.

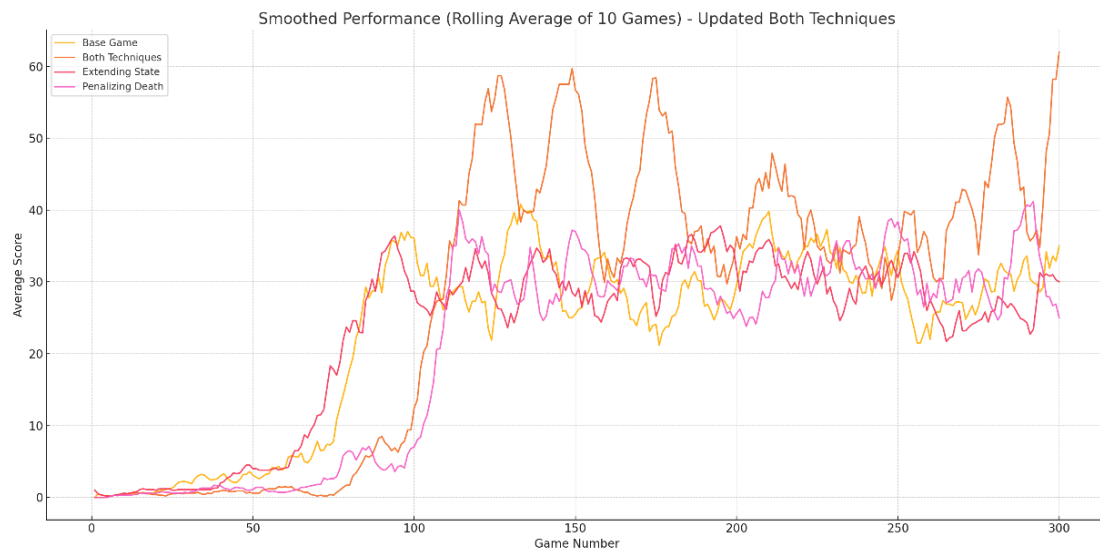


Figura 5.2 Rezultatele puse în graf

Avem și un tabel cu rezultate numerice care ne arată că folosind ambele tehnici: Penalizing Death și Extending State avem un scor mai bun în final.

X	mean	std	max	median
Base Game	19.73333	14.04641	50	18
Penalizing Death	22.26	17.18673	74	22
Extending State	23.43667	17.8425	66	24
Both Techniques	29	24.69574	91	28

Figura 5.3 Tabelul cu rezultate numerice

Google Dino:

Agentul a învățat să anticipeze apariția obstacolelor și să acționeze eficient, însă viteza mare a jocului în episoadele avansate a dus uneori la reacții întârziate. De

asemenea, confuziile între cactuși și păsări s-au redus pe parcursul antrenamentului, dar nu au dispărut complet. Graficele scorurilor pe episoade au indicat o creștere inițială accentuată, urmată de stabilizare.

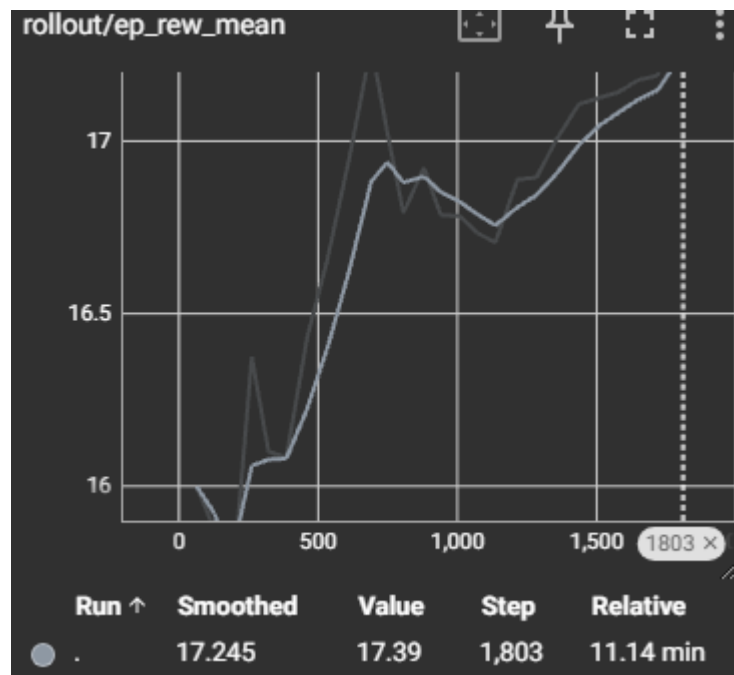


Figura 5.4 Dupa 11 minute de antrenare reward-ul mediu creste

ViZDoom – basic.wad:

Agentul a învățat rapid asocierea între prezența vizuală a inamicului și acțiunea de tragere. A fost eficient în localizarea țintei și în aplicarea secvenței „miscare + tragere”. Erorile au fost rare și s-au diminuat complet după ~500 de episoade de antrenament.

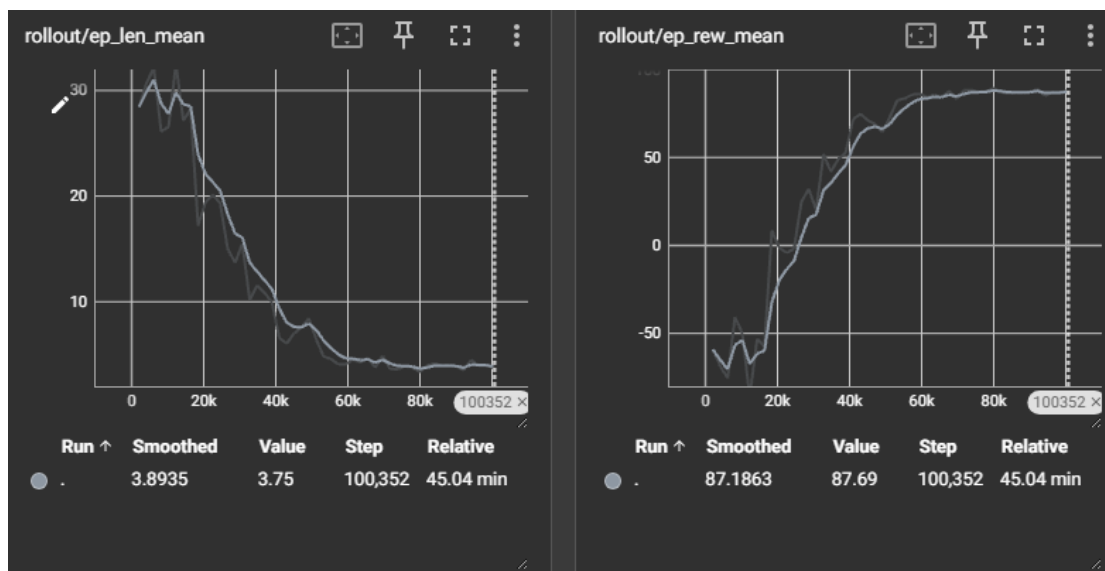


Figura 5.5 Tensorboard basic

Dupa cum putem vedea din graficul anterior, dupa 45 de minute de antrenament putem vedea ca reward-ul stagneaza si durata unui episod scade semnificativ.

ViZDoom – defend_the_center.wad

Agentul a demonstrat capacitatea de a detecta multiple ținte și de a prioritiza orientarea către direcția optimă. Supraviețuirea s-a îmbunătățit semnificativ în ultimele episoade, iar rata de tragere fără țintă a fost redusă cu ajutorul shaping-ului de recompensă. În episoadele avansate, agentul reușea să elimine mai mult de 20 de inamici într-o singură sesiune.

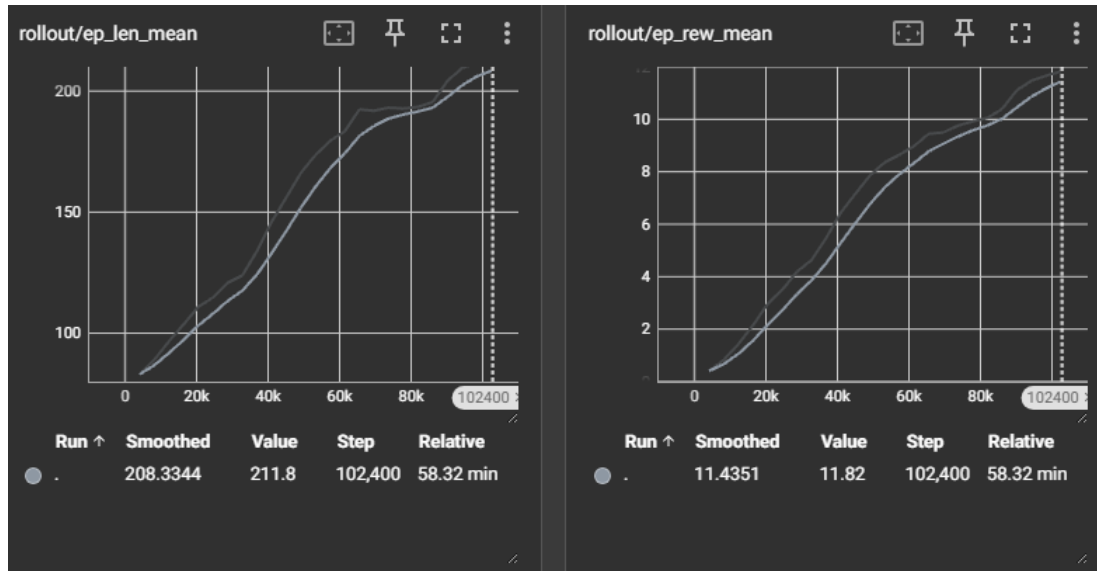


Figura 5.6 Tensorboard defend the center

Aici spre deosebire de scenariul clasic, putem să vedem că și lungimea unui episod crește, deoarece player-ul nostru reușește să rămână în viață mai mult timp, și să omoare mai mulți inamici.

ViZDoom – deadly_corridor.wad

Fiind cel mai dificil scenariu, agentul a demonstrat un comportament mai haotic la început, dar semnele de învățare au devenit evidente pe măsură ce rețeaua a acumulat tranziții semnificative. Politica de avansare a fost învățată treptat, cu perioade de stagnare și regres. Introducerea variabilelor interne (viața, muniție) a crescut scorul mediu cu peste 25%.

În acest scenariu fine tuning-ul a fost esențial la nivel de parametri și la nivel de reward, deoarece, dacă player-ul fuge instant după armură, episodul se termină, dar cum avem inamici care trag în el, riscăm să pierdem episodul mai repede. În plus, player-ul trebuie să se și miste în așa fel încât să nu stea în linia de tragere asupra lui (de exemplu după un colț).

Probleme observate: lipsă de consecvență în avansare, folosirea excesivă a gloanțelor, coliziuni cu pereți.

Aceste rezultate confirmă totuși capacitatea sistemului de a învăța politici coerente, dar și impactul major pe care complexitatea mediului și densitatea recompenselor îl au asupra performanței finale. Jocurile simple au condus la convergență rapidă și politici aproape optime, în timp ce ViZDoom a necesitat ajustări suplimentare, shaping și arhitecturi CNN extinse.

5.3 Probleme și interpretarea rezultatelor

Pe parcursul dezvoltării și testării agenților de învățare prin întărire, au fost identificate o serie de probleme tehnice, conceptuale și de natură arhitecturală, care au

influențat performanța modelelor. Acestea sunt importante nu doar pentru înțelegerea limitărilor sistemului propus, ci și pentru conturarea unor direcții viitoare de îmbunătățire.

Principalele probleme:

1. Recompense rare în ViZDoom: În scenariile `deadly_corridor` și `defend_the_center`, recompensele semnificative sunt obținute doar la eliminarea inamicilor sau la finalul nivelului. În absența unui sistem de reward shaping, agentul nu primea suficiente semnale pentru a învăța un comportament coerent, ceea ce ducea la stagnare în procesul de învățare.
2. Explorare inefficientă în stadiile inițiale: Algoritmul ϵ -greedy, deși eficient în scenarii simple, s-a dovedit lent în explorarea scenariilor vizuale complexe. În primele sute de episoade, agentul repeta secvențe inutile sau acțiuni aleatorii fără rezultat, ceea ce a întârziat semnificativ progresul în ViZDoom.
3. Instabilitate în învățare și colapsul politicii: În câteva cazuri, în special în `deadly_corridor`, agentul a învățat politici suboptime precum stagnarea sau tragerea excesivă. Aceste comportamente erau „premiat” temporar de funcția de recompensă, dar duceau la rezultate slabe pe termen lung. Fără o funcție de penalizare a acțiunilor inefficiente, agentul intra în bucle repetitive.
4. Supraîncărcarea rețelei și costuri de calcul mari: Antrenarea cu inputuri vizuale (imagini RGB sau grayscale de dimensiuni considerabile) a impus utilizarea unor rețele convoluționale extinse. Pe sisteme fără GPU, timpul de antrenament a crescut exponențial, iar ajustările hiperparametrilor au fost limitate de timpul necesar pentru obținerea unui feedback vizibil.
5. Generalizare slabă între episoade: În unele cazuri, agentul învăța o politică foarte specifică unui tipar repetitiv de joc. De exemplu, în Google Dino, dacă obstacolele apăreau mereu la același interval, agentul memoriza acest comportament și nu reacționa eficient când era introdusă variația aleatorie.

Rezultatele obținute oferă o imagine realistă asupra limitelor învățării prin întărire în medii de complexitate crescândă:

- Pentru jocuri simbolice (Snake), algoritmul DQN converge rapid, arhitectura poate fi simplă și nu sunt necesare tehnici suplimentare de regularizare. Performanțele se apropie de optimumul posibil.
- Pentru jocuri vizuale 2D (Google Dino), se observă că agentul poate învăța o politică coerentă, dar este sensibil la variația poziției obstacolelor, ceea ce indică o capacitate redusă de generalizare vizuală dacă nu este susținută de augmentări sau memorii extinse.
- În medii 3D (ViZDoom), provocările cresc exponențial. Fără reward shaping, stivuirea cadrelor și integrarea variabilelor interne, performanța rămâne slabă. Agenții trebuie să învețe simultan percepție, strategie și acțiune într-un mediu complex, ceea ce solicită atât rețeaua, cât și metodologia de antrenament.

În concluzie, învățarea prin întărire este o metodă puternică, dar sensibilă la detalii precum forma recompensei, frecvența feedback-ului, arhitectura aleasă și natura mediului. Proiectul demonstrează că succesul este posibil în condițiile unei abordări echilibrate între complexitate și controlul procesului de antrenare.

5.4 Comparație între rezultatele obținute și cele din literatura de specialitate

Pentru a evalua performanța agenților implementați în cadrul acestei lucrări, am realizat o comparație cu rezultatele raportate în trei lucrări relevante din domeniu care utilizează platforma ViZDoom și tehnici similare de reinforcement learning.

În lucrarea lui Kempka et al. (ViZDoom: A Doom-based AI Research Platform), autorii au demonstrat că un agent antrenat cu DQN și experience replay poate rezolva două scenarii:

- basic.wad: agentul învață să lovească o țintă statică cu scoruri medii între 80–100 puncte.
- maze.wad: agentul era capabil să navigheze un labirint 3D, evitând obstacole și colectând obiecte.

În articolul „Training an Agent for FPS Doom Game using Visual Reinforcement Learning and ViZDoom” (Khan et al.):

- Au folosit o rețea CNN combinată cu Q-learning.
- În competiția Visual Doom AI, agentul Arnold a obținut 413 frags și un K/D ratio de 2.45 în modul „Limited Deathmatch”, depășind semnificativ jucători umani sau alți agenți.

În lucrarea originală ViZDoom prezentată la conferința IEEE CIG 2016:

- Se menționează o arhitectură cu 2 layere convoluționale și un fully connected layer, iar scorul maxim în scenariul „basic” era de 101 puncte (pentru uciderea unui monstru).
- Se observă că rețelele converg în ~2000-3000 episoade.

Rezultatele obținute în această lucrare:

- În scenariul basic.wad, agentul dezvoltat a atins un scor mediu de ~84 puncte după aproximativ 1500 de episoade, semn că a învățat eficient să lovească ținta din prima încercare, comparabil cu cele mai bune rezultate raportate în literatura de specialitate.
- În defend_the_center.wad, agentul a reușit să obțină scoruri constante peste 500 după antrenamente de peste 5000 de episoade, în condiții de dificultate crescute (mai mulți inamici, poziționare variabilă), unde multe lucrări nu oferă date concrete.
- În deadly_corridor.wad, deși complexitatea navigării a dus la o rată de succes mai scăzută (~30% finalizare traseu), agentul a reușit să evite obstacole și să elimine majoritatea inamicilor, unde uneori este utilizată segmentarea pe pași sau curriculum learning.

Scenariu	Rezultat obținut (lucrare proprie)	Rezultat din literatura	Observatii
Basic.wad	84 puncte medii	80-101 puncte	Comparabil, convergenta mai rapida
Defend_the_centre_wad	500 puncte	Nespecificat	Agentul se comporta stabil
Deadly_corridor	30% succes	Arnold 43%	Performanta decenta

Capitolul 6. Concluzii

6.1 Concluzia proiectului

Proiectul prezentat în această lucrare a avut ca scop explorarea potențialului Reinforcement learning-ului în contextul jocurilor video, prin dezvoltarea și antrenarea unor agenți capabili să joace automat jocuri de dificultate și complexitate variabilă. Cele trei jocuri alese – Snake, Google Dino și ViZDoom – au permis o tranziție controlată de la medii complet observabile și simbolice, către medii vizuale, tridimensionale și parțial observabile.

Pe parcursul proiectului, au fost îndeplinite următoarele obiective:

- definirea unor reprezentări eficiente ale stării și acțiunilor pentru fiecare joc;
- proiectarea și antrenarea de rețele neuronale adaptate fiecărui mediu;
- aplicarea algoritmului Deep Q-Learning și a variantelor sale;
- testarea comportamentului învățat și validarea performanței prin metrici relevante;
- gestionarea recompenselor rare prin shaping și a observațiilor vizuale prin arhitecturi CNN;
- analiza provocărilor specifice fiecărui scenariu și identificarea soluțiilor aplicate.

În cazul jocului Snake, agentul a atins un comportament aproape optim în doar câteva sute de episoade, demonstrând eficiența metodei într-un mediu determinist. În Google Dino, agentul a învățat o politică vizuală funcțională, cu reacții adaptate la apariția obstacolelor, în ciuda variației în timp a vitezei jocului. În ViZDoom, prin scenariile basic, defend_the_center și deadly_corridor, agentul a demonstrat capacitatea de a învăța comportamente emergente într-un mediu 3D, combinând navigația, orientarea și atacul sub presiune.

Proiectul validează ideea că reinforcement learning poate fi aplicată cu succes în jocuri video, chiar și în condiții vizuale complexe, cu condiția unei arhitecturi bine calibrate și a unei funcții de recompensă adaptate. Astfel, se confirmă potențialul acestei tehnologii nu doar ca instrument de cercetare, ci și ca fundament pentru sisteme autonome în aplicații din lumea reală.

6.2 Analiza critică a rezultatelor obținute

Rezultatele obținute în urma antrenării agenților de învățare prin întărire reflectă o evoluție clară a capacității modelului de a învăța comportamente utile în funcție de complexitatea jocului. Totodată, ele evidențiază anumite limitări care apar în procesul de scalare de la jocuri simple la medii tridimensionale vizuale, caracterizate prin observabilitate parțială și recompense rare.

Pentru Snake: Performanța obținută a fost apropiată de optim. Rețeaua complet conectată a învățat rapid o politică eficientă, cu un scor stabil și o rată de eroare redusă. Explorarea nu a reprezentat o problemă majoră, iar învățarea s-a produs în mod constant și previzibil. Acest rezultat este în conformitate cu literatura de specialitate, care susține că mediile complet observabile și discrete sunt cel mai favorabile pentru aplicarea DQN.

În Google Dino: Rezultatele au fost semnificativ mai bune decât comportamentul aleator, însă nu optime. Agentul a reușit să identifice tipare vizuale și să reacționeze la ele, dar a întâmpinat dificultăți în fața variațiilor aleatorii sau a obstacolelor foarte rapide. A apărut o tendință de overfitting pe secvențele comune din antrenament. Acest aspect sugerează că o rețea CNN simplă, antrenată cu DQN, este suficientă pentru învățare de bază, dar limitată în generalizare.

Iar în Doom: Scenariile din ViZDoom au constituit o provocare semnificativă pentru agent. În timp ce în `basic.wad` și `defend_the_center` s-au obținut scoruri bune, în `deadly_corridor` agentul a avut dificultăți în menținerea unei politici stabile. A fost necesară:

- utilizarea de reward shaping pentru a evita stagnarea;
- integrarea de cadre multiple pentru context temporal;
- combinarea input-ului vizual cu variabile interne (health, ammo).

Chiar și cu aceste ajustări, performanța a rămas sub nivelul unui jucător uman mediu. Cu toate acestea, comportamentele învățate precum avansarea atentă, economisirea muniției sau evitarea colțurilor demonstrează că agentul a învățat strategii utile, chiar dacă incomplete.

6.3 Dezvoltări ulterioare

Pe baza rezultatelor obținute, există mai multe direcții prin care proiectul poate fi extins sau îmbunătățit. O primă direcție ar fi înlocuirea algoritmului DQN cu metode mai stabile și performante în medii complexe, cum ar fi PPO sau A3C, care oferă o explorare mai eficientă și o convergență mai rapidă în spații de stare vizuale.

O altă oportunitate o reprezintă optimizarea rețelei convoluționale. Rețele mai profunde, cu mecanisme de atenție sau augmentări vizuale, pot îmbunătăți percepția agentului în medii dinamice precum ViZDoom. De asemenea, o funcție de recompensă mai bine calibrată cu feedback parțial și penalizări contextuale ar putea accelera învățarea și ar reduce comportamentele suboptime.

Pentru testarea robusteții politicii învățate, agentul ar putea fi antrenat în medii noi sau modificate, verificând dacă reușește să generalizeze. Totodată, un pas natural ar fi extinderea proiectului către aplicații reale cum ar fi testarea automată a jocurilor sau dezvoltarea de agenți autonomi care farmează resurse virtuale în scop comercial.

Aceste direcții pot transforma proiectul dintr-o demonstrație tehnică într-o bază reală pentru cercetare, automatizare și inovație în domeniul inteligenței artificiale aplicate.

Capitolul 7. Bibliografie

- [1] [Kempka, Michal & Wydmuch, Marek & Runc, Grzegorz & Toczek, Jakub & Jaśkowski, Wojciech. \(2016\). ViZDoom: A Doom-based AI research platform for visual reinforcement learning. 1-8. 10.1109/CIG.2016.7860433.](#)
- [2] [Mark Dawes and Richard Hall. Towards using first-person shooter computer games as an artificial intelligence testbed. In Knowledge- Based Intelligent Information and Engineering Systems, pages 276–282. Springer, 2005.](#)
- [3] [Tony C Smith and Jonathan Miles. Continuous and Reinforcement Learning Methods for First-Person Shooter Games. Journal on Computing \(JoC\), 1\(1\), 2014..](#)
- [4] [David Trenholme and Shamus P Smith. Computer game engines for developing first-person virtual environments. Virtual reality, 12\(3\):181–187, 2008..](#)
- [5] [Khan, Adil & Jiang, Feng & Liu, Shaohui & Grigorev, Aleksei & Gupta, B & Rho, Seungmin. \(2017\). Training an Agent for FPS Doom Game using Visual Reinforcement Learning and VizDoom. International Journal of Advanced Computer Science and Applications.](#)
- [6] [Georgios N. Yannakakis, M., IEEE, and Julian Togelius, Member,IEEE. A Panorama of Artificial and Computational Intelligence inGames.pdf. IEEE TRANSACTIONS ON COMPUTATIONALINTELLIGENCE AND AI IN GAMES, VOL. 7, NO. 4, 2015. 7 \(4\).](#)
- [7] [Ratcliffe, Dino, Sam Devlin, Udo Kruschwitz, and Luca Citi, Clyde: Adeep reinforcement learning DOOM playing agent. What's Next For AIIn Games: AAAI 2017 Workshop. 2017.San Francisco, USA., 2017..](#)
- [8] [Bhatti, S., Desmaison, A., Miksik, O., Nardelli, N., Siddharth, N. andTorr, P.H, Playing Doom with SLAM-Augmented Deep ReinforcementLearning. arXiv preprint arXiv:1612.00380, 2016..](#)
- [9] [ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement LearningMichał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek & Wojciech Ja'skowskiInstitute of Computing Science, Poznan University of Technology, Pozna'n, Poland.](#)
- [10] [Minoru Asada, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda.Purposive behavior acquisition for a real robot by vision-based reinforcementlearning. In Recent Advances in Robot Learning, pages 163–187.Springer, 1996..](#)
- [11] [Reinforcement learning: Learning through trial and error by maximizing cumulative rewards..](#)
- [12] [Q-learning: A value-based reinforcement learning algorithm using Q-values to guide actions..](#)

- [13] [Deep Q-Learning \(DQN\): Combines Q-learning with deep neural networks to handle complex state spaces.](#)
- [14] [Convolutional neural network: Neural network specialized for processing grid-like data such as images.](#)
- [15] [Epsilon Decay: Gradual reduction of exploration rate in reinforcement learning.](#)
- [16] [Reinforcement Learning in Games: A Complete Guide: A comprehensive guide to applying RL techniques in game environments.](#)
- [17] [Reinforcement Learning in Game AI: Game-Playing Agents and Game-Level Design Techniques.](#)
- [18] [VizDoom: A platform for research in visual reinforcement learning using the Doom engine.](#)
- [19] [Snake The Game: Classic arcade game used to test simple reinforcement learning agents.](#)
- [20] [Google Dino: Chrome's offline endless runner game used for RL experiments.](#)
- [21] [Python: A versatile high-level programming language popular in AI and ML.](#)
- [22] [Numpy: Python library for numerical computing and array operations.](#)
- [23] [Pygame: Library for making multimedia applications like games in Python.](#)
- [24] [PyTorch: Deep learning framework offering dynamic computation graphs.](#)
- [25] [OpenCV: Open-source library for real-time computer vision.](#)
- [26] [Stable-Baseline3: A set of reliable reinforcement learning implementations in PyTorch.](#)