Reinforcement Learning - Pacman

Ciuperceanu Vlad-Mihai, Huzum Andreea, Miu Tudor-Gabriel
Grupa 351

1 Introducere

Pacman este un joc video lansat in 1980 pe consolele Atari, care a devenit rapid un simbol al industriei jocurilor video si o parte importanta a istoriei acestora.

Acest proiect are ca scop explorarea aplicarii Reinforcement Learning in cadrul jocului Pacman, utilizand diferiti algoritmi pentru a imbunatati performanta agentului. Vom analiza eficienta acestora si vom face o comparatie intre rezultatele obtinute.

2 Environment

Vom folosi mediul Pacman din Atari Gymnasium (Arcade Learning Environment).

Figura 1: ALE/Pacman-v5

2.1 Interactiunea cu mediul

La fel ca orice joc Atari din framework, starile sunt date ca frame-uri RGB de 250x160 pixeli din joc.

Jucatorul are 5 actiuni din care poate alege: NOOP (nu schimba pozitia) sau UP, DOWN, LEFT, RIGHT (pentru cele 4 directii).

Reward-ul corespunde scorului, fiind de 1 pentru fiecare pellet mancat. Pentru cele 4 power pellets, reward-ul este de 50, iar fiecare fantoma aduce 100 de puncte. In majoritatea frame-urilor, datorita functionarii jocului, reward-ul va fi 0, intrucat nu se inregistreaza nicio adaugare la scor.

In informatiile returnate se poate gasi si numarul de vieti ramase, vizibil si in coltul fiecarui frame.

2.2 Preprocesari

Pentru o imbunatatire in performanta, in special in metodele ce folosesc retele neuronale, este de ajutor o preprocesare a mediului si a spatiului de observatie.

Am ales sa reducem dimensiunea frame-urilor la 84x84 pixeli, transformand imaginea in grayscale. De asemenea, am ales sa nu folosim fiecare frame in parte pentru eficienta, grupandu-le cate 4 sau avand un frameskip de 4 asupra mediului. Pentru prima varianta am implementat manual transformarile, in timp ce pentru a doua am folosit wrapper-ul AtariPreprocessing din Gymnasium:

```
env = wrappers.AtariPreprocessing(env, frame_skip=4,
                grayscale_obs=True, screen_size=84)
```

Am preferat sa preprocesam datele astfel intrucat aceasta este modalitatea standard cu cele mai bune rezultate pentru mediile Atari, folosita si in paper-ul celor de la DeepMind.

3 Random Agent

Mai intai am implementat un agent care ia actiunile in mod random, pentru a ne face o idee despre felul in care functioneaza mediul si pentru a avea un punct de referinta in implementarea algoritmilor.

Acesta a prezentat un scor de 13.2, obtinut ca reward mediu in urma rularii a 10 episoade.

4 Monte Carlo Tree Search

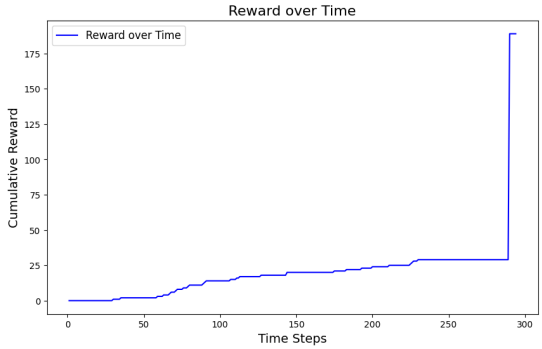
Un algoritm care foloseste principii de Reinforcement Learning este Monte Carlo Tree Search, pe care am ales sa il implementam pentru inca o referinta in antrenarea agentilor.

MCTS este potrivit in contextul unor medii mari, definite prin stari, actiuni si reward-uri. Se bazeaza pe explorarea unui arbore de decizie asupra starilor, folosindu-se de "statisticile" obtinute in urma experientei jocului. Acesta nu are nevoie de cunostinte specifice mediului si functioneaza cu informatie minima despre starile mediului. Totodata, mentine balanta intre explorare si exploatare prin formula de selectie a nodurilor - UCT (Upper Confidence Bound for Trees).

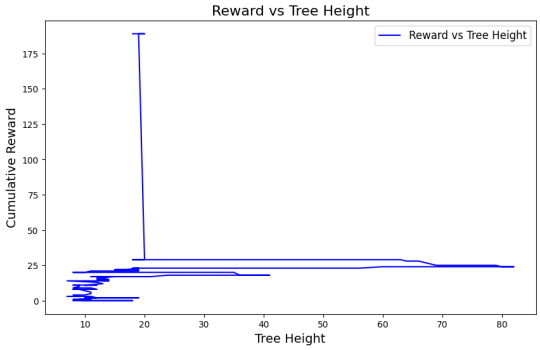
Desigur, are si dezavantaje, printre care se numara timpul ridicat de executie (datorita crearii arborelui de stari), numarului mare de episoade rulate pentru a descoperi actiunile potrivite, varianta mare (data de actiunile alese random) si memoria necesara rularii (tot datorita arborelui creat).

Totusi, ruland aproximativ 3 ore, am obtinut un rezultat foarte bun, ajungand la un reward de 189 in urma a 294 de actiuni decise.

Cresterea a fost una incrementala, pana cand a ajuns in starea in care mananca un Power Pellet, urmata de intalnirea cu o fantoma, ceea ce a dat o crestere masiva in reward.



(a) Reward in timp



(b) Reward pentru inaltimea arborelui

Figura 2: Reward-ul in urma actiunilor gasite

Se poate observa ca, in timp, este nevoie de o adancime mare a arborelui (aceasta ajunge pana la 80), pentru a putea gasi deciziile cu potential maxim, dupa care revine la o inaltime de 20, in urma gasirii actiunii optime.

Aceasta este o metoda care a dat rezultate, insa este limitata de dezavantajele timpului de executie si memoriei necesare, asa cum am mentionat mai sus.

5 Q-Learning

O prima incercare a fost de a implementa Q-Learning fara a folosi retele neuronale pentru aproximarea functiei.

Chiar daca am preprocesat mediul folosind AtariPreprocessing pentru a reduce dimensiunile, ajutand la invatare si la retinerea starilor, tot ne-am lovit de o problema de memorie, dimensiunea Q-table crescand repede.

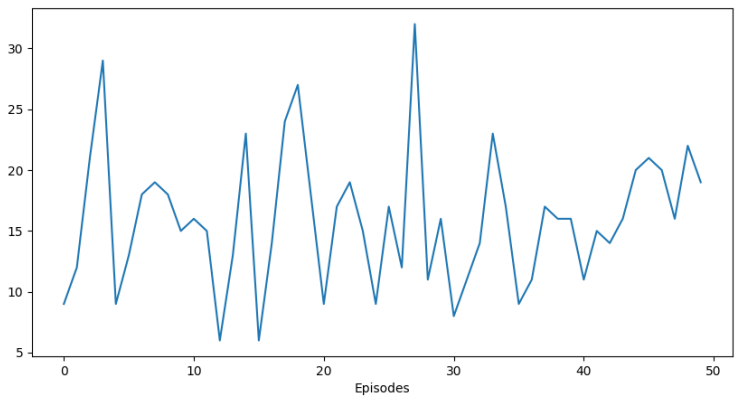


Figura 3: Reward in timp

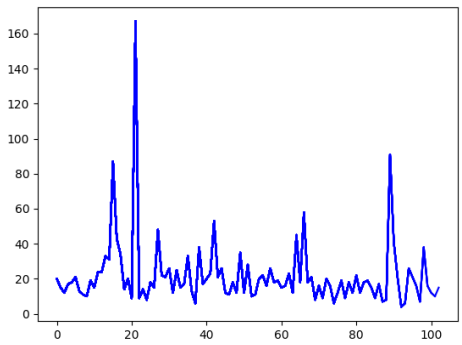
Dupa cum se poate observa, numarul de episoade trebuie sa fie unul mare pentru a putea atinge convergenta, insa timpul de executie a fost semnificativ chiar si pentru 50 de episoade.

6 DQN - Deep Q-Learning

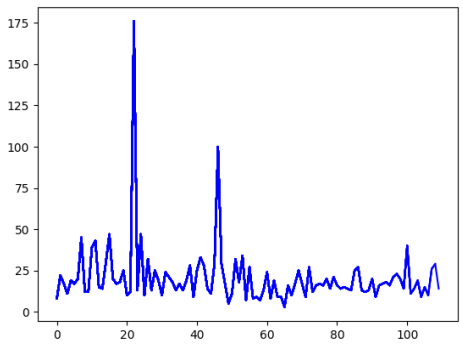
O prima imbunatatire a fost folosirea unei DQN pentru aproximarea functiei Q.

Am aplicat preprocesari asupra mediului, pentru a ajuta procesul de invatare al retelei neuronale. De asemenea, am normalizat valorile pixelilor din imagine, pentru o convergenta mai buna. Este de mentionat ca, in timpul jocului, pierderea unei vieti nu este penalizata, astfel ca am introdus manual un reward de -10 pentru astfel de situatii.

Modificand learning rate-ul, am obtinut urmatoarele reward-uri:



(a) Learning Rate de 0.001

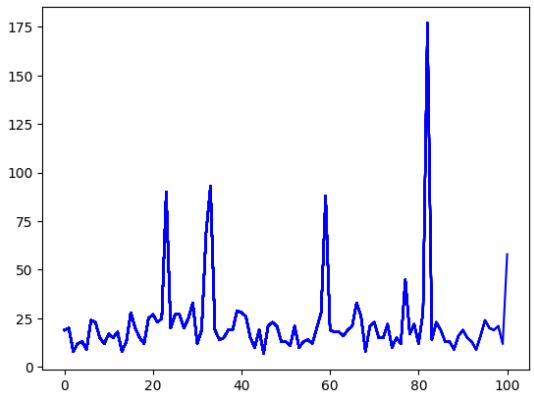


(b) Learning Rate de 0.01

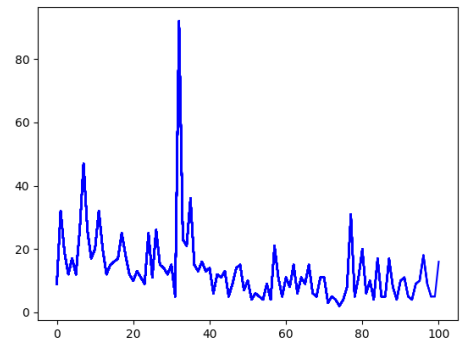
Figura 4: Reward in timp, alegand un batch de 32 in experience replay

Se poate observa ca, datorita actiunilor random, ambele modele ajung la reward-uri mari la inceput (probabil gasind succesiunea de actiuni ca la metoda MCTS), insa apoi reward-ul incepe sa scada. Totusi, pentru un reward mai mic, un spike asemanator are loc si mai tarziu, in jur de episodul 90. Acest lucru inseamna ca modelul incepe sa invete, insa are nevoie de mai multe episoade.

Apoi, ne-am gandit ca am putea ajuta procesul de invatare prin adaugarea unei recompense de 0.05 pentru ramanerea in viata (indiferent de castigarea unui pellet sau nu), din moment ce reward-ul, in absenta unui pellet, este 0. Adunat cu numarul frame-urilor de procesat, acest lucru poate ingreuna aproximarea functiei Q, afectand gradientii calculati. De asemenea, am modificat dimensiunea batch-ului ales din replay memory. Rezultatele obtinute au fost astfel:



(a) Learning Rate de 0.005



(b) Learning Rate de 0.01

Figura 5: Reward in timp, alegand un batch de 64 in experience replay

Se poate observa ca, in functie de learning rate, reward-ul obtinut nu creste neaparat, posibil din cauza unei discrepante intre valorile reward-urilor primite in joc (posibil sa fie nevoie de o penalizare mai mare pentru pierderea unei vieti sau de un bonus mai mic pentru pastrarea acestora pentru o actiune). De asemenea, un numar mai mare de episoade este necesar pentru a observa modul in care agentul invata, insa algoritmul necesita un timp de rulare de aproximativ 3 ore pentru cele 100 de episoade. Totusi, pentru un learning rate de 0.005, se poate observa o crestere in scorurile mari obtinute, urmand unul chiar in episodul 100. Astfel, se pare ca modelul invata sa obtina reward-uri din ce in ce mai mari, insa ar fi nevoie de mai multe episoade pentru a ne apropia de convergenta.

7 DDQN - Double Deep Q-Learning

Urmatoarea imbunatatire a performantei si a convergentei a fost implementarea unei DDQN.

De data aceasta, am modificat bonusul si penalizarea in legatura cu pastrarea/pierderea unei vieti la 0.01, respectiv -20.

Daca in algoritmii trecuti am folosit AtariPreprocessing, cu un frameskip de 4, in acest caz am ales sa grupam cate 4 frame-uri pentru a constitui o stare. Learning rate-ul a fost unul de 0.0005, favorizand o crestere mai sigura in invatare.

Ruland 250 de episoade timp de aproximativ 3 ore si 30 de minute, am obtinut urmatorul rezultat:

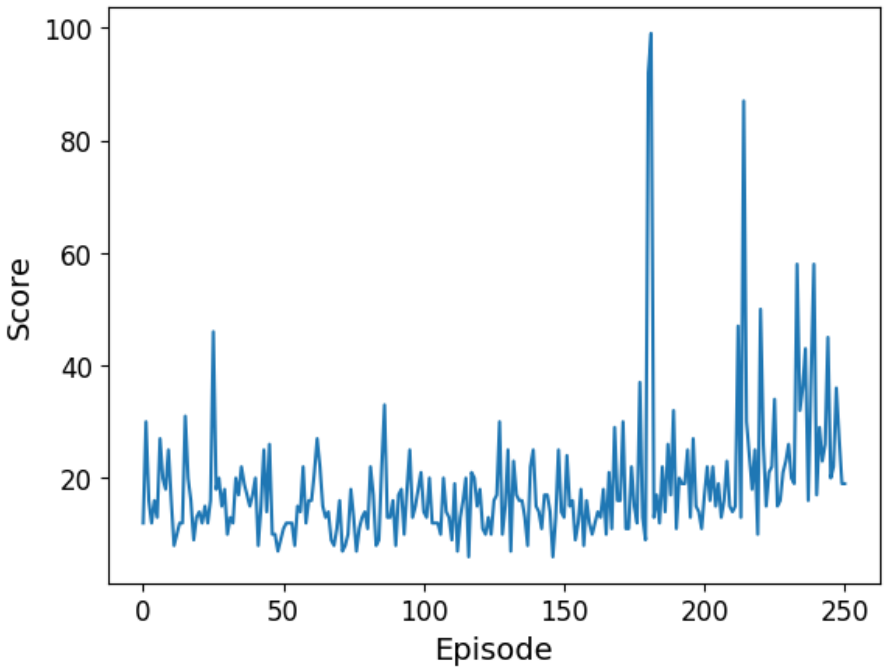


Figura 6: Reward in timp

Se poate observa cum, incepand cu primul spike, incepe un trend ascendent, semnificand faptul ca, pentru un numar mai mare de episoade, agentul va incepe sa obtina scoruri din ce in ce mai mari.

8 Stable Baselines3

Dupa cum s-a observat anterior, antrenarea modelelor DQN si DDQN dureaza extrem de mult, chiar si folosind GPU pe portiunile legate de retelele neuronale, astfel ca suntem limitati la un numar relativ mic de episoade. In majoritatea paper-urilor legate de acest subiect, cateva milioane de episoade sunt necesare pentru a atinge convergenta si pentru a obtine scoruri semnificative.

De aceea, am ales sa folosim si o biblioteca specializata pentru algoritmi de Reinforcement Learning - Stable Baselines3.

Aceasta pune la dispozitie arhitecturi de modele, asa cum este si cea utilizata de noi - DQN cu CNNPolicy, dar si medii vectorizate. Cu ajutorul acestora, putem antrena agentul pe mai multe medii in paralel, crescand atat performanta obtinuta, cat si timpul de rulare necesar.

Astfel, dupa ce am rulat pentru un numar de 3.3 milioane de pasi, cu replay memory de 100 000, pe 16 medii in paralel timp de 3 ore, am obtinut un reward mediu de 183.6 in urma a 10 episoade jucate.

9 Concluzii

Asa cum s-a vazut cel mai bine in graficul metodei DDQN, care a obtinut cel mai bun rezultat (excluzand Stable Baselines3), agentii incep sa invete, obtinand scoruri bune, insa au nevoie de mai multe episoade si de mai mult timp de antrenare.

In final, implementarea algoritmilor de Reinforcement Learning a prezentat o varietate de posibile solutii si de performante obtinute. Fiecare metoda a venit cu provocarile sale, dar si cu oportunitati de optimizare semnificative. Totusi, indiferent de aceste diferente, toate abordarile au contribuit la o intelegere mai profunda a modului in care agentii pot invata si se pot adapta la medii complexe.

10 Referinte

- https://ale.farama.org/environments/pacman/
- https://stable-baselines3.readthedocs.io/en/master/guide/examples.html