**Preambul**

In acest proiect vom implementa algoritmi din aria reinforcement learning precum iterarea valorilor sau Q-learning. Vom dezvolta algoritmii pe problema Gridworld (similara cu cea prezentata la curs), iar apoi o vom aplica la Pacman.

Codul sursă poate fi descărcat de pe Moodle.

Execuția autograderului se face cu comanda

python autograder.py

Testele pentru una din întrebării se pot rula cu comanda (de exemplu testele pentru întrebarea q2):

python autograder.py -q q2

Un anume test particular poate fi invocat cu comanda:

python autograder.py -t test\_cases/q2/1-bridge-grid

Putem forța execuția autograder-ului fără grafică folosind flagul –no-graphics

Din arhiva pusa la dispoziție puteți edita următoarele fișiere, pe care trebuie să le transmiteți ca și răspuns la assignment:

|  |  |
| --- | --- |
| valueIterationAgents.py | Agent de tip iterarea valorilor pentru rezolvarea unui MDP |
| qlearningAgents.py | Agent de tip Q-learning pentru rezolvarea problemei Gridworld, Crawler si Pacman |
| analysis.py | Fișier in care se transmit răspunsurile la întrebările din proiect |

Putem consulta si celelalte fișiere esențiale din proiect precum:

|  |  |
| --- | --- |
| mdp.py | Definește problema mdp (metodele acestei probleme) |
| learningAgents.py | Definește clasele de baza si anume ValueEstimationAgent si QLearningAgent care vor fi extinse de agenții realizați in proiect |
| gridworld.py | Implementează problema Gridworld |
| featureExtractors.py | Clase pentru a extrage proprietățile perechilor (stare, acțiune). Este folosita in agentul Q-learning aproximativ – adica in fișierul qlearningAgents.py |

Se vor modifica doar fișierele indicate. Pentru buna funcționare a autograderului pe Gradescope, recomandarea este sa nu se modifice fișiere suport.

Puteți lucra in echipe de maxim 2 persoane. Doar unul din membri echipei va face transmiterea fișierelor pe Gradescope si in assignment își va indica (pe Gradescope) colegul ca si membru al echipei. De asemenea, in comentariile funcțiilor din cadrul soluției transmise se va indica nominal autorul funcției respective.

Codul sursa transmis pe Gradescope va fi evaluat automat dpdv al corectitudinii tehnice. Nu schimbați numele funcțiilor respectiv a claselor deja existente in fișierele sursa. Pentru buna funcționare a Gradescope este necesar sa păstrați structura proiectului intacta. Scorul final va fi dat de corectitudinea implementării furnizate.

Gradescope permite detectarea similarităților de cod sursa si logica a codului. Prin urmare, va rugam sa lucrați independent si sa nu dați propriile coduri sursa altor colegi. Proiectele transmise care vor fi identificate cu grad ridicat de similitudine vor fi notate cu 0 puncte.

**MDP**

Puteți executa jocul Gridworld cu comanda (se pot folosi săgețile):

python gridworld.py -m

Agentul este reprezentat cu un punct albastru. La apăsarea tastei înainte, de fapt agentul va muta înainte in 80% din cazuri (la fel ca si in exemplul de la curs). -m înseamnă ca jocul se execută manual (cu taste).

Putem testa un agent aleatoriu:

python gridworld.py -g MazeGrid

Remarcăm faptul că Gridworld este construit astfel încât sa trebuiască să introducem o stare pre-terminală, șl apoi să luăm acțiunea exit înainte de terminarea episodului. Dacă executăm un episod manual, câștigul obținut poate să fie mai mic decât cel așteptat, deoarece avem discount rate. Discount rate implicit este 0.9, si poate fi schimbat cu flag-ul -d

La fel ca si in proiectele anterioare, poziția din grid este reprezentata prin tuplul (x,y).

**Întrebări la care trebuie sa răspundeți (prin implementare de cod) in cadrul acestui proiect:**

**Întrebarea 1. Iterarea valorilor**

Cerința este să scriem un agent bazat pe iterarea valorilor (vedeți formulele Bellman din curs). Agentul va fi scris în ValueIterationAgent din fișierul valueIterationAgent.py. Agentul este dezvoltat parțial. Agentul este de tip offline, deci antrenarea lui se face înainte de joc. Parametrul care controlează antrenarea este numărul de iterații din algoritmul de iterare a valorilor. Acesta se transmite la intrare cu opțiunea -i. ValueIterationAgent preia MDP-ul in constructorul clasei si executa algoritmul de iterare a valorilor pentru numărul de iterații specificat.

Algoritmul calculează estimări de tipul *k-steps* pentru valoarea optima Vk. Deci va trebui sa implementați si următoarele metode din clasa specificata:

* computeActionFromValues(state) care calculează cea mai bună acțiune pornind de la funcția de valoare existenta in self.values
* computeQValueFromValues(state,action) care returnează Q-valorile aferente unei perechi (state, action).

Aceste cantități vor fi afișate in interfața grafică. Valorile sunt numerele din pătrate, iar Q-valorile sunt numerele din colturile pătratelor si acțiunile stabilite de politica sunt indicate de săgeți.

Se va utiliza versiunea batch a algoritmului de iterare a valorilor in care fiecare vector Vk este calculat pe baza unui vector fix Vk-1, similar ca și in curs. Aceasta înseamnă faptul că atunci când valoarea unei stări este actualizată in iterația k pe baza valorilor stărilor succesor, valorile stărilor succesor vor fi cele calculate in iterația anterioara (k-1).

O politică care rezulta din valorile de la adâncimea k (care reflectă următoarele k bonificații) in fapt va reflecta următoarea bonificație (cea de-a k+1), deci de fapt se returnează Pik+1. La fel in cazul Q-valorilor. Acestea vor reflecta o bonificație in plus față de valori (deci de fapt returnam Qk+1).

Deci trebuie sa returnați politia sintetizată din valori, adică Pik+1

Trebuie sa tratați si cazul in care in MDP într-o stare nu sunt acțiuni disponibile.

Testarea autograderului se face cu

python autograder.py -q q1

comanda de mai jos încarcă ValueIterationAgent care va calcula o politica si apoi o va executa de 10 ori. Trebuie să descoperiți faptul că valoarea stării de start (V(start)) care poate fi citita de pe GUI si câștigul empiric obținut sunt destul de apropiate.

python gridworld.py -a value -i 100 -k 10

daca se folosește layout-ul implicit (BookGrid) trebuie să obțineți valorile prezentate in curs (la fiecare iterație).

*Notare*: agentul va fi executat pe un grid nou. Se vor verifica valorile, Q-valorile si politicile obținute după un număr fixat de iterații, si de asemenea la convergență (de exemplu după 100 iterații).

**Întrebarea 2. Analiza problemei trecerii punții**

Lucrăm pe setupul BridgeGrid in care avem o stare terminala cu un câștig destul de mic (1), la capătul opus avem o stare terminala cu câștig mare (10) iar puntea este o cale îngustă separată de stări terminale cu un câștig negativ foarte mare (-100). Agentul pornește de lângă starea cu câștig mic. Se consideră discount-ul implicit 0.9 si zgomotul implicit 0.2 (adica in 80% din cazuri agentul va merge in direcția acțiunii alese, iar in celelalte 10% el merge fie in stânga sau in dreapta perpendicular pe direcția aleasă de politică). Schimbați unul dintre discount sau parametrul de zgomot pentru a vedea daca politica optima calculată permite agentului să încerce să treacă puntea.

Puneți răspunsul in question2() din fișierul analysis.py

Setupul implicit se executa cu

python gridworld.py -a value -i 100 -g BridgeGrid –discount=0.9 –noise=0.2

Notare: vom verifica dacă ați schimbat doar unul din parametri, si daca cu aceea schimbare un algoritm corect de iterare a valorilor permite agentului să treacă puntea.

Verificați răspunsul furnizat pe autograder cu întrebarea q2.

**Întrebarea 3. Politici**

Fie layout-ul DiscountGrid. Starea de start este in pătratul galben. Distingem intre doua tipuri de căi alese:

1. risk the cliff: agentul alege să meargă pe calea scurtă, pe lângă nodurile terminale cu valoare negativă
2. avoid the cliff: agentul alege calea mai lunga, dar sigură, la distanță de nodurile terminale cu valoare negativă.

In această problemă trebuie să alegeți setări ale discount-ului, zgomotului, si a bonificației de supraviețuire pentru a produce politici optime de tipuri diferite. Setarea aleasă pentru parametrii indicați trebuie să aibă proprietatea că daca agentul urmează politica optimă fără să fie afectat de zgomot, atunci va arăta comportamentul stabilit de politică. Daca un anume comportament particular nu este atins pentru nicio setare a parametrilor, atunci trageți concluzia ca politica este imposibilă si returnați stringul “NOT POSSIBLE”

Mai jos sunt tipurile de politici optime pe care trebuie să încercați sa le produceți:

1. preferam ieșirea cea mai apropiata, (+1) dar in condițiile risk the cliffs (-10)
2. preferam ieșirea cea mai apropiata (+1) dar in condițiile avoid the cliffs (-10)
3. preferam ieșirea cea mai depărtată (+10) in condițiile risk the cliffs (-10)
4. preferam ieșirea cea mai depărtată (+10) in condițiile avoid the cliffs (-10)
5. evitam atât cele 2 ieșiri cat si cliffs, deci episodul nu se termina niciodată

pentru a vedea aceste comportamente rulam comanda de mai jos:

python gridworld.py -g DiscountedGrid -a value –discount [YOUR\_DISCOUNT] –noise [YOUR\_NOISE] –livingReward [YOUR\_LIVING\_REWARD]

executați autograderul pe întrebarea q3 pentru a vedea rezultatul.

question2a() pana la question2e() trebuie sa returneze un tuplu (discount, noise, livingreward) in analysis.py

Daca in GUI nu vedeți săgeata, atunci apăsați un buton pe tastatura pentru a schimba GUI-ul in afișarea Q-valorilor si apoi puteți in minte sa calculați poziția săgeții.

*Notare*: vom verifica daca politica dorita este returnata in fiecare caz particular

**Întrebarea 5: Q-learning**

Observam ca prin iterarea valorilor nu se învață din experiența. In schimb, se face un algoritm de calcul prin care se decide asupra politicii de acțiune înainte ca agentul sa interacționeze cu mediul. La interacțiunea cu mediul, agentul doar executa politica prestabilita. Deci, de fapt agentul este unul reflex. Aceasta diferență poate sa nu conteze in cazul unui MDP simulat cum este Gridworld, însă in problemele reale, de fapt definirea MDP este aproximativa, si in multe cazuri nu este disponibila.

Cerința la aceasta întrebare este sa scriem un agent de tipul Q-learning care face foarte puține procesări înainte de joc (deci in constructorul agentului), dar învață din erori (trial-and-error). Interacțiunea cu mediul de întâmpla in funcția update(state, action, nextState, reward). In fișierul qlearningAgent.py este scris un agent QLearningAgent, care trebuie dezvoltat. Acest agent poate fi selectat la execuție folosind flag-ul -a q. trebuie sa implementați funcțiile update, computeValueFromQValues, getQValue si computeActionFromQvalues.

In computeActionFromQValues daca avem egalitate trebuie sa o rezolvam prin selecție aleatorie (pentru a obține un comportament mai bun). Pentru aceasta trebuie folosita funcția random.choice(). Intr-o stare noua, pe care agentul inca nu a experimentat-o, acțiunile pe care agentul inca nu le-a luat vor avea toate o Q-valoare de 0. Daca totuși avem acțiuni experimentate care au Q-value negativ, atunci o acțiune ne-experimentata pare mai buna (are q-value 0).

De asemenea, trebuie sa fiți atenți in funcțiile computeValueFromQValues si computeActionFromQValues trebuie sa accesați Q-valorile apelând funcția getQValue.

Daca aveți definita actualizarea Q-valorilor, puteți vedea agentul care învață daca este controlat manual prin tastatura, folosind

python gridworld.py -a q -k 5 -m

-k determina numărul de episoade pe care agentul le folosește pentru a învăța. Puteți observa modul in care agentul învață informație despre stări. Pentru debug, puteți inhiba zgomotul punând flag-ul –noise 0.0

Notare: autograderul va executa agentul Q-learning si va verifica daca acesta învață aceleași Q-valori si politica ca si in implementarea de referință. Puteți executa autograder-ul cu testele din q5.

**Întrebarea 6. Epsilon Greedy**

Trebuie sa finalizați implementarea agentului Q-learning implementând o strategie de selecția a acțiunii de tip epsilon-greedy in metoda getAction. Agentul va selecta o acțiune aleatorie pentru o fracțiune epsilon din timp, si in rest va urma sugestia data de cele mai bune Q-valori. Observam faptul ca alegând o acțiune aleatorie poate de asemenea sa rezulte acțiunea cea mai buna. Deci de fapt alegeți o acțiune aleatorie din toate acțiunile legal posibile.

Puteți alege un element aleatoriu dintr-o lista prin apelul random.choice. puteți simula o variabila binara cu probabilitatea p de succes folosind util.flipCoin(p) care returnează True cu probabilitatea p si False cu probabilitatea 1-p.

Dupa implementarea funcției getAction puteți observa comportamentul agentului in GridWorld (cu epsilon = 0.3)

python gridworld.py -a q -k 100

valorile finale calculate pentru Q-valori trebuie sa se asemene cu cele obținute prin algoritmul de iterare a valorilor. Totuși, câștigurile medii vor fi mai mici decât Q-valorile prezise, deoarece avem o alege a unei acțiuni aleatorie, care este sub-optimala. Cu comenzi similare cu cele de mai jos se poate observa simularea jocului pentru diferite valori ale lui epsilon:

python gridworld.py -a q -k 100 –noise 0.0 -e 0.1

python gridworld.py -a q -k 100 –noise 0.0 -e 0.9

Pentru a testa autograderul, îl puteți rula cu folderul q6

Fără niciun cod adițional, trebuie sa puteți executa un robot crawler de tip Q-learning

python crawler.py

Daca comanda de mai sus nu funcționează, înseamnă că ați scris codul pentru Q-learning foarte specific pentru gridworld si trebuie sa îl faceți mai general pentru a fi disponibil si pentru alte probleme de tip MDP.

Pentru crawler agent puteți sa va jucați cu parametri pentru a vedea modul in care acești parametri influențează politicile si acțiunile agentului. Observam faptul ca pasul este un parametru al simulării, rata de învățare si epsilon sunt parametri specifici algoritmului de învățare iar factorul de discount este un parametru al mediului.

**Întrebarea 7. Trecerea punții**

Pentru început trebuie sa învățați un agent de tip Q-learning cu rata de învățare implicita pentru problema BridgeGrid fără zgomot, pentru 50 de episoade si sa observam daca acesta găsește politica optima

python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1

Încercați sa experimentați cu epsilon 0. Găsiți un epsilon si o rata de învățare pentru care avem o șansă foarte buna (peste 99%) ca politica optima sa fie învățată dupa 50 de iterații? Funcția question8() din analysis.py trebuie sa returneze fie un tuplu cu 2 valori (epsilon, learning rate) sau stringul ‘NOT POSSIBLE’ daca nu se găsește nicio combinație (epsilon, learning rate) potrivita. Epsilon este controlat de parametrul -e iar rata de învățare de parametrul -l.

Răspunsul pe care îl dați in question8() nu trebuie sa depindă de mecanismul prin care rezolvați cazurile de egalitate la luarea deciziei de acțiune. Deci răspunsul trebuie sa fie corect chiar daca, de exemplu, rotim puntea cu 90 de grade.

Executia autograderului se face cu

python autograder.py -q q7

**Întrebarea 8. Pacman si Q-learning**

In acest setup, pentru a juca Pacman vom folosi 2 faze.

1. Faza de învățare. Pacman va începe sa învețe valorile pozițiilor si acțiunile. Durează destul de mult ca sa învățam aceste valori chiar si pentru layout-uri mici. Simulările de învățare a lui Pacman rulează in mod silențios (quiet), fără GUI.
2. Dupa ce faza de învățare s-a terminat, începe faza de testare, in care proprietățile self.epsilon si self.alpha a lui Pacman vor fi setate la 0, astfel se oprește învățarea Q-learning si se dezactivează explorarea, si Pacman începe sa își exploateze politica învățată. Jocurile de test sunt rulate implicit cu interfața grafica.

Fără sa faceți nicio schimbare de cod ar trebui sa puteți executa agentul Pacman Q-learning pentru layout-uri mici cu comanda:

python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid

PacmanQAgent este deja scris, datorita existentei agentului QLearningAgent scris anterior. Agentul QLearning Pacman diferă doar in ceea ce privește parametrii impliciți care sunt potriviți specific pentru problema Pacman: epsilon = 0.05, alpha = 0.2, gamma = 0.8.

Primiți punctul pentru aceasta problema daca comanda de mai sus funcționează fără excepții si agentul câștigă in cel puțin 80% din situații. Autograder va executa 100 de jocuri de test dupa ce face (simulează) 2000 jocuri de învățare.

Daca qLearningAgent funcționează cu gridworld.py si cu crawler.py dar nu pare sa învețe o politica bună pentru Pacman pe smallGrid, poate fi deoarece getAction si computeActionFromQValues nu considera in mod corespunzător acțiunile care nu au fost întâlnite anterior. Prin definiție o acțiuni care nu a fost întâlnită are Q-value = 0, daca toate acțiunile au valori Q negative atunci o acțiune care nu a fost întâlnite devine optimala.

Executia autograderului se face cu

python autograder.py -q q8

Puteți experimenta cu parametri de învățare diferiți. Aceștia se transmit cu opțiunea -a. de exemplu: -a epsilon=0.,1,alpha=0.3,gamma=0.7

In comanda cu -x 2000 -n 2010 se decide executia a 2010 jocuri, din care 2000 pentru învățare, care nu se executa cu GUI. Deci veți vedea ca Pacman executa doar 10 jocuri (cele de testare). Daca totuși doriți sa vizualizați cu GUI 10 jocuri de învățare, puteți folosi comanda:

python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10

ca si benchmark, sunt necesare intre 1000 si 1400 de jocuri înainte ca Pacman sa aibă un câștig pozitiv pentru un segment de 100 episoade, ceea ce înseamnă ca Pacman deja câștigă mai des decât pierde. La sfarsitul învățării, acest câștig trebuie sa rămână in mod cert pozitiv si ridicat (intre 100 si 350).

Daca Pacman a terminat faza de învățare, atunci acesta trebuie sa fie capabil sa câștige in mod consistent in cel puțin 90% din jocurile de test.

Totuși, pe un layout mai complicat, precum mediumGrid, Pacman nu va funcționa corespunzător.

**Întrebarea 9. Q-learning aproximativ**

Cerința este sa implementați un agent QLearning aproximativ, care învață ponderile pentru caracteristicile stărilor, in condițiile in care stările pot sa partajeze caracteristici comune. Scrieți implementarea voastră in ApproximateQAgent din qlearningAgents.py . Se va folosi partea teoretica de la QLearning aproximativ.

Implicit, ApproximateQAgent folosește IdentityExtractor care asignează o singura caracteristica (feature) pentru fiecare pereche (state, action). Folosind acest extractor de caracteristici (feature extractor), agentul QLearning aproximativ poate sa funcționeze identic cu agentul PacmanQAgent. Putem testa funcționarea cu

python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid

ApproximateQAgent este moștenită din clasa QLearningAgent, si prin urmare moștenește o serie de metode precum getAction. Prin urmare, trebuie sa va asigurați ca metodele din QLearningAgent apelează getQValue (in mod necesar), in locul accesării valorilor Q in mod direct. Deci atunci când suprascriem getQValue in agentul aproximativ, noile valori aproximative de tip q vor fi cele folosite pentru a calcula acțiunea.

Puteți executa acest agent cu SimpleExtractor, si trebuie sa vedeți ca agentul câștigă cu ușurință:

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid

Chiar si pentru layout-uri mai mari, agentul trebuie sa se descurce cu succes (de exemplu pe mediumClassic).

Daca nu aveți alte erori, agentul trebuie sa câștige aproape de fiecare data, chiar cu aceste caracteristici simple, chiar si cu doar 50 de jocuri de învățare.

Notare: vom executa agentul QLearning aproximativ si vom verifica daca acesta învață aceleași Q-valori si ponderi pentru setul de caracteristici ca si in implementarea de referință. Executia autograderului se face cu

python autograder.py -q q9