Reinforcement Learning Pentru Jocul 2048

Hodivoianu Anamaria, Uceanu Alexandra, Robu Corina

15 ianuarie 2025

1 Descrierea Jocului

2048 este un joc single-player de tip puzzle. Pe o tabla 4x4 se afla placute cu valori puteri ale lui 2. Jucatorul poate sa le traga in sus, jos, stanga sau dreapta pentru a uni placutele identice, care vor deveni una singura cu valoarea dublata. Dupa fiecare mutare apare pe tabla o noua placuta cu valoare 2 (cu probabilitate 0.9) sau 4 (cu probabilitate 0.1) in oricare dintre pozitiile libere. Scorul este calculat insumand valorile placutelor unite (unire 2 si 2 rezulta 4 puncte). Scopul este obtinerea placutei 2048, insa se poate ajunge si la valori mai mari. Jocul se termina cand toata tabla este ocupata si nu mai exista miscari posibile (nu se mai pot uni placute). Un exemplu de tabla este in Figura 1. O strategie consta in mentinerea placutei cu cea mai mare valoare intr-un singur colt (de exemplu stanga-sus) pe tot parcursul jocului. Totusi, din cauza modului aleator in care sunt generate placutele, nu este garantata obtinerea placutei cu valoarea 2048.

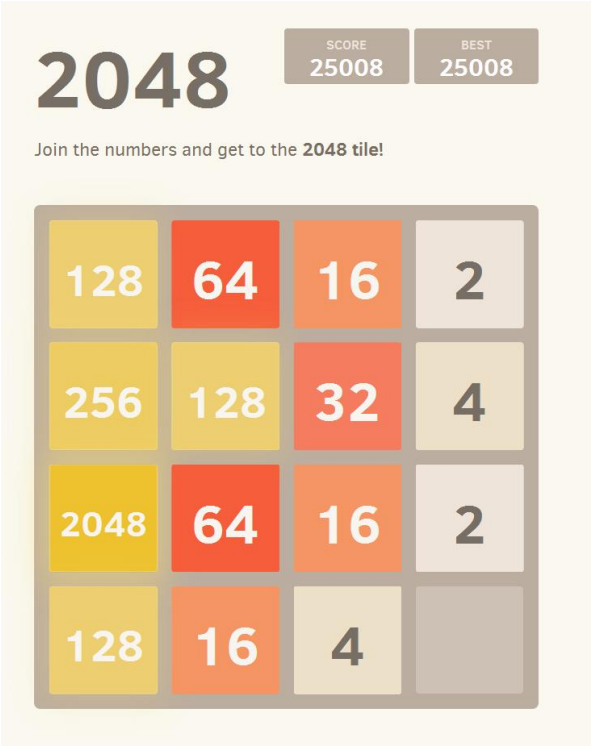

Figura 1: Exemplu de tabla de joc

2 Mediul

2.0.1 Implementare

Am implementat intai mediul, o clasa Game in care avem:

- tabla sub forma de matrice (reprezinta starea)
- scorul
- metode pentru a modifica tabla in functie de actiunea aleasa
- metoda pentru a adauga o placuta noua
- metoda step, care primeste o actiune, o aplica si calculeaza reward-ul (diferit de scor) si returneaza noua stare, reward-ul, boolean daca s-a terminat sau nu jocul si informatii precum cea mai mare valoare de pe tabla, scorul total
- metoda pentru a verifica daca jocul s-a terminat, nu mai sunt nici locuri libere pe tabla, nici mutari posibile
- metoda care returneaza actiunile invalide (care nu modifica tabla)
- metoda pentru a reseta tabla si scorul
- metoda care returneaza tabla intr-o forma mai convenabila pentru modelele de RL, logaritmand valorile nenule si adaugand o dimensiune noua pentru canal (1, 4, 4) pentru convolutii
- alte metode auxiliare

2.0.2 Reward

Reward-ul a constat in scorul obtinut, dublat daca s-a obtinut o valoare ≥ 256 sau o noua valoare maxima, logaritmat si la care s-a aplicat o mica penalizare/bonus in functie de numarul de locuri libere inainte si dupa mutare, sau -1 in cazul unei mutari invalide.

2.0.3 Observatii

Cateva observatii sunt:

- Pentru tabla am incercat si one-hot encoding, insa nu s-au observat diferente pozitive
- La reward am incercat sa luam in considerare si numarul de piese care si-au schimbat pozitia, dar nu s-au observat diferente semnificative

3 Algoritmii

Am testat 2 algoritmi de reinforcement learning pe acest joc, DQN si Actor Critic.

3.1 DQN

Primul algoritm implementat a fost DQN. DQN (Deep Q-Network) este un algoritm de RL care foloseste o retea neuronala pentru a aproxima functia de valoare-actiune $Q(s, a)$. Este un algoritm *off-policy*, care are o retea tinta (*target network*) pe care o actualizeaza periodic pentru a stabiliza invatarea si pentru a preveni oscilatiile in estimarile Q.

Explorarea si exploatarea sunt controlate printr-o politica *epsilon-greedy*, care alege o actiune aleatoare cu probabilitatea ϵ si actiunea cu cea mai mare valoare estimata $Q(s, a)$ cu probabilitatea $1 - \epsilon$.

DQN foloseste un mecanism de memorare a experientelor. Tranzitiile $(s, a, r, s', \text{done})$ sunt salvate intr-un buffer, iar reteaua este antrenata pe mostre aleatorii extrase din acest buffer.

3.1.1 Implementare

Am creat clasa DQNAgent in care am setat hiperparametri si cele 2 modele (model si target model) si am implementat urmatoarele metode:

- update target model: actualizarea lui target model cu parametri de la model
- remember: salvam in memorie starea, actiunea, reward-ul, starea rezultata si boolean pentru stare finala
- act: ori alege o actiune aleatoare dintre cele valide ori cea mai buna actiune valida cu ajutorul modelului (mascam actiunile invalide; am incercat si sa nu mascam actiunile invalide, insa antrenarea dura prea mult)
- replay: updateaza modelul utilizand experientele trecute, extragem din memorie tupluri aleatoare de forma stare, actiune, reward, stare rezultata, done, target model prezice Q values pentru starile rezultate, extragem valorile maxime si utilizam ecuatia Bellman pentru a calcula target Q values. Modelul prezice Q values pentru stari, calculam pierderea si actualizam modelul
- decay epsilon: miscoreaza epsilon

3.1.2 Retea

Am utilizat o retea convolutionala cu 3 straturi convolutionale si 2 fully connected (Figura 2).

```python
class QNetwork(nn.Module):
    def __init__(self, state_size, action_size):
        super(QNetwork, self).__init__()
        input_channels = state_size[0]
        self.conv1 = nn.Conv2d(in_channels=input_channels, out_channels=144, kernel_size=2, stride=1)
        self.conv2 = nn.Conv2d(in_channels=144, out_channels=288, kernel_size=2, stride=1)
        self.conv3 = nn.Conv2d(in_channels=288, out_channels=432, kernel_size=2, stride=1)

        self.fc1 = nn.Linear(432, 512)
        self.fc2 = nn.Linear(512, action_size)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.relu(self.conv3(x))
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Figura 2: QNetwork

3.1.3 Hiperparametri

Hiperparametrii utilizati:

- Gamma: 0.999 (importanta mare pentru reward-uri long term)
- Epsilon: 0.4

- Epsilon minim: 0.005
- Epsilon decay: 0.9999
- Rata de invatare: 0.0001
- Batch size: 96
- Memory size: 100000
- Optimizator: Adam
- Functie de pierdere: MSE
- Frecventa replay: 4 pasi
- Frecventa actualizare target model: 10 episoade

3.1.4 Antrenare

Numarul maxim de episoade de antrenare a fost 30000. Am salvat cele mai bune 7 modele si le-am testat pe 500 de episoade.

Rezultate partiale (mai detaliate in fisierul info.txt):

- model 1, episod antrenare: 29937, scor: 25888:
- cel mai mare scor: 27940
- piesa 2048: 8 ori
- piesa 1024: 199 ori
- piesa 512: 206 ori

Figura 3 ilustreaza evolutia scorurilor si a placutelor maxime de-a lungul antrenarii.

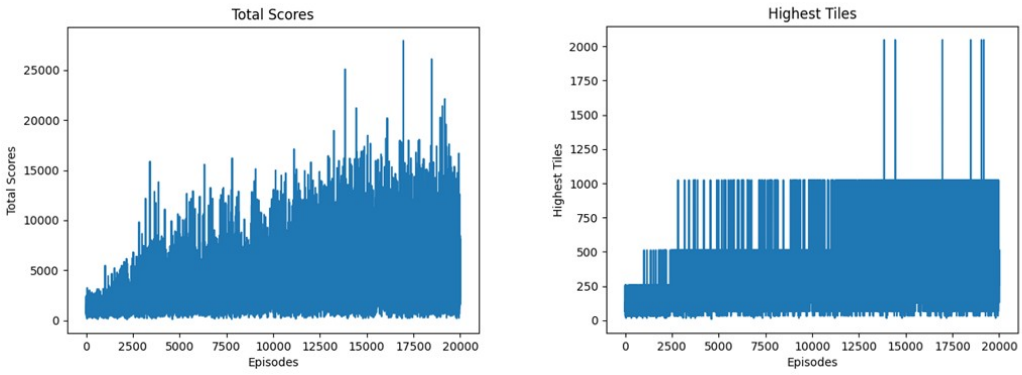

Figura 3: Evolutia scorurilor si a placutelor

3.1.5 Observatii

Cateva observatii sunt:

- Cele mai bune episoade de antrenare au fost printre ultimele, sugerand ca modelul s-ar mai fi putut imbunatati, insa antrenarea deja durase peste 24 de ore. In unele articole gasite pe internet agentii reuseau sa treaca de 2048, insa era nevoie de 60000-100000 episoade si o retea mai adanca sau cu mai multi neuroni.

- Nu am observat imbunatari semnificative pana ce nu am ales un epsilon mic de la inceput. Initial incepeam cu epsilon 1 si il scadeam pe parcurs pana la 0.01, insa agentul ajungea greu la valoarea 512 la testare. Probabil pentru ca actiunile aleatoare duceau la stari nefavorabile, si deci la terminarea jocului prea repede, inainte ca agentul sa invete o strategie benefica pe termen lung.

La final am modificat si probabilitatile de generare a placutelor noi si am testat modelul pe 200 episoade.

- 2 cu 0.9, 4 cu 0.1: Scor maxim: 27796, 512: 88 ori, 1024: 71 ori, 2048: 4 ori
- 2 cu 0.5, 4 cu 0.5: Scor maxim: 14036, 512: 61 ori, 1024: 18 ori, 2048: 0 ori
- 2 cu 1, 4 cu 0: Scor maxim: 33992, 512: 72 ori, 1024: 102 ori, 2048: 12 ori
- 2 cu 0.1, 4 cu 0.9: Scor maxim: 11028, 512: 45 ori, 1024: 7 ori, 2048: 0 ori

3.2 Actor Critic

Al doilea algoritm implementat a fost Actor Critic. Actor Critic este un algoritm de RL care combina abordarea bazata pe valori cu cea bazata pe politici. In acest algoritm, exista doua componente principale: actorul si criticul.

Actorul este cel care alege actiunile si are o politica $\pi(a|s;\theta)$ care mapeaza starea s la probabilitatile actiunilor a. Criticul evalueaza actiunile luate de actor, estimand valoarea starii curente sau a starii-actionii folosind o functie de valoare $V(s;\theta_c)$.

Actorul si criticul sunt antrenati simultan. Actorul isi actualizeaza politica pentru a maximiza recompensa pe termen lung, iar pentru asta foloseste estimarile criticului. Criticul isi actualizeaza functia de valoare utilizand temporal difference error (TD) pentru a evalua performanta actorului.

3.2.1 Implementare

Am creat clasa ActorCriticAgent unde am setat hiperparametri si am initializit cele 2 modele, actor si critic. Am implementat metodele:

- act: avand o stare actorul calculeaza probabilitatile actiunilor (mascam actiunile invalide) si alege o actiune conform distributiei
- compute loss: avand un episod calculam reward-urile cumulative cu discount si utilizand valorile prezise de critic in timpul episodului calculam avantajele (cat de buna/ proasta fost o actiune luata comparat cu estimarea criticului) pentru a afla pierderile. La actor utilizam si entropia pentru a incuraja explorarea
- decay entropy beta: scadem entropy beta pe parcurs, minimizand explorarea

3.2.2 Retele

Retelele pentru actor si critic au fost similare cu cea utilizata la primul algoritm (Figurile 4 si 5).

3.2.3 Hiperparametri

Hiperarametrii utilizati:

- Gamma: 0.999
- Rata de invatare actor: 0.0001

```python
class Actor(nn.Module):
    def __init__(self, state_size, action_size):
        super(Actor, self).__init__()
        input_channels = state_size[0]
        self.conv1 = nn.Conv2d(in_channels=input_channels, out_channels=128, kernel_size=2, stride=1)
        self.conv2 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=2, stride=1)
        self.conv3 = nn.Conv2d(in_channels=256, out_channels=384, kernel_size=2, stride=1)

        self.fc1 = nn.Linear(384, 432)
        self.fc2 = nn.Linear(432, action_size)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.relu(self.conv3(x))
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        action_probs = torch.softmax(self.fc2(x), dim=-1)
        return action_probs
```

Figura 4: Actor Network

```python
class Critic(nn.Module):
    def __init__(self, state_size):
        super(Critic, self).__init__()
        input_channels = state_size[0]

        self.conv1 = nn.Conv2d(in_channels=input_channels, out_channels=144, kernel_size=2, stride=1)
        self.conv2 = nn.Conv2d(in_channels=144, out_channels=288, kernel_size=2, stride=1)
        self.conv3 = nn.Conv2d(in_channels=288, out_channels=432, kernel_size=2, stride=1)

        self.fc1 = nn.Linear(432, 512)
        self.fc2 = nn.Linear(512, 1)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.relu(self.conv3(x))
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        state_value = self.fc2(x)
        return state_value
```

Figura 5: Critic Network

- Rata de invatare critic: 0.0001
- Entropy beta: 0.01
- Entropy beta minim: 0.001
- Entropy decay: 0.9999
- Optimizator actor: Adam
- Optimizator critic: Adam

3.2.4 Antrenare

Am antrenat modelul timp de 3000 de episoade. Am salvat cele mai bune 3 modele si le-am testate pe 150 de episoade. Rezultatele nu au fost la fel de bune ca cele ale agentului DQN, insa nici nu am rulat pe un numar foarte mare de episoade.

- model 1, episod antrenare: 2610, scor: 8272
- cel mai mare scor: 6616

- piesa 2048: 0 ori
- piesa 1024: 0 ori
- piesa 512: 11 ori

4 Documentare / Cercetare

Am analizat proprietatile si constrangerile jocului 2048 pentru a alege modele de Reinforcement Learning potrivite.

De exemplu, o provocare o reprezinta numarul urias de stari posibile, intrucat pe fiecare pozitie poate fi orice putere a lui 2, nu puteam folosi un algoritm care foloseste un tabel de stari. Alta dificultate a constat in factorul aleator al jocului, avand o anumita stare si alegand o anumita actiuni poate duce in stari diferite, unele mai mult sau mai putin favorabile, ceea ce poate induce agentul in confuzie. De asemenea, jocul necesita o strategie pe termen lung de la inceput, este foarte usor sa se ajunga intr-o stare terminala alegand actiuni aleatoare sau nepotrivite, insa in acelasi timp agentul are nevoie sa ajunga la stari mai avansate pentru a putea descoperi o strategie.

Balansarea explorarii cu alegerea actiunii identificata ca fiind cea mai buna a fost extrem de importanta, intrucat agentul nu reusea sa invete daca era lasat sa exploreze prea mult la inceput.

De asemenea, pentru a vedea rezultate a fost necesar un numar mare de episoade, fiind dificil uneori sa ne dam seama daca problema era la model/ hiperparametri/ mediu sau la numarul prea mic de episoade. In cele 2 articole pe care le-am citit agentii obtineau rezultate bune si ajungeau la valoarea 2048 dupa 20000-60000 de episoade, insa acest lucru poate fi dificil de pus in practica din cauza timpului de antrenare, mai ales daca reteaua utilizata este complexa.

Bibliografie

[1] https://medium.com/@qwert12500/playing-2048-with-deep-q-learning-with-pytorch-implementation-4313

[2] https://towardsdatascience.com/a-puzzle-for-ai-eb7a3cb8e599