UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Reinforcement learning pentru jocul Pong

propusă de

Moisă Eduard Gabriel

Sesiunea: *iulie, 2019*

Coordonator științific

Conf. Dr. Mădălina Răschip

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

Reinforcement learning pentru jocul Pong

Moisă Eduard Gabriel

Sesiunea: *iulie, 2019*

Coordonator științific

Conf. Dr. Mădălina Răschip

Avizat,

Îndrumător Lucrare de Licenţă

Titlul, Numele şi prenumele _____

Data _____ Semnătura _____

DECLARAŢIE privind originalitatea conţinutului lucrării de licenţă

Subsemntatul(a) ……

domiciliul în ……..

născut(ă) la data de ……., identificat prin CNP ……., absolvent(a) al(a) Universităţii „Alexandru Ioan Cuza" din Iaşi, Facultatea de …… specializarea ……., promoţia ……., declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul: _____

_____

_____elaborată sub îndrumarea dl. / d-na _____, pe care urmează să o susţină în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării fasificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie răspundere că lucrarea de faţă nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, …… Semnătură student ……

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul „*Reinforcement learning pentru jocul Pong*”, codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc 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 Iaşi, să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi,

Absolvent *Moisă Eduard Gabriel*

_____

Cuprins

Abstract

În această lucrare vom folosi o rețea neuronală multi-strat pentru a îmbunătăți o politică (tactică) folosind *reinforcement learning*. *Reinforcement learning* este una dintre multele metode din domeniul învățării automate. Această metodă poate fi utilizată într-o varietate de domenii. În cazul de față este vorba despre strategii ale jocurilor, în special pentru jocul Pong. Fiind dat un *frame*, folosim rețeaua neuronală antrenată cu ajutorul metodei *Policy Gradients* pentru a aproxima recompensele viitoare pentru toate acțiunile posibile, efectuând acțiunea care conduce la cel mai bun rezultat. După un timp de antrenare suficient, agentul ajunge să se comporte mai bine decât un jucător uman.

1. Introducere

1. 1. Descriere joc

Pong este unul dintre primele jocuri pe calculator care au fost create vreodată. Acest joc simplu ce simulează un joc de tenis sau tenis de masă constă în două palete și o bilă. Scopul este să ajungi la 21 de puncte înaintea adversarului. Un jucător primește un punct când celălalt ratează bila. Jocul a fost dezvoltat original de către Allan Alcorn și lansat în anul 1972 de către Atari corporations. Astăzi, Pong este considerat jocul care a pornit industria jocurilor video și este dovada că piața jocurilor video poate produce un venit semnificativ având în vedere că în anul 1975 Atari a lansat o varianta *home edition* ce s-a vândut în peste 150,000 de exemplare.

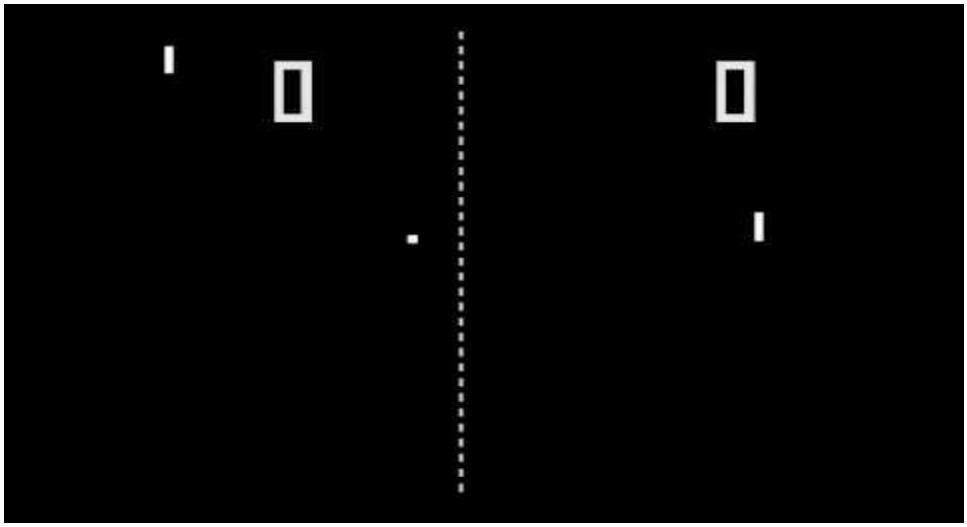


Fig. 1 – Interfața grafică a jocului Pong

În lucrarea prezentă jocul se va desfășura între jucătorul implementat de cei de la OpenAI și un agent antrenat cu ajutorul unei rețele neuronale prin *Reinforcement learning* și *Policy gradients.*

2. *Reinforcement learning*

2. 1. Rețele neuronale

Structural, o rețea neuronală din punct de vedere biologic reprezintă o serie de neuroni interconectați care formează un tot unitar.

Funcțional, aceasta se poate defini ca o mulțime de elemente interconectate (adesea de dimensiuni mari) ce conlucrează pentru rezolvarea unei probleme. Cea mai importantă caracteristică a acestor rețele este faptul că pot fi îmbunătățite prin învățare.

Câtea dintre multele aplicații care folosesc rețele neuronale sunt:

- Recunoașterea scrisului de mână;
- Google, pe baza unor cuvinte date ca intrare la căutare furnizează imagini cu ajutorul unui algoritm bazat pe rețele neuronale;
- Microsoft a dezvoltat un sistem prin care limba engleza poate fi tradusă in chineza;
- Largă aplicabilitate în afaceri, matematică, medicină, știință etc;

2. 1. 1. Neuronul artificial

Primul model matematic al unui neuron a fost propus de McCulloch și Pitts (1943). [1]

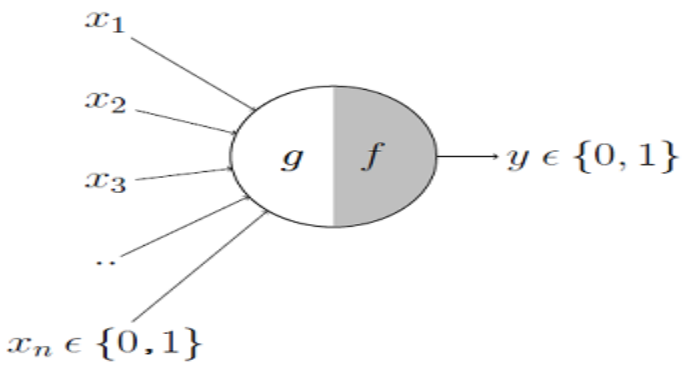


Fig. 2 - Primul model matematic al unui neuron

Poate fi împărțit în două părți. În prima parte g primește o intrare $x_1, x_2 \dots x_n$ pe care o calculează, iar în a doua parte bazat pe rezultat, f ia o decizie.

Exemple:

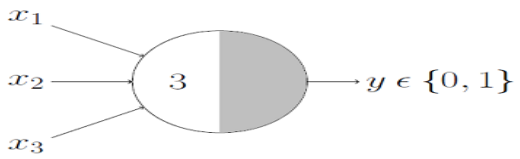


Fig. 3 - Funcția AND

Funcția *AND* va fi activată numai dacă toți neuronii sunt 1, $g(x) \geq 3$ în cazul de față.

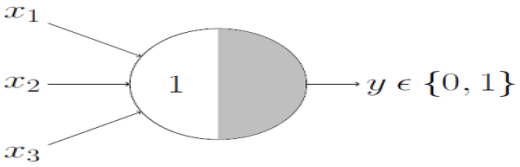


Fig. 4 - Funcția OR

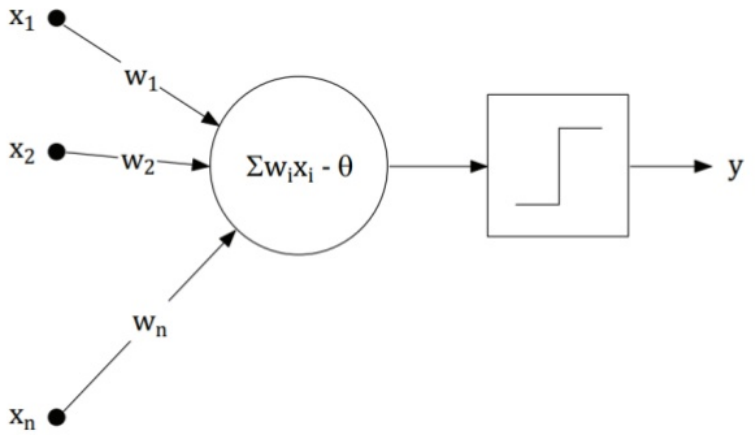
Funcția *OR*, activată dacă oricare din neuroni este 1, $g(x) \geq 1$ aici.

2. 1. 2. Perceptronul originar

În 1958 Rosenblatt, încercând să rezolve problema învățării, a propus un model de neuron numit perceptron, prin analogie cu sistemul vizual uman.

2. 1. 3. Perceptronul standard

Semnalele de intrare sunt însumate, iar neuronul generează un semnal doar dacă suma depașește pragul.



$$y = F\left(\sum_{i=1}^{n} w_i x_i - \theta\right)$$

Perceptronul are ponderi sinaptice ajustabile și funcție de activare semn sau treaptă.

$$F(a) = \begin{cases} -1, & dacă\ a < 0 \\ 1, & dacă\ a \geq 0 \end{cases} \qquad F(a) = \begin{cases} 0, & dacă\ a < 0 \\ 1, & dacă\ a \geq 0 \end{cases}$$

Procesul învatarii într-un perceptron:

- Învățarea are loc prin ajustarea succesivă a ponderilor pentru a reduce diferența dintre ieșirile reale și ieșirile dorite, pentru toate datele de antrenare.
- Dacă la iterația p ieșirea reală este Y(p) iar ieșirea dorită este Y_d (p), atunci eroarea este: e(p) = Y_d(p) – Y(p)
- Iterația p se referă la vectorul de antrenare cu indexul p
- Dacă eroarea este pozitivă, trebuie să creștem ieșirea perceptronului Y(p)
- Dacă eroarea este negativă, trebuie să micșoram ieșirea Y(p)

2. 1. 4. Perceptronul multistrat

- Perceptronul multistrat este o rețea neuronală de tip feed-forward cu unul sau mai multe straturi ascunse
    - Un strat de intrare
    - Unul sau mai multe straturi ascunse/ intermediare
    - Un strat de ieșire
- Calculele se realizează numai în neuronii din straturile ascunse și din stratul de ieșire
- Semnalele de intrare sunt propagate înainte succesiv prin straturile rețelei

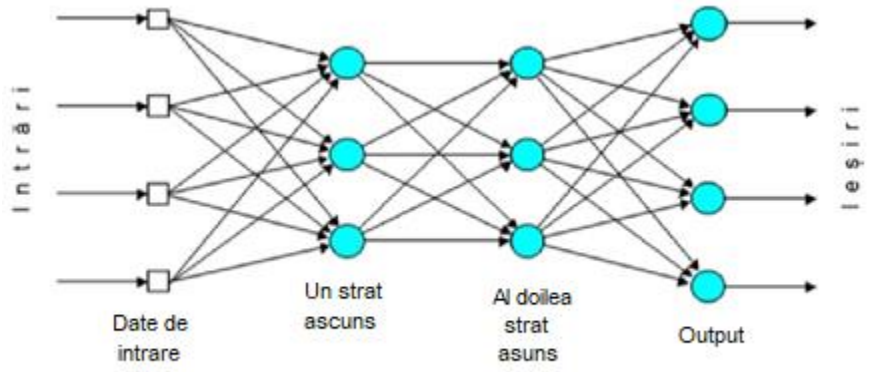


Fig. 5 - Perceptronul multistrat

Procesul învățării într-un perceptron multistrat:

- O rețea multistrat învață într-un mod asemănător cu perceptronul
- Rețeaua primește vectorii de intrare și calculează vectorii de ieșire
- Dacă există o eroare (o diferență între ieșirea reală și ieșirea dorită), ponderile sunt ajustate pentru a reduce eroarea

2. 1. 5. Algoritmul *backpropagation*

Algoritmul *backpropagation* se bazează pe minimizarea diferenței dintre ieșirea dorită și ieșirea reală, prin metoda gradient descent. Acesta a fost lansat la mijlocul anilor ’80 de către Rumelhart, Hinton și Williams (1986) ca instrument general de antrenare a perceptronului multistrat. Ideea de bază este găsirea minimului funcției de eroare în raport cu ponderile conexiunilor.

Algoritmul are două faze:

- Rețeaua primește vectorul de intrare și propagă semnalul înainte, strat cu strat, până se generează ieșirea
- Semnalul de eroare este propagat înapoi, de la stratul de ieșire către stratul de intrare, ajustandu-se ponderile rețelei

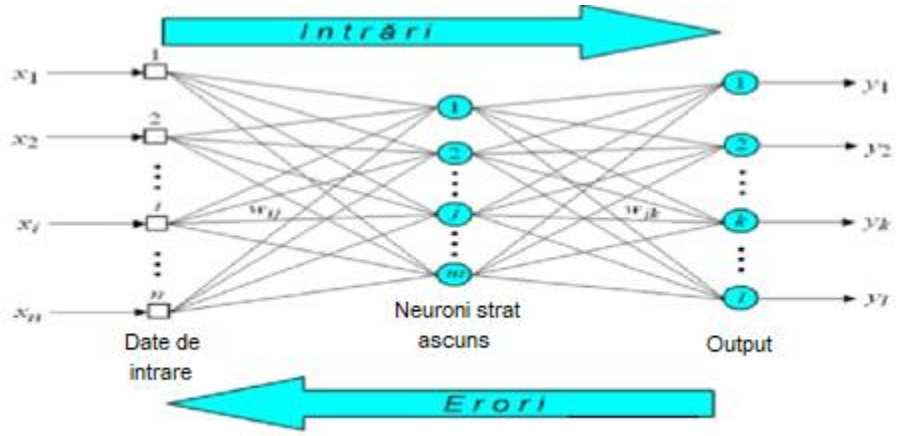


Fig. 6 - Backpropagation

- Ponderile și pragurile se initializeaza cu valori aleatorii mici, dar diferite de 0.
- Inițializarea ponderilor se face în mod independent pentru fiecare neuron.

2. 1. 6. Tipuri de antrenare

- *Online learning*
  - Ponderile se actualizează după prelucrarea fiecărui vector de antrenare
- *Batch learning*
  - După prelucrarea unui vector de antrenare se acumulează gradienții de eroare în corecțiile ponderilor Δw
  - Ponderile se actualizează o singură dată la sfârșitul unei epoci (după prezentarea tuturor vectorilor de antrenare)
  - Avantaj: rezultatele antrenării nu depind de ordinea în care sunt prezentați vectorii de antrenare

2. 1. 7. Funcţii de activare

Funcţiile de activare (transfer) transformă semnalul de activare a reţelei în semnal de ieşire. În funcţie de caracteristicile dorite ale semnalelor de ieşire, se pot folosi anumite tipuri de funcţii de activare:

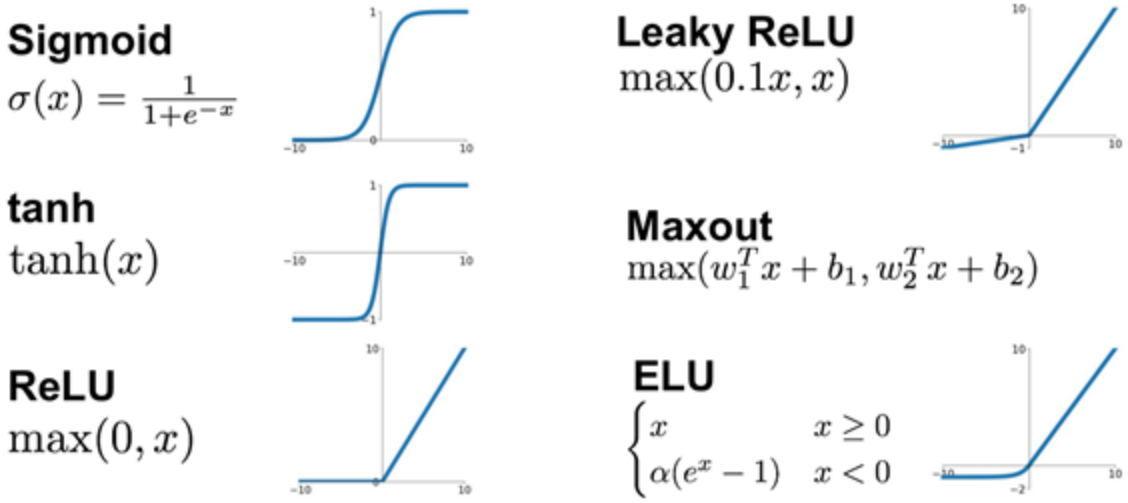


Fig. 7 - Funcții de activare[1]

2. 2. *Reinforcement learning*

2. 2. 1. *Markov Decision Processes* (MDPs)

Markov Decision Processes [2] trebuie să satisfacă proprietatea Markov (*Markov Property*): Având în vedere prezentul, stările viitoare sunt independente de cele din trecut.

Proprietate: Starea S_t este Markov daca si numai daca: $P[S_{t+1}|S_t] = P[S_{t+1}|S_1, \dots S_t]$

Putem defini toate tranzițiile stărilor cu ajutorul unei matrici de tranziție P ale cărei linii vor conține probabilitățile de a ajunge dintr-o anumită stare în toate stările succesoare ei.

$$P = \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{bmatrix}$$

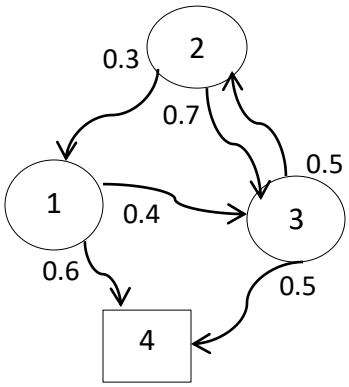
Primul și cel mai simplu MDP este Procesul Markov / Lanț Markov.

[1] https://medium.com/@krishnakalyan3/introduction-to-exponential-linear-unit-d3e2904b366c

2. 2. 1. 1. Procesul Markov / Lanț Markov

Un proces Markov reprezintă o secvență de stări aleatoare S_1, S_2 ... care îndeplinesc proprietatea Markov.

Mai jos avem o ilustrare a unui lanț Markov în care fiecare nod reprezintă o stare cu o probabilitate de tranziție de la o stare la alta, unde 4 este stare terminală.



Acest lanț are următoarea matrice de tranzitie P:

P =

	1	2	3	4
1			0.4	0.6
2	0.3		0.7	
3	0.5			0.5
4				

Pentru fiecare stare, suma probabilităților tranzițiilor este 1.

2. 2. 1. 2. *Markov Reward Process (MRP)*

MRP este un lanț Markov cu recompense.

Un MRP este definit ca un tuplu de forma $\langle S, P, R, \gamma \rangle$ unde:

S este un set finit de stări;

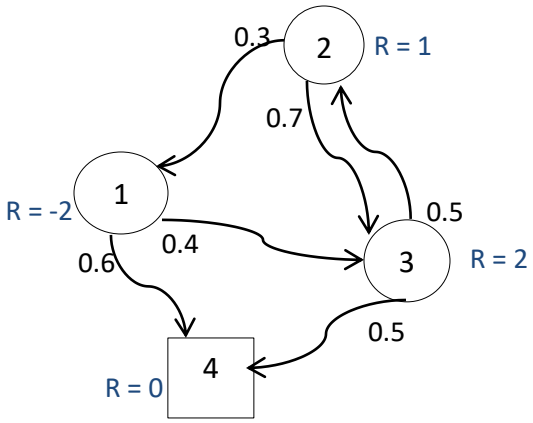
P este o matrice de tranziție, $P_{ss'}^{a} = P[S_{t+1} = s' | S_t = s]$;

R este funcția de recompensă, $R_s = [R_{t+1}|S_t = s;$

γ se numește *discount factor*, $\gamma \in [0,1]$.

Scopul este să maximizăm recompensa pe termen lung, G_t, unde:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$



Având în vedere faptul că într-un MRP nu avem nicio acțiune de luat, G_t este calculat efectuând tranziții în mod aleator.

2. 2. 1. 3. *Markov Decision Process (MDP)*

Este un MRP cu decizii. Este un mediu în care toate stările sunt Markov.

MDP = $\langle S, A, P, R, \gamma \rangle$ unde:

S este un set finit de stări; A este un set finit de acțiuni;

P este o matrice de tranziție, $P_{ss}^a, = P[S_{t+1} = s'|S_t = s, A_t = a];$

R este o funcție de recompensă, $R_s^a = E[R_{t+1}|S_t = s, A_t = a];$

γ se numește *discount factor*, $\gamma \in [0,1]$.

2. 2. 2. *Reinforcement Learning*

Este învățarea a ceea ce trebuie să facem – cum să facem legătură între situații și acțiuni – astfel încât să maximizăm o recompensă. Celui ce învață nu i se spune ce acțiuni anume să facă, ca în majoritatea formelor de învățare a mașinilor, ci trebuie să descopere ce acțiuni îi oferă „recompensa" cea mai mare prin încercarea acestora.

Reinforcement learning folosește arhitectura formală a MDPs (*Markov Decision Processes*) pentru a defini interacțiunea dintre un agent și un mediu în termeni de stări, acțiuni și recompense.

Arhitectura este destinată în special pentru a reprezenta caracteristici esențiale din domeniul inteligentei artificiale.

- Agentul
  - Efectuează acțiuni
- Mediul
  - Acordă recompense
  - Îi prezintă agentului situații numite stări

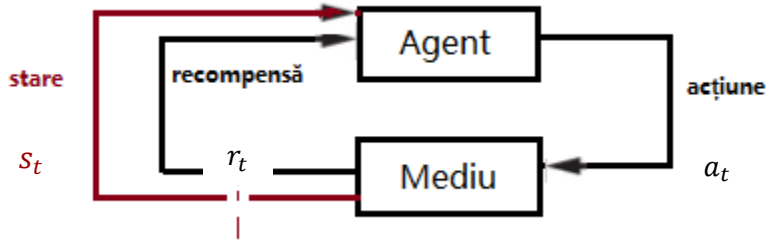


Fig. 8 - Ilustrare algoritm *Reinforcement Learning*

Un agent mai poate conține și una sau mai multe componente:

- Politica: definește comportamentul agentului.
- Funcție valoare (*value function*): definește cât de bună este fiecare stare și/ sau acțiune.
- Model: reprezentarea agentului pentru mediu.

Politica definește comportamentul agentului și este o asociere de tip stare-acțiune. Aceasta este de două feluri:

- Politică deterministă : $\pi(s) = a$
- Politică stocastică : $\pi(a|s) = P[A = a|S = s]$ (probabilitatea de a efectua acțiunea a în starea s.

Funcția valoare este o predicție a viitoarelor recompense și este folosită la evaluarea stărilor și prin urmare la alegerea dintre acțiuni:

$v_\pi(s) = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots | S_t = \text{s}]$ (cu ajutorul funcției valoare v pentru politica π putem estima din starea în care ne aflăm care va fi suma recompenselor viitoare, unde γ se numește *discount factor* ($\gamma \in [0, 1]$).

Modelul prezice ce se va întâmpla în continuare în mediu luând în considerare viitoarele situații înainte că ele să se întâmple.

Metodele care rezolvă probleme ce implică *reinforcement learning* cu ajutorul unor modele și planificări se numesc *model-based methods*, în opoziție cu *model-free methods* în care baza învățării este explicită prin metoda *trial-and-errors*.

Pentru a rezolva o problemă ce implică *reinforcement learning*, trebuie să găsim o politică care acumulează foarte multe recompense pozitive pe termen lung. Pentru MDPs (*Markov Decision Processes*) finite, putem să definim precis o politică optimă în felul următor. Funcțiile valoare definesc o ordine parțială peste politici. O politică π este definită ca fiind mai eficientă sau egală decât o politică π' dacă recompensa pe termen lung obținută de către π este mai mare sau egală pentru toate stările, S. Cu alte cuvinte, $\pi \geq \pi'$ dacă si numai dacă $v_\pi(s) \geq v_{\pi'}(s)$ pentru toate $s \in S$.

Tot timpul va exista o politică care este mai bună sau egală cu o alta. Aceasta este o politică optimă. Deși s-ar putea să existe mai multe politici optime, notăm politica optimă cu π_*. Ele împart aceeași funcție de tip stare-valoare, numită *optimal state-value function*, notată v_* și definită prin:

$v_*(s) = max_\pi v_\pi(s)$, pentru toate $s \in S$.

Politicile optime împart totodată și aceeași funcție optimă de tip acțiune-valoare, notată q_* definită astfel:

$q_*(s, a) = max_\pi q_\pi(s, a)$, pentru toate $s \in S \; si \; a \in A(s)$

Pentru perechea de tip stare-acțiune (s, a), această funcție oferă recompensa viitoare pentru efectuarea acțiunii a în starea s, după care urmează politica optimă.

Una din provocările întâlnite în *reinforcement learning* și care nu este prezentă în alt tip de învățare este problema explorării și exploatării. Pentru a obține multe recompense pozitive, un agent de tip RL trebuie să prefere acțiuni pe care le-a încercat în trecut și care s-au dovedit a fi eficiente în obținerea recompensei. Dar pentru a descoperi astfel de acțiuni trebuie să execute mișcări pe care nu le-a mai realizat până atunci. Agentul trebuie să exploateze experiențele pe care le deține deja și care au adus recompense, dar în același timp trebuie să exploreze pentru a alege acțiuni și mai bune în viitor. Dilema este că niciuna dintre ele nu pot fi cercetate fară a le încerca. Agentul trebuie să încerce o varietate de acțiuni și progresiv să le favorizeze pe cele care se dovedesc ca fiind cele mai bune. Într-o sarcină stocastică, fiecare acțiune trebuie încercată de mai multe ori pentru a aproxima viitoarea recompensă.

Dilema explorare-exploatare a fost intens studiată de matematicieni de-a lungul anilor, rămânând totuși nerezolvată.

2. 2. 3 *Reinforcement learning* și *Policy gradients*

Avem politica π cu parametrul θ.

$\pi_\theta(a|s) = P[a|s]$ => Probabilitatea de a efectua acțiunea a in starea s cu parametrii θ.

Cum știm dacă politica noastra este bună?

Trebuie să găsim cei mai buni parametri θ pentru a maximiza o funcție de scor, $J(\theta)$.

$$J(\theta) = E_{\pi\theta}[\sum \gamma r]$$

Ideea este că $J(\theta)$ ne va spune cât de bună este politica, iar gradientul ne va ajuta să gasim cei mai buni parametri pentru politica noastră pentru a maximiza alegerea acțiunilor bune.

Acum că avem funcția de scor care ne spune cât de bună este politica, trebuie să gasim parametrii θ care maximizează această funcție.

Pentru a realiza asta, trebuie să efectuăm *gradient ascent* pe parametrii politicii.

Gradient ascent este inversul metodei *gradient descent.* De ce abordăm problema in felul acesta? Pentru că folosim metoda *gradient descent* atunci cand vrem să minimizăm o funcție de eroare.

Dar noi având o funcție de scor și pentru că dorim să maximizăm scorul, avem nevoie de *gradient ascent* [3].

Politica : π_θ
Funcția de scor: $J(\theta)$
Gradient: $\nabla_\theta J(\theta)$
Actualizare parametri: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

Vrem să gasim cei mai buni parametri θ^*, care maximizează scorul:

$$\theta^* = argmax_\theta \, E_{\pi\theta}[\sum_t R(s_{t,} a_t)]$$

Unde $E_{\pi\theta}[\sum_t R(s_{t,} a_t)] = J(\theta) = E_\pi[R(\tau)]$, τ – tuplu de stări, acțiuni, recompense (s_0, a_0, r_0) (s_1, a_1, r_1)
În continuare folosim *Policy gradient theorem*:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau \pi(\tau;\theta) R(\tau)$$

$$= \sum_\tau \nabla_\theta \pi(\tau;\theta) R(\tau) \;\; (am\ introdus\ gradientul\ \text{în}\ sum\text{ă})$$

$$\nabla_\theta \pi(\tau;\theta) = \pi(\tau;\theta) * \frac{\nabla_\theta \pi(\tau;\theta)}{\pi(\tau;\theta)}$$

$$\nabla \log x = \frac{\nabla x}{x}$$

$$\Rightarrow \sum_\tau \pi(\tau;\theta) \, \nabla_\theta (\log \pi(\tau;\theta)) R(\tau)$$

$$\nabla_\theta J(\theta) = E_\pi[\, \nabla_\theta (\log \pi(\tau;\theta)) R(\tau)]$$

Policy gradient: $E_\pi[\, \nabla_\theta (\log \pi(s, a, \theta)) R(\tau)]$

Regula de actualizare a parametrilor: $\Delta\theta = \alpha * \nabla_\theta(\log\pi(s,a,\theta))R(\tau)$, unde α este rata de invățare iar $\Delta\theta$ este schimbarea facută parametrilor.

2. 2. 4. Alte strategii existente

Policy Gradients nu este singura metodă care poate rezolva astfel de probleme.

2. 2. 4. 1. *Deep-Q-Networks (DQN)*

Deep Learning + Reinforcement Learning = Deep-Q-Networks

Ce este *Deep Learning*?

Să presupunem că deja cunoaștem recompensele viitoare pentru orice stare. Prin urmare, agentul ar ști întotdeauna ce acțiune să efectueze.

Va efectua seria de acțiuni care va duce în final la maximizarea totalului recompenselor viitoare. Acest total (maximum total reward) poartă denumirea și de *Q-value*:

$$Q(s,a) = r(s,a) + \gamma max_a Q(s',a)$$

În ecuația precedentă *Q-value* este egal cu recompensa imediată $r(s,a)$, obținută în starea s, efectuând acțiunea a + cea mai mare valoare posibilă (*Q-value*) din starea următoare. *Gamma* controlează contribuția recompenselor în viitor și poartă numele de *discount factor*.

De ce să folosim *deep-q-learning*?

Q-learning este un algoritm simplu, dar destul de puternic pentru a ști exact ce acțiune să efectueze un agent la un moment dat.

Dar dacă avem mai multe date? Să ne imaginăm un model cu 15,000 de stări și 1000 de acțiuni pe stare. Ar trebui creat un tabel de 15,000,000 de celule.

Este evident că nu putem să ne dăm seama de funcția *Q-value* potrivită pentru stări noi știind stările vechi. Există două probleme:

- Memoria necesară pentru a salva și actualiza tabelul ar crește odată cu numărul stărilor
- Timpul ar fi exagerat de mare pentru a crea respectivul tabel

În *deep-q-networks* [4] este folosită o rețea neuronală pentru a aproxima funcția Q-value. Starea reprezintă datele de intrare, iar *Q-value* pentru toate acțiunile reprezintă ieșirea.

În imaginea de mai jos putem observa diferența dintre cele două metode:

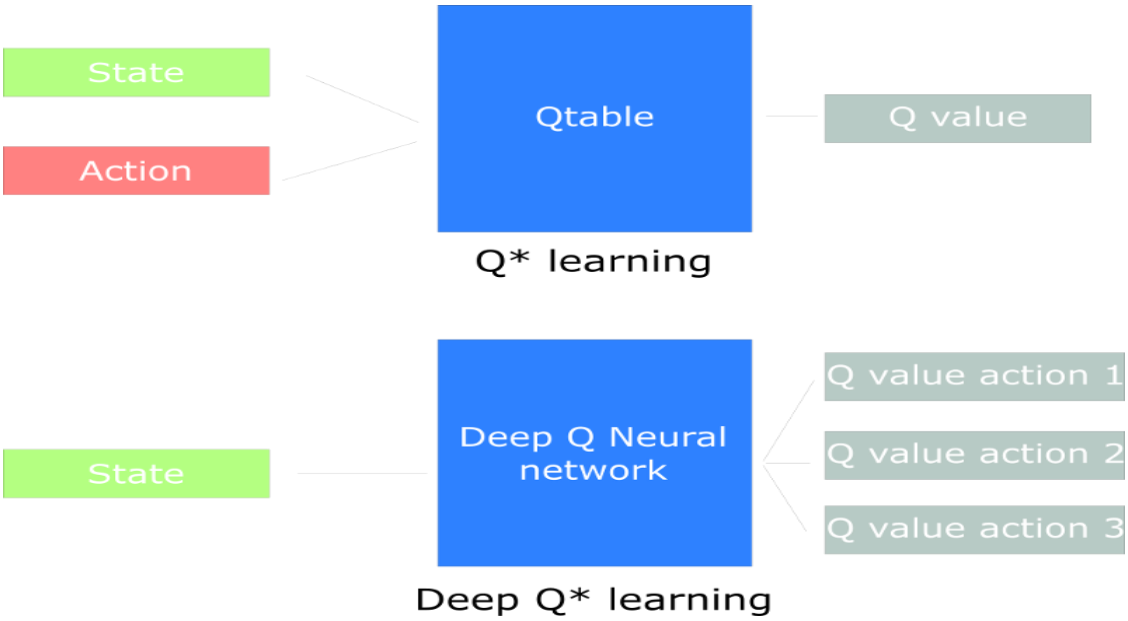


Fig. 9 - Ilustrare metode *Qtable* și *Deep Q Networks*[2]

Rețelele de tip *deep-q-networks [4]* sunt diferențiate de celelalte rețele neuronale uzuale cu un singur strat ascuns datorită adâncimii lor. Versiuni anterioare de rețele neuronale (primii perceptroni) erau superficiale, compuse din date de intrare, o ieșire și maxim un strat ascuns între ele. Mai mult de trei straturi (incluzând datele de intrare și ieșirea) califică o rețea neuronală ca fiind *deep q network*. Așadar termenul *deep* este folosit pentru a defini rețele cu mai mult de un strat ascuns.

În DQN, fiecare strat de noduri se antrenează pe un set distinct de date, acestea reprezentând ieșirea stratului anterior. Cu cât avansăm mai mult în rețeaua neuronală, aceasta este capabilă de mai mult având în vedere că recombină informațiile anterioare și le finisează.

[2] https://sergioskar.github.io/Deep_Q_Learning/

În loc să memorăm câte o asociere de tipul (stare-acțiune) diferită pentru fiecare combinație de pixeli de pe ecran, folosim rețele convoluționale (*ConvNets*) pentru ca aceste asocieri să fie memorate pentru combinații similare.

ConvNets poate indica agentului faptul că o poziție (p1) este în esență aceeași cu o altă poziție (p2) și să efectueze acțiunea pe care a ales-o și în p2.

Fiecare experiență (constând în starea curentă, acțiune, recompensă, starea următoare) este memorată în ceea ce numim *experience replay memory* (ERM).

Avantajul acestei metode este următorul:

Atunci când învățam din experiențe imediate (*policy learning*), parametrii curenți determină următoarea mulțime de date pe care vom efectua antrenarea. Acest comportament poate conduce la blocarea într-un minim local.

Folosind ERM, experiențele folosite pentru a antrena rețeaua provin din diferite momente. Acest lucru facilitează învățarea și evită eșecul.

2. 2. 4. 2. O altă abordare a jocului Pong cu *Reinforcement Learning* folosind un alt model de recompense

În abordarea folosită de mine, recompensa este +1 sau -1. +1 Dacă agentul trimite bila după oponent și -1 dacă mingea depășește paleta agentului. Având în vedere că un jucător are nevoie de 21 de puncte pentru a câștiga, agentul fiind la început va avea destul de des un scor total aproape de -21. Antrenarea constă în maximizarea acestor recompense. Dacă totalul ajunge să fie pozitiv înseamnă că agentul nostru a castigat. (Exemplu: un meci se termină 21 la 16 pentru agentul antrenat de noi, atunci vom avea 21 (recompense de tip +1) și -16 (recompense de tip -1); însumate acestea vor rezulta într-un scor total de 5.)

Aceasta nu este singura variantă.

Dacă mai sus încercam să maximizam acel scor, o altă abordare ar fi să ne ghidăm după pozitia paletei.

Recompensele pot consta în:

- Agentul primește o recompensă negativă proporțională cu distanta de la mijlocul paletei până la mijlocul bilei în momentul în care bila trece de paletă.
- Agentul primește o recompensă de 10 dacă nimerește bila.
- Agentul primește o recompensă de 0 în orice alt caz.

În cazul de față, maximizarea recompensei reprezintă minimizarea distanței dintre paletă și bilă.

Dacă mingea este aproape de paletă, recompensa este mare, iar dacă este departe, recompensa este mică.

2. 2. 4. 3. *FRIQ-learning (Fuzzy Rule Interpolation-based Q-learning) reinforcement learning algorithm*

În abordarea FRIQ-learning [5] se folosește tot tehnica de mai sus cu distanța dintre paletă și bilă.

Pentru a folosi această metodă trebuie să găsim valorile stărilor caracteristice jocului Pong în orice moment dat.

Pot fi folosiți următorii descriptori de stări: distanta de la minge la paletă, unghiul de direcție a mingii și poziția verticală a paletei.

Aceste patru valori (s_1, s_2, s_3, s_4) descriu starea în care se află jocul și variabila a descrie acțiunea în felul următor:

s_1: poziția orizontală a bilei

s_2: poziția verticală a bilei

s_3: unghiul de direcție a bilei

s_4: poziția paletei

a : mișcarea paletei (-1, 0, 1)

Aici antrenarea se face cu poziția și direcția bilei fiind aleatoare pentru fiecare episod, nejucându-se împotriva altui jucător. Practic se simulează o jumătate din jocul original Pong,

deoarece cealaltă jumătate este identică și pot fi folosite aceleași reguli pentru ambele palete, cu valorile inversate.

Scopul acestei abordări este de a construi niște reguli de mutare a paletei în Pong, fiind dată o poziție și o traictorie a bilei pentru a o returna cu succes înapoi în suprafață de joc.

3. Utilizarea metodei *reinforcement learning* pentru jocul Pong

Din perspectiva *tabular reinforcement learning* [2], în jocul Pong, produsul cartezian al poziției celor două palete, poziția bilei și controlul paletei jucătorului reprezintă toate stările posibile. Când dimensiunea tabelului de căutare este destul de mică, atunci toate intrările din tabel pot fi găsite ușor, metoda *tabural RL* ar putea să învețe cu succes rezolvarea optimă a cerințelor.

Rețelele nuronale convolutionale au fost prima dată implementate pentru a reprezenta stări ale jocurilor de tip Atari-2600: Pong, Breakout, Space Invaders etc. Reprezentarea stărilor din joc folosind informația vizuală creează un volum mare de stări. În Pong, numărul de stări posibile dintr-un *screenshot* de dimensiune 80 x 80 pixeli poate urca până la 2^{6400} stări (având în vedere că un pixel poate reprezenta cel mult două stări).

Metoda de învățare *Policy Gradient* [6] diferă de cea pe bază de tabel, deoarece cea din urmă actualizează tabelul celulă cu celulă. Metoda tabelară este ineficientă dacă spațiul stărilor este mare. În contrast, învățarea pe baza unei tactici, aproximează politica eficient ajustând ponderile rețelei cu ajutorul algoritmului *backpropagation*.

Jocul funcționează în felul următor: noi primim o imagine („frame”) de dimensiune 210x160x3 și trebuie să decidem în ce parte să mutăm paleta, în sus sau în jos. Dacă un jucător a atins 21 de puncte, spunem că s-a terminat un episod. Prin urmare, o secvență de imagini („frame” –uri) s_t ,o secvență de acțiuni $a_t \in \{1, 2, 3\}$ (acestea reprezintă acțiunile paletei, și anume: stă pe loc, se mișcă în sus, respectiv în jos) pe care „agentul” le-a efectuat și o secvență de recompense $r_t \in \{-1\ , 0\ , 1\}$, constituie un lot de experiențe pentru episodul respectiv.

Având în vedere că jucăm împotriva unui NPC (*non-player charachter)*, dacă acesta câștigă, recompensa de la frame-ul t va fi $r_t = -1$, iar dacă agentul câștigă $r_t = 1$. Cât timp jocul rulează vom avea mii de recompense evaluate la 0 și ocazional 1 sau -1 .

Informația primită la timpul t trebuie propagată la acțiuni anterioare, $t - 1$, având in vedere că recompensa obținută la momentul t a fost ajutată de acțiuni luate în trecut.

Prima dată trebuie să definim o politică pe care o va utiliza jucătorul nostru. Aceasta va lua decizia (daca să mutăm paleta în sus sau în jos) cu referire la starea în care ne aflam. Vom folosi o rețea neuronala cu un strat ascuns care primeste ca intrare imaginea furnizata (100,800 pixeli (210*160*3)) și produce o singură valoare reprezentând probabilitatea de a muta paleta în sus.

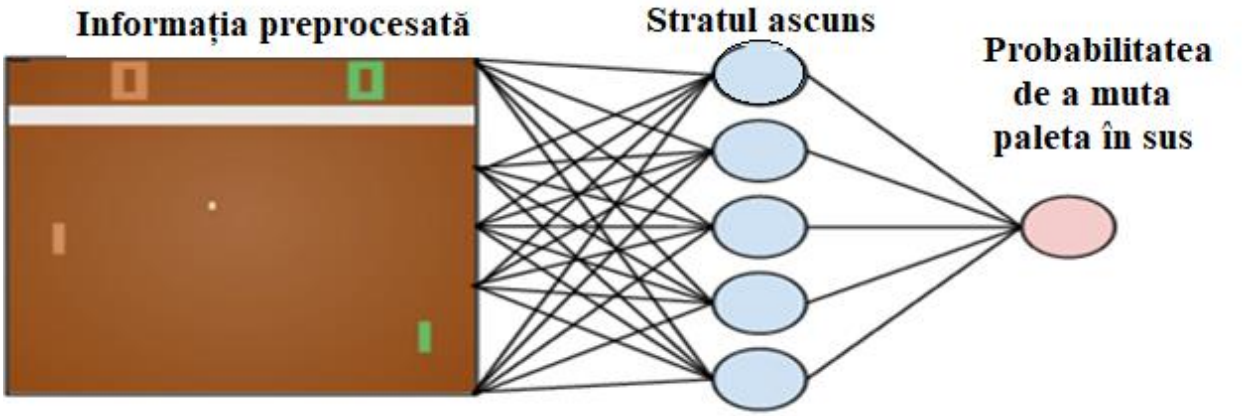


Fig. 10 - Preprocesare imagine

Exemplu:

Presupunem că avem o rețea neuronală cu un singur strat ascuns, cu ponderile W_{ij}^1 si W_{jk}^2, conectând vectorul preprocesat (adică matricea rezultată din procesarea imaginii (80x80), transformată într-un vector unidimensional (6400)) cu stratul ascuns și stratul ascuns cu *output*-ul nostru . Fie funcțiile de activare pentru W^1 si W^2 ReLu și Sigmoid și x_i reprezentând diferența dintre două frame-uri ($s_t - s_{t-1}$). Această diferență capteaza informatția temporală. Având în vedere cele de mai sus, acțiunea prezisă y_k poate fi scrisă ca:

$$y_k = Sigmoid(\sum_j [ReLU(\sum_i [x_i \, W_{ij}^1 \,])W_{jk}^2])$$

Formula anterioară este implementată în Python în modul următor:

```
def activare_neuron(imagine_procesata):
  """Implementare pentru forward propagation"""
  produs_imagine_strat_ascuns = np.dot(multime_date_antrenare['W1'], imagine_procesata) #(1)
  produs_imagine_strat_ascuns[produs_imagine_strat_ascuns<0] = 0 # (2)
  produs_strat_ascuns_iesire = np.dot(multime_date_antrenare['W2'], produs_imagine_strat_ascuns) # (3)
  sus_prob = f.functia_sigmoidala(produs_strat_ascuns_iesire)  # (4)
  return sus_prob, produs_imagine_strat_ascuns #
```

Presupunem că stratul ascuns are H = 200 de neuroni.

Dimensiunea modelului este D = 80 * 80 adică 6400 de pixeli.

W1 și W2 sunt două matrici inițializate aleator. În W1 sunt reprezentate ponderile fiecărui pixel din imaginea furnizată, către neuronii stratului ascuns, iar în W2 sunt prezente ponderile de la fiecare neuron din stratul ascuns către valoarea de ieșire.

Pașii realizați în implementarea de mai sus sunt următorii:

(1) Calculăm activarea neuronilor. Având în vedere că W1 este o matrice de dimensiune (H x D), iar X este un vector unidimensional (D x 1), înmulțindu-le obținem o matrice de dimensiune (H x 1) adică (200 x 1)

(2) Valorile negative sunt setate ca fiind 0 iar celelalte sunt lăsate la fel. (ReLU)

(3) Calculăm probabilitatea logaritmică de a muta paleta în sus. W2 este o matrice de dimensiune (1 x H), iar rezultatul de la punctul unu, adică h (stratul ascuns) este de dimensiune (H x 1). Înmulțind cele doua matrici rezultă (1 x H) * (H x 1) = 1 (scalar)

(4) Aplicăm funcția Sigmoid unde sus_prob va determina probabilitatea paletei de a fi mutată în sus (0 < sus_prob < 1)

Considerăm în continuare scenariul din Fig. 11:

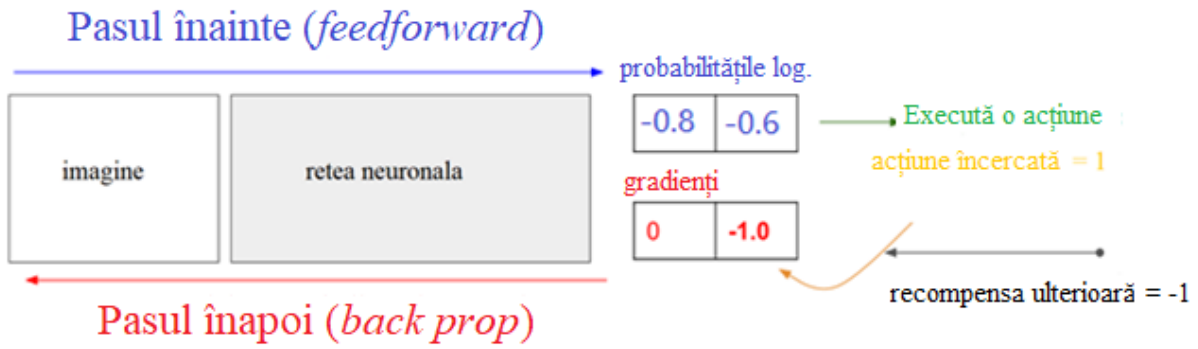


Fig. 11 – Scenariu posibil

Prin politica aleasă, probabilitatea calculată ca paleta să urce este 45% (logprob -0.8) și ca să coboare 55% (logprob -0.6). Acum vom muta paleta; să spunem că am ales să o mutam în jos. Încă nu știm dacă această acțiune este bună sau rea, putem aștepta până la sfârșitul unui joc (un episod are cel puțin 21 de jocuri) și să vedem ce recompensă primim (+1 dacă am învins, -1 dacă am pierdut). În exemplul de mai sus, coborând, a dus la pierderea jocului (-1 recompensă). Deci dacă punem valoarea -1 ca probabilitatea logaritmică a acțiunii de coborâre și efectuăm *backpropagation* , vom avea o rețea neuronală care va descuraja alegerea acestei acțiuni pentru imaginea (intrarea) respectiva pe viitor din moment ce aceasta a cauzat pierderea jocului.

Politica noastră presupune incearcarea unor acțiuni, iar când acele acțiuni se prezintă a fi bune, le încurajam să mai fie luate când avem o imagine asemănătoare procesată, iar când nu, sunt descurajate.

Cum antrenăm rețeaua?

Inițializăm modelul cu W_1 și W_2 conținând valori aleatoare și jucăm un număr de episoade de Pong, să spunem 100.

Să presupunem că fiecare joc are 200 de frame-uri, deci avem de luat 20,000 de decizii (sus sau jos).

Ce ne rămâne acum de făcut este să etichetăm fiecare decizie pe care am luat-o cu pozitiv sau negativ.

De exemplu, am câștigat 12 episoade și am pierdut 88.
Considerăm toate 200 * 12 = 2400 decizii pe care le-am luat în episoadele câștigătoare și realizăm o actualizare pozitivă (efectuăm *backpropagation* și actualizam parametrii, încurajând astfel toate acțiunile pe care le-am luat în acele stări.)

Acum luăm toate 200 * 88 = 17600 decizii pe care le-am luat în episoadele pierdute și descurajăm orice decizie pe care am luat-o făcând un update negativ.

Rețeaua va deveni puțin mai încrezătoare în a repeta decizii care au funcționat, și invers pentru cele cu recompensă negativă. Jucăm din nou 100 de episoade cu noua noastră tactică (politică) puțin îmbunătățită și repetăm până devine din ce în ce mai buna.

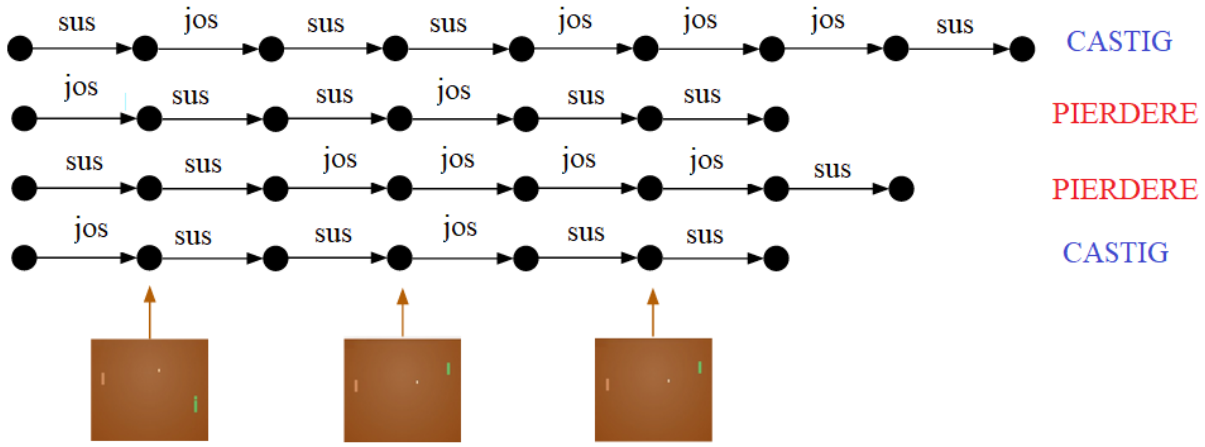


Fig. 12 – Diagramă pentru 4 jocuri

In Fig. 12, fiecare cerc negru reprezintă o stare din joc (ca în cele 3 imagini) și fiecare săgeată este o tranziție notată cu acțiunea (sus, jos) încercată atunci. Avem deci două câștiguri și două înfrângeri.

4. Rezultate experimentale

4.1. Tehnologii folosite

Pentru implementare am folosit Pyhton, un limbaj de programare foarte bine închegat, cu o mulțime de module și *framework*-uri foarte folositoare.

Grafica a fost realizată cu ajutorul *framework*-ului Gym[3]. Gym a fost folosit la dezvoltarea și compararea algoritmilor de tip *reinforcement learning*, *framework* care oferă suport pentru antrenarea agenților, inclusiv pentru învățarea acestora de a juca Pong.

Unul dintre cele mai folosite module ale limbajului Python[4] este NumPy. Acesta reprezintă un pachet fundamental pentru calculele științifice și nu numai. NumPy a jucat un rol foarte important în implementarea mea având foarte multe funcții predefinite cum ar fi: înmulțirea dintre matrici, funcții ca radical și exponențial, initializarea de matrici, generarea de numere aleatoare etc.

Pentru a salva modelul antrenat și pentru a se putea porni jocul cu agentul antrenat sau nu, am avut nevoie de modulul Pickle. Acesta ajută la serializarea și deserializarea obiectelor din Python. Formatul datelor dintr-un fișier de tip Pickle este de tip binar.

Cu ajutorul CSV și DateTime am realizat statistica, înregistrând scorul pentru fiecare episod și timpul când s-a întâmplat asta într-un fișier. Timpul fiind monitorizat cu funcții ale modulului DateTime.

Pentru a putea realiza partea experimentală, după terminarea fiecărui joc am salvat într-un fișier de tip CSV (*comma separated values*) scorul și numărul episodului urmând ca într-un fișier Excel să fie realizată o diagramă pentru a urmări evoluția învățării agentului.

[3] https://gym.openai.com/docs/

[4] https://www.python.org/

4.2. Rezultate experimentale

În cadrul analizei strategiei de învatare, am ajuns la urmatoarele rezultate:

După 5000 de episoade în care agentul a învățat să joace Pong am ajuns la o medie a scorului de -3, adică agentul pierde cu o medie de 21 la 18. Aceste episoade s-au derulat pe o perioadă de aproximativ 24 de ore.

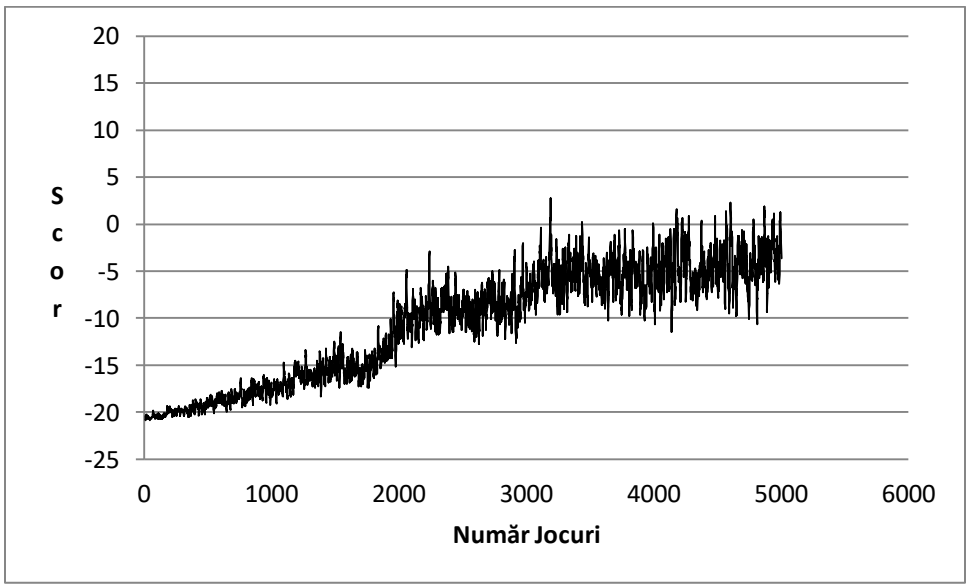


Fig. 13 - Analiză a scorului după 5000 de episoade

După alte 7500 de episoade media scorului devine 5. Timpul alocat fiind de circa 48 de ore.

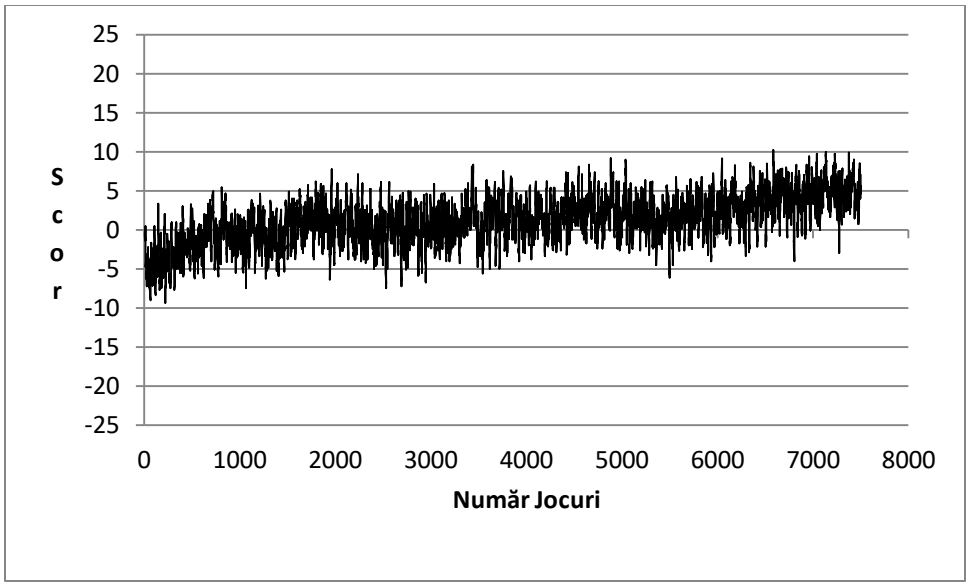


Fig. 14 - Analiză a scorului după încă 7500 de episoade

După aproximativ 12500 de episoade, agentul reușește să învingă cu o medie a scorului de 5 (21 la 16). Primele 5000 de episoade s-au derulat mult mai rapid având în vedere că agentul nu era antrenat, scorul de 21 fiind atins cu ușurință de oponent, dar cu cât numărul episoadelor creșteau și agentul învață să joace, timpul alocat chiar și unui singur episod se putea tripla. Un alt aspect ar fi platforma pe care s-a realizat antrenarea. Dacă agentul antrenat pe un laptop cu un procesor Intel Core I5-4200H , 2.8GHz, 4gb RAM și o placă video NVIDIA GEFORCE GTX 950M a obținut o medie de 5 a scorului după aproximativ 3 zile, cu siguranță pe un dispozitiv de ultimă generație acest timp ar fi scurtat semnificativ.

4. 3. Comparații cu alte metode

În lucrarea [7] este prezentată o statistică a metodelor de tip *deep-q-networks* pentru a rezolva jocul Pong.

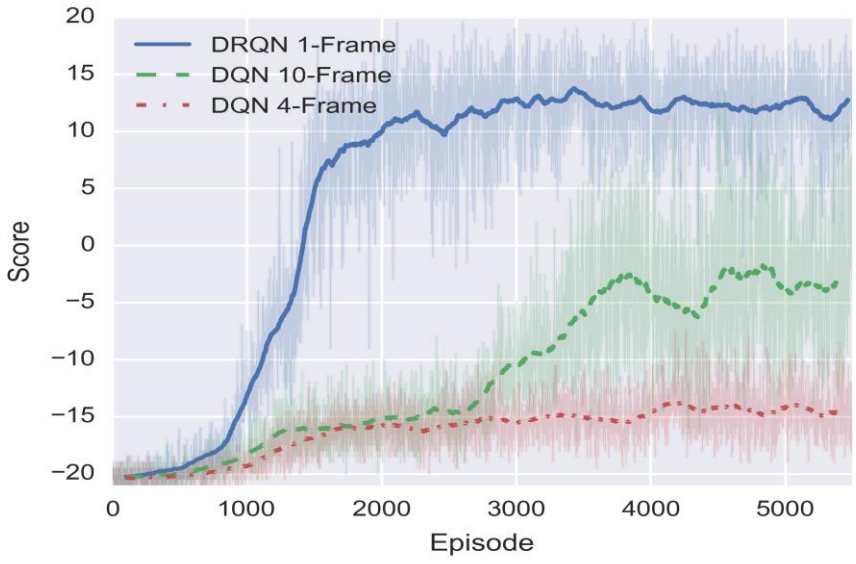
Prezentăm mai jos această statistică în Fig. 15 pentru a putea realiza comparații cu metoda implementată în această lucrare.

În Fig. 15, X-Frame se referă la faptul că datele de intrare sunt alcătuite din X frame-uri.

Putem observa că după 5000 de episoade, implementarea cu 10 frame-uri primite la intrare ajunge la o medie a scorului apropiată de cea a algoritmului propus în lucrarea curentă.

Metoda DRQN 1-Frame (Deep-Recursive-Q-Network) se prezintă a fi cea mai eficientă variantă, ajungând la o medie a scorului de 10 în doar 5000 de episoade.

Cea cu 4 frame-uri este cea mai ineficientă, metoda ajungând doar la un scor de -15.



[5]

Fig. 15 – Strategii folosind *deep-q-networks*

[5] https://www.cs.utexas.edu/~pstone/Papers/bib2html-links/SDMIA15-Hausknecht.pdf

Concluzii

Scopul acestei lucrări este de a învață un agent să joace binecunoscutul joc Pong și să obțină victoria în fața unui NPC (*non-player-character*) antrenat de OpenAI.

Pentru a realiza acest lucru am utilizat metoda *Reinforcement Learning* cu *Policy Gradient*, metodă care se dovedește a fi destul de eficientă pentru problema noastră.

Pentru a putea aborda această problemă am aprofundat noțiuni de rețele neuronale și am folosit diferite librării implementate în python.

Ca și îmbunătățiri aș dori să abordez problema intru-un mod diferit. Adică jocul să se desfășoare între doi agenți, ambii antrenați de mine. Să înceapă de la 0 fiecare și pe parcurs să împrumute unul de la celălalt experiențele dobândite. În felul acesta cred că antrenarea ar fi mai eficientă din punct de vedere al timpului.

Bibliografie

1. https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1

2. **R. S. Sutton, A. G. Barto,** *Reinforcement Learning:An Introduction second edition.* 2018.

3. T. **Simonini,** *An introduction to Policy Gradients with Cartpole and Doom.* 2018. (https://www.freecodecamp.org/news/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f/)

4. **V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra și M. Riedmiller,** *Playing Atari with Deep Reinforcement Learning.* 2013.

5. **T. Tompa, D. Vincze, & S. Kovács,** *The Pong game implementation with the FRIQ-learning reinforcement learning algorithm.* s.l. : Carpathian Control Conference (ICCC), 2015.

6. **S. Phon-Amnuaisuk,** *Learning to Play Pong using Policy Gradient Learning.* 2018.

7. **M. Hausknecht and P. Stone**, Deep Recurrent Q-Learning for Partially Observable MDPs