Universitat de les Illes Balears

Aprenentatge per Reforç

Intel·ligència Artificial

Marc Ferrer i Harpo Joan

Novembre 2024

Índex

1 Descripció del problema

El projecte es basa en un joc desenvolupat sobre un tauler de 8×8 caselles, on un agent ha de trobar el camí òptim des de la seva posició inicial fins a una casella objectiu. Aquest escenari planteja un entorn de laberint, en el qual les parets impedeixen el moviment a través d'algunes caselles. L'agent només pot desplaçar-se en quatre direccions possibles: nord, sud, est o oest, seguint una connectivitat de quatre veïns.

En el plantejament inicial, la configuració del laberint és estàtica: la posició de l'agent, la ubicació de les parets i la casella objectiu romanen inalterables al llarg del procés.

L'objectiu inicial és comprendre l'algorisme de *Q-learning* per resoldre el problema proposat. Això serveix com a base per a una segona fase, que consisteix en modificar-lo i obtenir l'algorisme SARSA.

2 Marc teòric en el context de l'aprenentatge per reforç

Es poden considerar dues estratègies distintes a la categoria d'algorismes d'aprenentatge per reforç sense model (degut a que no tenim accés a la distribució de probabilitat de transició). A continuació s'indiquen quines diferències hi ha entre aquestes dues estratègies i s'exemplifiquen amb els dos algorismes protagonistes de la pràctica.

2.1 Algorismes fora-política: *Q-Learning*

Els algorismes fora-política permeten a un agent aprendre de les observacions sobre la política òptima fins i tot quan no la segueix. Això serveix per aprendre a partir d'un conjunt de dades fixe o una política d'ensenyament. Particularment, l'algorisme *Q-learning* desenvolupat per Watkins el 1989 va suposar un dels primers avenços en el camp de l'aprenentatge per reforç en quant als algorismes fora-política es refereix. L'agent aprèn una funció acció-valor $Q(S, A)$ que retorna un *q-value*. Aquest valor indica quina és l'utilitat esperada d'estar a l'estat S i realitzar l'acció A. La forma d'actualitzar la taula Q prendrà especial rellevància a l'hora de comparar els dos algorismes que tractam en aquesta pràctica, i en el cas del *Q-learning* ve donada per la següent equació:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, A) - Q(S_t, A_t)] \tag{1}$$

on:

- α és la taxa d'aprenentatge
- γ és el factor de descompte
- R_{t+1} és la recompensa inmediata
- $\max_a Q(S_{t+1}, A)$ és el valor màxim possible en el següent estat

Tenint en compte l'equació 1, la traducció a pseudocodi quedaria:

Algorisme 1 ***Q-learning***

Paràmetres de l'algorisme: taxa d'aprenentatge $\alpha \in (0, 1]$, petita $\varepsilon > 0$

Inicialitzar $Q(s, a)$, per a tots $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitràriament excepte que $Q(\text{terminal}, \cdot) = 0$

Iterar per a cada episodi:

  Inicialitzar S

  Iterar per a cada passa de l'episodi:

    Escollir A de S usant la política derivada de Q (p.e., ε-greedy)

    Realitzar l'acció A, observar R, S'

    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

    $S \leftarrow S'$

  fins que S és terminal

2.2 Algorismes dins-política: SARSA (*State-Action-Reward-State-Action*)

L'algorisme SARSA (*State-Action-Reward-State-Action*) és un mètode d'aprenentatge per reforç que pertany a la categoria d'algorismes dins-política. Això vol dir que l'agent aprèn basant-se en la política que segueix en el moment d'explorar l'entorn. Ara l'agent considera les transicions de l'estat-acció a un altre parell i aprèn dels valors dels parells estat-acció. La forma d'actualitzar la funció acció-valor $Q(S, A)$, en el cas de SARSA, ve donada per la següent equació:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]. \tag{2}$$

on:

- α és la taxa d'aprenentatge
- γ és el factor de descompte
- R_{t+1} és la recompensa inmediata
- $Q(S_{t+1}, A_{t+1})$ és el valor en el següent estat i la següent acció

Anàlogament al que hem fet a l'apartat anterior, de l'equació 2 obtenim:

Algorisme 2 SARSA

Paràmetres de l'algorisme: taxa d'aprenentatge $\alpha \in (0, 1]$, petita $\varepsilon > 0$
Inicialitzar $Q(s, a)$, per a tots $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitràriament excepte que $Q(\text{terminal}, \cdot) = 0$
Iterar per a cada episodi:
    Inicialitzar S
    Escollir A de S usant la política derivada de Q (p.e., ε-greedy)
    Iterar per a cada passa de l'episodi:
        Realitzar l'acció A, observar R, S'
        Escollir A' de S usant la política derivada de Q (p.e., ε-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma\ Q(S', A') - Q(S, A)]$
        $S \leftarrow S'; A \leftarrow A';$
    fins que S és terminal

2.3 Comparativa dels dos algorismes

Un cop exposats els deos algorismes, podem distingir que la principal diferència és com s'actualitza el valor Q. SARSA actualitza el valor Q fent servir Q del següent estat i la següent acció que marca la política; en canvi, *Q-Learning* ho fa mitjançant el valor Q de l'estat següent i l'acció cobdiciosa posterior (en el cas d'usar la política ε-greedy. A més, si observam ambdós pseudocodis, trobam que SARSA realitza la primera elecció de A fora del bucle que recorr totes les passes del episodi. Al contrari que *Q-Learning*, ja que aquest mètode duu a terme una única elecció de A i es troba dins del bucle anteriorment mencionat.

Analitzant en perspectiva, es pot dir que SARSA aprèn una política avaluant i ajustant les accions que executa l'agent, mentre que *Q-Learning* aprèn una política òptima de manera independent. Per això es sol dir que ün agent SARSA s'arrisca més".

3 Implementació del codi

3.1 Punt de partida

En quant a la traducció dels algorismes anteriors a codi `Python` per al cas real de la pràctica on un agent ha de trobar la sortida d'un laberint, ens podem basar en el codi del repositori de l'assignatura. Concretament hem d'analitzar la classe `Agent`, on hi s'hi encapsula el mètode `train`[1]:

```
    def train(self, discount, exploration_rate, learning_rate, episodes,
    stop_at_convergence=False):

        cumulative_reward = 0
        cumulative_reward_history = []
        win_history = []

        for episode in range(1, episodes + 1):

            state = self.environment.reset()

            while True:
                if np.random.random() < exploration_rate:
                    action = random.choice(self.environment.actions)
                else:
                    action = self.predict(state)

                next_state, reward, status = self.environment._aplica(action)
                cumulative_reward += reward

                if (state, action) not in self.Q.keys():
                    self.Q[(state, action)] = 0.0

                max_next_Q = 0

                for a in self.environment.actions:
                    if (next_state, a) in self.Q and self.Q[(next_state, a)] > max_next_Q:
                        max_next_Q = self.Q[(next_state, a)]

                self.Q[(state, action)] = self.Q[(state, action)] + learning_rate *
                (reward + discount * max_next_Q - self.Q[(state, action)])

                if status in (Status.WIN, Status.LOSE):
                    break

                state = next_state

            cumulative_reward_history.append(cumulative_reward)

        return cumulative_reward_history, win_history, episode
```

[1] En el codi mostrat s'han omés alguns comentaris i instruccions per a major claretat.

3.2 Algorisme resultant

Partint del codi base anterior que se'ns ha proporcionat i sense perdre de vista els pseudocodis explicats anteriorment, podem adaptar aquest mètode `train` de tal manera que ens serveixi per a la implementació de l'algorisme SARSA. Per això, podem distingir cinc canvis principals a l'hora d'escriure la nova funció `train`[2]:

```
def train(self, discount, exploration_rate, learning_rate, episodes,
stop_at_convergence=False):

    cumulative_reward = 0
    cumulative_reward_history = []
    win_history = []

    for episode in range(1, episodes + 1):

        state = self.environment.reset()
        # 1r canvi: seleccionar l'acció derivada de la política epsilon-greedy
        if np.random.random() < exploration_rate:
            action = random.choice(self.environment.actions)
        else:
            action = self.predict(state)

        while True:
            # 2n canvi: executar l'acció A i observar la recompensa R i el nou estat S'
            next_state, reward, status = self.environment._aplica(action)
            cumulative_reward += reward

            # 3r canvi: seleccionar la següent acció derivada de la política epsilon-greedy
            if np.random.random() < exploration_rate:
                next_action = random.choice(self.environment.actions)
            else:
                next_action = self.predict(next_state)

            if (state, action) not in self.Q.keys():
                self.Q[(state, action)] = 0.0

            if (next_state, next_action) not in self.Q.keys():
                self.Q[(next_state, next_action)] = 0.0

            # 4t canvi: actualització segons SARSA
            self.Q[(state, action)] = self.Q[(state, action)] + learning_rate *
            (reward + discount * self.Q[(next_state, next_action)] - self.Q[(state, action)])

            if status in (Status.WIN, Status.LOSE):
                break

            # 5è canvi: actualitzar l'estat i l'acció
            state = next_state
            action = next_action

        cumulative_reward_history.append(cumulative_reward)

    return cumulative_reward_history, win_history, episode
```

[2]En el codi mostrat s'hi tornen a ometre alguns comentaris i instruccions per a major claretat.

3.3 Justificació dels canvis realitzats

Els canvis mencionats s'han realitzat de tal forma que es compleixi el panorama teòric que hem explicat a l'apartat anterior, doncs les justificacions de cada canvi són les següents:

- **1r canvi: seleccionar l'acció derivada de la política ε-greedy**
  El primer canvi realitzat per a adaptar la funció és seleccionar una acció A seguint la política establerta. A diferència de *Q-learning*, SARSA selecciona aquesta acció amb anterioritat a iterar cada passa de l'episodi. Segons el pseudocodi 2 l'elecció de l'acció A és anterior a l'iteració per a cada passa de l'episodi, al contrari que a algorisme 1

Q-Learning	**SARSA**

Inicialitzar S
Iterar per a cada passa de l'episodi:
--- 	
`state = self.environment.reset()`
`while True:`
	Inicialitzar S
	Escollir A de S usant la política derivada de Q (p.e., ε-greedy)
	Iterar per a cada passa de l'episodi:
	--- 	
	`state = self.environment.reset()`
	`if np.random.random() < exploration_rate:`
	`  action = random.choice (self.environment.actions)`
	`else:`
	`  action = self.predict(state)`
	`while True:`

- **2n canvi: executar l'acció A i observar la recompensa R i el nou estat S'**
  El segon canvi implementat és executar l'acció A que hem escollit anteriorment amb el mètode `aplica`. A continuació actualitzam la recompensa general amb el *reward* generat a l'hora d'escollir l'acció.

Q-Learning	**SARSA**

Iterar per a cada passa de l'episodi:
Escollir A de S usant la política derivada de Q (p.e., ε-greedy)
--- 	
`while True:`
`  if np.random.random() < exploration_rate:`
`  action = random.choice (self.environment.actions)`
`  else:`
`  action = self.predict(state)`
	Iterar per a cada passa de l'episodi:
	Realitzar l'acció A, observar R, S'
	--- 	
	`next_state, reward, status = self.environment._aplica(action)`
	`cumulative_reward += reward`

- **3r canvi: seleccionar la següent acció derivada de la política ε-greedy**
  La tercera modificació és clau en quant al desenvolupament de l'algorisme SARSA. Similarment al primer canvi, ara hem de triar la pròxima acció. Aquesta seria una de les principals diferències entre els dos algorismes, ja que es té en compte el següent moviment a realitzar, contràriament que a l'algorisme de partida.

Q-Learning	**SARSA**
Iterar per a cada passa de l'episodi: Escollir A de S usant la política derivada de Q (p.e., ε-greedy)	Realitzar l'acció A, observar R, S' Escollir A' de S usant la política derivada de Q (p.e., ε-greedy)

`while True:`
`  if np.random.random() <`
`  exploration_rate:`
`  action =`
`  random.choice`
`  (self.environment.actions)`
`  else:`
`  action = self.predict(state)`
	`if np.random.random() <`
	`exploration_rate:`
	`  next_action =`
	`  random.choice`
	`  (self.environment.actions)`
	`else:`
	`  next_action =`
	`  self.predict(next_state)`

- **4t canvi: actualització segons SARSA**
  Aquest canvi tambe és considerablament important per què s'actualitza la taula Q en funció de les equacions 1 i 2 respectivament. A la primera, s'inicialitza la variable `max_next_Q` i es cerca quin serà el màxim valor Q. Contràriament, SARSA no fa aquest procediment i aplica la fórmula tenint en compte el parell format pel següent estat i la següent acció.

Q-Learning	**SARSA**
$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$	$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma\, Q(S',A') - Q(S,A)]$

`max_next_Q = 0`
 	
`for a in self.environment.actions:`
`  if (next_state, a) in self.Q and`
`  self.Q[(next_state,a)] > max_next_Q:`
`  max_next_Q =`
`  self.Q[(next_state, a)]`
 	
`self.Q[(state, action)] =`
`self.Q[(state, action)] + learning_rate`
`*(reward + discount * max_next_Q -`
`self.Q[(state, action)])`
	`self.Q[(state, action)] =`
	`self.Q[(state, action)] +`
	`learning_rate *`
	`(reward + discount *`
	`self.Q[(next_state, next_action)] -`
	`self.Q[(state, action)])`

- **5è canvi: actualitzar l'estat i l'acció**
  Finalment, a aquesta darrera modificació, SARSA guarda l'estat S' i l'acció A', a diferència de l'algorisme *Q-learning*, que només té en compte el pròxim estat.

Q-Learning	**SARSA**
$S \leftarrow S'$	$S \leftarrow S'; A \leftarrow A'$
`state = next_state`	`state = next_state` `action = next_action`

4 Proves i resultats

4.1 Configuració experimental

El programa realitza primer un entrenament de l'agent per, després validar l'aprenentatge. Aquest entrenament ve definit principalment per uns hiperparàmetres que influeixen en el resultat final.

- `discount`: representa el valor que l'agent dona a les recompenses futures per sobre de les recompenses immediates. Un valor proper a 0 enfoca a l'agent en les recompenses immediates, ignorant el llarg termini, mentre que un valor proper a 1, centra l'agent a trobar una estratègia més sostenible pel futur. El valor definit per aquest hiperparàmetre ha estat de 0.90.
- `exploration_rate`: controla la possibilitat que l'agent elegeixi una acció aleatòria en lloc d'una acció basada en la seva política. Un valor proper a 1 promou l'exploració, mentre que un valor proper a 0, fomenta l'explotació del que l'agent coneix. El valor predeterminat és 0.10, xifra coherent amb el paradigma de les estrategies dins-política.
- `learning_rate`: controla quant canvia el que ja sap l'agent en funció de la informació que acaba d'aprendre. Un valor proper a 1, canvia completament l'estratègia fetn cas a la informació que acabes de obtenir; en canvi, un valor proper a 0 quasi no ajusta l'estratègia. El valor usat és 0.6.
- `episodes`: determina el nombre d'iteracions que realitza l'agent durant l'entrenament. Si el nombre d'iteracions és major, més oportunitats hi ha que l'agent trobi una política correcta i òptima. El valor inicial és de 1000 episodis, encara que hem fet proves amb 2000 i 4000 episodis.
- `stop_at_convergence`: és un boolea que atura l'entrenament quan l'agent ha aconseguit un rendiment òptim en el cas de que l'assoleixi abans d'acabar els episodis.

4.2 Anàlisi dels resultats

Les successives execucions del codi resultant ens permeten concloure certs aspectes relatius al funcionament del programa desenvolupat. En primer lloc cal dir que el funcionament és exitós i l'agent en la majoria dels casos assoleix la `casella_desti` com es mostra a l'imatge 1.

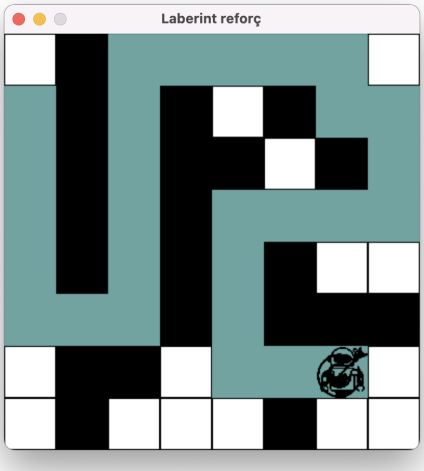

Figura 1: L'agent assoleix la meta.

D'altra banda, cal dir que en determinades execucions l'agent no troba la meta i pareix que es queda dins un bucle del qual no pot sortir. Això es pot explicar com a conseqüencia de la naturalesa del problema i el paradigma del mètode que ho resol. Com es tracta d'un mètode predictiu, no es pot assolir un 100% d'èxit. Tot i que modificar

paràmetres clau com hem explicat a l'apartat anterior és una estrategia útil per a obtenir uns millors resultats de cara a trobar la solució, es pot considerar un comportament *acceptable* el fet que a vegades l'agent no trobi solució. Concretament, les imatges 2 i 3 es corresponen amb una execució en la que l'agent es va quedar enrocat entre les cel·les $(0, 6)$ i $(1, 6)$. Aquest exemple només demostra que tot i que es configurin paràmetres com el nombre d'episodis, sempre hi existirà una probabilitat $p > 0$ que ens farà oscil·lar entre un resultat favorable i un de negatiu (tot i que aquesta probabilitat en la nostra solució és molt petita).

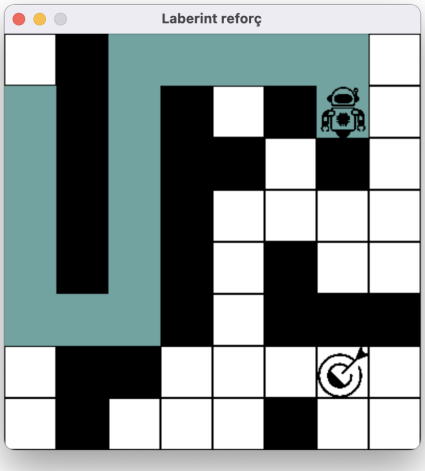

Figura 2: Agent a la cel·la $(1, 6)$.

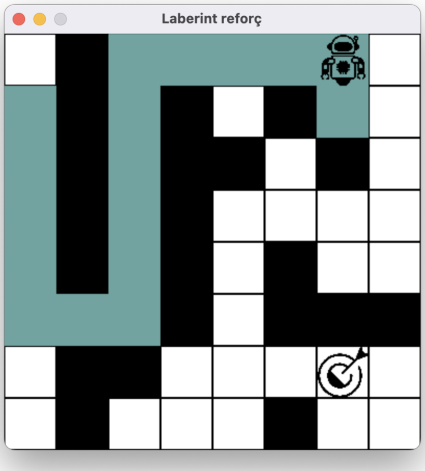

Figura 3: Agent a la cel·la $(0, 6)$.

5 Conclusió

El procés de desenvolupament i la confecció de la memòria d'aquesta pràctica ens ha permès treure varies conclusions de distint caràcter.

En primer lloc, hem comprès la distinció entre els algorismes fora-política i dins-política, i hem establert un marc teòric rigurós per a exemplificar dos algorismes suficientment representatius de cada categoria. A més, després de diverses proves, hem vist que l'algorisme que hem implementat funciona correctament, encara que hi ha algunes poc probables excepcions sorgides de l'ús de tècniques predictives que no garenteixen l'èxit complet en tots els escenaris. Hem visit què signifiquen els hiperparàmetres de les expressions matemàtiques i com afecten els valors de cada un d'ells al rendiment de l'algorisme.

L'ús de LaTeXper a redactar aquest informe ha estat molt satisfactori, ajudant-nos a presentar i estructurar les nostres idees de la manera més clara i professional que se'ns ha ocorregut.

En resum, consideram que hem resolt el problema plantejat de manera exitosa, a més d'haver entès més profundament els conceptes teòrics tractats aquí i hem conegut una eina útil i professional amb la que segur tornarem a fer feina pròximament.