UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Abordări din domeniul inteligenței artificiale pentru jocul ”Frogger”

propusă de

Ioana-Ingrid Ghimpu

Sesiunea: februarie, 2025

Coordonator științific

Conf. Dr. Răschip Mădălina

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Abordări din domeniul inteligenței artificiale pentru jocul ”Frogger”

Ioana-Ingrid Ghimpu

Sesiunea: februarie, 2025

Coordonator științific

Conf. Dr. Răschip Mădălina

Avizat,
Îndrumător lucrare de licență,
Conf. Dr. Răschip Mădălina.
Data: ..... Semnătura: .....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Ghimpu Ioana-Ingrid** domiciliat în **România, jud. Iași, mun. Iași, strada Prof. I. Simionescu, nr. 13, bl. F11, et. parter, ap. 1**, născut la data de **01 mai 2002**, identificat prin CNP **6020501226724**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2018, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Abordări din domeniul inteligenței artificiale pentru jocul "Frogger"** elaborată sub îndrumarea domnului **Conf. Dr. Răschip Mădălina**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Abordări din domeniul inteligenței artificiale pentru jocul "Frogger"**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea "Alexandru-Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Ioana-Ingrid Ghimpu**

Data: ..... Semnătura: .....

Cuprins

Motivație

Această lucrare are ca motivație dorința de a experimenta practic cu doi algoritmi fundamentali din domeniul inteligenței artificiale: Deep Q-Learning și algoritmi genetici. Analizarea performanței acestora într-un context aplicat a oferit oportunitatea de a înțelege avantajele, limitările și principiile funcționalității fiecăruia.

Jocul Frogger a fost selectat ca platformă de testare datorită complexității sale, care îl face o provocare interesantă pentru antrenarea unui agent. Față de alte jocuri mai simple, cum ar fi ”Flappy bird”, acesta are un spațiu mai mare de acțiuni posibile și necesită o adaptare constantă din partea agentului. Jocul include, de asemenea, două părți distincte: traversarea șoselei prin evitarea mașinilor și navigarea râului folosind bușteni, ceea ce creează un mediu ideal pentru a testa capacitatea algoritmilor de a se adapta la provocări.

Acest proiect nu doar că explorează potențialul acestor algoritmi, ci contribuie și la dezvoltarea abilităților mele practice în implementarea și evaluarea metodelor de inteligență artificială.

Capitolul 1

Introducere

Inteligența artificială reprezintă abilitatea unui computer sau a unui robot controlat de un computer de a îndeplini sarcini de obicei specifice ființelor inteligente.[1] Studiul acestui domeniu a fost început, la nivel teoretic, de Alan Turing, care, în 1950, a propus un test pentru a verifica dacă o mașinarie "gândește". Pe parcursul timpului, inteligența artificială s-a dezvoltat în mai multe direcții, printre care învățarea automată, învățarea profundă, algoritmi genetici și sisteme expert. Tehnicile dezvoltate în cadrul acestor subdomenii permit dezvoltarea de aplicații care pot recunoaște voci și imagini, diagnostica pacienți, procesa limbajul natural și eficientiza procese industriale.

Ceea ce stă la baza acestui domeniu este capacitatea de a crea sisteme ce pot învăța și se pot îmbunătăți pe baza experiențelor anterioare fără intervenția directă a programatorului pentru fiecare sarcină.

1.1 Q-Learning

Q-Learning sau Reinforcement Learning este o tehnică din domeniul inteligenței artificiale ce se rezumă la învățarea unui agent să ia decizii optime în explorarea unui spațiu, fără a avea informații predefinite despre acesta. Pentru a realiza acest lucru, agentul primește feed-back sub formă de recompense, facilitând astfel găsirea celei mai optime strategii pentru obținerea celor mai bune rezultate pe termen lung.

Această tehnică a fost dezvoltată de Chris Watkins în 1989 și printre cele mai bune exemple pentru folosirea acesteia sunt AlphaGo (Google Deepmind, 2015), proiectul

[1] B.J.Copeland, artificial intelligence, 2025, https://www.britannica.com/technology/artificial-intelligence

”DQN” în care agenții sunt antrenați pentru a învinge în performanță jucători umani la jocuri Atari 2600 (Google Deepmind, 2015) dar și în șah unde a ajuns la nivelul uman expert (Buro, 2002).

Cel mai comun algoritm de Reinforcement Learning a fost dezvoltat de Watkins și Dayan în 1992 și a fost folosit pentru dezvoltarea aplicațiilor pentru controlul traficului (Abdoos, Mozayani și Bazzan în 2011) și pentru tranzacționare de acțiuni (Lee, Park, O, Lee, și Hong în 2007).

De asemenea, combinarea acestuia cu rețele neuronale are rezultate excepționale pentru jocuri de tip arcade, de multe ori chiar depășind performanțele umane: Ms. Pac-Man (Bom, Henken, și Wiering, 2013).

1.2 Algoritmi genetici

Algoritmii genetici reprezintă o metodă de optimizare care imită principiul lui Darwin al ”supraviețuirii celui mai adaptat” aplicată asupra unui set (generații) de soluții candidat (indivizi) care evoluează de la o generație la alta.[2] Aceștia au capacitatea de a rezolva probleme greu sau imposibil de rezolvat de metodele tradiționale, prin explorarea unui spațiu mare de soluții candidat, selectând și îmbunătățind constant soluțiile cele mai bune prin mecanisme precum încrucișarea (crossover) și mutațiile.

Datorită versatilității lor, ei au fost aplicați cu succes într-o varietate de domenii, precum ingineria (pentru proiectarea structurilor aerodinamice), medicina (pentru a alinia secvențele de ADN) și cel financiar (pentru identificarea strategiilor de tranzacționare eficiente). Ei joacă un rol important în domeniul inteligenței artificiale și roboticii, fiind folosiți în antrenarea agenților care învață să rezolve sarcini complexe, inclusiv jocuri și probleme de control.

1.3 Frogger

Frogger este un joc video clasic lansat în 1981, care a devenit un reper al industriei de jocuri. Acesta prezintă o premisă simplă, dar captivantă: jucătorul trebuie să controleze o broscuță ce trebuie să traverseze un drum aglomerat și un râu pentru a ajunge cu bine la destinație. Jocul necesită reflexe rapide, planificare și coordonare,

[2] Applied Energy, Genetic Algorithm, 2015, https://www.sciencedirect.com/topics/engineering/genetic-algorithm

având un mediu dinamic de mașini care circulă pe drumuri și bușteni care plutesc pe apă, care forțează jucătorul să ia decizii rapide pentru a evita obstacolele și a găsi calea optimă.

Datorită mecanicii sale bazate pe luarea deciziilor în timp real și a mediului dinamic și imprevizibil, Frogger a devenit un punct de interes în cercetarea inteligenței artificiale, fiind adesea utilizat ca o platformă pentru testarea algoritmilor de învățare automată.

În cadrul acestei lucrări a fost utilizat un environment realizat de Ricardo Henrique Remes de Lima în anul 2017[3], utilizând biblioteca "pygame". Jocul rulează cu 30 de frame-uri pe secundă (fps), iar la fiecare frame, agentul poate alege din 5 acțiuni: să se deplaseze în sus, în jos, în stânga, în dreapta sau să stea pe loc. Spațiul de explorare este reprezentat de o grilă de percepție (*vision grid*) de 13 x 12. Cu alte cuvinte, agentul se poate deplasa din locul de start, de 6 ori la stânga sau dreapta și de 12 ori în sus. Acesta este împărțit în culoare (*lanes*) care conțin obstacole. Obstacolele sunt definite atât de direcția și viteza cu care se deplasează, caracteristici ce diferă de la un culoar la altul, cât și de efectul pe care îl au atunci când agentul se intersectează cu ele. În momentul în care broasca se intersectează cu o mașină, aceasta moare, iar punctajul acumulat până atunci și poziția ei sunt resetate, însă în a doua parte a jocului supraviețuirea acesteia depinde de intersectarea cu bușteni, întrucât contactul cu "apa" duce la pierderea vieții și resetarea jocului. Aceste două părți, drumul și râul, sunt delimitate de un spațiu (două culoare) sigure. Ajungerea agentului pe cel de-al 12-lea culoar duce la resetarea poziției broaștei fără a pierde punctajul (broasca poate juca în continuare).

Jocul original este mai divers:jucătorul are trei vieți, pe râu unii bușteni sunt înlocuiți cu un șir de țestoase care se pot scufunda, există mai multe nivele care adaugă pe parcurs mai multe obstacole ce devin tot mai diverse, dar și avantaje pentru jucător, acestea rezultând ca dificultatea jocului să fie mereu în creștere. Am ales în această lucrare să renunț la aceste mecanici, pentru a nu complica excesiv procesul de învățare.

[3]https://github.com/rhrlima/frogger.git

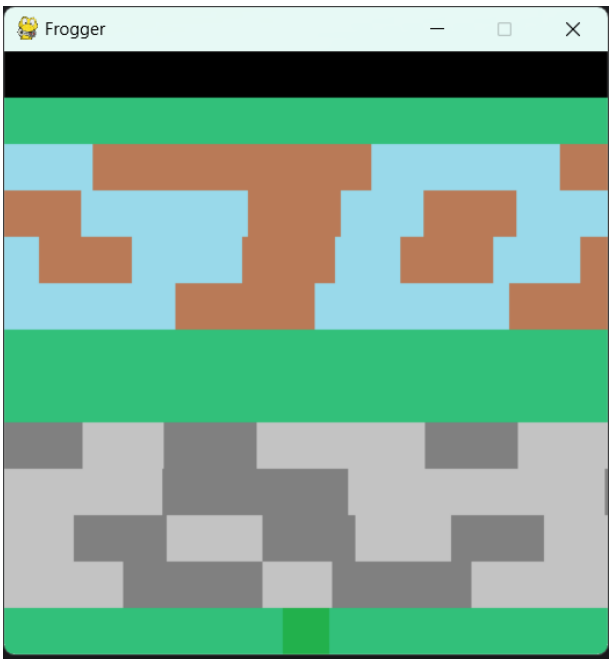

Figura 1.1: Reprezentare vizuală a jocului

1.4 Structura tezei

Această teză cuprinde descrierea algoritmilor folosiți, datele experimentale utilizate și concluziile rezultate pe baza comparațiilor făcute.

În capitolul 2 are scopul de a prezenta algoritmul Q-Learning și felul în care acesta a fost implementat în cadrul acestei lucrări.

Capitolul 3 prezintă algoritmii genetici și conține descrierea reprezentării indivizilor, funcției fitness și a operatorilor genetici folosiți.

Capitolul 4 expune experimentele efectuate pe baza celor doi algoritmi și prezintă rezultatele obținute pe baza acestora.

Capitolul 5 are rolul de a oferi o perspectivă amplă asupra metodelor de învățare, oferind exemple despre abordări existente, diferite de cea prezentată.

Capitolul 2

Algoritmul Deep Q-Learning

Q-learning, propus pentru prima dată de Watkins în 1989, este un algoritm fundamental de învățare prin consolidare. Algoritmul se bazează pe premisa estimării valorilor pe termen lung ale acțiunilor potențiale într-o anumită stare prin utilizarea unei funcții de valoare cunoscută sub numele de funcția Q. Prin actualizări succesive bazate pe experiența acumulată, agentul își rafinează deciziile pentru a optimiza recompensa totală primită în timp.

Procesul realizat de Q-learning poate fi conceptualizat ca un echilibru între explorare și exploatare. Pe de o parte, agentul trebuie să exploreze noi căi de acțiune, cu scopul de a identifica strategii mai avantajoase, iar, pe de altă parte, să selecteze acțiunile care promit cele mai mari recompense în momentul prezent, pe baza „experiențelor" existente ale agentului. O provocare semnificativă în antrenarea unui agent este riscul ca acesta să atingă un optim local, adică o strategie suboptimală care maximizează recompensele pe termen scurt, dar nu conduce la succesul pe termen lung. Pentru a evita această situație, agentul utilizează o strategie de explorare. O astfel de strategie este metoda ε-*greedy*, în care agentul selectează o acțiune aleatoare cu o probabilitate ε, spre deosebire de selectarea obișnuită a acțiunii cu cea mai mare valoare Q estimată. Această strategie este de o importanță capitală pentru a se asigura că agentul identifică căi noi care pot conduce la recompense mai mari.

2.1 Q-Learning

Algoritmul Q-Learning estimează valorile Q ale potențialelor acțiuni dintr-o anumită stare. În acest scop, acesta folosește perechi de tip stare - acțiune cărora le asociază

valori Q, inițial generate aleatoriu și care se actualizează pe parcursul antrenării cu scopul de a ajunge cât mai aproape de valorile lor reale.

Acest algoritm utilizează tabele pentru reținerea perechilor de tipul stare-acțiune și a valorilor Q corespunzătoare. Pe parcursul antrenării, când agentul întâlnește din nou o anumită pereche stare-acțiune, valoarea Q a acesteia este actualizată utilizând ecuația Bellman:

$$Q(s,a) = Q(s,a) + \alpha \cdot [R + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)] \tag{2.1}$$

- $Q(s,a)$: reprezintă valoarea curentă a funcției Q pentru starea s și acțiunea a, care reflectă estimarea pe termen lung a recompenselor pentru această pereche de stare și acțiune.

- α: este rata de învățare, un parametru care controlează cât de mult influențează informațiile noi actualizarea valorii curente. Valorile mari ale α duc la o adaptare rapidă la noi informații.

- R: este recompensa imediată primită după efectuarea acțiunii a în starea s.

- γ: reprezintă factorul de discount, care ajustează influența recompenselor viitoare față de cele imediate. Un γ apropiat de 1 sugerează că agenții țintesc recompensele pe termen lung, în timp ce un γ mic favorizează recompensele imediate.

- $\max_{a'} Q(s',a')$: este valoarea maximă a funcției Q pentru starea următoare s', considerând toate posibilele acțiuni a' ce pot fi luate din acea stare. Acesta reflectă valoarea estimată a celei mai bune acțiuni în starea s'.

Această abordare funcționează bine pentru medii cu un spațiu redus de stări și acțiuni, însă devine ineficientă pentru probleme mai complexe. Utilizarea tabelelor pentru reținerea și actualizarea valorilor Q devine cu atât mai problematică în cazul mediilor cu un spațiu mare de stări, unde șansa de a înâlni două stări identice este extrem de mică. Pentru a face față acestor provocări, algoritmul Q-Learning poate fi extins prin integrarea unei rețele neuronale, conducând astfel la dezvoltarea algoritmului Deep Q-Learning[8][7]. În acest caz, neuronii de ieșire reprezintă valorile funcției Q pentru o pereche stare-acțiune, iar ponderile sunt inițializate aleatoriu, la fel ca în cazul valorilor din tabel.

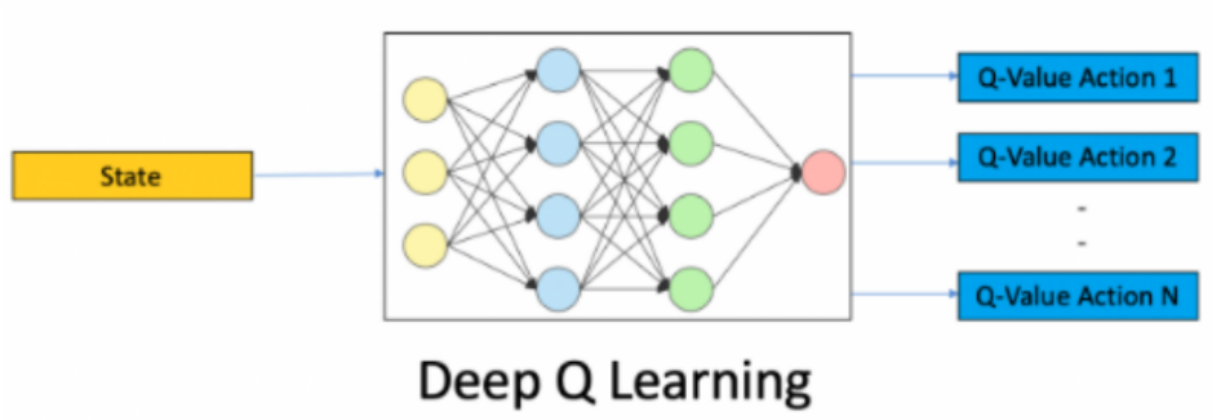

Figura 2.1: Model arhitectură rețea neuronală folosită în Deep Q-Learning

În cadrul algoritmului implementat pentru această lucrare, agentul reține perechi de tipul (s, a, s', r) într-un buffer de experiențe și se antrenează pe o submulțime aleatoare a acestuia. Având în vedere dimensiunea mare a spațiului de stări, valorile Q pentru submulțimea de perechi aleasă, sunt aproximate de rețeaua neuronală.

Pentru a putea utiliza algoritmul de Deep Q-Learning, este necesară calcularea unei valori target. Astfel, formula 2.1 devine:

$$Q_{target} = R + \gamma \cdot \max_{a'} Q(s', a') \tag{2.2}$$

Acest target este folosit apoi pentru calcularea erorii care ajută la actualizarea ponderilor din rețea. În acest context, formula 2.2 nu este suficientă, deoarece nu acoperă cazurile în care s' este stare terminală, în care agentul nu mai are posibilitate de decizie. Pentru a evita ca aceste cazuri să influențeze negativ funcționalitatea algoritmului, considerăm:

$$Q_{target} = R \tag{2.3}$$

Odată ce valoarea pentru target a fost calculată, agentul poate calcula eroarea pentru a actualiza ponderile scăzând valoarea Q obținută din target. În cadrul algoritmului implementat, eroarea este calculată astfel:

$$loss = \sum_{n=0}^{b} [Q(s_n, a_n) - Q_{target}]^2 \tag{2.4}$$

Metoda prezentată este mult mai eficientă pentru problema propusă, însă poate duce la instabilitate, deoarece valoarea Q_{target} este modificată la fiecare pas al antre-

namentului. Din acest motiv, algoritmul implementat este unul de Double Deep Q-Learning[10].

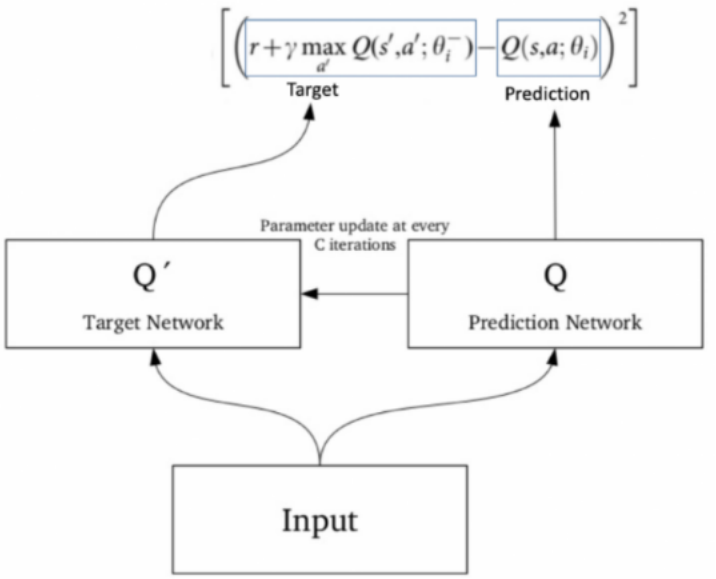

Figura 2.2: Model arhitectură rețea neuronala folosită în Double Deep Q-Learning (sursa [5])

Acesta folosește o rețea neuronală suplimentară care ajută la calcularea Q_{target} astfel:

- rețeaua principală returnează valorile $Q(s, a)$ și $Q(s', A)$, unde A reprezină spațiul acțiunilor posibile

- se reține a' acțiunea pentru care s-a obținut cea mai mare valoare $Q(s', A)$

- se obține de la rețeaua suplimentară (rețeaua target) $Q(s', a')$

- se calculează valoarea target modificând formula 2.2:

$$Q_{target} = R + \gamma \cdot Q(s', a') \tag{2.5}$$

Pe cea de a doua rețea neuronală nu se aplică back-propagation pentru actualizarea ponderilor, în schimb, o dată la ceva timp, se copiază aceste valori din rețeaua principală. Această abordare reduce oscilațiile bruște ale valorilor $Q(s, a)$ și previne actualizările incoerente care pot apărea din cauza corelațiilor puternice dintre eșantioanele succesive. Prin menținerea unei rețele țintă separate, algoritmul Deep Q-Learning îmbunătățește convergența și permite o estimare mai robustă a valorilor Q, ceea ce duce la o învățare mai stabilă și eficientă.

2.2 Rețea neuronală

Rețeaua neuronală reprezintă un factor extrem de important în cadrul algoritmilor de Deep Q-Learning. Pentru funcționalitatea eficientă a acestor metode este necesară folosirea unei rețele capabile să aproximeze valorile funcției Q într-un mod robust și stabil. Arhitectura rețelei joacă un rol esențial în performanța algoritmului, influențând atât viteza de învățare, cât și precizia estimărilor.

În cadrul acestei implementări, rețeaua neuronală are rolul de a primi ca intrare o reprezentare a stării s și de a produce la ieșire estimările $Q(s, a)$ pentru fiecare acțiune posibilă. Structura acesteia este aleasă astfel încât să echilibreze complexitatea modelului și capacitatea de generalizare, evitând atât subajustarea (*underfitting*), cât și supraajustarea (*overfitting*).

Implementarea rețelei neuronale în cadrul acestei lucrări a fost făcută utilizând biblioteca PyTorch[1], o alegere populară datorită suportului său pentru optimizarea automată a gradientului, paralelizare pe GPU și ușurința în definirea arhitecturilor de rețele neuronale. PyTorch permite antrenarea eficientă a modelelor și oferă instrumente utile pentru manipularea tensorilor.

Rețeaua neuronală folosită pentru acest algoritm este una compusă din trei straturi liniare, complet conectate:

- primul strat are 84·84 neuroni de intrare, aceștia reprezentând pixelii care compun o stare (o imagine 1D)

- un strat ascuns care are 128 de neuroni

- ultimul strat cu 5 neuroni de ieșire, reprezentând spațiul de acțiuni

```
self.fc1 = nn.Linear(84*84, out_features: 128)
self.fc2 = nn.Linear( in_features: 128, out_features: 64)
self.fc3 = nn.Linear( in_features: 64, num_actions)
```

Figura 2.3: Arhitectura rețelei neuronale implementate

[1]https://pytorch.org

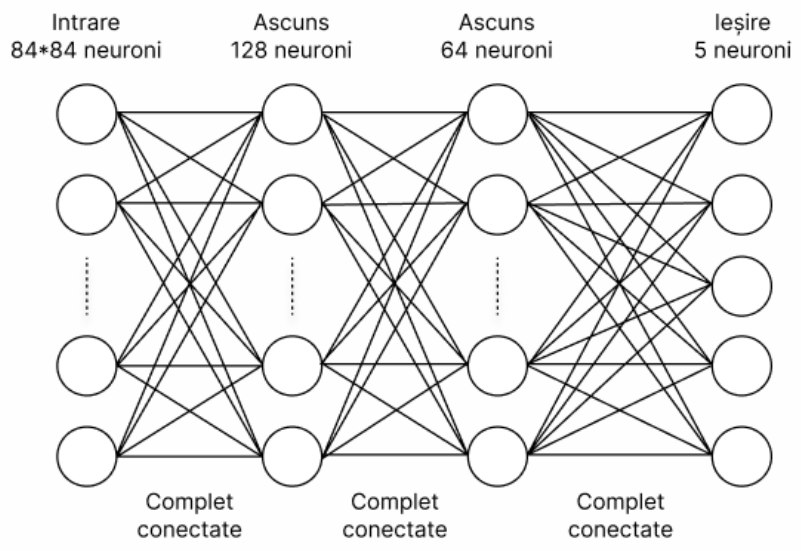

Figura 2.4: Arhitectura rețelei neuronale utilizate

Primele două straturi utilizează funcția de activare Rectified Linear Unit (ReLU)[4]:

$$f(x) = \begin{cases} x, & \text{dacă } x > 0 \\ 0, & \text{altfel} \end{cases}$$

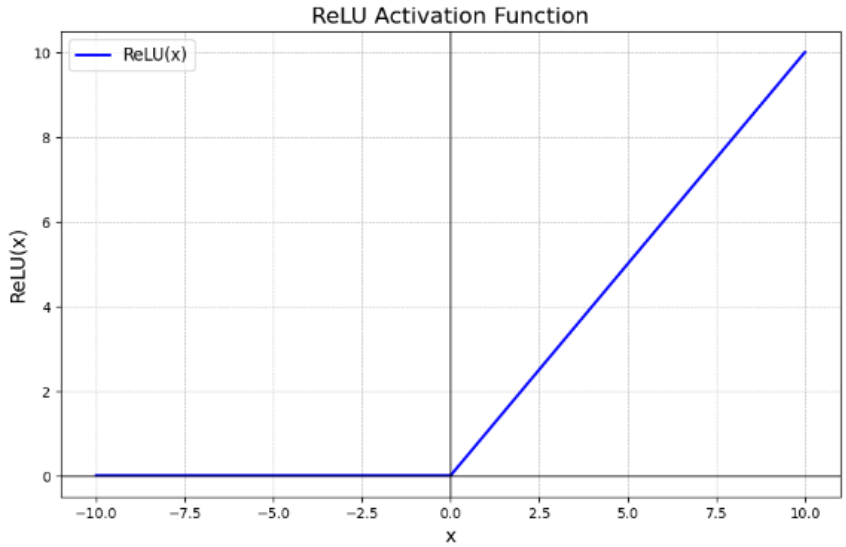

Figura 2.5: Funcția de activare ReLU

Asupra ultimului strat se aplică o funcție liniară de activare, decizie motivată de evitarea încadrării valorilor $Q(s, a)$ în intervalul [0, 1].

Ponderile primelor două straturi sunt generate folosind inițializarea Kaiming, fiind potrivită pentru funcția de activare aleasă, iar cele pentru ultimul strat sunt generate folosind inițializarea Xavier[3].

Bias-urile tuturor straturilor sunt inițializate cu zero, iar rețeaua se actualizează folosind optimizatorul Adam.

2.3 Sistemul de recompense

Sistemul de recompense reprezintă un alt factor important în performanța algoritmului Deep Q-Learning. Acesta influențează în mod direct comportamentul agentului. O recompensă mare reprezintă acțiunile care trebuie prioritizate, iar o recompensă mică, acțiunile care trebuie evitate.

Având în vedere dorința ca agentul să îndeplinească sarcina dată, acțiunile care îl apropie pe acesta de atingerea obiectivului, în cazul problemei propuse și anume acțiunea de avansare pe culoare, este recompensată, iar cele care îl îndepărtează de obiectiv, cum ar fi devansarea pe culoare, este penalizată. Pentru ca agentul să învețe eficient, trebuie luate în considerare și stările terminale. În cazul propus, acestea sunt reprezentate de trecerea în siguranță a tuturor culoarelor, pentru care se primește o recompensă generoasă și pierderea vieții care duce la restartarea jocului care este penalizată sever.

La fel ca în cazul alegerii acțiunilor, alegerea valorii recompenselor trebuie guvernată de echilibru. Spre exemplu, o penalizare mult prea mare pentru pierderea unei vieți poate duce agentul să înțeleagă faptul că avansarea, deși apropie de îndeplinirea scopului, aduce riscul unei pierderi extreme, deci nu merită. Astfel, comportamentul obținut este cel de a rămâne în zona sigură, fără a progresa. Pe de altă parte, o penalizare prea mică determină agentul să nu ia în considerare pericolul reprezentat de obstacole.

O altă situație nefavorabilă este reprezentată de o recompensă fixă pentru avansare care încurajează agentul să exploateze recompensa dintr-o zonă sigură, avansând și devansând succesiv.

Luând în considerare toate situațiile prezentate, sistemul de recompense ales este prezentat în următoarele tabele.

După cum se poate observa în tabela 2.2 există o coloană cu recompense în cazuri speciale. Pentru mișcarea ”sus” agentul va primi o recompensă mult mai mare atunci când avansează pentru prima dată pe un nou culoar, pentru a-l încuraja să avanseze cât mai mult și să nu încerce să primească puncte pentru avansări și devansări succesive. Agentul este penalizat în loc să fie recompensat atunci când ajunge într-o stare impor-

Recompensa pentru stări	
Pierderea unei vieți	-50
Traversarea drumului	+50
Traversarea râului	+100

Tabela 2.1: Recompense pentru stări importante

Recompensa pentru acțiuni		
acțiune	recompensă	recompensă specială
sus	+2	$+10 \cdot new - max - lane$
jos	-2	-50
stânga	0	
dreapta	0	
nimic	0	

Tabela 2.2: Recompense pentru acțiuni

tantă favorabilă prin intermediul acțiunii ”jos”, în loc să folosească acțiunea ”sus”.

Folosind acest sistem de recompense, agentul învață faptul că avansarea este cel mai important aspect, dar nu ignoră strategii care necesită evitarea obstacolelor folosind alte acțiuni, deoarece nu este pedepsit pentru acest lucru, iar penalizarea pentru pierderea unei vieți este costisitoare.

Capitolul 3

Algoritm Genetic

Algoritmii genetici (AG) sunt o clasă de tehnici de optimizare inspirate de principiile selecției naturale. Conceptul fundamental care stă la baza utilizării algoritmilor genetici poate fi înțeles ca o imitație a procesului de evoluție naturală, în care cei mai potriviți indivizi sunt selectați în vederea producerii descendenților generației următoare, asigurându-se astfel supraviețuirea celor mai relevante trăsături. Iterația acestui proces permite algoritmilor genetici să exploreze eficient un spațiu de căutare vast, obținând adesea soluții optime sau aproape optime la probleme care altfel ar fi greu de rezolvat prin metode tradiționale de optimizare.

Algoritmii genetici au un punct forte deosebit în capacitatea lor de a gestiona probleme de optimizare multi-obiectiv, în care obiectivele conflictuale trebuie optimizate simultan. În plus, versatilitatea lor permite adaptarea la diferite tipuri de probleme, inclusiv optimizarea continuă și discretă, selectarea caracteristicilor în învățarea automată.

Cu toate acestea, trebuie remarcat faptul că algoritmii genetici nu sunt lipsiți de limitările lor. Printre acestea se numără predispoziția la convergență prematură, în plus față de necesitatea ajustării riguroase a parametrilor. Echilibrul optim între explorare (căutarea de noi soluții) și exploatare (rafinarea soluțiilor optime) este de o importanță capitală pentru eficacitatea unui algoritm genetic.

3.1 Indivizi - reprezentare

Algoritmii genetici se bazează pe selectarea celor mai potriviți indivizi pentru a crea o nouă generație bazată pe cele mai relevante gene. Acești indivizi sunt, în esență,

reprezentarea soluțiilor candidat, dintr-o anumită generație.

În cadrul algoritmului implementat pentru această lucrare, indivizii sunt reprezentați prin intermediul unei clase ce conține caracteristici specifice fiecărei soluții folosind limbajul de programare Python. Genomul este structurat sub forma unei liste care conține valori naturale, reprezentând acțiuni pe care agentul le face din fiecare stare. Alte variabile importante pentru a reține cât este de potrivit un individ în rezolvarea problemei propuse sunt culoarul maxim pe care acesta a ajuns și indexul mișcării care a dus la pierderea unei vieți.

Cum indivizii dintr-o generație sunt responsabili de crearea indivizilor din generațiile următoare, fiecare dintre aceștia are atribuită valoarea obținută la aplicarea funcției fitness, care este utilizată apoi, în procesul de selecție. Pentru a facilita aplicarea selecției de tip ruletă, fiecărei soluții îi este asociată o probabilitate relativă calculată ca raport între valoarea sa fitness și suma totală a valorilor fitness ale populației și un scor cumulativ utilizat pentru a construi intervale de selecție, facilitând alegerea indivizilor în funcție de valoarea lor precedentă. De asemenea, în scopul evitării alegerii repetate ale aceleiași soluții pentru formarea noii generații, fiecare individ este marcat după alegere.

Genomii pentru fiecare dintre indivizii care alcătuiesc generația inițială sunt generați aleatoriu cu condiția ca aceștia să nu ducă la pierderea unei vieți mai devreme de cea de-a 10-a acțiune. Această decizie este motivată de evitarea erorilor de calcul întâlnite la alegerea punctelor de încrucișare. În cadrul procesului de încrucișare, punctele de tăiere sunt alese pe baza indicelui acțiunii care a dus la pierderea unei vieți. Dacă un individ moare foarte devreme, nu există un interval suficient de mare pentru a selecta puncte de tăiere valide. Procesul alegerii punctelor de tăiere este explicat pe larg în cadrul secțiunii despre operatori genetici (3.2). Dacă un individ ar muri imediat după prima mișcare, în procesul de încrucișare, acesta ar fi copiat în eșantionul de selecție, deoarece nu există puncte de tăiere semnificative pentru a recombina genele cu un alt individ. În loc să generăm un nou individ cu caracteristici îmbunătățite, am obține o copie exactă a unei soluții deja slab performante, ceea ce ar încetini progresul evoluției algoritmului genetic.

3.2 Funcția fitness

Funcția fitness reprezintă un element crucial în dezvoltarea unui algoritm genetic. Aceasta are rolul de a evalua calitatea soluțiilor candidat și de a determina cât de bine rezolvă acestea problema propusă. Alegerea funcției influențează atât viteza de convergență a algoritmului, cât și calitatea indivizilor din fiecare generație. Astfel, o funcție bine aleasă poate duce la convergența rapidă spre soluția cea mai optimă și asigură diversitatea genetică a populației, pe când o funcție prost definită poate duce la convergența prematură a algoritmului sau la lipsa unui progres semnificativ în timp.

Cum problema propusă spre rezolvare în această lucrare constă în antrenarea unui agent să avanseze cât mai mult folosind un număr minim de acțiuni, funcția fitness aleasă pentru a evalua soluțiile candidat este următoarea:

$$individual.fitness = individual.lane - \frac{individual.death}{genome.length} \tag{3.1}$$

În acest context, $individual.lane$ reprezintă culoarul maxim pe care a ajuns agentul, urmând acțiunile reprezentate de individul pentru care se calculează fitness-ul, $individual.death$ reprezintă indexul acțiunii care a dus la pierderea vieții, iar $genome.length$ semnifică lungimea genomului, sau numărul maxim de acțiuni pe care le are la dispoziție fiecare individ.

În acest fel, se calculează nu doar cât de mult a avansat agentul urmând un set de acțiuni, ci și viteza cu care acesta a progresat.

3.3 Operatori genetici

Fiind inspirați din procesul evolutiv biologic, algoritmii genetici utilizează o serie de operatori genetici pentru a modifica și diversifica populația. Scopul acestora este de a genera o nouă populație de indivizi mai adaptați, îmbunătățind progresiv soluțiile către optimul problemei. Operatorii genetici sunt mecanisme care modifică structura indivizilor și influențează procesul de optimizare. Acești operatori sunt responsabili atât pentru explorarea unor soluții noi, cât și pentru exploatarea celor mai bune caracteristici deja existente, procentul cu care sunt aplicați influențând direct acest echilibru. Principalii operatori genetici sunt încrucișarea, mutația și selecția, fiecare având un rol esențial în dinamica algoritmului.

Încrucișarea reprezintă principala metodă de exploatare în cadrul unui algoritm

genetic. Aceasta extinde spațiul soluțiilor prin crearea unor noi indivizi ce moștenesc caracteristicile a doi indivizi părinți. Încrucișarea poate fi aplicată prin alegerea a unul sau mai multe puncte de tăiere.

Mutația, pe de altă parte, determină rata de explorare a algoritmului. Aceasta constă în mici modificări aleatorii în genomului unui individ, aducând schimbări minore asupra structurii acestuia. O rată de mutație prea mare, duce la instabilitatea algoritmului, iar o rată mult prea scăzută poate duce la blocarea într-un punct de optim local.

În cele din urmă, selecția reprezintă, de asemenea, un pas crucial în performanța pe termen lung a algoritmului. Aceasta are rolul de a alege cei mai potriviți indivizi care să alcătuiască următoarea generație de soluții candidat. La fel ca ceilalți operatori genetici, și selecția se poate face aplicând mai multe metode, printre care se numără elitismul, roata norocului sau ruletă și selecția de tip turneu.

Elitismul este o strategie utilizată pentru a conserva cei mai puternici indivizi și de a-i transfera, fără modificări, în următoarea generație. Aceasta previne pierderea unor soluții calitative și poate determina o convergență rapidă spre o soluție optimă, cu riscul de a fi una prematură și de a duce la o stagnare în calitatea soluțiilor.

Ruleta este o metodă de selecție care se bazează pe valoarea fitness-ului indivizilor, dar care poate duce la convergența prematură a algoritmului sau la lipsa diversității în populație. Aceasta are la bază ideea tragerii la sorți, însă fiecare individ are șanse proporționale cu valoarea fitness-ului. În esență, indivizii ce au obținut un fitness mai mare, au mai multe șanse de a fi selectați pentru formarea generației următoare. Probabilitatea de selecție a unui individ este:

$$P_i = \frac{fitness(i)}{fitness_{total}} \tag{3.2}$$

- P_i reprezintă probabilitatea individului i de a fi selectat

- $fitness(i)$ este fitness-ul individului i

- $fitness_{total}$ reprezintă media fitness-ului întregii populații

După calcularea probabilităților tuturor indivizilor dintr-o populație, se poate imagina o roată de ruletă pe care fiecare individ ocupă o zonă corespunzătoare valorii calculate anterior. Se generează apoi un număr aleatoriu și se alege individul corespunzător acestuia de pe roata de ruletă.

Selecția de tip turneu este o altă metodă populară de selecție în cadrul algoritmi-

lor genetici. Față de metoda selecției prin ruletă, aceasta nu este bazată pe fitness-ul fiecărui individ în raport cu fitness-ul total al populației. Pentru a aplica selecția prin turneu se selectează o submulțime aleatoare a populației și se selectează individul cu fitness-ul cel mai ridicat pentru a face parte din generația următoare.

Algoritmul utilizat în cadrul acestei lucrări implementează două variante de încrucișare, una de mutație și toate cele trei strategii de selecție enumerate mai sus. În cele ce urmează vor fi prezentate pe larg metodele folosite.

3.3.1 Încrucișare cu un punct de tăiere

Această funcție se aplică pe toți indivizii populației. Pentru fiecare dintre aceștia, va fi ales un alt individ părinte fie aleatoriu, fie cu probabilitate $cross\_weak$. Această variabilă este calculată după formula:

$$cross\_weak+ = 2^{\frac{n+1}{population\_size}\cdot 10-10} \tag{3.3}$$

- n reprezintă indexul individului curent

Variabila $cross\_weak$ are rolul de a controla partenerii cu care un individ poate fi supus funcției de încrucișare. Pentru indivizii cu fitness-ul mic, probabilitatea de a alege un partener cu fitness-ul mare este mai mare, pentru a asigura evoluția populației spre soluția optimă. Pentru indivizii cu fitness-ul mare, crește probabilitatea de a avea un partener cu fitness-ul mai scăzut, pentru a asigura diversitatea populației și pentru a evita convergența prematură.

Astfel, în locul alegerii unui partener aleatoriu din întreaga populație, se alege o valoare aleatoare între 0 și 1. Următorl pas este de a sorta populația după valoare fitness a fiecărui individ. În situația în care valoarea aleasă este mai mică decât $cross\_weak$, se obține din populația sortată submulțimea indivizilor cu un fitness mai scăzut decât cel al soluției actuale (toți indivizii cu un index mai mic decăt cel al individului curent), iar dacă numărul obținut este mai mare de acest prag, se obține submulțimea indivizilor cu un fitness superior. Având această submulțime a populației, se alege aleatoriu un partener pentru individul actual.

Această funcție implementează încrucișarea cu un singur punct de tăiere. Acest punct este ales aleatoriu dintr-un interval determinat de poziția acțiunii, din genomul primului părinte, care duce la pierderea unei vieți. Alegerea intervalului se face astfel:

- intervalul $[0, individual.death]$, unde $individual.death$ reprezintă indexul acțiunii care a dus la pierderea unei vieți, se împarte în trei secțiuni astfel:

- $$0 \rightarrow individual.death \cdot 0.25$$
- $$individual.death \cdot 0.25 \rightarrow individual.death \cdot 0.75$$
- $$individual.death \cdot 0.75 \rightarrow individual.death$$

- fiecăreia dintre aceste secțiuni îi este atribuită o probabilitate astfel $(0.25, 0.5, 0.25)$

- se alege un număr aleatoriu între 0 și 1 și se compară cu valorile probabilităților anterioare. Dacă valoarea este între 0 și 0.25, punctul de tăiere va fi ales din prima secțiune. Dacă acest număr este între 0.25 și 0.75, se va alege din cea de-a doua secțiune, altfel, din cea de-a treia secțiune.

Acest mod de a alege punctul de tăiere este motivat de dorința de a îmbunătăți noul individ format prin încrucișare, iar dacă am alege punctul fără a lua în considerare acțiunea care a dus la pierderea unei vieți, acesta ar avea aceeași valoare fitness ca părintele său. De asemenea, împărțind acest interval în trei secțiuni cu probabilitățile respective, ne asigurăm că tăierea capetelor se face cu probabilitate mică, crescând astfel diversitatea soluțiilor din populație. Tot în acest scop, algoritmul descris mai sus se aplică până când individul rezultat este diferit de părinții săi.

Aplicând această funcție pentru fiecare individ al populației, obținem atâția indivizi noi câți sunt în orice generație, pe care îi adăugăm în eșantionul de selecție.

3.3.2 Încrucișare cu două puncte de tăiere

Pentru fiecare individ al populației, se alege un număr aleatoriu între 0 și 1. Dacă acesta este mai mic decât probabilitatea $cross2\_rate$, atunci individul este supus încrucișării. Se alege, la întâmplare, din restul populației, un al doilea părinte. Se aleg două puncte de tăiere astfel încât să fie înainte și după acțiunea care cauzează pierderea unei vieți în genomul primului părinte.

Se crează un individ nou, luând din primul părinte genele până la primul punct de tăiere, din al doilea părinte trăsăturile până la al doilea punct de tăiere, iar restul

trăsăturilor aparțin primului părinte. Algoritmul se repetă până se obține un individ diferit de părinții săi. Soluțiile candidat astfel obținute se adaugă eșantionului de selecție.

Motivul pentru care a fost aleasă o probabilitate cu care această funcție să se aplice pe un individ, a fost acela de a putea evita conversia prematură rezultată din popagarea unor secvențe de gene foarte puternice de la o generație la alta. Având în vedere faptul că funcția de încrucișare reprezintă componenta de exploatare, aceasta crează indivizi tot mai puternici de-a lungul generațiilor. Încrucișarea între indivizi puternici ar putea duce la crearea unor noi indivizi cu aceeași secvență de acțiuni ca și parinții săi, pe termen lung ducând la lipsa diversității populației. Datele experimentale arată însă faptul că aplicarea acestei funcții cu probabilitatea 1 duce la cele mai bune rezultate.

3.3.3 Mutație

Funcția de mutație implementată în această soluție încearcă să echilibreze explorarea și obținerea de soluții mai apropiate de cea optimă. Pentru acest algoritm, mutația se aplică asupra a trei gene ale indivizilor.

Pentru fiecare individ al populației se alege o valoare aleatorie între 0 și 1. Aceasta se compară cu rata de mutație pentru a decide dacă se va aplica acest operator genetic asupra individului.

În cazul fiecărei soluții candidat supusă mutației, acțiunea care duce la pierderea unei vieți are probabilitatea de 0.7 de a fi modificată, pentru a asigura posibilitatea de a se obține și soluții cu valoarea fitness mai mare decât individul original. Se aleg aleatoriu alte gene asupra cărora se va aplica mutația. Fiecare dintre aceste gene este înlocuită cu o altă acțiune din spațiul de acțiuni, aleasă aleatoriu.

Indivizii astfel obținuți sunt adăugați eșantionului de selecție.

3.3.4 Combined-selection

Selecția noii generații de soluții candidat se aplică, în principal, pe indivizii rezultați în urma încrucișărilor și a mutației, însă prin aplicarea elitismului se păstrează și cei mai puternici indivizi din generația precedentă. În acest sens, noua generație este formată astfel:

- se aleg după metoda elitistă primii doi indivizi din generația anterioară și primii doi din eșantionul de selecție.

- se aleg prin ruletă 50% din indivizii care vor forma noua generație, după cum s-a prezentat anterior, folosind probabilitățile și pragurile calculate pentru fiecare individ, marcând soluțiile selectate pentru a nu duplica rezultatele.

- se aleg restul indivizilor prin selecția de tip turneu, după cum a fost descris în secțiunea precedentă, marcând fiecare individ după ce a fost ales

Prin marcarea indivizilor aleși se asigură diversitatea populației și se evită convergența prematură la o soluție suboptimală. Împărțind astfel noua generație, rezultă soluții diverse, prioritizând indivizii puternici fără a-i neglija complet pe cei mai slabi care ar putea duce totuși la rezultate optime în viitor.

Capitolul 4

Rezultate experimentale

Așa cum este cazul multor algoritmi din domeniul inteligenței artificiale, performanța algoritmilor abordați în această lucrare este strâns legată de alegerea valorilor unor variabile esențiale, precum dimensiunea populației, rata de mutație și metoda de selecție (pentru AG), rata de învățare și sistemul de recompense (în cazul DQL). Aceste variabile influențează direct capacitatea algoritmului de a găsi soluții optime și de a echilibra exploatarea soluțiilor bune cu explorarea unor alternative noi.

Pentru a înțelege impactul fiecărei variabile asupra performanței agentului Frogger, în această lucrare se prezintă o serie de experimente utilizând o abordare de tip *trial and error*. Prin varierea sistematică a acestor parametri și compararea rezultatelor obținute, se urmărește să se identifice combinațiile care conduc la o evoluție mai bună și la performanțe mai bune ale agentului.

În această secțiune, se prezintă detaliat aceste experimente, se analizează rezultatele obținute și implicațiile fiecărei alegeri asupra procesului de antrenare.

4.1 Deep Q-Learning

Pentru acest algoritm, scopul experimentelor este de a compara performanțele diferitelor abordări și felul în care acestea afectează performanța agentului și viteza de convergență. În acest scop, se prezintă rezultatele obținute de arhitecturi diferite ale rețelei neuronale, de folosirea unei rețele separate pentru calcularea valorii Q_{target} în contrast cu abordarea Deep Q-Learning și de aplicarea unor rate de învățare diferite. Valorile obținute în urma acestor experimente sunt prezentate în tabelul 4.1.

4.1.1 Rata de învățare

Unul dintre aspectele acestui experiment este de a testa performanța algoritmului Q-Learning în raport cu valoarea ratei de învățare. Această variabilă joacă un rol important în antrenarea agentului și în viteza cu care acesta învață. O rată de învățare prea mare poate duce la convergență prematură, pe când o rată de învățare prea mică rezultă în învățarea foarte lentă.

Pentru a testa impactul acestei variabile și pentru a determina valoarea cea mai potrivită pentru antrenarea agentului să joace jocul Frogger, am rulat algoritmul cu trei valori distincte 0.02, 0.002 și 0.0002. Figurile 4.1 și 4.2 reprezintă rezultatele obținute de-a lungul a 5000 de episoade. Analizând acest grafic, se poate observa faptul că valoarea ratei de învățare duce la rezultate diferite între cei doi algoritmi. Pentru Deep Q-Learning, rata de învățare care duce la obținerea celor mai bune rezultate are valoarea 0.0002, valoarea 0.002 duce la o convergență prematură, iar valoarea cea mai mare obține rezultate medii. Acest lucru indică faptul că agentul învață cel mai eficient cu o rată de învățare mai mică. Pentru algoritmul Double Deep Q-Learning se observă convergența înainte de jumătatea numărului de episoade pentru valoarea 0.02, iar valoarea de mijloc are cele mai bune rezultate. Pentru valoarea 0.0002 se poate observa că agentul a avut o performanță mai proastă. Acest lucru indică faptul că, în cazul algoritmului Double Deep Q-Learning, rata de învățare nu trebuie să se apropie de extreme.

4.1.2 Deep Q-Learning și Double Deep Q-Learning

Cum scopul acestei lucrări este de a compara performanțele diferitelor abordări din domeniul inteligenței artificiale, au fost testate cele două variante prezentate în capitolul 1: Deep Q-Learning și Double Deep Q-Learning.

Pentru a putea face o comparație între performanțele celor doi algoritmi, experimentele prezentate anterior au fost aplicate pe ambele variante. Rezultatele acestor metode sunt reprezentate în figurile 4.1 și 4.2. Pe baza acestor grafice putem evidenția faptul că algoritmul DDQL obține rezultate mai bune decât DQL. Urmărind traiectoriile definite de valoarea ratei de învățare care a dus la cele mai bune rezultate în cazul fiecărui algoritm, se observă faptul că algoritmul DDQL obține rezultate mai bune mai devreme și mai consistent decât DQL. Diferența dintre mediile rezultatelor obținute de cei doi algoritmi nu este, însă, una notabilă, după cum se poate observa urmărind

datele din tabel.

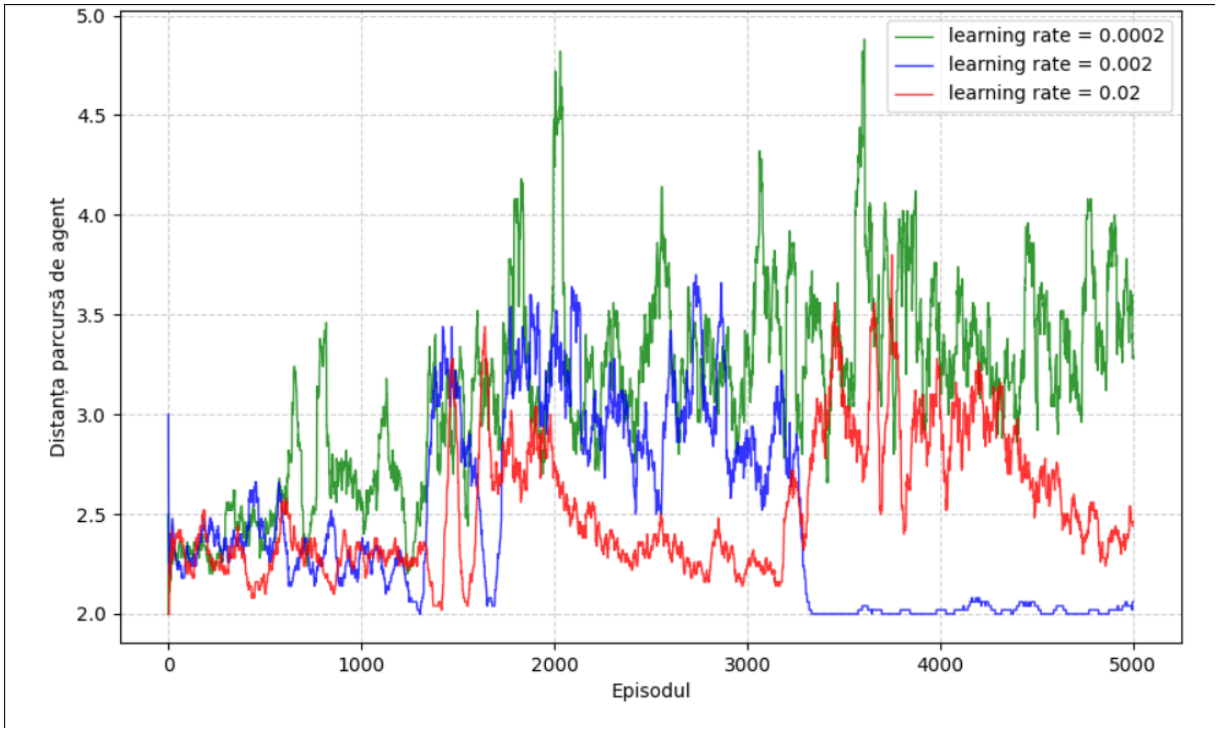

Figura 4.1: Rezultate experimentale obținute pentru algoritmul Deep Q-Learning

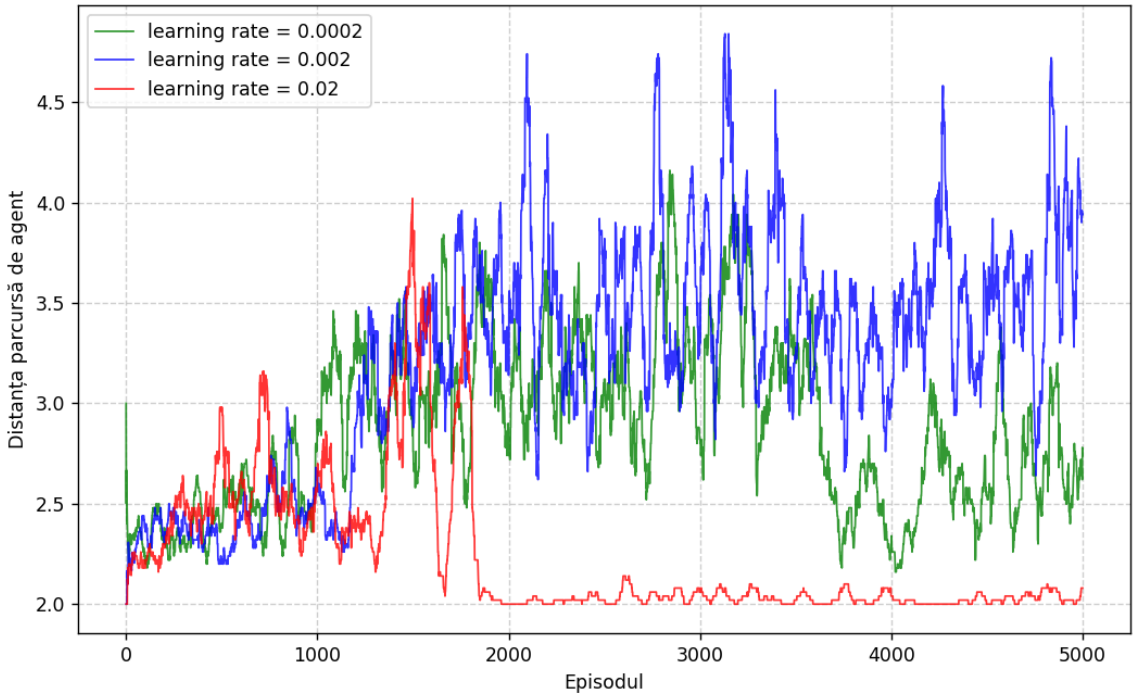

Figura 4.2: Rezultate experimentale obținute pentru algoritmul Double Deep Q-Learning

Model	Rata de învățare	Culoar maxim	Culoar minim	Culoar mediu
DQL	0,02	14	2	2.5332
DQL	0,002	24	2	2.4624
DQL	0,0002	38	2	3.1414
DDQL	0,02	27	2	2.24
DDQL	0,002	27	2	2.8724
DDQL	0,0002	41	2	3.2314

Tabela 4.1: Rezultatele experimentelor efectuate pentru Q-Learning

4.2 Algoritm genetic

Experimentele efectuate în cadrul algoritmului genetic sunt bazate pe modul de utilizare al operatorilor genetici, aceștia fiind cei care influențează în mod direct performanța algoritmului. În acest scop, tabelul 4.2 prezintă rezultatele obținute după mai multe rulări ale algoritmului pentru diferite valori ale:

- $cross\_weak$ menționat în secțiunea 3.3.1

- $cross2\_rate$ menționat în secțiunea 3.3.2

- rata de mutație

Aceste rezultate reflectă influența fiecărui operator genetic asupra apropierii de optimul problemei și viteza cu care acesta converge spre o valoare optimă.

4.2.1 Probabilitatea încrucișării cu un individ mai slab

Valoarea variabilei semnifică probabilitatea unui individ de a se încrucișa cu un individ cu fitness-ul mai slab, astfel având control asupra ratei de exploatare a soluțiilor. O valoare prea mică a lui $cross\_weak$ duce la convergența prematură a algoritmului, pe când o valoare prea mare rezultă în lipsa evoluției.

Inițial, scopul a fost de a descuraja încrucișarea cu indivizi cu fitness scăzut, motiv pentru care valoarea acestei probabilități a fost aleasă aproape de 0. După testare s-a observat lipsa diversității populației rezultată de propagarea celor mai puternice secvențe de acțiuni. Atât indivizii cu fitness slab, cât și cei cu fitness mare se încrucișau cu indivizi cu fitness mare, ceea ce ducea la exploatarea soluțiilor găsite, limitând ex-

plorarea. Astfel, se ajungea ca noile populații să fie formate, aproape în întregime, din indivizi cu aceleași secvențe de gene, ducând la convergența spre soluții suboptimale.

Pentru a evita acest comportament, valoarea $cross\_weak$ este dată de ecuația 3.3. Cum $\frac{n+1}{population\_size}$, unde n reprezintă indexul individului, ia valori între 0 și 1, graficul funcției este reprezentat în figura 4.3.

Folosirea acestei funcții asigură faptul că indivizii cu un fitness mic au șanse mai mari de a se încrucișa cu indivizii cu fitness mai mare, iar cei cu fitness mare au mai multe șanse de a se încrucișa cu cei cu un fitness mai mic. Astfel, se evită propagarea acelorași secvențe de acțiuni și oferă o șansă și indivizilor mai slabi.

Cu scopul evidențierii influenței acestui parametru asupra performanței algoritmului, a fost testat și cazul în care indivizii se încrucișează fără constrângeri legate de valoarea fitness-ului. Rezultatele acestui test sunt marcate cu valoarea *random* în tabelul final.

Performanța agentului de-a lungul generațiilor pentru toate cele trei variante testate ale valorii $cross\_weak$ sunt reprezentate în figura 4.4 . Pe baza acestui grafic, se observă faptul că valoarea fixă 0.2 obține cele mai slabe rezultate, însă încrucișarea aleatorie a indivizilor duce la convergență prematură. Pe de altă parte, obținem o performanță mai bună calculând valoarea $cross\_weak$ după formula 3.3.

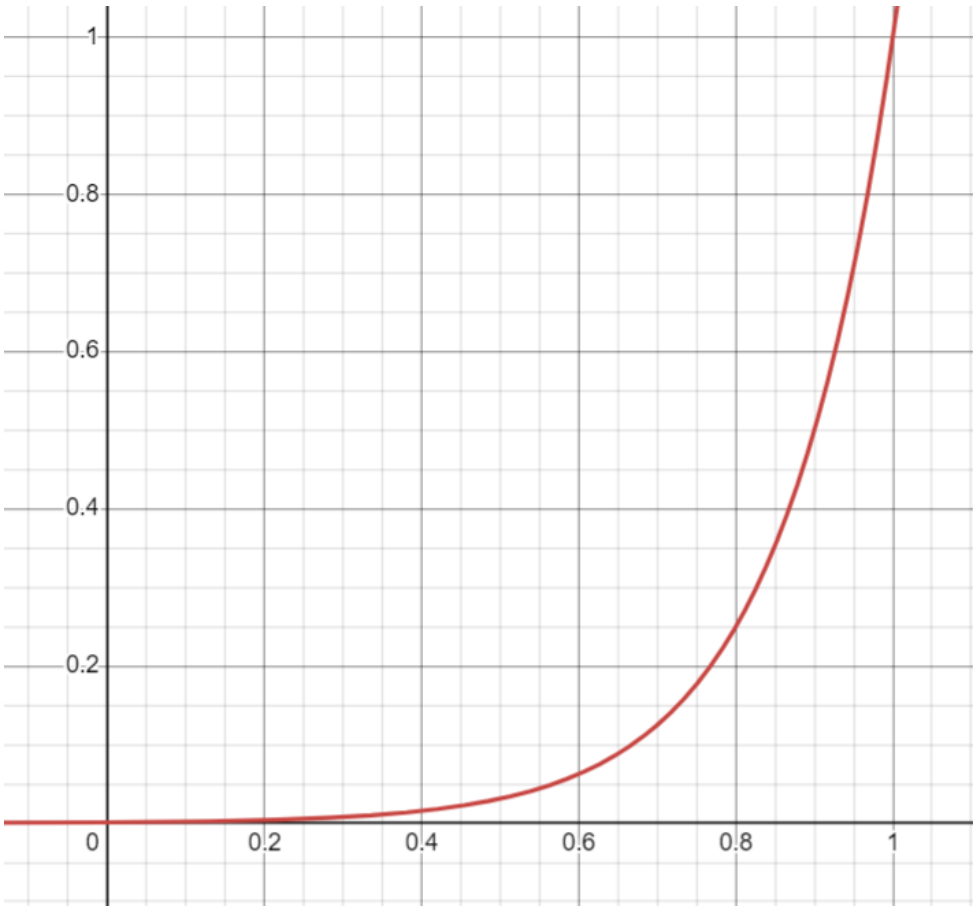

Figura 4.3: Graficul $cross\_weak$

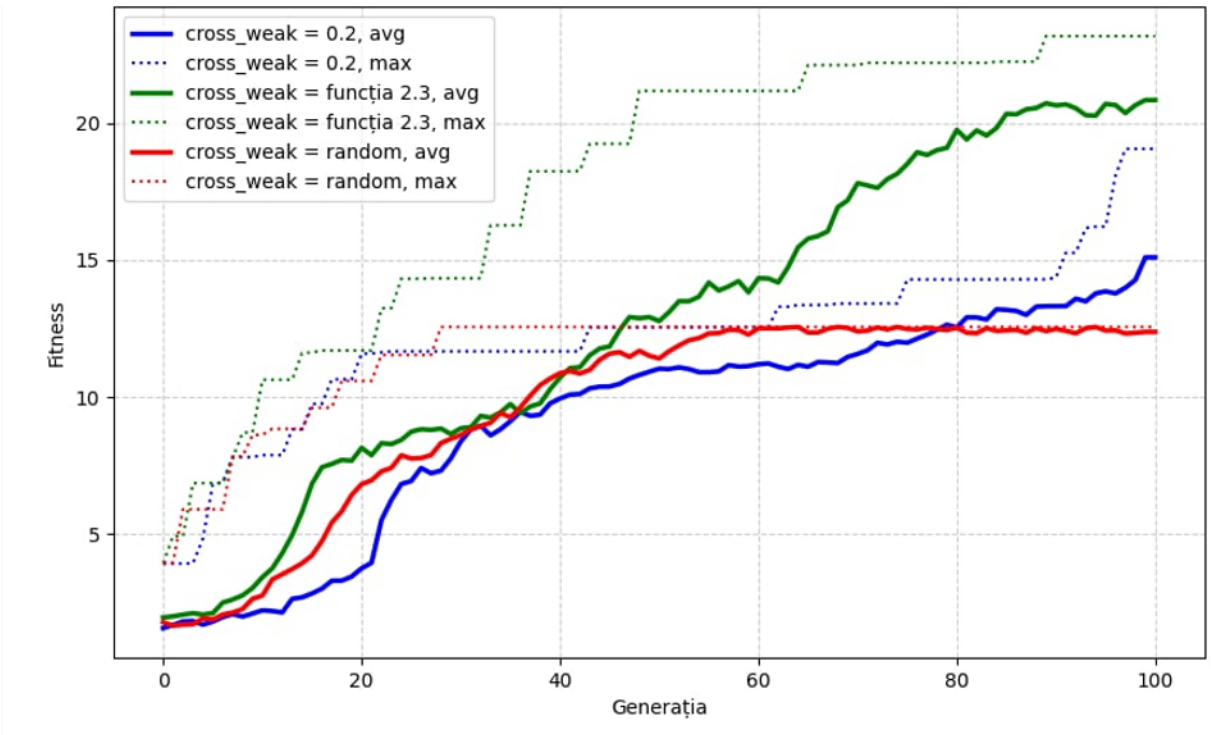

Figura 4.4: Valorile fitness obținute la fiecare generație în funcție de valoarea $cross\_weak$

4.2.2 Probabilitatea încrucișării cu două puncte de tăiere

Variabila reprezintă probabilitatea ca pe un individ să se aplice funcția de încrucișare cu două puncte de tăiere. Asemenea variabilei $cross\_weak$, și valoarea variabilei $cross2\_rate$ afectează echilibrul dintre exploatare și explorare, favorizând exploatarea.

În figura 4.5 sunt reprezentate valorile maxime și medii ale fitness-ului obținut în fiecare generație, pentru valorile 0, 0,35, respectiv 1 ale variabilei $cross2\_rate$. Pe baza acestui grafic se observă, în primul rând, lipsa de evoluție atunci când nu se aplică această funcție pe niciunul dintre indivizi. Pe de altă parte, atât din datele prezentate în tabel, cât și din reprezentarea grafică se observă faptul că aplicarea acestei funcții pe toți indivizii populației duce la obținerea unor rezultate mai bune. De asemenea, analiza figurii 4.5 evidențiază viteza de convergență în cazul celor trei valori alese, anume a faptului că aceasta crește cu cât valoarea este mai apropiată de 0.

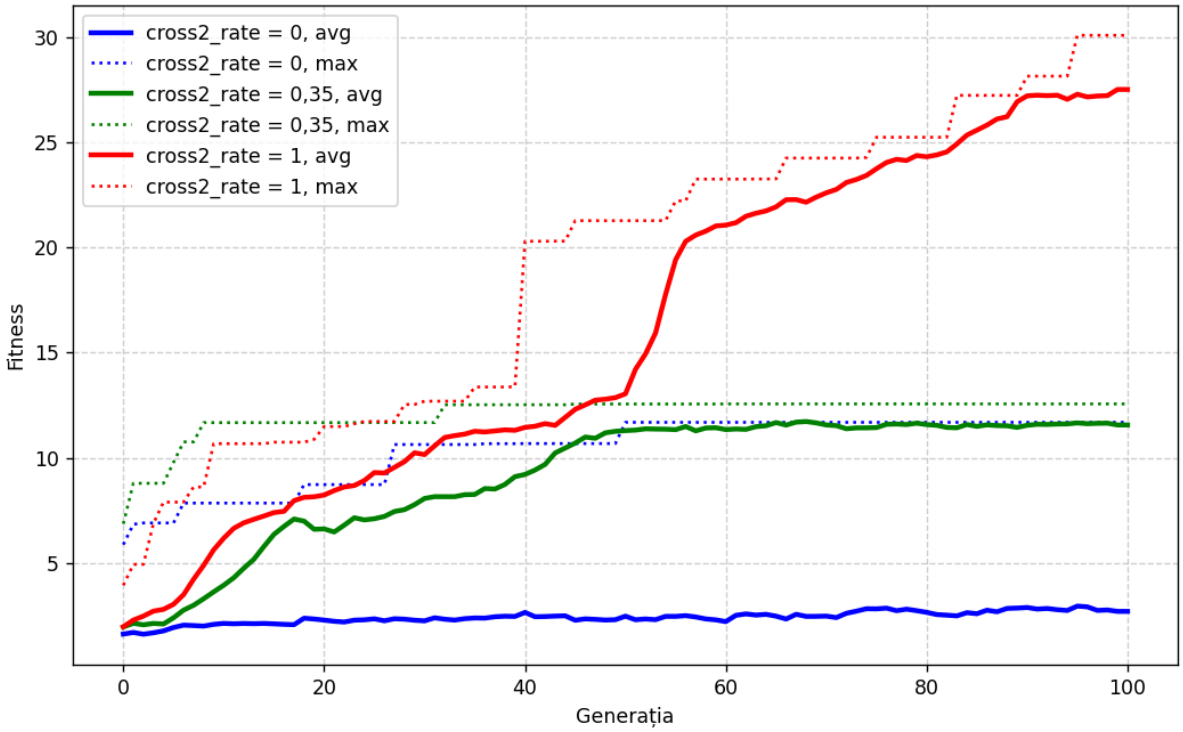

Figura 4.5: Valorile fitness obținute la fiecare generație în funcție de rata metodei 2 de încrucișare

4.2.3 Rata de mutație

Rata de mutație reprezintă componenta de explorare a algoritmului genetic. Aceasta asigură posibilitatea de a ieși dintr-un optim local. Rata mutației trebuie însă aleasă cu

atenție, deoarece o valoare prea ridicată a acesteia duce la instabilitatea algoritmului, iar lipsa acesteia poate rezulta în convergență prematură.

Pentru a stabili ce valoare duce la antrenarea optimă a agentului, a fost testată performanța algoritmului cu diferite valori ale ratei de mutație. Rezultatele obținute pe parcursul generațiilor sunt reprezentate în figura 4.6 . Analizând acest grafic, se poate ajunge la concluzia că o rată mai mare de mutație duce la rezultate mai bune, însă aduce și instabilitate algoritmului. Această instabilitate este reprezentată de existența unor fluctuații în curba care reprezintă valorile medii ale fitness-ului. Traiectoria albastră, reprezentând lipsa completă a mutațiilor, este mai lină, pe când celelalte prezintă oscilații, acestea fiind mai frecvente și mai ample cu cât valoarea este mai apropiată de 1.

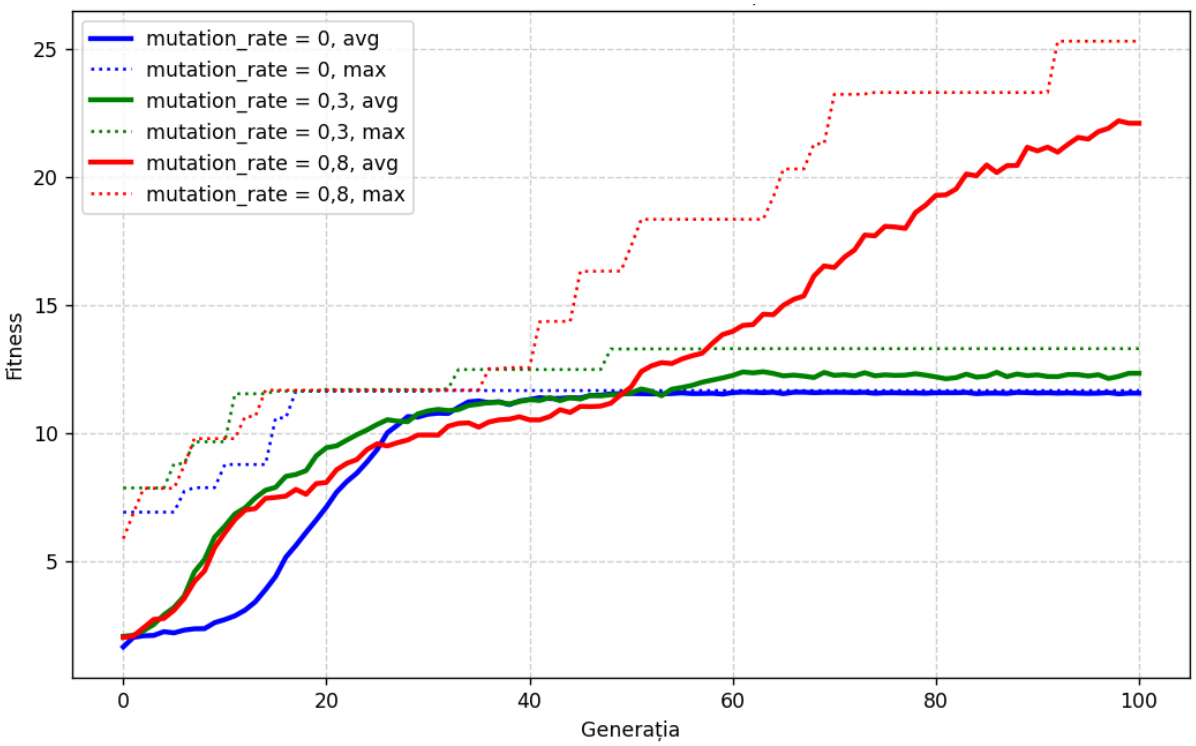

Figura 4.6: Valorile fitness obținute la fiecare generație în funcție de rata mutației

cross_weak	probabilitate crossover2	rata de mutație	fitness maxim	fitness minim	fitness mediu
0.2	0,35	0,3	22,18	12,675	16,2075
random	0,35	0,3	23,2	12,705	16,96
formula 3.3	0,35	0,3	32,18	12,52	16,5515
formula 3.3	0	0,3	13,7	11,67	12,6505
formula 3.3	1	0,3	34,15	12,51	17,9995
formula 3.3	0,35	0	13,34	7,865	11,888
formula 3.3	0,35	0,8	33,085	16,475	24,7565

Tabela 4.2: Rezultatele experimentale ale algoritmului genetic

Capitolul 5

Abordări existente

Antrenarea unui agent pentru a învăța strategii pentru un joc poate fi abordată prin mai multe metode. Această secțiune are rolul de a prezenta tehnici de învățare utilizate în literatura de specialitate și de a analiza principiile fundamentale și performanța fiecărei abordări. Scopul acestei analize este de a oferi o perspectivă mai amplă asupra metodelor de învățare existente și de a evidenția diferențele dintre acestea și metoda prezentată în această lucrare.

5.1 Deep Q-Learning

O abordare diferită a algoritmului Q-Learning este prezentată de Carter B. Burn, Frederick L. Crabbe și Rebecca Hwa în cadrul lucrării "Let's Do the Time Warp Again: Human Action Assistance for Reinforcement Learning Agents"[2] din 2021. Această lucrare prezintă rezultatele obținute de un algoritm Q-Learning asistat[6] pentru a antrena un agent să joace jocul Frogger. Autorii au dorit să investigheze diferențele dintre profesorii umani și cei digitali, dar și eficiența unui agent astfel antrenat și unul antrenat cu un algoritm Q-Learning clasic.

Algoritmii Q-Learning asistați presupun antrenarea unui agent cu ajutorul unui profesor. Acest profesor poate fi atât uman, cât și un alt program deja antrenat pentru rezolvarea problemei. Scopul profesorului este de a interveni atunci când agentul face o greșeală, pentru a-i oferi feed-back și a-l îndruma spre o soluție mai bună. Există mai multe metode prin care profesorii pot interveni, acestea fiind oferirea unei recompense drept feed-back, prezentarea unei subprobleme pe care agentul se poate antrena inițial și ajutor în alegerea unei acțiuni. Descrierea funcționalității acestora cât

și implementări existente cu profesori virtuali sunt prezentate în lucrarea menționată. Autorii au dorit să testeze efectul învățării asistate uman asupra performanței agentului. Pentru a rezolva problema timpului de reacție uman în contrast cu cel al agentului, aceștia au implementat un cadru prin care profesorul poate opri temorar antrenamentul. După oprirea antrenamentului, îndrumătorul poate lua controlul asupra agentului pentru a-l ghida spre rezultate mai bune. Cu scopul evitării ajutorului excesiv, aceștia au decis să limiteze numărul de interacțiuni pe care profesorul le poate face cu agentul. După ce profesorul a terminat procesul de îndrumare sau când procesul a dus la o stare terminală, agentul își continuă antrenamentul din punctul opririi acestui proces.

Abordarea a fost testată pe un număr de 49 profesori umani diferiți iar rezultatele arată faptul că această metodă are rezultate mai bune decât cele ale unui algoritm Q-Learning clasic. În raport cu profesorii reprezentați de programe pre-antrenate, rezultatele performanței nu diferă foarte mult, însă atenția oferită de către îndrumător este semnificativ mai scăzută în cazul celor umani. Aceste date indică faptul că algoritmul Q-Learning asistat este mult mai potrivit în cazul problemelor complexe, iar aplicarea acestuia folosind profesori umani are potențialul de a întrece performanțele agenților ajutați de alte programe pre-antrenate.

5.2 Algoritm Genetic

O altă metodă de învățare bazată pe algoritmi genetici este cea descrisă de Davis Ancona și Jake Weiner în cadrul lucrării numite "Developing Frogger Player Intelligence Using NEAT and a Score Driven Fitness Function"[1] din 2014. Scopul acesteia este de a demonstra eficiența algoritmului NeuroEvolution of Augmenting Topologies[9] (NEAT) în antrenarea unui agent să joace jocul Frogger.

În cadrul acestui algoritm, indivizii reprezintă rețele neuronale, la început simple cu aceeași arhitectură. Pe parcursul generațiilor, acestea evoluează modificând atât ponderile conexiunilor, cât și topologia rețelei prin adăugarea unor noi neuroni și conexiuni. La fel ca algoritmii genetici clasici, NEAT folosește operatori genetici pentru evoluția indivizilor, cât și o funcție fitness care determină valoarea unui individ. Un aspect important este atribuirea unui ID unic fiecărei arhitecturi noi care apare în populație. Acest concept este numit *inovație* și contribuie la gestionarea încrucișărilor, astfel încât acestea să aibă loc doar între indivizi cu aceeași topologie.

Autorii lucrării menționate testează felul în care un agent astfel antrenat se des-

curcă în diferite variații ale jocului prin conducerea a trei experimente. Aceste experimente adaugă diferite constrângeri asupra universului jocului pentru a demonstra adaptabilitatea algoritmului implementat.

Jocul, la fel ca în cadrul acestei lucrări, este reprezentat ca o grilă ce conține 5 culoare cu mașini care trebuie evitate și 5 culoare cu bușteni. În primul experiment, aceștia au ales ca bușteniii să aibă lungimea a trei segmente din lungimea grilei, însă au observat faptul că agentul învață doar parțial felul în care ar trebui navigat râul, existând spații mici prin care acesta să cadă "în apă". Astfel, au decis ca în cel de-al doilea experiment să ajusteze această mărime la un segment din grilă. Testând această abordare, au reușit să obțină rezultate favorabile care indică faptul că agentul a învățat să navigheze segmentul de râu în mod eficient. Cel de-al treilea experiment introduce oportunitatea câștigării unui număr mai mare de puncte prin plasarea aleatorie a unei insecte pe ultimul culoar. Dacă agentul ajunge pe poziția pe care se află această insectă, punctele acumulate se dublează.

Printre diferențele notabile față de abordarea prezentată în cadrul acestei lucrări, se numără spațiul acțiunilor și calcularea valorii fitness. Autorii lucrării menționate oferă agentului un spațiu de 4 acțiuni posibile, eliminând mersul înapoi, deoarece aceștia îl consideră nefavorabil. Funcția fitness din cadrul algoritmului NEAT este una exponențială, în funcție de distanța pe verticală parcursă de agent. De asemenea, agentul este echipat cu un set de senzori care au scopul de a intercepta obstacolele din apropiere. Autorii au definit senzori separați pentru identificarea mașinilor din jurul agentului, pentru a identifica existența bușteniilor în fața lui și spațiul de deplasare, cât și pentru a intercepta insecta dacă aceasta se află într-o rază de trei segmente de grilă.

Prin aplicarea acestui algoritm, autorii au obținut rezultate favorabile în urma conducerii repetate ale celor trei experimente. Aceste rezultate au demonstrat faptul că algoritmul NEAT este eficient în antrenarea unui agent în cadrul unui mediu cu un spațiu mare de stări.

Concluzii

Luând în considerare rezultatele experimentale obținute pentru algoritmii de tip Q-Learning implementați, putem spune că aceștia, deși foarte potriviți pentru antrenarea unui agent. Testarea a arătat faptul că agenții obțin rezultate cu atât mai bune, cu cât numărul episoadelor este mai mare. De asemenea, putem concluziona faptul că agenții antrenați cu ajutorul acestor algoritmi au reușit să învețe, într-o oarecare măsură, strategii potrivite pentru câștigarea jocului Frogger.

Având în vedere faptul că algoritmul genetic implementat obține rezultate doar pe testarea pe același *seed* de generare a obstacolelor, putem spune că acesta a reușit să antreneze agentul pentru rezolvarea unei instanțe. Algoritmul genetic a obținut rezultate foarte bune, iar testarea a arătat faptul că potențialul agentului de a găsi strategia optimă este limitată de numărul genelor unui individ.

În concluzie, ambii algoritmi din domeniul inteligenței artificiale au reușit să antreneze cu succes un agent pentru a câștiga jocul Frogger. În cazul algoritmului genetic există abordări mai potrivite care ar putea duce la rezultate mai bune pe o generare aleatorie a obstacolelor.

Bibliografie

[1] Davis Ancona and Jake Weiner. Developing frogger player intelligence using neat and a score driven fitness function. Online, 2014. URL: https://www.cs.swarthmore.edu/~meeden/cs81/s14/papers/DavisJake.pdf.

[2] Burn C., Crabbe F., and Hwa R. Let's do the time warp again: Human action assistance for reinforcement learning agents. Online, 2021. doi:10.5220/0010258700920100.

[3] Leonid Datta. A survey on activation functions and their relation with xavier and he normal initialization. 2020. URL: https://doi.org/10.48550/arXiv.2004.06632, arXiv:2004.06632.

[4] Konrad Kording David Rolnick. Reverse-engineering deep relu networks. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 8178–8187. PMLR, 2020. URL: https://proceedings.mlr.press/v119/rolnick20a.html.

[5] Universitatea "Alexandru Ioan Cuza" Iași Facultatea de informatică. Cursuri inteligență artificială. *Reinforcement Learning II*, 2024. URL: https://sites.google.com/view/iafii/home?authuser=0.

[6] Torrey L. and Taylor M. Teaching on a budget: Agents advising agents in reinforcement learning. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, page 1053–1060, 2013.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013. URL: https://doi.org/10.48550/arXiv.1312.5602.

[8] Sanskriti Singh. How are neural networks used in deep q-learning? URL: https://www.turing.com/kb/how-are-neural-networks-used-in-deep-q-learning.

[9] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. doi:10.1162/106365602320169811.

[10] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2015. URL: https://arxiv.org/abs/1509.06461, arXiv:1509.06461.