Project 3 - Reinforcement Learning

Adela Morar

Ianuarie 2026

1 Introducere

Acest proiect urmărește implementarea unor agenți inteligenți capabili să ia decizii optime în medii stochastice, folosind conceptele de Învățare prin Recompensă (Reinforcement Learning - RL). Vom explora atât tranziția de la planificarea offline, unde modelul mediului este cunoscut (MDP), cât și de la învățarea online, unde agentul trebuie să exploreze pentru a învăța (Q-Learning).
Obiectivul principal este controlul unor agenți în mediile *Gridworld* și *Pacman*, trecând prin 9 cerințe care acoperă algoritmi fundamentali precum Value Iteration și Q-Learning.

2 Procese Decizionale Markov (MDP)

Un Proces Decizional Markov (MDP) oferă cadrul matematic pentru modelarea problemelor de luare a deciziilor în situații în care rezultatele sunt parțial aleatoare și parțial sub controlul unui agent.
Un MDP este definit formal de tuplul (S, A, T, R, γ):

- **S (States):** Mulțimea tuturor stărilor posibile în care se poate afla agentul.
- **A (Actions):** Mulțimea acțiunilor disponibile agentului.
- **T (Transition Function):** $T(s, a, s') = P(s'|s, a)$, probabilitatea de a ajunge în starea s' din starea s executând acțiunea a.
- **R (Reward Function):** $R(s, a, s')$, recompensa primită pentru tranziție.
- γ **(Discount Factor):** Un număr între 0 și 1 care determină importanța recompenselor viitoare față de cele imediate.

Scopul într-un MDP este găsirea unei *politici* $\pi(s)$ care maximizează recompensa totală cumulată.

3 Planificare Offline: Value Iteration

Primele cerințe ale proiectului se concentrează pe rezolvarea MDP-urilor cunoscute folosind algoritmul *Value Iteration*.

3.1 Conceptul Teoretic

Value Iteration este un algoritm de programare dinamică ce calculează iterativ valorile optime $V^*(s)$ pentru fiecare stare. Se bazează pe ecuația de actualizare Bellman:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')] \tag{1}$$

3.2 Q1: Implementarea Value Iteration

Această cerință presupune dezvoltarea unui agent de planificare offline care calculează valorile optime ale stărilor într-un MDP cunoscut. Agentul rulează un algoritm iterativ înainte de a începe interacțiunea cu mediul.

3.2.1 Modelul Matematic

Algoritmul se bazează pe ecuația de actualizare Bellman (1). La fiecare iterație k, valoarea unei stări s este actualizată pe baza valorilor succesorilor din iterația anterioară $k-1$. Aceasta asigură propagarea informației despre recompense prin întregul spațiu al stărilor.
O vizualizare a convergenței valorilor după 5 iterații pe un grid simplu poate fi observată mai jos:

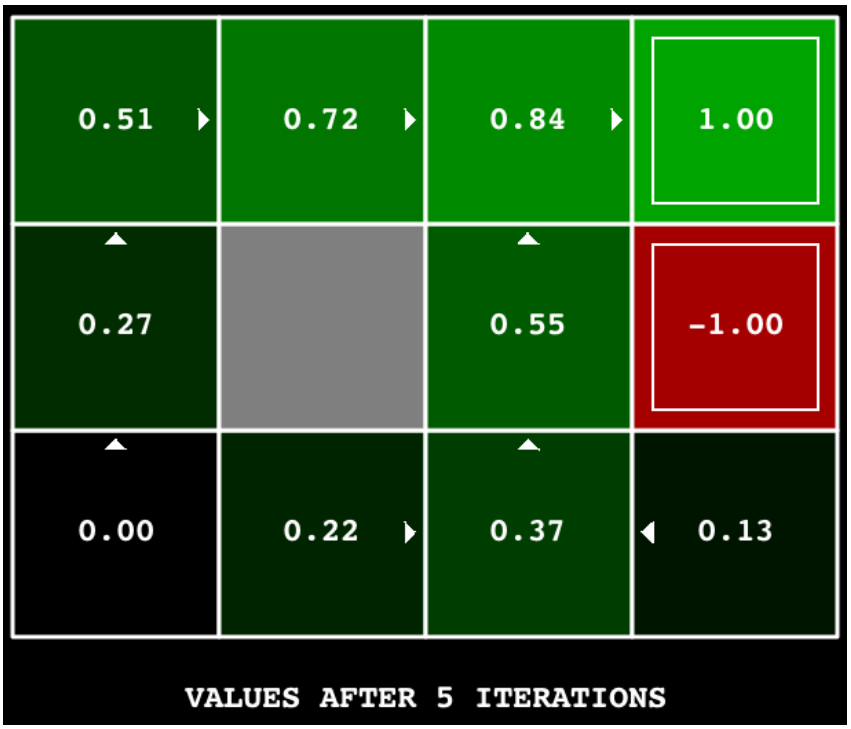

Figure 1: Vizualizarea valorilor și a politicilor (săgeți) după 5 iterații

3.2.2 Algoritmul și Implementarea

Implementarea din `ValueIterationAgent` respectă principiul "Batch Update". Aceasta înseamnă că valorile din iterația curentă (V_{k+1}) sunt calculate exclusiv pe baza unei copii a valorilor din iterația anterioară (V_k).

Pseudocodul de mai jos ilustrează logica centrală implementată:

Algorithm 1 Batch Value Iteration

Input: MDP (S, A, T, R, γ), număr iterații K
Inițializare: $V[s] \leftarrow 0, \forall s \in S$
for $i \leftarrow 1$ to K **do**
    $V_{new} \leftarrow$ vector gol ▷ Stocăm noile valori temporar
    for fiecare stare $s \in S$ **do**
        if s este stare terminală **then**
            $V_{new}[s] \leftarrow 0$
        else
            $Q_{max} \leftarrow -\infty$
            for fiecare acțiune $a \in A(s)$ **do**
                $q \leftarrow$ ComputeQValue(s, a, V)
                if $q > Q_{max}$ **then**
                    $Q_{max} \leftarrow q$
                end if
            end for
            $V_{new}[s] \leftarrow Q_{max}$
        end if
    end for
    $V \leftarrow V_{new}$ ▷ Actualizare sincronă
end for
return V

Calculul efectiv al valorii Q (funcția auxiliară utilizată mai sus) se realizează prin însumarea recompenselor ponderate cu probabilitățile de tranziție:

Algorithm 2 Calculul Q-Value ($Q(s, a)$)

function ComputeQValue(s, a, V)
    $q \leftarrow 0$
    for fiecare s' și probabilitate P din $T(s, a)$ **do**
        $reward \leftarrow R(s, a, s')$
        $q \leftarrow q + P \cdot (reward + \gamma \cdot V[s'])$
    end for
    return q
end function

Pentru determinarea politicii optime într-o stare (metoda `computeActionFromValues`), agentul alege pur și simplu acțiunea care maximizează funcția ComputeQValue:

$$\pi^*(s) = \arg\max_a Q(s, a)$$

3.3 Q2: Analiza Podului (Bridge Crossing)

Această cerință analizează comportamentul agentului într-un mediu specific, `BridgeGrid`. Harta prezintă o situație de risc versus recompensă:

- O stare terminală sigură, dar cu recompensă mică (+1), aflată aproape de start (stânga).
- O stare terminală valoroasă (+10), aflată la capătul unui "pod" îngust (dreapta).
- De ambele părți ale podului există prăpastii cu recompensă negativă majoră (-100).

O reprezentare vizuală a acestui mediu și a valorilor (care arată riscul traversării în condiții implicite) este prezentată mai jos:

Figure 2: Mediul BridgeGrid: Valorile arată riscul ridicat (negativ) pe pod în cazul zgomotului mare

Cu parametrii impliciți, agentul are șansa să cadă de pod. Sarcina a fost ajustarea unui singur parametru pentru a încuraja traversarea.

3.3.1 Semnificația Parametrilor

Pentru a înțelege decizia agentului, trebuie să definim cei doi parametri care influențează calculul utilității:

- **Factorul de Discount (γ):** Acesta determină cât de mult prețuiește agentul recompensele viitoare comparativ cu cele imediate.
  - $\gamma \approx 0$: Agentul este "lacom" și miop; preferă recompensa imediată (+1).
  - $\gamma \approx 1$: Agentul este dispus să aștepte și să călătorească mai mult pentru o recompensă mai mare (+10).
- **Zgomotul (Noise):** Reprezintă natura stochastică a mediului. Într-un gridworld, dacă agentul alege acțiunea "Nord", zgomotul este probabilitatea ca el să ajungă, de fapt, în stările adiacente (Est sau Vest).
  - *High Noise:* Incertitudine mare. Pe un pod îngust, zgomotul mare înseamnă o probabilitate ridicată de a cădea în prăpastie, indiferent de intenția agentului.
  - *Low Noise:* Mișcare deterministică. Agentul ajunge exact unde intenționează.

3.3.2 Soluția Implementată

În configurația implicită (Noise = 0.2), riscul de a cădea de pe pod este prea mare. Chiar dacă recompensa de la capăt este +10, speranța (Expected Value) traversării devine negativă din cauza penalizării enorme (-100) aplicate în cazul căderii.
Pentru a rezolva problema, am modificat parametrul de zgomot, reducându-l drastic (foarte aproape de 0), păstrând factorul de discount neschimbat.

```
answerDiscount = 0.9
answerNoise = 0.01
```

Justificare: Prin setarea zgomotului la o valoare aproape nulă (0.01), eliminăm incertitudinea tranzițiilor. Agentul poate acum să planifice traversarea podului știind că acțiunea *Move East* va rezulta în deplasarea spre Est și nu în căderea în prăpastie (Nord sau Sud). Astfel, utilitatea drumului lung devine maximă, iar agentul alege să traverseze.

3.4 Q3: Politici și Parametri (DiscountGrid)

Această cerință explorează modul în care modificarea parametrilor fundamentali ai unui MDP poate schimba radical politica optimă a agentului, chiar dacă structura hărții rămâne neschimbată.
Mediul `DiscountGrid` prezintă o dilemă interesantă:

- Există două ieșiri: una apropiată dar mică (+1) și una îndepărtată dar mare (+10).
- Există două rute: una scurtă dar riscantă (pe lângă prăpastie, marcată cu roșu) și una lungă dar sigură (ocolitoare, marcată cu verde).

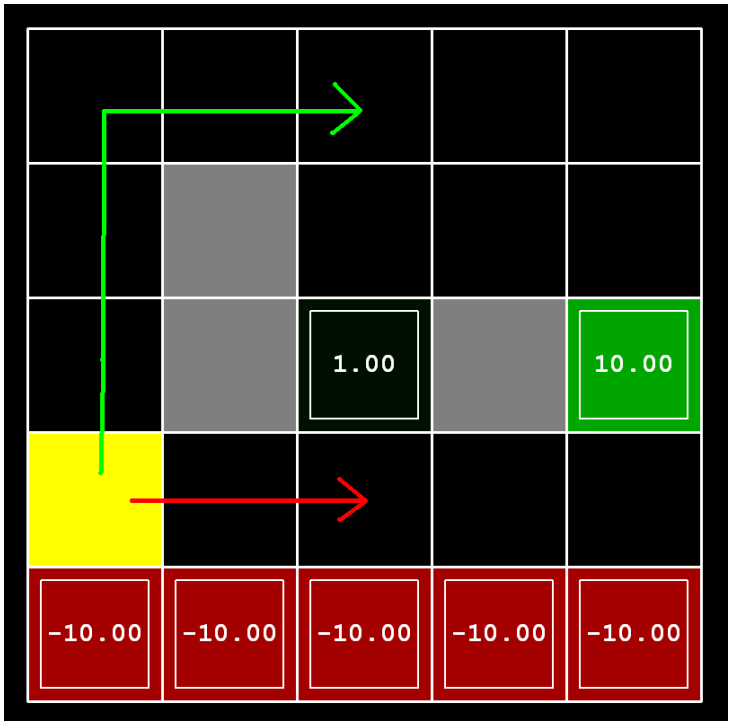

Figure 3: Cele două tipuri de rute în DiscountGrid: Riscantă (Roșu) vs. Sigură (Verde)

Scopul a fost identificarea combinațiilor de parametri ($Discount, Noise, LivingReward$) pentru a obține 5 comportamente distincte.

3.4.1 Rolul Parametrilor

Pe baza experimentelor, am definit influența fiecărui parametru astfel:

- **Discount (γ):** Controlează "răbdarea" agentului. O valoare mică devalorizează recompensele îndepărtate (precum cea de +10), făcând agentul să prefere ieșirea apropiată (+1).

- **Noise:** Controlează percepția riscului. Cu un zgomot mic, agentul are încredere să meargă pe marginea prăpastiei. Cu zgomot mare, marginea devine prea periculoasă statistic.
- **Living Reward:** Costul sau beneficiul trecerii timpului. O valoare mică forțează ieșirea rapidă. O valoare mare poate încuraja agentul să nu termine jocul niciodată.

3.4.2 Soluțiile Identificate

Am determinat valorile optime pentru fiecare dintre cele 5 scenarii cerute:

a) Preferă ieșirea apropiată (+1), riscând prăpastia

- *Valori:* Discount=0.2, Noise=0, LivingReward=0.01
- *Explicatie:* Discount-ul mic face ca ieșirea de +10 să pară neimportantă. Noise-ul 0 elimină riscul căderii, permițând drumul cel mai scurt pe lângă prăpastie.

b) Preferă ieșirea apropiată (+1), evitând prăpastia

- *Valori:* Discount=0.1, Noise=0.05, LivingReward=0.2
- *Explicatie:* Din nou, discount-ul mic prioritizează ieșirea apropiată. Totuși, introducerea unui mic zgomot (0.05) face drumul de pe margine prea riscant, forțând agentul să ocolească.

c) Preferă ieșirea îndepărtată (+10), riscând prăpastia

- *Valori:* Discount=0.9, Noise=0.01, LivingReward=0.01
- *Explicatie:* Discount-ul mare (0.9) face ca recompensa de +10 să fie foarte atractivă. Zgomotul redus permite asumarea riscului pentru a ajunge acolo rapid.

d) Preferă ieșirea îndepărtată (+10), evitând prăpastia

- *Valori:* Discount=0.9, Noise=0.2, LivingReward=0.5
- *Explicatie:* Dorim +10 (Discount mare), dar zgomotul ridicat (0.2) face ruta scurtă "sinucigașă" statistic. Agentul alege ruta lungă și sigură.

e) Evită ambele ieșiri (Nu termină jocul)

- *Valori:* Discount=0, Noise=0, LivingReward=1
- *Explicatie:* Deoarece a trăi (Living Reward = +1) este la fel de valoros ca a ieși prin cea mai apropiată ieșire, și cum ieșirea oprește acumularea de puncte, agentul preferă să acumuleze puncte la infinit rămânând în joc.

3.5 Q4: Prioritized Sweeping Value Iteration (Extra Credit)

Standard Value Iteration actualizează toate stările la fiecare pas, chiar dacă valoarea multora dintre ele nu se schimbă semnificativ. **Prioritized Sweeping** optimizează acest proces concentrând efortul de calcul asupra stărilor care suferă modificări majore și asupra predecesorilor acestora.

3.5.1 Conceptul Algoritmului

Ideea centrală este menținerea unei cozi de priorități cu stările care trebuie actualizate, ordonate după mărimea "erorii" (diferența dintre valoarea curentă și cea recalculată).

Parametrii cheie sunt:

- **Predecesori:** Pentru o stare s, predecesorii sunt toate stările p din care se poate ajunge în s cu probabilitate nenulă ($T(p, a, s) > 0$).
- **Theta (θ):** Un prag de toleranță mic. Dacă modificarea valorii unei stări este mai mică de θ, nu considerăm necesară propagarea schimbării către predecesori.

3.5.2 Implementare și Pseudocod

Am implementat clasa `PrioritizedSweepingValueIterationAgent`. Deoarece folosim modulul `util.PriorityQueue` (care este un min-heap), am introdus prioritățile cu semn negativ ($-diff$) pentru a simula un max-heap (procesăm întâi erorile cele mai mari).

Algoritmul implementat este descris mai jos:

Algorithm 3 Prioritized Sweeping

Input: MDP, θ, Iterations K
Pasul 1: Calcul Predecesori
for fiecare stare $s \in S$ **do**
    for fiecare acțiune a și succesor s' **do**
        if $P(s'|s, a) > 0$ **then**
            Adaugă s în lista de predecesori a lui s' ($Predecessors[s'] \leftarrow s$)
        end if
    end for
end for
Pasul 2: Inițializare Coadă
$PQ \leftarrow$ PriorityQueue()
for fiecare stare $s \in S$ (non-terminală) **do**
    $diff \leftarrow |V[s] - \max_a Q(s, a)|$
    $PQ.push(s, -diff)$ ▷ Prioritate negativă pentru Max-Heap behavior
end for
Pasul 3: Bucla Principală
for $i \leftarrow 0$ to K **do**
    if PQ este goală **then**
        break
    end if
    $s \leftarrow PQ.pop()$ ▷ Scoate starea cu cea mai mare eroare
    if s nu este terminală **then**
        $V[s] \leftarrow \max_a Q(s, a)$ ▷ Actualizează valoarea
    end if
    for fiecare predecesor p in $Predecessors[s]$ **do**
        if p nu este terminală **then**
            $diff \leftarrow |V[p] - \max_a Q(p, a)|$
            if $diff > \theta$ **then**
                $PQ.update(p, -diff)$
            end if
        end if
    end for
end for

Această abordare asigură că propagarea valorilor se face "organic", urmărind fluxul schimbărilor mari prin spațul stărilor, ceea ce duce la o convergență mult mai rapidă decât Value Iteration standard în medii vaste.

4 Învățare Online: Q-Learning

Următoarele cerințe fac trecerea la Reinforcement Learning propriu-zis, unde agentul nu cunoaște modelul tranzițiilor T sau al recompenselor R.

4.1 Conceptul Teoretic

Q-Learning este o metodă care învață direct valorile $Q(s, a)$ (utilitatea de a face acțiunea a în starea s). Regula de actualizare pe baza eșantioanelor (sample-uri) (s, a, r, s') este:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') \right) \tag{2}$$

Unde α este rata de învățare (learning rate).

4.2 Q5: Implementarea Q-Learning Agent

Spre deosebire de Value Iteration, agentul Q-Learning este *model-free*. Acesta nu cunoaște funcțiile de tranziție T sau de recompensă R inițial. Învață valorile $Q(s, a)$ exclusiv prin interacțiunea "încercare și eroare" cu mediul.

4.2.1 Vizualizare și Rezultate

Agentul explorează grid-ul, actualizând valorile stărilor pe măsură ce le vizitează.

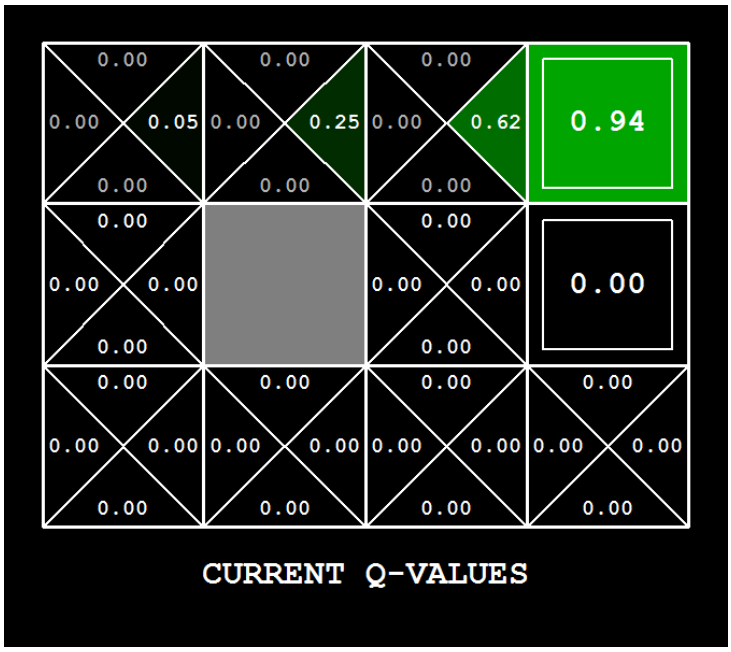

Figure 4: Valorile Q (în triunghiuri) convergând către valorile optime

4.2.2 Algoritmul de Actualizare

Esența implementării mele în `QLearningAgent` este metoda `update`. Aceasta aplică formula (2) de învățare temporală (Temporal Difference Learning) de fiecare dată când agentul face o mișcare și observă o recompensă.

4.2.3 Pseudocod Implementare

Am stocat valorile Q într-un dicționar (Counter). Pentru selecția acțiunii (`computeActionFromQValues`), am implementat o logică de spargere aleatorie a egalității (`random tie-breaking`) pentru a evita blocarea agentului în cicluri.

Algorithm 4 Q-Learning Update

function UPDATE(s, a, s', r)
  $Q_{old} \leftarrow$ getQValue(s, a)
  $V_{next} \leftarrow \max_{a'}$ getQValue(s', a') ▷ Valoarea stării viitoare
  $Sample \leftarrow r + \gamma \cdot V_{next}$
  $Q_{new} \leftarrow (1 - \alpha) \cdot Q_{old} + \alpha \cdot Sample$
  setQValue(s, a, Q_{new})
end function

4.3 Q6: Strategia Epsilon Greedy

Un agent Q-Learning pur greedy alege întotdeauna acțiunea cu valoarea Q maximă. Aceasta poate duce la o problemă majoră: agentul poate rămâne blocat într-un optim local, refuzând să încerce rute noi care ar putea fi mai bune pe termen lung.

4.3.1 Explorare vs. Exploatare

Pentru a rezolva această problemă, am implementat strategia ϵ-greedy în metoda `getAction`. Aceasta introduce un element de aleatoriu controlat:

- **Exploatare (Probabilitate $1 - \epsilon$):** Agentul alege cea mai bună acțiune cunoscută ($\max Q$) pentru a maximiza recompensa.
- **Explorare (Probabilitate ϵ):** Agentul alege o acțiune aleatoare din cele disponibile, pentru a descoperi noi stări și recompense potențiale.

4.3.2 Implementare (Pseudocod)

Algorithm 5 Selecția Acțiunii (ϵ-Greedy)

function GETACTION(s, ϵ)
  $A \leftarrow$ getLegalActions(s)
  if A este goală **then return** None
  end if
  $random\_val \leftarrow$ random$(0, 1)$
  if $random\_val < \epsilon$ **then**
    return RandomChoice(A) ▷ Explorare
  else
    return $\arg\max_a Q(s, a)$ ▷ Exploatare
  end if
end function

4.3.3 Impactul parametrului ϵ

Testând cu diferite valori în GridWorld și pe robotul Crawler:

- ϵ **mic (0.0 - 0.1):** Agentul este stabil, dar învață lent rute noi.
- ϵ **mare (0.5 - 0.9):** Agentul explorează rapid tot mediul, dar obține un scor mic deoarece face multe mișcări aleatoare inutile (se lovește de pereți etc).
- Valoarea optimă găsită a fost în jur de $0.05 - 0.1$ pentru faza de testare, după antrenament.

4.4 Q7: Bridge Crossing Revisited

Această cerință a testat limitele algoritmului Q-Learning într-un scenariu cu resurse limitate (doar 50 de episoade de antrenament) pe harta `BridgeGrid`.

Răspuns: `'NOT POSSIBLE'`

4.4.1 Justificare

Nu există o combinație de parametri (ϵ, α) care să garanteze găsirea politicii optime în peste 99% din rulări, din următoarele motive:

- **Dacă ϵ este mare (ex: 1.0):** Agentul se comportă haotic (random). Probabilitatea ca acesta să traverseze întregul pod secvențial, fără a cădea și fără a se întoarce, doar prin mișcări aleatoare, este extrem de mică.
- **Dacă ϵ este mic (ex: 0.0):** Agentul nu explorează suficient. Acesta va descoperi rapid recompensa mică de lângă start (+1) și, fiind "greedy", se va mulțumi cu ea, fără a descoperi vreodată recompensa mare de la capătul podului.
- **Numărul redus de episoade:** Chiar și cu un ϵ echilibrat, 50 de episoade sunt insuficiente pentru ca informația despre recompensa mare (+100) să se propage (prin update-uri succesive) de la capătul podului până la starea de start, având în vedere lungimea acestuia.

4.5 Q8: Q-Learning și Pacman

În această etapă, am aplicat agentul `PacmanQAgent` (bazat pe clasa `QLearningAgent` implementată anterior) în jocul Pacman pe harta `smallGrid`.

4.5.1 Faze de Execuție

Procesul este împărțit în două etape distincte:

- **Antrenament (Training):** Primele 2000 de jocuri. Agentul explorează mediul ($\epsilon > 0$) și își actualizează tabelul Q.
- **Testare (Testing):** Ultimele 10 jocuri. Parametrii de învățare și explorare sunt setați la 0 ($\epsilon = 0, \alpha = 0$). Agentul folosește strict politica învățată ("exploatare") și jocul este afișat în GUI.

4.5.2 Rezultate și Limitări

Pe harta mică (`smallGrid`), agentul obține o rată de câștig de peste 90%.
Totuși, pe hărți mai mari (`mediumGrid`), această abordare eșuează. Motivul este **explozia spațiului stărilor**: fiecare configurație a tablei (poziția lui Pacman + pozițiile fantomelor + poziția fiecărui punct de mâncare) este o stare unică în tabelul Q. Agentul nu poate vizita toate aceste stări într-un timp rezonabil pentru a învăța ce să facă, și nu are capacitatea de a *generaliza* (de exemplu, nu știe că a fi prins între două fantome este rău indiferent unde se află pe hartă).

4.6 Q9: Approximate Q-Learning

Aceasta este componenta finală a proiectului, care permite agentului Pacman să performeze pe hărți mari (ex: `mediumClassic`), unde numărul stărilor este prea mare pentru Q-Learningul anterior.

4.6.1 Conceptul de Aproximare

În loc să menținem un tabel imens cu valori $Q(s, a)$ pentru fiecare stare posibilă, aproximăm această valoare folosind o combinație liniară de *trăsături* (features) și *greutăți* (weights).

Formula funcției aproximative este:

$$Q(s, a) = \sum_{i=1}^{n} w_i \cdot f_i(s, a) \tag{3}$$

Unde:

- $f_i(s, a)$: O funcție care extrage o caracteristică a stării (ex: "distanța până la cea mai apropiată mâncare", "distanța până la fantome").
- w_i: Greutatea asociată acelei trăsături (cât de importantă este).

Această abordare permite **generalizarea**: dacă agentul învață că "a fi aproape de o fantomă este rău" într-un colț al hărții, va aplica această cunoștință peste tot, deoarece trăsătura "distanță la fantomă" este aceeași.

4.6.2 Implementare și Actualizare

În clasa `ApproximateQAgent`, am suprascris două metode esențiale:

1. **getQValue(state, action):** Calculează produsul scalar (dot product) dintre vectorul de greutăți și vectorul de trăsături returnat de `featExtractor`.

$$Q(s, a) = \text{weights} \cdot \text{features}$$

2. **update(state, action, nextState, reward):** În loc să actualizăm o valoare fixă într-un tabel, ajustăm greutățile w_i pentru a reduce eroarea de predicție. Logica implementată este:

Algorithm 6 Approximate Q-Learning Update

```
function UpdateWeights(s, a, s', r)
    Difference ← (r + γ max_a' Q(s', a')) − Q(s, a)
    Features ← extractFeatures(s, a)
    for fiecare trăsătură f_i și valoare val din Features do
        w_i ← w_i + α · Difference · val
    end for
end function
```

4.6.3 Rezultate

Folosind `SimpleExtractor`, agentul reușește să învețe o politică victorioasă pe `mediumGrid` în doar 50 de episoade, demonstrând eficiența net superioară față de Q-Learning-ul standard care nu a reușit să convergă nici după 2000 de episoade.

5 Concluzii

Principalele concluzii desprinse în urma implementării și testării celor trei categorii de algoritmi sunt:

- **Planificare vs. Învățare:** Algoritmul *Value Iteration* garantează găsirea politicii optime, dar este dependent de cunoașterea perfectă a modelului (MDP). În contrast, *Q-Learning* oferă flexibilitatea necesară aplicațiilor reale, permițând agentului să învețe "din mers" (model-free), deși necesită un echilibru atent între explorare și exploatare.
- **Limitările Metodelor Standard:** Pe măsură ce am trecut de la *Gridworld* la hărți complexe de *Pacman*, am observat că memorarea valorilor pentru fiecare stare individuală (Q-Learning standard) devine ineficientă computațional și imposibilă din punct de vedere al memoriei ("curse of dimensionality").
- **Puterea Generalizării:** Implementarea *Approximate Q-Learning* a demonstrat că soluția pentru medii complexe nu este memorarea brută, ci extragerea de trăsături relevante (features). Aceasta a permis agentului să generalizeze experiența, obținând performanțe superioare cu un timp de antrenament drastic redus.