Documentație- 2048

Partea I – 2048

2048 este un joc video puzzle cu plăci glisante pentru un jucător, scris de dezvoltatorul web italian Gabriele Cirulli și publicat pe GitHub. Jocul constă in glisarea unor piese (numerotate cu puteri ale lui 2) in una din direcțiile sus, jos, stânga sau dreapta pe o grilă de 4 pe 4 cu obiectivul de a le combina pentru a crea o piesă cu numărul 2048. Cu toate acestea, se poate continua jocul după ce a fost atins obiectivul, pentru a crea piese cu numere cat mai mari.

Jocul începe cu două piese deja în grilă, având o valoare fie de 2, fie de 4 (in implementarea din acest proiect se vor folosi doar 2 pentru simplicitate), iar o altă astfel de piesă apare într-un spațiu liber aleatoriu după fiecare tură. Piesele alunecă cât mai departe posibil în direcția aleasă până când sunt oprite fie de o altă piesa, fie de marginea grilei. Dacă două piese de același număr se ciocnesc în timpul mișcării, ele se vor îmbina într-o piesa cu valoarea totală a celor două piese care s-au ciocnit. Piesa rezultată nu se poate îmbina cu o altă piesă creata în aceeași mișcare. Cea mai mare plăci posibilă este 131.072, din cauza numărului limitat de spatii.

Dacă o mișcare face ca trei plăci consecutive de aceeași valoare să alunece împreună, se vor combina numai cele două plăci aflate cel mai departe de-a lungul direcției de mișcare. Dacă toate cele patru spații dintr-un rând sau coloană sunt umplute cu plăci de aceeași valoare, o mișcare paralelă cu acel rând/coloană va combina primele două și ultimele două piese. Când jucătorul nu are mișcări legale (nu există spații goale și nici piese adiacente cu aceeași valoare), jocul se termină.

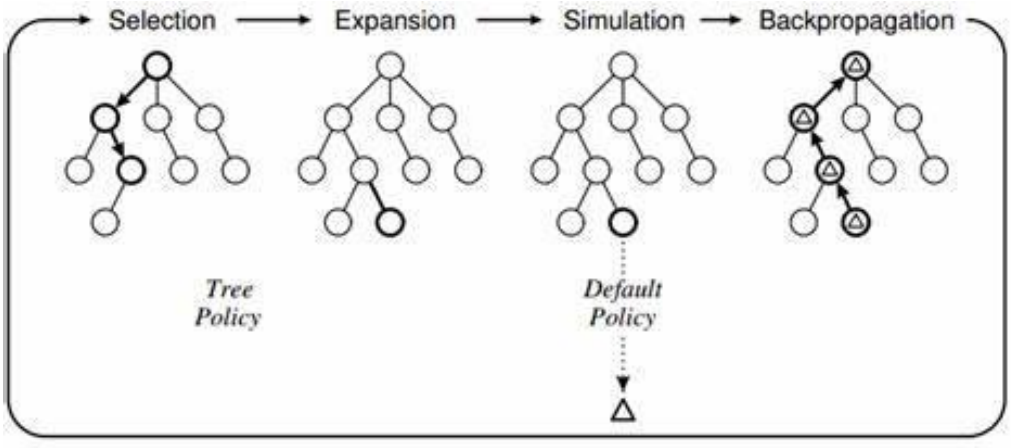
A fost scris în JavaScript și CSS într-un weekend și a fost lansat pe 9 martie 2014 ca software gratuit și open-source, cu licența MIT. Versiunile pentru iOS și Android au urmat în mai 2014. 2048 trebuia să fie o versiune îmbunătățită a altor două jocuri, ambele fiind clone ale jocului iOS “Threes” lansat cu o lună mai devreme. Cirulli însuși a descris 2048 ca fiind „similar din punct de vedere conceptual” cu Threes.

Partea II – Pure Monte Carlo tree search

Generalități algoritm

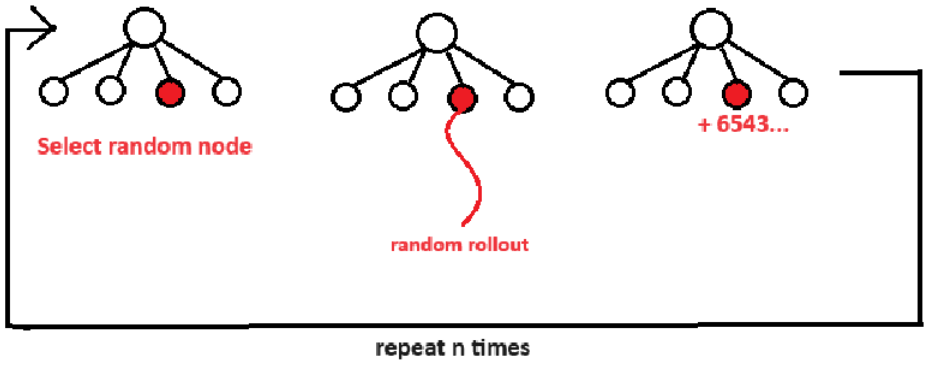
Monte Carlo Tree Search (MCTS) este un algoritm de luare a deciziilor utilizat în mare parte în inteligența artificială pentru jocuri și alte domenii cu spații de decizie vaste. Este popular în special jocuri precum șah, Go și alte jocuri similare. In acest proiect, o varianta a MCTS-ului este folosita pentru a crea un AI care joacă 2048.

Monte Carlo Tree Search se bazează pe un arbore de căutare prin expansiunea și explorarea iterativă a nodurilor. Algoritmul începe cu un nod rădăcină și expandează succesiv arborele, adăugând noduri pentru fiecare acțiune posibilă și simulând jocuri pentru a estima valoarea acestora. Această simulare se face de obicei printr-un proces de selecție, expansiune, simulare și back-propagation.



Aplicarea căutării arborelui Monte Carlo în jocuri se bazează pe multe playout-uri, numite și roll-outs. În fiecare playout, jocul se rulează până când ajunge la o stare finala, selectând fiecare mișcare la întâmplare. Rezultatul final al jocului din fiecare playout este apoi folosit pentru a da valori nodurilor din arbore, astfel încât nodurile mai bune sunt mai probabil să fie alese în playout-uri viitoare.

Cel mai elementar mod de a folosi aceasta tehnica este de a aplica același număr de playout-uri după fiecare mișcare legală a jucătorului curent, alegându-se apoi mutarea care a dus la cele mai multe victorii (sau a adunat cel mai mare scor). Eficiența acestei metode – numită Pure Monte Carlo Game Search – crește adesea cu timpul, odată ce numărul de playout-uri creste.



Această procedură de bază poate fi aplicată oricărui joc ale cărui poziții au în mod necesar un număr finit de mișcări și o lungime finită. 2048 este un astfel de joc, din moment ce mișcările sunt foarte limitate: sus, jos, stânga, dreapta, iar jocul ori se termina la obținerea tile-ului 2048, ori la ultimul tile posibil, 131,072(2^17). Pure Monte Carlo nu are nevoie de o funcție de evaluare explicită: această simpla implementare a mecanicii jocului este suficientă pentru a explora tabla de 4x4.

Implementare

Algoritmul de pure MCTS a fost implementat in C++, folosindu-se de un environment public de pe GitHub, scris tot in C++ si modificat unde a fost necesar.

In primul rând, algoritmul funcționează prin citirea unei table de joc dintr-un fișier, “tree.txt”, si returnarea următoarei mișcări, cea pe care o considera a fi optima.

State-urile din arbore sunt reprezentate printr-un struct care retine board-ul curent, iar valorile sunt calculate prin asocierea fiecărei valori de pe piesa cu un scor separat intr-un unordered map, apoi doar adunând valorile de pe tabla. Funcția “score” calculează punctajul astfel, luând un nod ca parametru.

```
{1, 0},
{2, 4},
{3, 16},
{4, 48},
{5, 128},
{6, 320},
{7, 768},
{8, 1792},
{9, 4096},
{10, 9216},
{11, 20480},
{12, 45056},
{13, 98304}
```

Valorile sunt reținute in board, pentru a salva memorie, in formă de log2(x), din moment ce piesele sunt toate puteri ale lui 2. De asemenea, simulările din fiecare nod sunt șterse odată ce valoarea lor finala e calculata, memoria fiind alocata dinamic. De asemenea, in algoritmii cu beam search, nu se reținea tot arborele, ci doar 2 vectori care interschimbau cate un sir de copii cu un sir de stări cu generări de 2 pe acești copii. Funcția generate_2 generează un 2 (de fapt este un 1 = log2(2)) intr-un loc aleatoriu pe o tabla valida de joc.

Environmentul de 2048 implementat avea următoarele mișcări, in sens orar:

move_up -> corespunde cu 0;

move_right -> corespunde cu 1;

move_down -> corespunde cu 2;

move_left -> corespunde cu 3

Acestea returnau o tabla de 2048 după ce una din cele 4 mișcări a fost aplicata pe aceasta, si modifica variabila “ok” pentru a semnala daca mișcarea este valida sau nu. Am modificat funcțiile astfel încât sa modifica direct tabla trimisa ca parametru funcției, pentru o implementare mai ușoară.

MCTS-ul este implementat in funcția “MCTS”, care ia ca parametru nodul root de unde sa înceapă căutarea. Inițial, se calculează numărul de spatii goale din matrice, pentru a stabili cate random rollouts sa facă algoritmul: in caz ca sunt multe spatii goale, poate sa facă mai puține, șansa de pierdere fiind minima, însă daca sunt foarte puține locuri libere, de exemplu doua, face mai multe rollouts pentru a fi sigur ca mutarea aceea este cea buna. Urmează sa se simuleze numărul selectat de rollouts din root-ul dat, iar depinzând de ce mutare inițială a luat fiecare simulare (dintre copiii de sus, jos, stânga, si dreapta ai root-ului), se adaugă la scorul total al acelei mutări scorul final al rulării, dar creste si numărul de vizite in acel nod. Astfel, se calculează la final media scorurilor simulărilor trecute prin fiecare nod, iar cel cu media cea mai mare este ales.

Simularea din stare este data de funcția simulate_random_play, care alege aleator dintre cele 4 direcții, aplicând apoi funcția de mișcare specifica, si repeta alegerea unei mișcări in caz ca cea din urma nu era legala. In caz ca nu se mai găsește nicio mișcare valida, se returnează scorul actual.

```
switch (move) {
    case 0: move_up(legal_move, new_state->board); break;
    case 1: move_right(legal_move, new_state->board); break;
    case 2: move_down(legal_move, new_state->board); break;
    case 3: move_left(legal_move, new_state->board); break;
}
```

Algoritmul mai are o varianta in care calculează scorul după cate mișcări a făcut pana a ajuns la game over, adică cat a supraviețuit, care are performante similare, însă scorul bazat pe tile-uri este mult mai intuitiv si potrivit jocului, care are ca scop crearea tile-ul de 2048.

De asemenea, folosind un UI public, tot de pe GitHub, am implementat o interfață in Python pentru a testa manual algoritmul, care afișează pas cu pas ce mișcări sunt decise de algoritm. Modificând fișierul “tree.txt” ajutând-se de mișcarea oferita de MCTS, pentru a citi repetat din el, se face legătura dintre stări.

Acesta nu a fost primul algoritm implementat, ideea inițială fiind implementarea unui arbore cu min-max beam search (ordonam copii generați descrescător, iar generările de 2 crescător, pentru a antrena pe cele mai nefavorabile situații) de lungime si lățime variabila (apoi doar beam search), care urma sa se creeze la fiecare mutare a modelului, cu mai multe variante de funcție de evaluare, însă nu ajungea la scoruri prea mari, trebuindu-i prea multe euristice.

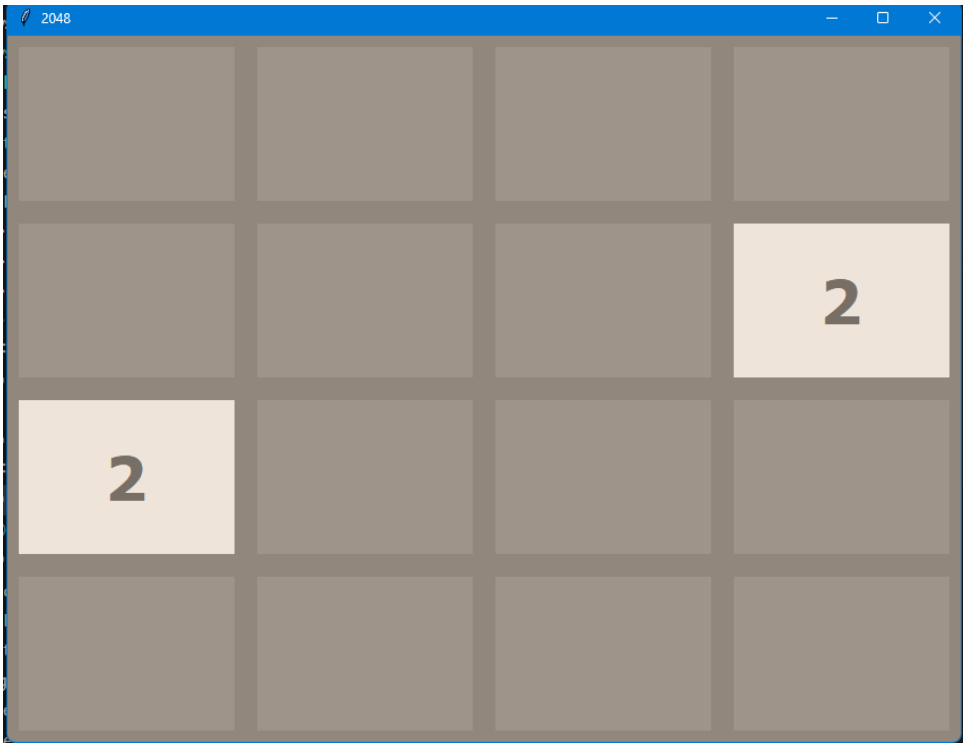
Funcția de evaluare a acestui algoritm de bazează pe o matrice tip “snake”, care încearcă sa mențină valoarea cea mai mare in coltul dreapta jos, celelalte urmând-o intr-o forma in care s-ar putea combina cu piese de valori consecutive.

```
int w_matrix[4][4]= {
        {0, 0, 0, 0},
        {1, 2, 3, 4},
        {8, 7, 6, 5},
        {9, 10, 11, 12}
};
```

Mai jos se vede sortarea crescătoare (după scorul evaluat cu funcția de evaluare) a tablelor cu generări de 2 aleatorii după o anumita mișcare, si cea descrescătoare a copiilor generați după aplicarea tuturor mișcărilor valide pe generările anterioare. Vor urma sa fie selectate, pentru fiecare sortare, primele n noduri, n fiind un număr dat ca parametru. De asemenea, se va repeta procesul de m ori (adâncimea arborelui), m fiind de asemenea dat. Pentru arbore sunt reținuți si interschimbați mereu doi vectori.

```
number_of_generations = make_generations(generations, children, min(child_nr, width/20));
// sort the generations increasingly
sort(generations, generations + number_of_generations+1, child_cmp);
// generate all possible moves
child_nr = make_children(children, generations, min(number_of_generations, width));
if(!child_nr) break;
// sort all moves decreasingly
sort(children, children + child_nr, gen_cmp);
best_move=children[0].root_move;
```

O stare inițială a unui board de 2048 folosind UI-ul implementat:



Starea finala a board-ului după ce a fost aplicat o singura data algoritmul Pure MCTS pe el:

Performanțe

Pure MCTS:

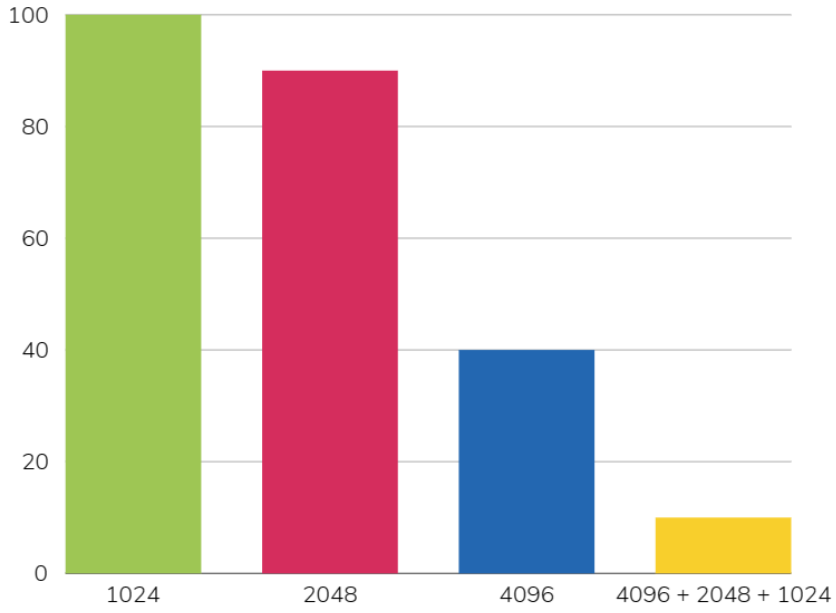
1024: 100% din cazuri

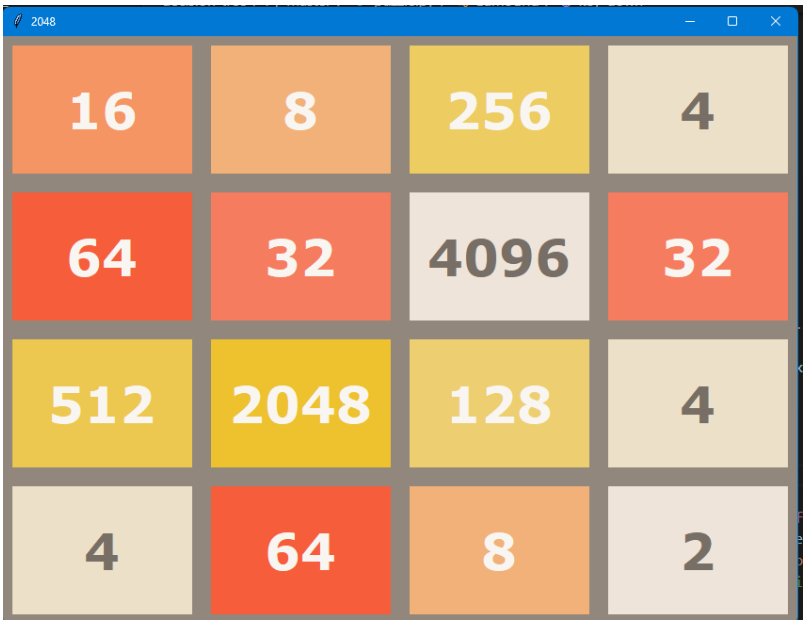
2048: 90% din cazuri

4096: 40% din cazuri

4096 + 2048 + 1024 pe același board: sub 10% din cazuri

8192: nicio simulare nu a ajuns la aceasta piesa





Minmax beam search:

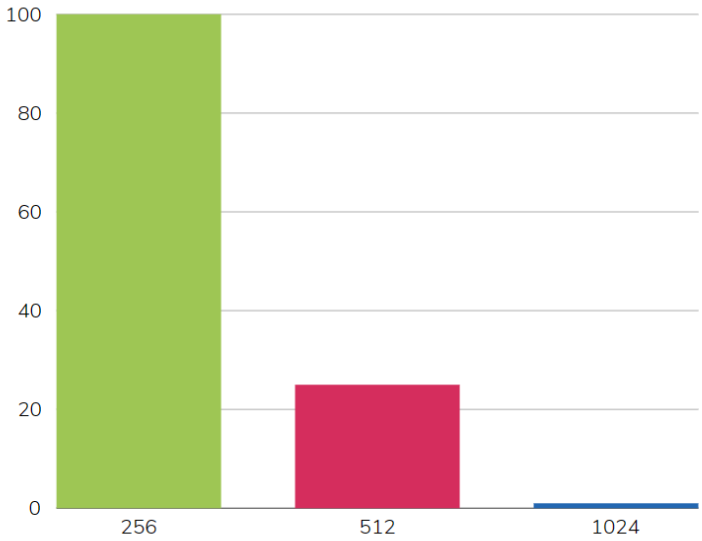
Algoritmul, selecționând doar anumite stări, aproximativ asemănătoare in scor, pentru a le explora, nu are o viziune foarte îndepărtată legata de următoarea mișcare, fiind puțin mai greedy ca MCTS. Astfel, rezultatele nu sunt la fel de bune, cea mai mare valoare pe care am obținut-o fiind 1024. Deși am experimentat cu diferite width-uri si depth-uri ale arborelui, rezultatele au fost surprinzător aceleași, uneori chiar mai rele din cauza overflow-ului.

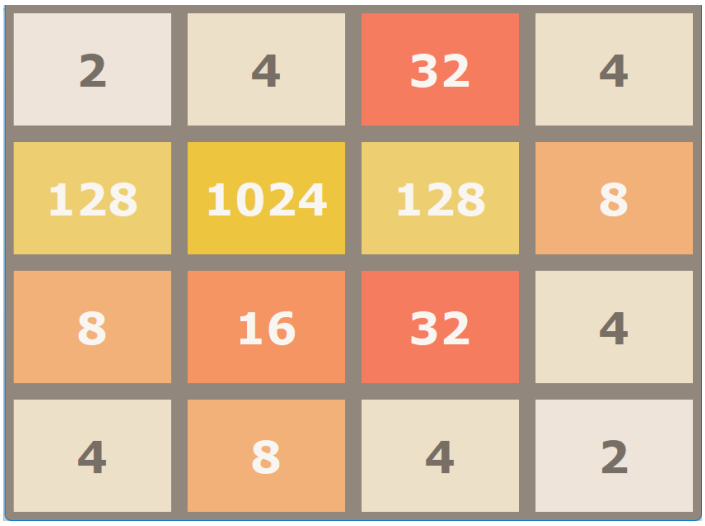
256: 100% din cazuri

512: 25% din cazuri

1024: 1% din cazuri

2048: nicio simulare nu a ajuns la aceasta piesa





Partea III – Deep-Q Network

Generalitati algoritm

Q-Learning este o tehnica de reinforcemenet learning, de tip off-policy, care identifica cea mai buna actiune posibila, avand in vedere starea curenta a agentului. Aceasta este utilizata in jocuri sau controlul robotilor. Q-learning foloseste o funtie care mapeaza starile si actiunile la o valoare numerica ce indica cat de benefica este o actiune intr-o anumita stare. Pentru actualizarea valorilor, algoritmul Q-Learning foloseste ecuatia Bellman: $Q(s,a)=(1-\alpha)\cdot Q(s,a)+\alpha\cdot[R(s,a)+\gamma\cdot \max Q(s',a')]$.

Antrenarea modelului consta in simularea repetata a programului(jocului). In timpul antrenarii, agentul trebuie sa gaseasca un echilibru intre explorare (incercarea de actiuni noi) si exploatare (alegerea actiunilor pe baza cunostiintelor existente). In general se utilizeaza o strategie epsilon-greedy, unde cu o probabilitate ϵ se efectueaza o actiune aleatoare si cu o probabilitate de 1- ϵ se alege actiunea cu valoarea Q maxima.

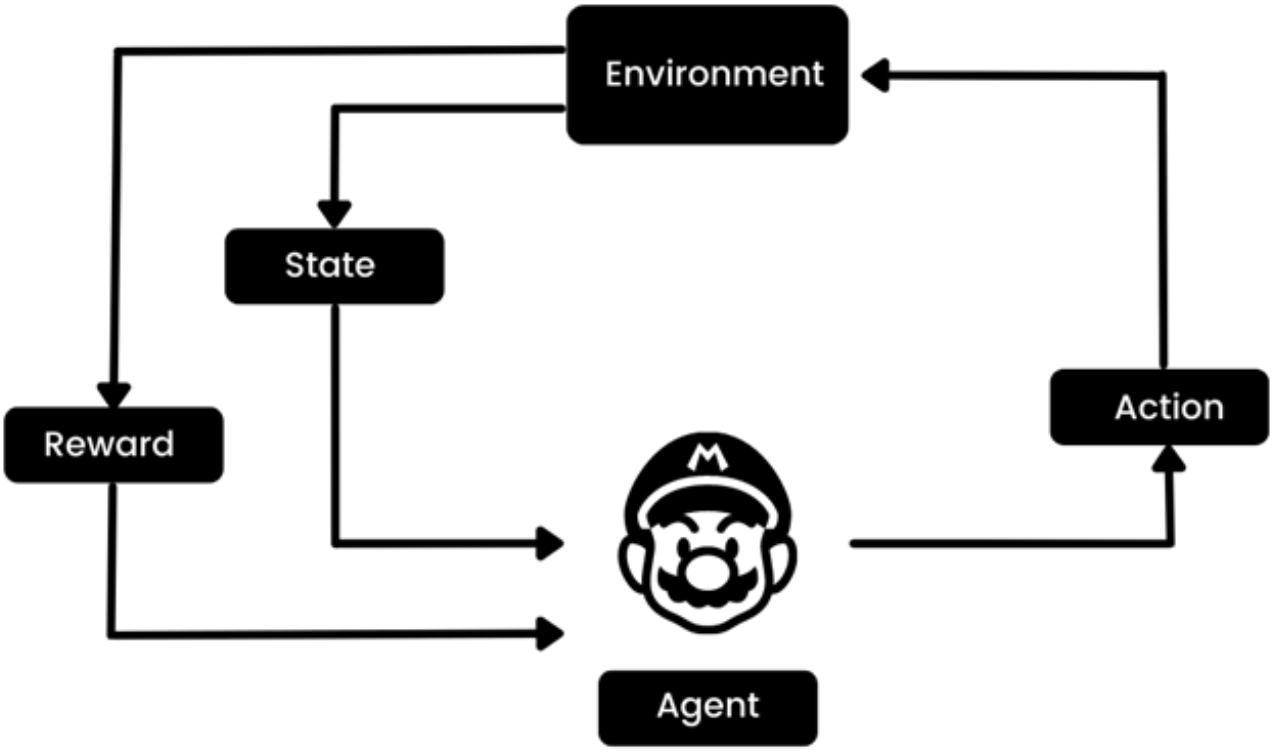


Fig 1- Diagrama Q-Learning

Deep Q-Learning este o extensie a algoritmului Q-learning care utilizeaza o retea neuronala pentru a invata si a aproxima functia Q. Se prefera utilizarea Deep Q-Learning in locul algoritmului de Q-Learning datorita dimensiunilor spatiale reduse si a convergentei rapide.

Primul pas in realizarea algoritmului este initializarea unei retele neuronale care va servi drept functie Q aproximativa. Antrenarea modelului se va face pe parcursul a mai multor episoade, agentul plecand mereu din starea initiala a mediului. La fiecare pas, agentul decide ce actiune va face, obseva starea curenta, actiunea, recompensa si starea urmatoare pe care le va stoca sub forma unui tuplu intr-un buffer.

Actualizarea parametrilor retelei se realizeaza urmand urmatorii pasi: se preia un batch aleator de experiente din Replay Memory, se calculeaza valoarea Q pentru fiecare tuplu, utilizandu-se reteaua tinta (o copie a retelei principale ai carei parametrii se actualizeaza periodic pentru a oferi persistenta), se aplica o functie de pierdere intre Q estimat de catre reteua principala si Q calculat de modelul tinta, iar valoarea obtinuta este utilizata pentru a face backpropagation in modelul principal.

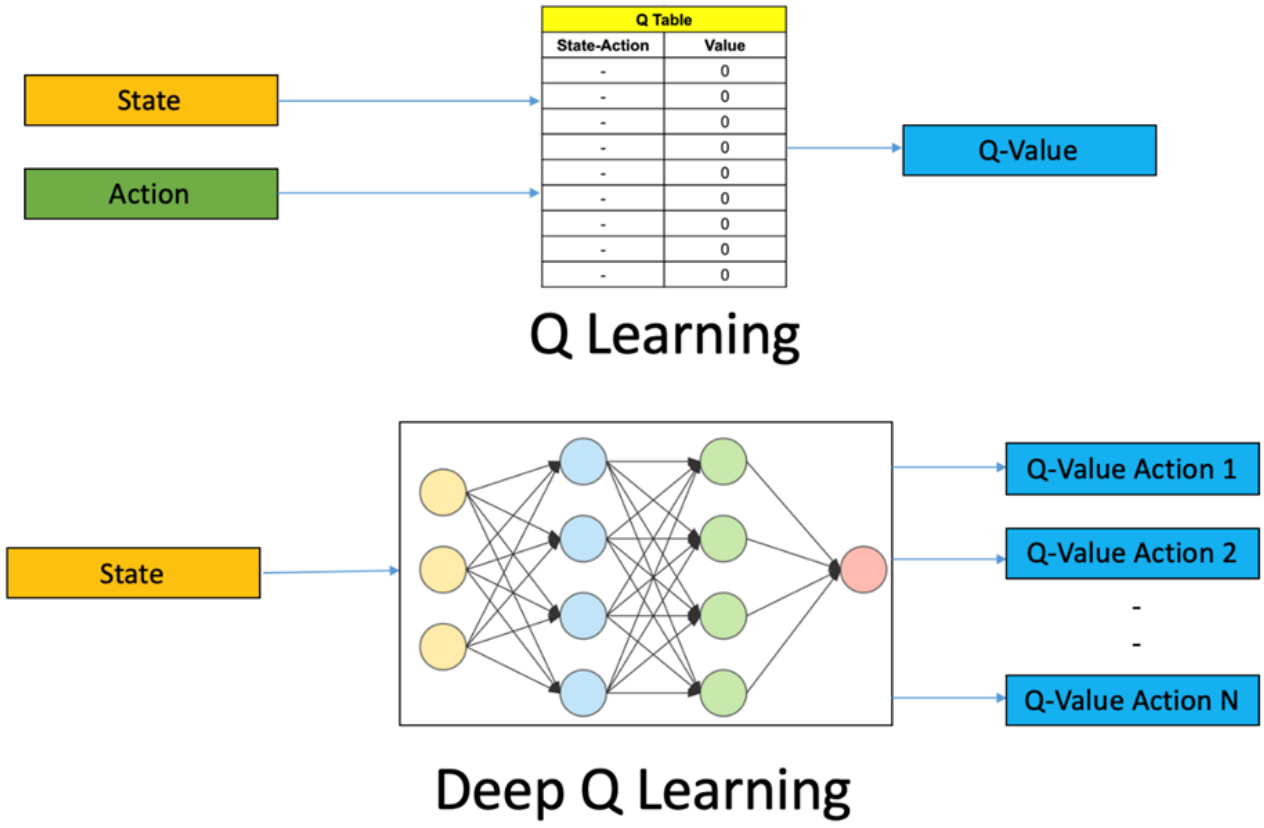


Fig 3- Diferente Deep Q-Learning vs Q-Learning

Implementare

Implementarea algoritmului DQL a fost realizata in Python, utilizandu-se libraria Pytorch pentru partea de antrenare si de constructie a retelei neuronale.

Fisierul DQL_Networks contine modelul de deep-learning si implementarea Replay Buffer-ului. Modelul este alcatuit din trei straturi lineare, non-liniaritatea fiind asigurata cu ajutorul functiei de activare ReLU. Replay Buffer-ul este definit ca o lista dublu inlantuita, deoarece operatiile de adaugare si de stergere a primului si a ultimului element sunt executate rapid. Functia sample din clasa ReplayBuffer returneaza un numar de batch_size sample-uri alese random din buffer.

```
class DQN(nn.Module):
    def __init__(self, state_size, action_size, hidden_size=64):
        super(DQN, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(state_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size * 2),
            nn.ReLU(),
            nn.Linear(hidden_size * 2, action_size)
        )

    def forward(self, x):
        return self.net(x)
```

Fig 4-Implementare model

Fisierul DQL_Agent utilizeaza modelul si buffer-ul definite in fisierul DQL_NETWORK pentru a construi agentul. In momentul in care este initializat, agentul defineste urmatoarele variabile: numarul de stari, numarul de actiuni (4), Reply Buffer-ul cu o dimensiune egala cu 10.000, modelul principal, modelul target (copie a modelului principal), optimizatorul de tip Adam ce are un learning rate egal cu 10^-4 si functia de pierdere (mean square error).

Metoda choose_action va primi, in momentul antrenarii starea curenta si va selecta urmatoarea actiune. In selectarea actiunii, agentul utilizeaza epsilon pentru a gasi un echilibru intre alegerea unui actiuni aleatoare (exploatare) si alegerea actiunii cu Q-value maxim (explorare).

```
def choose_action(self, state):
    if np.random.rand() <= self.epsilon:
        return np.random.choice(self.action_space)
    else:
        state = torch.from_numpy(state).float().unsqueeze(0)
        with torch.no_grad():
            action_values = self.model(state)
        return np.argmax(action_values.cpu().data.numpy())
```

Fig 5- choose_action din DNQ Agent

Metoda learn are ca parametru numarul de sample-uri pe care le extrage din buffer. Din buffer se extrag starile, actiunile, reward-urile, starile urmatoare si valorile variabilelor done care indica daca jocul s-a terminat. Ulterior se extrage Q_expected pentru fiecare stare si se calculeaza Q_target folosind modelul target. Cu ajutorul celor doua variabile se calculeaza eroarea folosind MSE. Cu ajutorul erorilor se modifica parametrii modelului principal. Ultimul pas este scaderea valorii lui epsilon (parametru care manageriaza alegerea unei actiuni aleatoare sau celui cu Q-value maxim), astfel incat acesta sa se axeze mai mult pe exploatare.

Metoda update_target_network este utilizata pentru update-ul parametriilor modelului target.

Fisierul Environment.py creaza environmentul pentru jocul de 2048. In momentul in care este creat un Environment sunt initializate numarul de stari, spatiul de actiuni (up, down, left, right) si starea jocului (Finalizat <=> done=true , In derulare <=> done=false).

Functia reset reseteaza jocul la starea initiala.

Functia get_next_state primeste ca parametru o actiune si verifica daca actiunea este legala, daca este adauga intr-un patratel aleator un 2.

Functia calculate_score_increase primeste ca parametrii starea veche si starea noua si calculeaza diferenta de scor.

Functia calculate_tile_merges primeste ca parametrii starea veche si cea noua si returneaza scorul obtinut prin combinarea a doua valori.

Functia evaluate_board primeste ca parametru o stare si returneaza un scor in functie de pozitia pe care se afla cele mai mari valori.

Functia calculate_reward primeste ca parametrii starea veche si cea noua. Aceasta va returna reward-ul, combinand rezultatele obtinute din functiile calculate_score-_increase, calculate_tile_merge, evaluate_board la care adauga un bonus pentru casetele cu valori mai mari de 512 si daca jocul inainteaza, scazand din valoare daca jocul sta pe loc sau este pierdut.

Functia step primeste ca parametru o actiune pe care o executa, rezultand o noua stare. Se verifica daca jocul este finalizat, iar apoi se calculeaza reward-ul starii curente. Functia returneaza noua stare, reward-ul acesteia, starea jocului si informatii despre joc .

```
def step(self, action: int) -> Tuple[np.ndarray, int, bool, str]:
    """
    ::next_state:: -> n x n matrix containing the state resulted from using action
    ::action:: -> integer from [0,1,2,3] representing which way we slide the tiles
    ::self.done:: -> indicates if the game is over
    ::info:: -> tells us if the game is won/lost, empty string if terminated = False
    """
    info = ""
    next_state = self.get_next_state(action)
    game_state = logic.game_state(next_state)

    if game_state != "not over":
        info = game_state
        self.done = True

    reward = self.calculate_reward(self.state, next_state)
    self.state = next_state
    return next_state, reward, self.done, info
```

Fig 6-Step Function

Fisierul dqn_workspace este fisierul in care are loc antrenarea. Se initializeaza un Environment de 2048 si se stabileste numarul de stari si actiuni. Ulterior este initializat agentul folosind clasa DQLAgent.

Antrenarea se desfasoara in 1000 de episoade. Pentru fiecare episod se reseteaza environmentul care reprezinta starea initiala. Cat timp numarul de pasi maxim nu a fost atins sau jocul nu s-a incheiat, se alege urmatoarea actiune, acestea este executata, rezultand o noua stare cu un nou reward. Acestea impreuna cu starea initiala, actiunea si starea jocului sunt introduse in buffer.
Noua stare devine starea curenta. La fiecare 20 de episoade se va face update-ul modelului target.

```
sys.path.append("../decision_tree/Py_Master")
from Environment import Environment_2048
  env = Environment_2048()
state_size = env.size * env.size
action_size = len(env.action_space.action_space)
agent = DQNAgent(state_size, action_size)
def train_dqn(agent, env, episodes, batch_size, max_steps):
    for e in range(episodes):
        state = env.reset()
        state = np.reshape(state, [1, agent.state_size])
        for time in range(max_steps):
            action = agent.choose_action(state, env.action_space.action_space)
            next_state, reward, done, _ = env.step(action)
            next_state = np.reshape(next_state, [1, agent.state_size])
            agent.memory.add(state, action, reward, next_state, done)

            state = next_state
            if done:
                print(f"episode: {e}/{episodes}, score: {time}, e: {agent.epsilon:.2}")
                break

            if len(agent.memory) > batch_size:
                agent.learn(batch_size)

        # Update target model every 20 episodes
        if e % 20 == 0:
            agent.update_target_network()
    train_dqn(agent, env, 1000, 64, 3000)
```

Fig 7-Antrenarea Agentului