Machine Learning

Università Roma Tre
Dipartimento di Ingegneria

Anno Accademico 2021 - 2022

Esercitazione: Reinforcement Learning (Ex 16)

Sommario

- Richiami RL e Q-learning
- Esempio taxi
- Ambiente OpenAI Gym
- Approccio non RL
- Approccio Q-Learning
- Approccio Epsilon-Greedy Q-Learning
- Valutazione e iperparametri

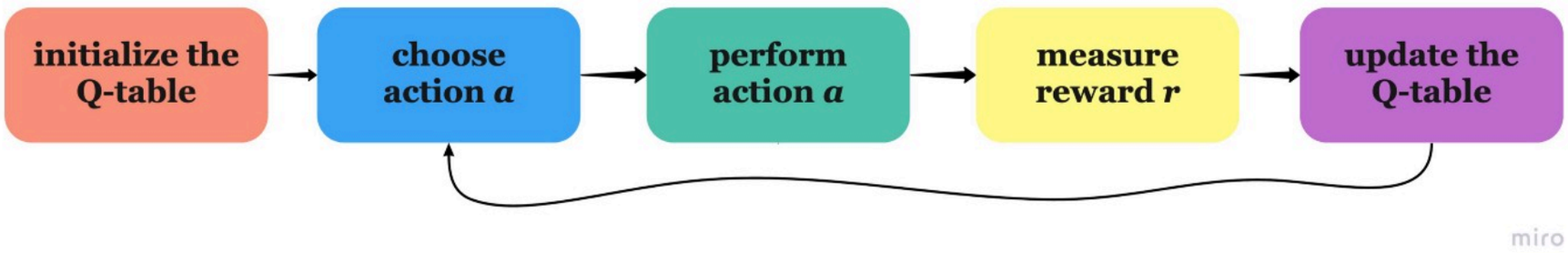
Richiami: Reinforcement Learning

- Nell'ambito del Reinforcement Learning (RL), la *policy* è la strategia per scegliere una azione nello stato corrente che determini la massima ricompensa (*reward)*.
- Il *Q-learning* è un algoritmo che mira a determinare col tempo la migliore azione (*best action*), dato lo stato corrente (*current state),* in base alla stima di *reward* attesa.
- Misura la bontà di una combinazione stato-azione in termini di *reward*.

  Impiega una *Q-table* aggiornata dopo ogni episodio, dove la riga corrisponde allo stato e la colonna all'azione. Il Q-value dentro la tabella indicano quanto una azione è stata buona (alto reward) in passato.
- È un algoritmo model-free, poiché l'agente non conosce il valore di una azione prima di effettuarla.
- Non segue un approccio greedy poiché scegliere sempre l'azione con reward immediato massimo potrebbe determinare sequenze di azioni non ottime.

Richiami: Reinforcement Learning

- Step #1: inizializzo la Q-table con valori pari a 0, ogni azioni e equiprobabile.
- Step #2: scegli l'azione in modo random, o sfrutta l'eventuale informazione che hai al principio
- Step #3: esegui l'azioni e colleziona il reward
- Step #4: aggiorna la Q-table di conseguenza



Richiami: Reinforcement Learning

- L'equazione di **Bellman** aggiorna i Q-values determinando il valore massimo di *reward* atteso per ogni stato nella Q-table.

$$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma max_a Q(s_{t+1}, a_t))$$

- Il primo termine *Q()* indica il valore dell'azione corrente nello stato corrente. Il secondo combina il reward corrente e il valore discount dello stato futuro caratterizzato da reward massima.

- Il *discount factor lambda* [0,1] permette di ridurre il reward col tempo e indica quanta importanza assegnamo ai futuri reward: valori vicini allo 0 indicano che l'agente si limita a valutare i reward immediati, vicini al 1 permettono di valutare l'effetto a lungo termine dei reward.

- Il valore *alpha* (learning rate (0,1]) determina l'importanza che assegniamo ai valori futuri rispetto a quelli attuali.

Esempio: taxi che guida da solo

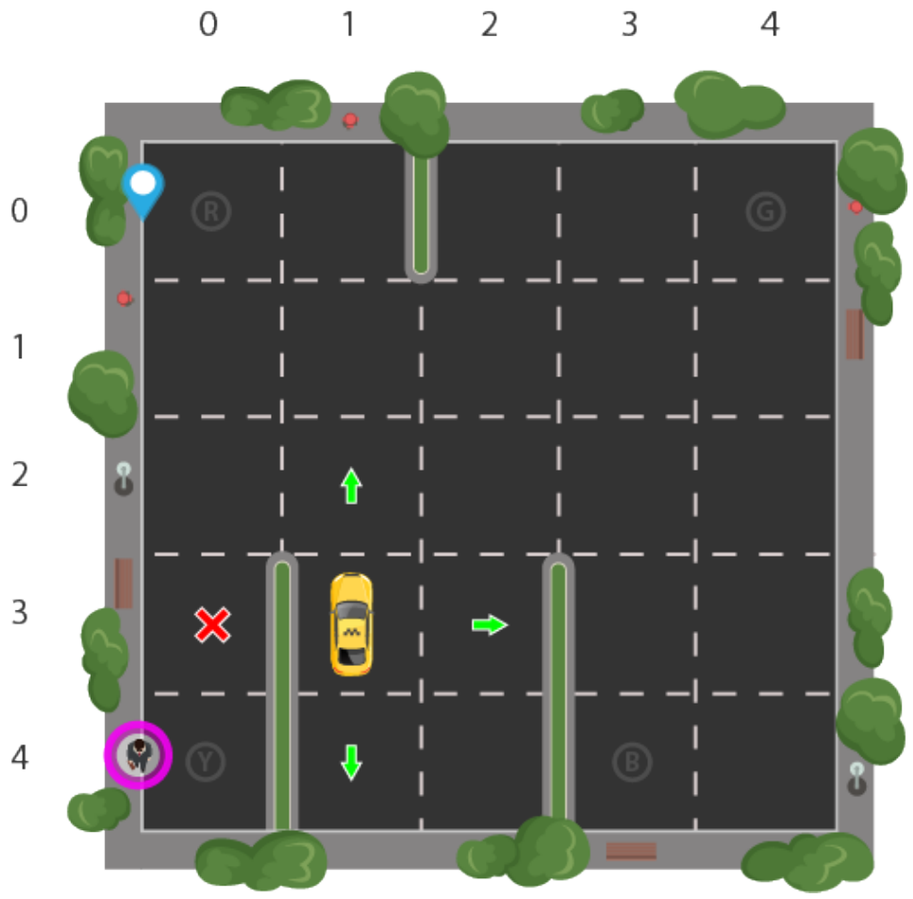
- Definiamo un ambiente (environment) semplificato dove un taxi deve prendere un cliente in una certa locazione e lasciarlo in un'altra.
- Vogliamo altresì:
  - Lasciare il cliente nel luogo giusto
  - Minimizzare il tempo per il trasporto
  - Seguire le regole della strada
- Dobbiamo definire: rewards, states, actions.
- Quali puoi ipotizzare?

Taxi che guida da solo: reward

- Per i reward possiamo ipotizzare:
  - Alto reward se il cliente viene lasciato correttamente.
  - Penalizzazione se il cliente viene lasciato nella location sbagliata.
  - Per ogni istante di tempo trascorso, una piccola penalità.

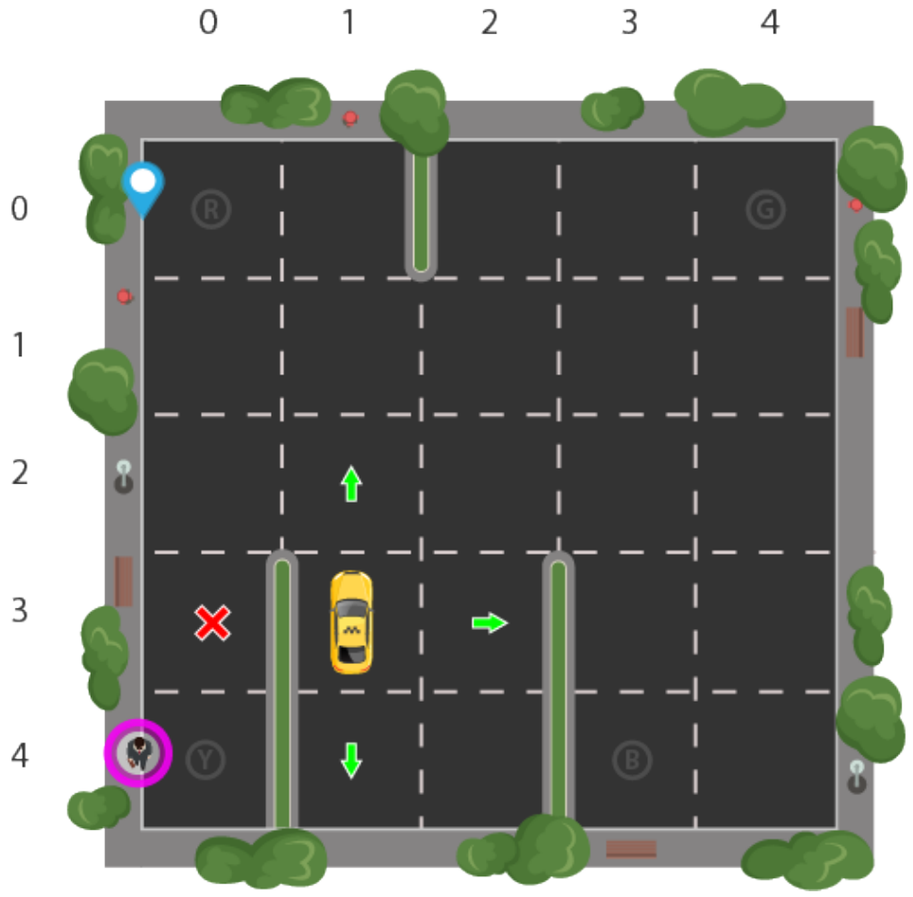
Taxi che guida da solo: state space

- Lo state space corrisponde a tutte le possibili situazioni in cui un taxi si può trovare. Ogni stato deve contenere abbastanza informazioni per permettere all'agente di decidere una azione.
- Supponiamo il taxi sia l'unico veicolo.
- Suddividiamo l'ambiente in una griglia 5x5
- Posizione corrente (3,1)
- 4 location per il pick up e drop off: R,G,Y,B; cioè [(0,0), (0,4), (4,0), (4,3)]
- Il cliente è in Y e vuole andare in R.
- Uno stato aggiuntivo che rappresenta il cliente all'interno del taxi.
- Quanti sono il numero dei possibili stati?



Taxi che guida da solo: state space

- Lo state space corrisponde a tutte le possibili situazioni in cui un taxi si può trovare. Ogni stato deve contenere abbastanza informazioni per permettere all'agente di decidere una azione.
- Supponiamo il taxi sia l'unico veicolo.
- Suddividiamo l'ambiente in una griglia 5x5
- Posizione corrente (3,1)
- 4 location per il pick up e drop off: R,G,Y,B; cioè [(0,0), (0,4), (4,0), (4,3)]
- Il cliente è in Y e vuole andare in R.
- Uno stato aggiuntivo che rappresenta il cliente all'interno del taxi.
- Possibili stati: 5 x 5 x 5 x 4 = 500



Taxi che guida da solo: action space

- L'agente può in ogni stato fare una delle seguenti azioni:
  - muoversi a nord
  - muoversi a su
  - muoversi a est
  - muoversi a ovest
  - prendere il cliente
  - lasciare il cliente
- Se l'agente non può fare una certa azione in uno stato (es. presenza di un muro) possiamo assegnare una penalità di -1.

OpenAI Gym

- Sono delle API in Python che permettono di sperimentare approcci RL.
  - https://www.gymlibrary.ml
- La libreria include già l'ambiente Taxi già costruito.

```
!pip install cmake 'gym[atari]' scipy

import gym

# carichiamo l'environment taxi
env = gym.make("Taxi-v3").env

env.render()

>>
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : |▒: |
|Y| : |B: |
+---------+
```

OpenAI Gym

```
...

env.reset() # reset environment to a new, random state
env.render()

print("Action Space {}".format(env.action_space))
print("State Space {}".format(env.observation_space))

>>
+---------+
|R: | : :G|
| : | : : |
| : :▒: : |
| | : | : |
|Y| : |B: |
+---------+

>> Action Space Discrete(6)
>> State Space Discrete(500)

...
```

OpenAI Gym

- Le azioni sono codificate con interi:
  - 0 = south, 1 = north, 2 = east, 3 = west, 4 = pickup, 5 = dropoff

```
state, reward, done, info = env.step(0) # azione: verso south
env.render()
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | :▒| : |
|Y| : |B: |
+---------+
```

```
state, reward, done, info = env.step(0) # azione: verso south
env.render()
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | :▒| : |
|Y| :▒|B: |
+---------+
```

OpenAI Gym

- Sono delle API in Python che permettono di sperimentare approcci RL.

```
# parametri (taxi row, taxi column, passenger index, destination index)
state = env.encode(3, 1, 2, 0)
print("State:", state)

env.s = state
env.render()

>> State: 328
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| |▒: | : |
|Y| : |B: |
+---------+
```

OpenAI Gym

- Possiamo rappresentare un certo stato dell'environment esplicitamente. Allo stato sarà associato un id numerico (328).

```
state = env.encode(3, 1, 2, 0)
# (taxi row, taxi column, passenger index, destination index)

print("State:", state)

env.s = state
env.render()
```

```
State: 328
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| |▓: | : |
|Y| : |B: |
+---------+
```

OpenAI Gym

- La reward table rappresenta coppie stati x azioni.
- Ad esempio, per lo stato 328 otteniamo il seguente dizionario:

```
env.P[328]

>>
{0: [(1.0, 428, -1, False)],
 1: [(1.0, 228, -1, False)],
 2: [(1.0, 348, -1, False)],
 3: [(1.0, 328, -1, False)],
 4: [(1.0, 328, -10, False)],
 5: [(1.0, 328, -10, False)]}
```

- Dove il dizionario ha la struttura:
  - {action: [(probability *sempre_1*, next-state, reward, done)]}.

Esercizio Taxi: senza RL

- Supponi che l'agente abbia accesso unicamente la *reward table* P per decidere quale azioni compiere. Perciò non apprende dall'esperienza acquista nel passato.
- Crea un loop che prosegua finché il cliente non sia arrivato a destinazione.
- Suggerimento: la funzione *env.action_space.sample()* restituisce una azione in modo casuale.

Esercizio Taxi: senza RL

```
env.s = 328  # stato iniziale

epochs = 0
penalties, reward = 0, 0
frames = [] # per animazione

done = False
while not done:
    action = env.action_space.sample()
    state, reward, done, info = env.step(action)
    if reward == -10:
        penalties += 1

    frames.append({
        'frame': env.render(mode='ansi'),
        'state': state,
        'action': action,
        'reward': reward
        }
    )
    epochs += 1

print("Timesteps taken: {}".format(epochs))
print("Penalties incurred: {}".format(penalties))
```

Esercizio Taxi: senza RL

- Per l'animazione:

```
from IPython.display import clear_output
from time import sleep

def print_frames(frames):
    for i, frame in enumerate(frames):
        clear_output(wait=True)
        print(frame['frame'].getvalue())
        print(f"Timestep: {i + 1}")
        print(f"State: {frame['state']}")
        print(f"Action: {frame['action']}")
        print(f"Reward: {frame['reward']}")
        sleep(.1)

print_frames(frames)
```

- L'algoritmo può impiegare molti step (oltre 1000) incorrendo in molte penalty (oltre 300).

Esercizio: Q-learning in Python

- *Esercizio*: modifica il codice precedente implementando l'algoritmo Q-learning.

Esercizio: Q-learning in Python

- *Esercizio*: modifica il codice precedente implementando l'algoritmo Q-learning.

```
%%time   # stampa il tempo trascorso al termine dell'esecuzione

import numpy as np
import random
from IPython.display import clear_output

# iperparametri
alpha = 0.1
gamma = 0.6

# per il report
all_epochs = []
all_penalties = []

q_table = np.zeros([env.observation_space.n, env.action_space.n])

...
```

Esercizio: Q-learning in Python

```
...

for i in range(1, 100001):
    state = env.reset()

    epochs, penalties, reward, = 0, 0, 0
    done = False

    while not done:
        action = np.argmax(q_table[state])
        next_state, reward, done, info = env.step(action)

        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state])

        # eq Bellman
        new_value = (1 - alpha) * old_value + alpha *
                    (reward + gamma * next_max)
        q_table[state, action] = new_value

...
```

Esercizio: Q-learning in Python

```
...

        if reward == -10:
            penalties += 1

        state = next_state
        epochs += 1

    if i % 100 == 0:
        clear_output(wait=True)
        print(f"Episode: {i}")

print("Training finished.\n")
```

```
>> Episode: 100000
>> Training finished.

>> CPU times: user 1min 25s, sys: 15 s, total: 1min 40s
>> Wall time: 1min 29s
```

```
q_table[328] # l'azione migliore è north -2.27
```

```
>> array([-2.31436727, -2.27325184, -2.31164458, -2.3090025 ,
          -2.8816    , -2.8816    ])
```

Esercizio: Q-learning in Python

- *Esercizio*: valuta nuovamente l'algoritmo con le best action ricavate dalla Q-table.

Esercizio: Q-learning in Python

- Valutazione dell'algoritmo con le best action ricavate dalla Q-table:

```
total_epochs, total_penalties = 0, 0
episodes = 100

for _ in range(episodes):
    state = env.reset()
    epochs, penalties, reward = 0, 0, 0

    done = False
    while not done:
        action = np.argmax(q_table[state])
        state, reward, done, info = env.step(action)
        if reward == -10:
            penalties += 1
        epochs += 1
    total_penalties += penalties
    total_epochs += epochs

print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs / episodes}")
print(f"Average penalties per episode: {total_penalties / episodes}")
```

```
>> Results after 100 episodes:
>> Average timesteps per episode: 13.01
>> Average penalties per episode: 0.0       <-- nessuna penalty con 100 clienti
```

Epsilon-Greedy Q-learning

- Con l'approccio Epsilon-greedy Q-learning introduciamo il bilanciamento tra *exploration* e *exploitation*.
- Nei modelli model-free è fondamentale esplorare l'ambiente per ottenere informazioni su cui basare le successive decisioni informate.
- Nella versione Espilon-greedy, con probabilità epsilon l'agente sceglie una azione in modo casuale (esplorazione) e segue l'azioni valutata migliore nell'altro caso (1-epsilon).

Algorithm 2: Epsilon-Greedy Action Selection

Data: Q: Q-table generated so far, : a small number, S: current state
Result: Selected action
Function *SELECT-ACTION(Q, S, ϵ)* **is**
  n $\leftarrow$ uniform random number between 0 and 1;
  if $n < \epsilon$ **then**
    A $\leftarrow$ random action from the action space;
  else
    A $\leftarrow$ maxQ(S,.);
  end
  return selected action A;
end

- *Esercizio*: modifica il codice introducendo questa versione.

Epsilon-Greedy Q-learning

- *Esercizio*: modifica il codice introducendo questa versione.

- ...

```
# iperparametri
alpha = 0.1
gamma = 0.6
epsilon = 0.1

...
```

```
while not done:
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample() # Explore action space
        else:
            action = np.argmax(q_table[state]) # Exploit learned values

...
```

```
>> Results after 100 episodes:
>> Average timesteps per episode: 12.81      <-- invece di 13.01
>> Average penalties per episode: 0.0        <-- nessuna penalty con 100 clienti
```

Valutazione approccio RL

- Alcune metriche da considerare nella valutazione sono:
  - Numero medio di *penalità* per episodio (ideale --> 0)
  - Numero medio di *timesteps* per percorso
  - Valore medio di *reward* per mossa

Measure	Random agent's performance	Q-learning agent's performance
Average rewards per move	-3.9012092102214075	0.6962843295638126
Average number of penalties per episode	920.45	0.0
Average number of timesteps per trip	2848.14	12.38

Iperparametri

- *Alpha*: da decrementare con l'incremento dell'esperienza acquisita
- *Gamma*: se ci avviciniamo all'obiettivo dobbiamo ridurre l'importanza della reward a lungo termine
- *Epsilon*: con l'accumularsi dei tentativi, epsilon deve ridursi.

- Esercizio: applica un approccio *grid search* per ricavare una approssimazione degli iperparametri nello scenario del taxi che guida da solo.

Testi di Riferimento

- OpenAI Gym https://www.gymlibrary.ml