Q-Learning

Stefano Zanatta[1,*]

[1]corso Intelligenza Artificiale, Università di Padova, Dipartimento di Matematica, Padova, Italia
[*]stefano.zanatta@studenti.unipd.it

ABSTRACT

Implementazione dell'algoritmo Q-Learning descritto a lezione, con valutazione del risultato tramite grafo e rappresentazione grafica.

Introduzione

Ho implementato l'algoritmo Q-learning visto a lezione, in Python. Un agente esplora l'environment, mentre impara la funzione action-utility ed infine estrae la policy migliore.
Per rappresentare visivamente la policy ho utilizzato il framework Turtle[1].
Per valutare la policy, la rappresento come un grafo e conto le sue componenti connesse.

Algoritmo Q-Learning

L'algoritmo Q-Learning fa parte degli algoritmi di apprendimento con rinforzo attivi, cioè quelli che non conoscono la policy e devono impararla. In particolare, Q-Learning impara una funzione che mappa le azioni eseguibili in ogni stato alla loro utilità, rapprentata dalla matrice Q. L'obiettivo di Q-Learning è imparare la matrice Q, dalla quale è possibile ottenere la policy migliore, scegliendo l'azione con reward massimo in ogni stato. Per imparare la matrice Q ottima servirebbero infinite iterazioni, per la convergenza di Q-learning vista a lezione. É quindi necessario approssimare il risultato, limitando il numero epochs.

Pseudo-codice

Pseudo-codice di Q-Learning (visto a lezione).

```
function Q–Learning
    per ogni s,a) inizializza la entry della tabella Q̂(s,a) ← 0
    osserva lo stato corrente s
    esegui per sempre
        seleziona una azione a ed eseguila
        ricevi la ricompensa immediata r
        osserva il nuovo stato s'
        aggiorna la entry Q̂(s,a) come segue:
        Q̂(s,a) ← r + max_a Q̂(s',a')
        s ← s'
```

La selezione dell'azione può avvenire in due modi:

- **esplorazione:** le utility vengono ignorate e si sceglie una azione random;
- **sfruttamento:** si utilizzano le policy imparate per raffinare la matrice Q.

Nell' implementazione ho inserito l'iperparametro *exploit*, che indica la probabilità di utilizzare la politica di *sfruttamento* rispetto all' *esplorazione*.

```
x = random number ∈ (0,1)
if x < exploit then
```

[1] https://docs.python.org/3.3/library/turtle.html?highlight=turtle

sfruttamento
else
esplorazione

Implementazione

Ho implementato le seguenti classi/moduli:

classe Environment

Matrice caratterizzata da spazi vuoti, muri e un goal. Ogni stato è associato ad azioni; i muri sono implementati rimuovendo le azioni verso il muro (tra stati adiacenti).
Metodi principali:

- execute: esegue una azione a partire da uno stato, ritorna il nuovo stato;
- reward: ritorna il reward (positivo o negativo) dato uno stato (-0.01 per uno spazio vuoto; +1 per il goal);

classe QLearning

Metodi principali:

- learn: esegue l'algoritmo Q-Learning, iperparametri:
  - epochs: numero di iterazioni dell'algoritmo Q-Learning;
  - exploit: probabilità di utilizzare la politica di sfruttamento invece dell'esplorazione;
  - random_start: per ogni epoch, inizia l'esplorazione dell'environment da uno stato random. Se false, inizia sempre dallo stato (0,0).
- policy: ritorna la miglior policy corrente.

Rappresentazione della Policy: modulo graphics

La policy è una matrice $\{0,1,2,3,4\}^{h,w}$, dove ogni numero rappresenta, rispettivamente, {up, right, down, left, stay} (stay è utilizzato per il goal) e (h, w) sono le dimensioni dell'environment.
La rappresentazione della policy è una matrice delle stesse dimensioni, dove ogni casella contiene una freccia ($\uparrow, \rightarrow, \downarrow, \leftarrow$) che indica la strada da seguire. Il goal è rappresentato da un quadrato rosso. I muri sono una linea tra una cella e l'altra.

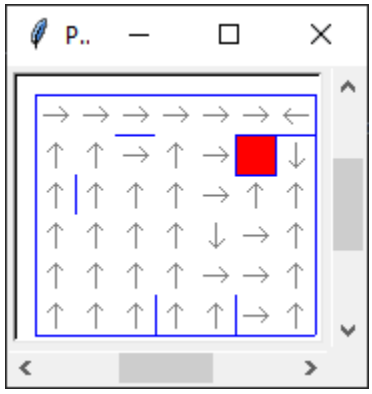

Figure 1. Esempio di policy random

Valutazione della Policy: modulo graph

Trasforma la policy in un grafo e calcola il numero di componenti connesse. Idealmente, deve esserci una componente connessa, cioè da ogni punto del grafo è possibile raggiungere il goal. Se il grafo contiene più componenti connesse, significa che è necessario più training. In un caso reale, si potrebbe includere il check del grafico all'interno della funzione "QLearning::learn", cioè dovrebbe continuare a raffinare la matrice Q fino a quando il grafo contiene una componente connessa (invece di avere le epochs fissate).
n.b.: Questa metrica non controlla l'efficienza della policy (cioè se le strade portano al goal con il percorso minimo).

Visualizzazione policy errata

Di seguito è possibile visualizzare il grafo di una policy errata. Come si può vedere dalla figura, sono presenti 9 componenti connesse. Significa che alcune "strade" non portano al goal.

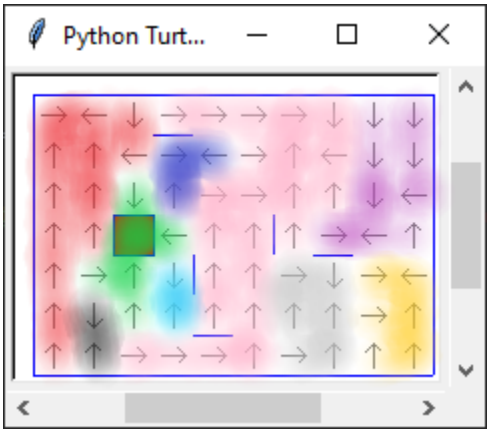

Figure 2. 3 epochs, 7x10

In questo caso è presente solo una componente connessa, quindi la policy è corretta (tutte le "strade" portano al goal).

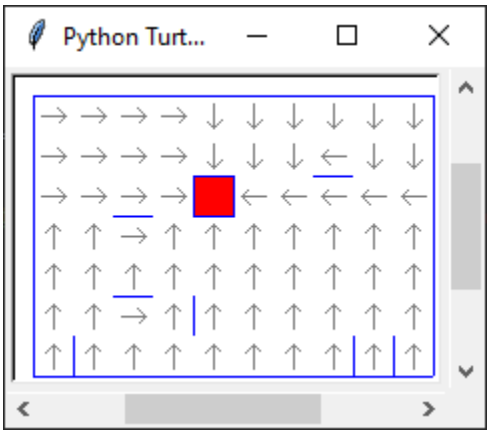

Figure 3. 3 epochs, 7x10

Risualtati

Ho provato ad allenare degli agenti su input diversi. Per ogni riga ho allenato 5 agenti diversi, su environment diversi, per avere una media delle componenti connesse (che uso come metrica).
Colonne:

- exploit: probabilità di utilizzare la policy di sfruttamento;
- environment: dimensioni dell'environment nel quale si muove l'agente;
- epochs: numero di epochs per ogni esecuzione
- delay: tempo di esecuzione (in media tra i 5 agenti)
- components: media del numero di componenti connesse della policy risultante (metrica).

exploit	environment	epochs	delay (sec.)	components (rounded avg.)
0	30x40	16	1.6	57
0	30x40	128	15	1
0	30x40	256	40	1
0.02	30x40	16	1.6	66
0.02	30x40	128	9.2	4
0.02	30x40	256	21.2	1
0.05	30x40	16	1.8	102
0.05	30x40	128	6.6	20
0.05	30x40	256	12.6	13

Table 1. experiments

Rappresentazione di una policy corretta per un particolare environment 30x40:

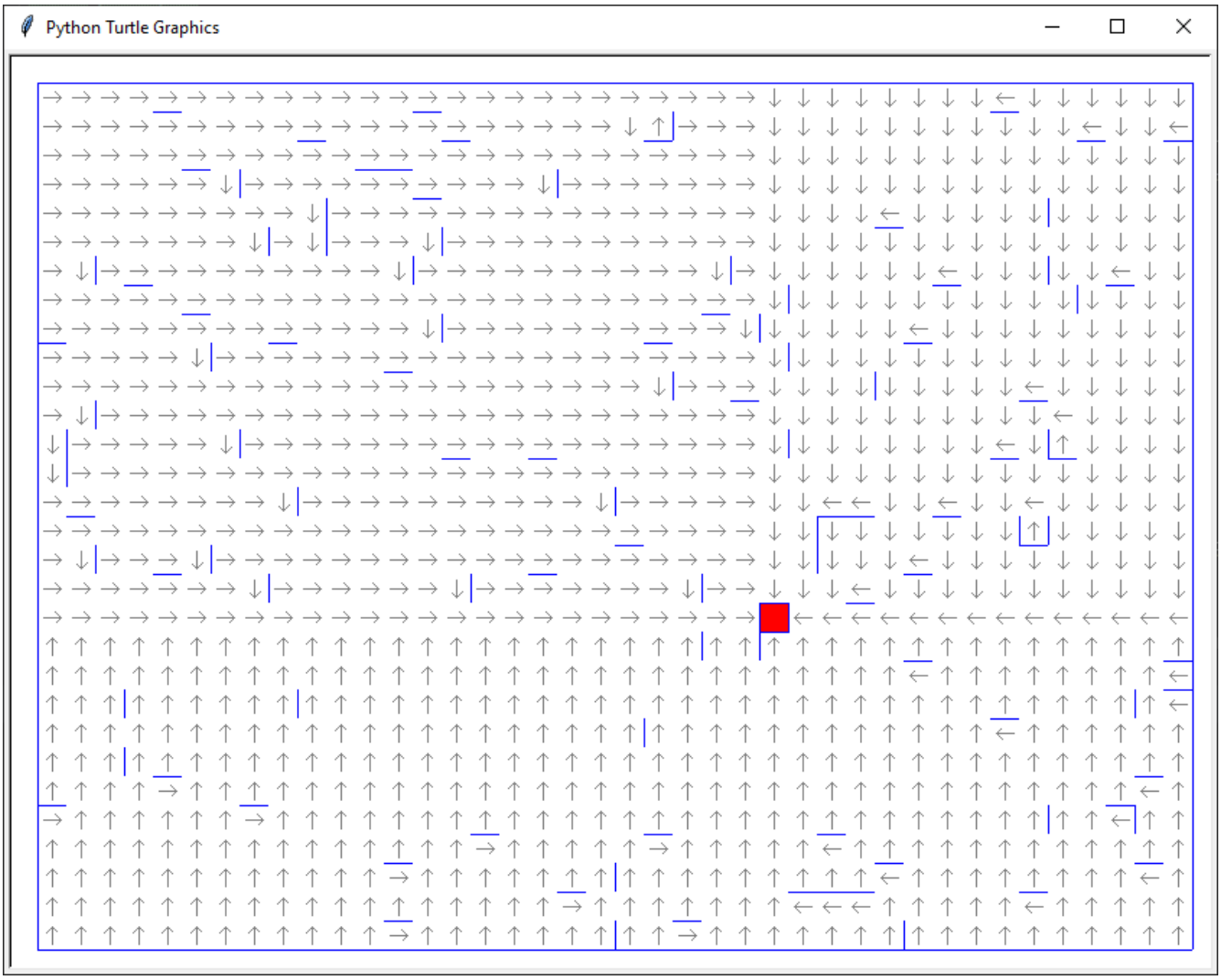

Figure 4. correct policy, 30x40