Reinforcement Learing (beginner)

- Sessione 1 (Markov Decision Process)
- Sessione 2 (Dynamic Programming)
- Sessione 3 (Metodo Montecarlo)
- Sessione 4 (Temportal Difference)

Sessione 1 (Markov Decision Process)

Task di controllo

Il *Task di Controllo* è una sequenza di stati e azioni utili per addestrare l'algoritmo a risolvere un determinato problema.

Il task di controllo è compostato da:

1) Stati S(t) - sono dei momenti nel tempo che assumono un certo valore

2) Azioni A - sono le azioni eseguibili nell'ambiente basate sullo stato del task ed eseguite in un determinato momento nel tempo

3) Ricompensa R - è un valore che l'agente riceve dall'ambiente dopo aver eseguito un'azione

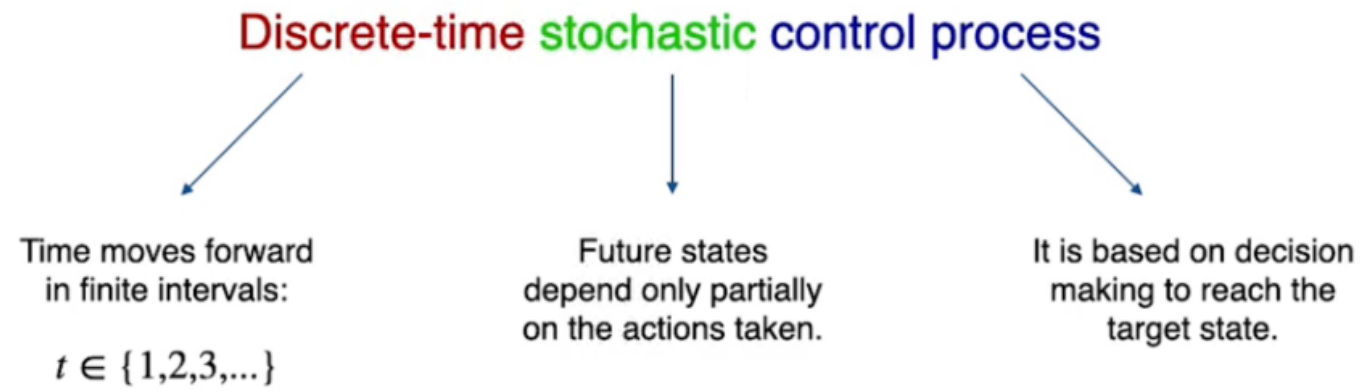
4) Agente - è l'entita che partecipa al task ne osserva lo stato ed effettua le azioni.

5) L'ambiente - è dove vengono eseguite le azioni da parte dell'agente a fronte delle quali vengono restituite *ricompense* e *osservazioni*.

Markov Decision Process

MDP sta per processo decisionale di Markov, è un template che serve per descrivere e gestire i task di controllo.

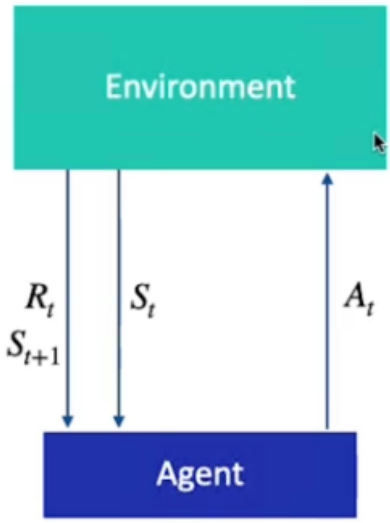
E' un controllo di processo *stocastico* basato su un *tempo discreto.*



L'MDP si basa su 4 elementi detti (S,A,R,P) ovvero:

- il set di possibili stati appartenenti al task
- il set di possibili azioni che possono essere intraprese in ciascun stato
- il set di possibili ricompense restituite a fronte di una azione intrapresa nello stato
- la probabilità di passare da uno stato all'altro eseguendo ogni possibile azione.

Di seguto viene data una rappresentazione grafica:



L'MDP ha una importante proprietà, ovvero la probabilità di visitare lo stato successivo dipende esclusivamente dallo stato attuale, il processo di Markov **NON ha quindi memoria del passato**.

MDP si diverifica nelle seguenti tipologie, ovvero:

- Finito
- Infinito
- Episodico
- Continuo

Nella versione versione *finita* le azioni, gli stati e le ricompense sono finite, mentre nella versione *infinita* uno o più dei valori citati sono infiniti. (un esempio di task finito è il labirinto griglia, mentre

infinito può essere la guida automatica perchè il valore della velocità è infinito)

Nella versiona episofica l'episodio ha uno stato terminale mentre (come per es. l'uscita dal labirinto) nella versione continua non esiste uno stato stato terminale.

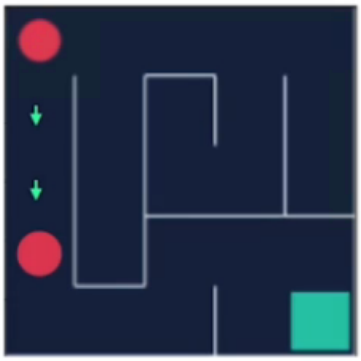
Traiettoria vs Episodio

La **traiettoria** è la seguenza: stato-azione-ricompensa, partendo da uno stato iniziale fino a giungere ad un determinato stato che può anche essere quello finale. La traiettoria di identifica con la lettera greca tau □□

Trajectory:

Elements that are generated when the agent moves from one state to another.

$$\tau = S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3$$



L'episodio in invece è una particolare *traiettoria* che inizia da uno stato e finisce nello stato ***finale*** .

Ricompensa vs Ritorno

La **ricompensa** è il risultato **immediato** che l'azione produce.

Il **ritorno** è la somma delle ricompense che l'agente ottiente da un certo momento nel tempo (t) finchè lo stato è completato. Il ritorno è identificato dalla lettera **G**

Ovvero:

Reward:

$$R_t$$

Return:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T$$

> L'intento è quindi quello di massimizzare il ritorno atteso **Gt**

Fattore di sconto

Il fattore di sconto serve per massimizzare il ritorno nella maniera più veloce e ottimale possibile.

Per ottimizzare il ritorno bisogna quindi moltiplicare la rimpensa ottenuta da ciascuna azione per un fattore di sconto gamma (rappresentato dal carattere γ) compreso tra zero e uno **γ** ∈ [0,1] elevato al momento nel tempo T, ovvero:

$$G_0 = R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \ldots + \gamma^{T-t-1} R_T$$

In questo modo vengono ridotti i valori di ricompensa ottenuti nel futuro, più è alto l'esponente più il valore nel futuro è ridotto, il tutto fino alla fine dell'episodio. (concetto derivato dalla matimatica finanziaria dove lo stesso valore nominare assume maggiore rilevanza quando è ravvicinato nel tempo, minore quando è distante rispetto a T0)

Se il valore di gamma valesse zero, allora non verrebbero considerate le ricompense future (perchè tutte portate a zero), e questa sarebbe una visione miope del task; al contrario del gamma valesse uno, allora non verrebbe calcolato l'episodio ottimale in quanto le ricompense non verrebbero scontate. Un valore gamma spesso utilizzato è 0.99 che garantisce - in genere - di massimizzare il ritorno, ovvero la somma delle ricompense scontate.

Policy

La policy è una funzione che prende in input uno stato e ritorna l'azione che deve deve essere eseguita nello stato stesso. E' rappresentata la lettera greca pigreco (π)

π : S -> A

Le due tipiche rappresentazioni in questo corso sono **π(a|s)** e **π(s)**

π(a|s) è la *probabilità* di eseguire un'azione "a" nello stato "s" -> ritorna una %

π(s) è invece *l'azione* "a" da eseguire nel dato stato "s" -> ritorna una azione "a"

Dipende dal contesto, in alcuni casi verrà restituita la probabilità in altri l'azione.

Tipologie di Policy

Le policy possono essere stocastiche o deterministiche.

Una policy *deterministica* sceglie sempre la stessa azione nello stesso stato, ovvero: π(s) -> **a**

es. π(s) = a1

Una policy *stocatica*, sceglie un'azione sulla base di determinate probabilità, ovvero: π(s) = **[p(a1),p(a1),.....p(an)]**

es. π(s) = [0.3,0.2,0.5] che significa che la policy ha la probabilità del 30% di scegliere la prima azione, del 20% la seconda e del 50% la terza.

La policy **ottimale,** rappresentata da **π*** il cui scopo è massimizzare la somma dei valori di ricompensa scontati alla lunga.

Stato valore

Il valore dello stato è il ritorno dello stato ottenuto interagendo con l'ambiente e seguendo la policy **π** fino alla fine dell'episodio, che si indica:

Vπ(s) = □□[Gt | **S**t=**s]**

di cui il dettaglio della formula:

$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T \,|\, S_t = s\right]$$

dove è visibile l'esplosione del ritorno **Gt** come la somma delle ricompense scontate partendo dallo stato "s" per giungere allo stato finale "T".

Stato-azione (Q-Value)

Se invece vogliamo valutare una azione in un determinato stato, lo facciamo attraverso lo stato-azione Q.

Il valore Q nello stato "s" con l'azione "a" è il ritorno ottenuto eseguendo l'azione "a" partendo dallo stato "s" interagendo con l'ambiente tramite la policy **π** che **seguiremo** fino alla fine dell'episodio.

Di cui la formula generica

$$q_\pi(s,a) = \mathbb{E}\left[G_t \,|\, S_t = s, A_t = a\right]$$

e la versione dettagliata:

$$q_\pi(s,a) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T \,|\, S_t = s, A_t = a\right]$$

L'equazione di Bellman

L'equazione di Bellman serve per trovare la policy ottimale per risolvere i task di controllo.

Bellman: Stato valore

Questa è l'equazione di Bellman che determina il **valore di uno stato** V**π**(s) associato alla policy, ovvero:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}\left[G_t \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\
&= \sum_a \pi(a \mid s) \sum_{s',r} p(s, r \mid s, a)\left[r + \gamma v_\pi(s')\right]
\end{aligned}
$$

Le prime tre formule descrivono che il valore dello stato è il ritorno atteso **Gt** ottenuto seguendo la policy **π** partendo dallo stato "s". (prima riga)

Possiamo espandere la definzione del ritorno Gt come la somma delle ricompense scontate da **γ** fino allo stato terminale. (seconda e terza riga)

Nell'espressione finale che ne deriva, troviamo la probabilità di eseguire ogni azione "a" condizionata nello stato "s", seguendo la policy **π** -> $\sum_a \pi(a \mid s)$ il tutto moltiplicato per $\sum_{s',r} p(s, r \mid s, a)\left[r + \gamma v_\pi(s')\right]$ che rappresenta la probabilità di raggiungere ogni stato successivo (indicato con s') moltiplicato per la ricompensa ottenuta arrivando nello stato successivo sommata al valore dello stato successivo scontato dal fattore gamma.

Notare l'esistenza della ricorsione per quanto concerne il valore **Vπ(s) -> Vπ(s')** che può essere utilizzato durante la stesura dell'algoritmo.

Quindi vengono eseguite tutte le possibili azioni nello stato e calcolato il valore moltiplicando la probabilità nell'azione per il ritorno simmato al valore dello stato succissivo.

Bellman: Stato-Azione (Q-Value)

Questa è l'equazione di Bellman che determina il valore di uno **stato-azione** associato alla policy, ovvero:

$$
\begin{aligned}
q_\pi(s,a) &= \mathbb{E}\left[G_t \mid S_t = s, A_t = a\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t-1} R_T \mid S_t = s, A_t = a\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s', a')\right]
\end{aligned}
$$

Le prime tre formule descrivono che il Q-Value è il ritorno atteso **Gt** associato all'azione "a" ottenuto seguendo la policy **π e** partendo dallo stato "s". (prima riga)

Possiamo espandere la definzione del ritorno Gt come la somma delle ricompense scontate da **γ** fino allo stato terminale condizionate allo stato "s" per l'azione "a". (seconda e terza riga)

Nell'espressione finale che ne deriva, viene indicata la *probabilità* di raggiungere ogni stato successivo **s'** sapendo che abbiamo scelto l'azione "a" -> $\sum_{s',r} p(s', r \mid s, a)$ il tutto moltiplicato per $\left[r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s', a')\right]$ che rappresenta la ricompensa ottenuta raggiungendo lo stato successivo s' sommanta alla somma scontata di **γ** dei Q-values di ciascuna azione eseguita negli stati successivi pesata per la probabilità di eseguire l'azione in s' dettata dalla policy.

Anche qui viene espresso il Q-Value in termine di altri Q-Values in maniera ricorsiva.

Soluzione di un MDP

Per risolvere un MDP bisogna risolvere un *task di controllo*, il che significa **massimizzare** il ritorno atteso. Nella pratica bisogna massimizzare il valore di ogni stato V*(s) -> $v_*(s) = \mathbb{E}_{\pi_*}\left[G_t \mid S_t = s\right]$ o il valore di ogni Q-Value Q*(s,a) -> $q_*(s,a) = \mathbb{E}_{\pi_*}\left[G_t \mid S_t = s, A_t = a\right]$

Per massimizzare questi ritorni bisognerà quindi trovare la policy ottimale **π*** che è quella che sceglie le **azioni ottimali** in **ogni stato** e che porta il **massimo ritorno atteso.**

Di seguto viene rappresentata la formula che sceglie l'azione che restituisce il ritorno più alto

nel caso valore dello stato:

$$\pi_*(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s)\right]$$

e nel caso stato-azione:

$$\pi_*(s) = \arg\max_a q_*(s, a)$$

> Ma pare che ci sia un problema, per trovare la policy ottimale servono dei valori ottimali, e vice versa, ovvero per trovare il valore dello stato ottimale o dello stato-azione ottimale (V* o Q*) bisogna avere una policy ottimale... come fare?

Dobbiamo ottimizzare le equazioni di Bellman:

Per il valore degli stati V*:

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s')\right]$$

ovvero:

Il valore ottimale dello stato è il ritorno atteso ottenuto seguendo la policy ottimale.

La policy ottimale in pratica sceglie **l'azione** che ritorna il valore più alto (di qui la max) associato al ritorno attesso.

Il ritorno atteso è la probabilità di accedere ad ogni stato successivo effettuando l'azione ottimale moltiplicato per la ricompensa ottenuta al raggiungemento dello stato s', sommato al valore di quello stato s' ottimale scontato da gamma.

o nel caso di dello stato-azione Q*

$$q_*(s, a) = \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right]$$

Nel caso invece del valore Q* ottimale per una *azione* "*a*" nello *stato* "*s*":

E' la somma pesata dei ritorni ottenuti raggiungendo ogni possibile stato successivo.

Il ritorno è la ricompensa ottenuta raggiungendo lo stato successivo sommato al valore ottimale massimo dello stato-azione successivo scontato da gamma.

Sessione 2 (Dynamic Programming)

La programmazione dinamica (aka DP) è il primo metodo in grado di risolvere il task di controllo.

Lo scopo del DP è tovare la policy ottimale **π*** per ogni stato V dell'ambiente (che nel nostro caso è discreto)

Per determinare quindi la policy ottimale bisogna rispettare la catena di dipendenze tra stato-azione e stato-valore, ovvero:

$$\pi_* \iff \pi_*(s)$$

$$q_* \iff q_*(s,a)$$

$$v_* \iff v_*(s)$$

Nella pratica l'equazione di Bellman si dettagglia, nel caso di V* come il valore ottimable ricavato dal'azione che lo massimizza, ovvero:

$$v_*(s) = \max_a \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma v_*(s')\right]$$

e nel caso dello stato azione come la probabilità più alta che massimizzi di ritorno:

$$q_*(s,a) = \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right]$$

Partiamo dallo stato valore, per poter tovare il valore ottimale bisogna inzializzare tutti gli stati con dei valori a piacere, poi attraverso un processo detto di "**sweep**" gli stati subiscono una serie di "**passate**" che ne affina man mano i valori fino a farli *convergere* verso l'ottimo. Ogni volta che quindi aggiorniamo i valori stimati andiamo a migliorarli e per questo la nuova stima sarà migliore della precedente.

NOTA: uno dei problemi del DP è che necessita di un "modello perfetto" che descriva con precisione la transizione da uno stato all'altro, cosa che in genere non avviene nella realtà. Il modello perfetto ritorna quindi le probabilità di transizione degli stati, come evidenziato nella parte rossa della formula sotto riportata:

$$V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$$

Un dei limiti che del DP è che parte dal presupposto che si conosca il modello e che quindi se ne possa verificare con esattezza il comportamento e quindi i ritorni. Nella realtà non è possibile fare questo, serviranno altri metodi che analizzano il comportamento dell'ambiente e cercano di stimare un modello.

Iterazione del valore (value iteration V*)

Ora vediamo l'algoritmo di "iterazione del valore" utile per determinare la policy ottimale **π***. Tale algoritmo calcolo lo stato valore ottimale attraverso N passate (dette anche sweep)

$$\pi_*(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s')\right]$$

Questa policy fa in modo che per ciascuno stato venga eseguita l'azione che massimizza il ritorno. Per trovare l'azione migliore dobbiamo però conoscere il valore ottimale dello stato che potrebbe seguire lo stato attuale. Per determinare il valore ottimale dobbiamo implementare un processo iterativo sugli stati, per far questo manterremo una tabella con i valori stimati per ciascuno stato. L'inizializzazione iniziale non deve essere suibito ottimale, può anche contenere valori randomici che attraverso le varie passate (sweeps) migliorano convergendo all'ottimale.

La reqola di aggiornamento degli stati sarà quindi:

$$V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$$

Formula che tradotta significa: determinare l'azione con la probabilità più alta di ottenere la reward che massimizza il ritorno per andare dallo stato s allo stato s'.

Di seguito lo pseudo codice che determina la policy ottimale attraverso la massimizzazioni degli stati valore:

Algorithm 2 Value Iteration

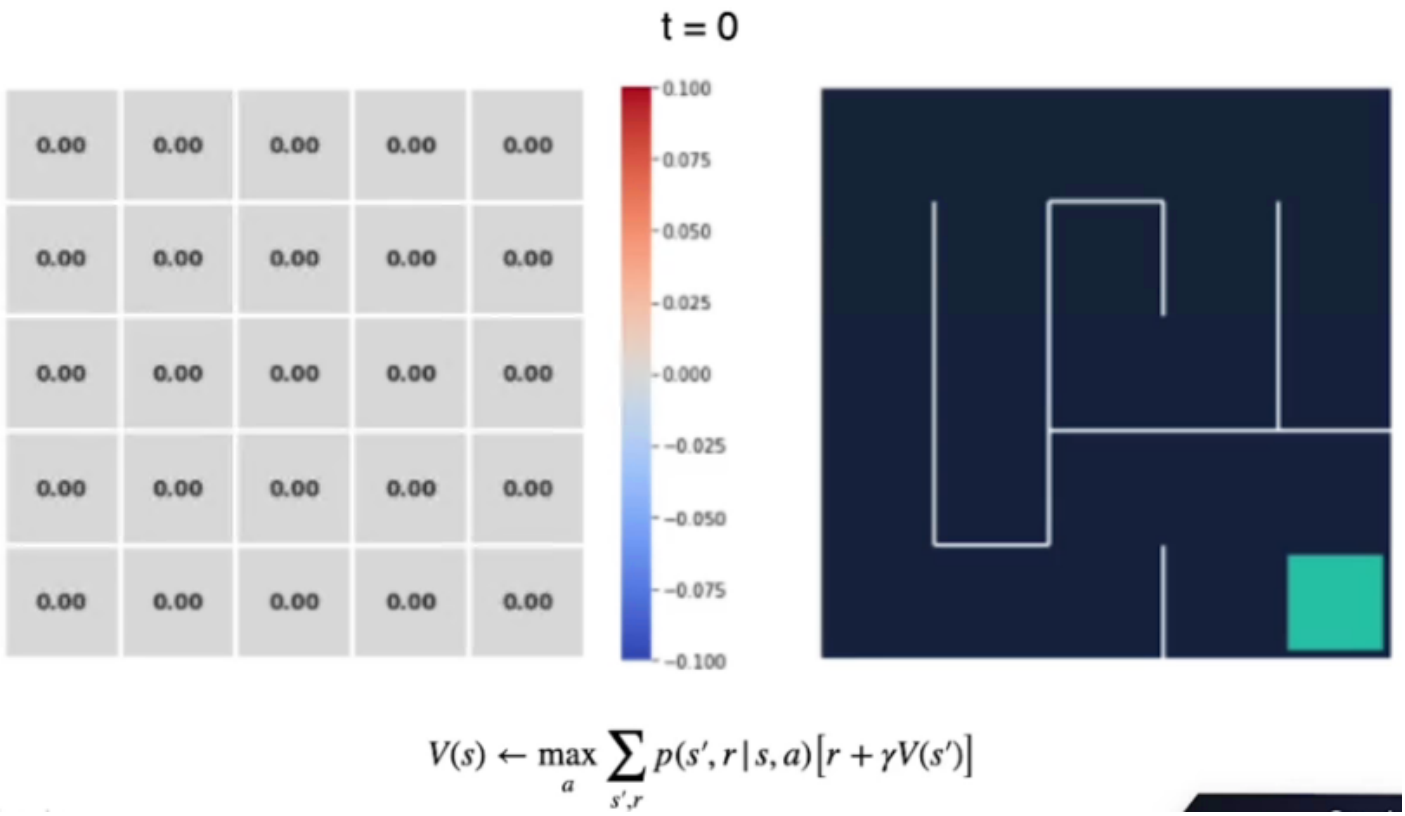
Input: $\theta > 0$ tolerance parameter, γ discount factor
Initialize $V(s)$ arbitrarily, with $V(terminal) = 0$
repeat
    $\Delta \leftarrow 0$
    for $s \in S$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_{a \in A(s)} \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
until $\Delta > \theta$
Output: π: greedy policy w.r.t. $V(s)$

Come si può vedere si tratta due due cicli innestati, il primo che looppa fino a che il valore precedente di ciascun stato si avvicina al valore attuale, il che significa che lo stato valore non sta migliorando più di tanto e quindi possiamo assumere una convergenza.

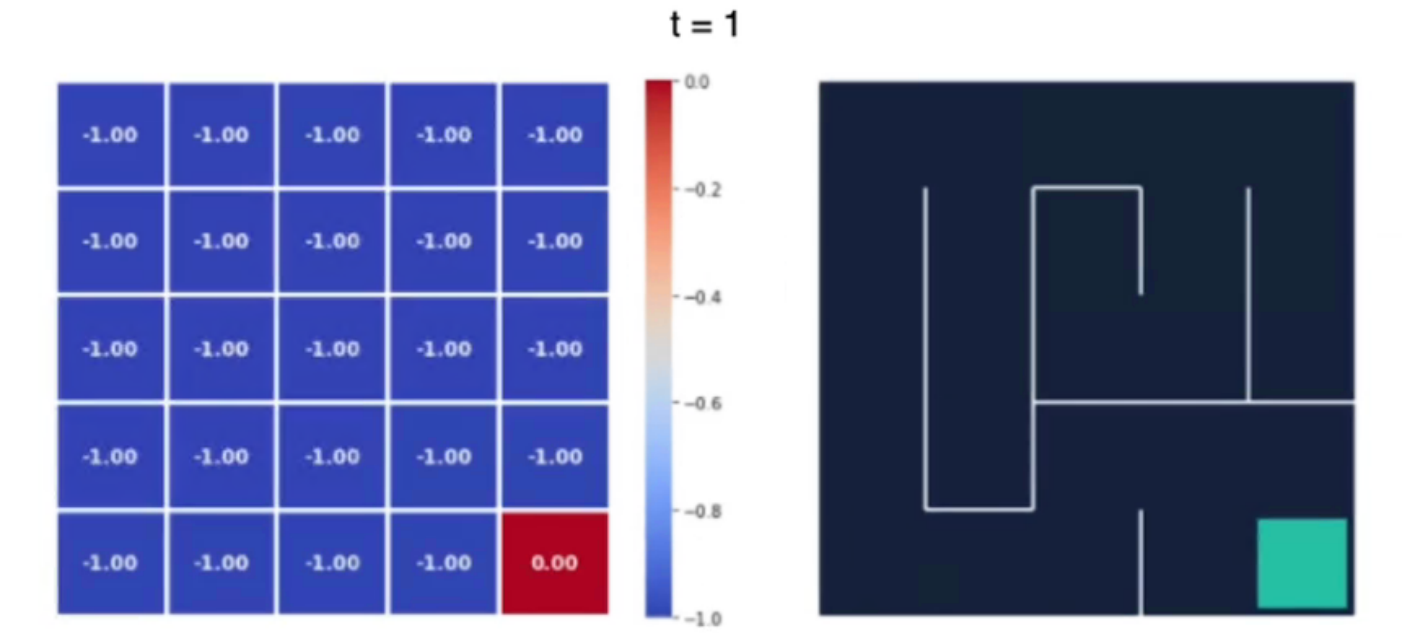
Il secondo loop, fa passare tutti gli stati dell'ambiente e per ciascun stato calcola l'equazione di bellman assumento una policy di default che assegna la stessa probabilità per ciascuna azione. In questo modo a tendere alcune di queste probabilità, pur essendo uguali, andranno a trovare il ritono migliore. **Nella pratica quindi per ogni stato vengono eseguite tutte le azioni possibili e considerata quella che possiede il ritorno più grande.**

Vediamo con un esempio, qui abbiamo il labirinto di 5x5 al tempo t0 i cui valori degli stati sono inizializzati a zero. (sebbene avremmo potuto scegliere altri valori anche random)

La ricompensa per ogni azione è -1 tranne che per lo stato finale che è pari a zero.

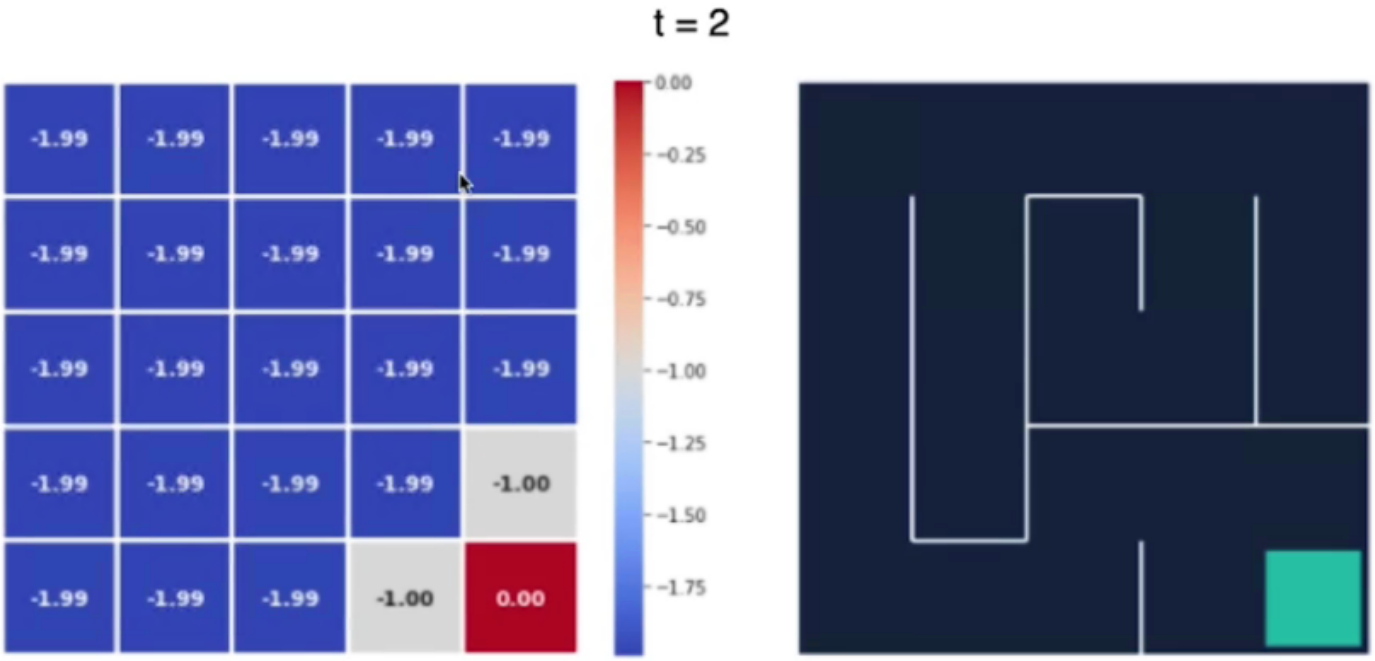


Quindi iniziamo a iterare su tutti gli stati fino all'ultimo che è quello finale, al tempo t1 il valore degli stati sarà:

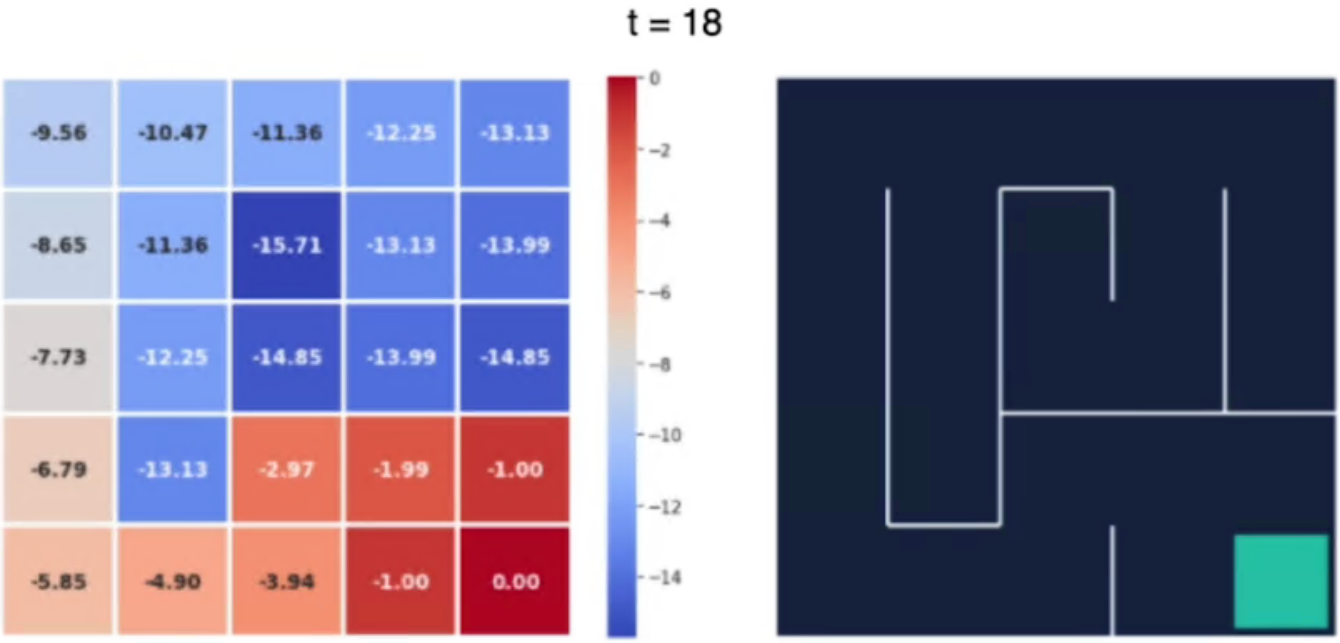


si può notare come le ricompense sono tutte -1 tranne lo stato goal finale.

Al tempo t2 notiamo che il valore degli stati si "propaga" dallo stato goal a quello successivo:



e alla fine dopo 18 iterazioni abbiamo i valori non cambiano in maniera sostanziali, siamo quindi arrivati vicini al valore ottimale:



Possiamo notare che più siamo lontani dal goal e più è basso il valore dello stato.

> L'agente raccoglie più ricompense negative quando è più **lontano** dal goal, invece più il percorso che lo porta al goal è breve, meno saranno le ricompense negative che verrano assegnate allo stao.

Per esempio il valore -15,71 è il più alto perchè da quello stato il percorso per arrivare al goal è il più lungo.

Di seguito l'algoritmo implementato in Python:

Innanzitutto definiamo la policy che inizialialmente sceglie con la stessa probabilità una azione tra le 4 effettuabili per ciascun stato del labirito:

Define the policy $\pi(\cdot|s)$

Create the policy $\pi(\cdot|s)$

In [5]:
```
policy_probs = np.full((5, 5, 4), 0.25)  # 25 states with 4 possible actions [0.25, 0.25, 0.25, 0.25]
```

In [6]:
```
def policy(state):
    return policy_probs[state]
```

Test the policy with state (0, 0)

In [7]:
```
action_probabilities = policy((0, 0))
for action, prob in zip(range(4), action_probabilities):
    print(f"Probability of taking action {action}: {prob}")
```

```
Probability of taking action 0: 0.25
Probability of taking action 1: 0.25
Probability of taking action 2: 0.25
Probability of taking action 3: 0.25
```

inzializziamo anche il valore degli stati:

Create the $V(s)$ table

:
```
state_values = np.zeros(shape=(5,5))
```

implementiamo ora l'algoritmo che va a migliorare il valore degli stati e cambia in base al valore migliorato, la policy scegliendo quella che massimizza il risultato.

```python
def value_iteration(policy_probs, state_values, theta=1e-6, gamma=0.99):
    delta = float("inf")

    while delta > theta:
        delta = 0

        for row in range(5):
            for col in range(5):
                old_value = state_values[(row, col)]
                action_probs = None
                max_qsa = float("-inf")

                for action in range(4):
                    next_state, reward, _, _ = env.simulate_step((row, col), action)
                    qsa = reward + gamma * state_values[next_state]

                    if qsa > max_qsa:
                        max_qsa = qsa
                        action_probs = np.zeros(4)
                        action_probs[action] = 1.

                state_values[(row, col)] = max_qsa
                policy_probs[(row, col)] = action_probs

                delta = max(delta, abs(max_qsa - old_value))
```

I primo while serve per verificare se il valore degli stati è arrivato alla convergenza e quindi non migliora ulteriromente.

I due for servono per sweeppare tutte le righe-colonne, mentre il terzo for innestato esegue tutte le azioni nello stato selezionato dai due for eserni.

Il cuore dell'algoritmo è nel terzo "for" interno dove per ogni azione:

- viene interrogato l'ambiete con l'azione e restituito il risultato (-1 o 0) e lo stato successivo
- viene calcolato il valore dato dall'equazione di Bellman scontata dal fattore gamma
- delle 4 azioni viene salvato il valore calcolato da Bellman più alto e l'azione ad esso associata

Fuori dal terzo "for" delle azioni possibili, viene aggiornato il valore dello stato e aggiornato l'array con l'azione associata al valore dello stato più alto calcolato con Bellman.

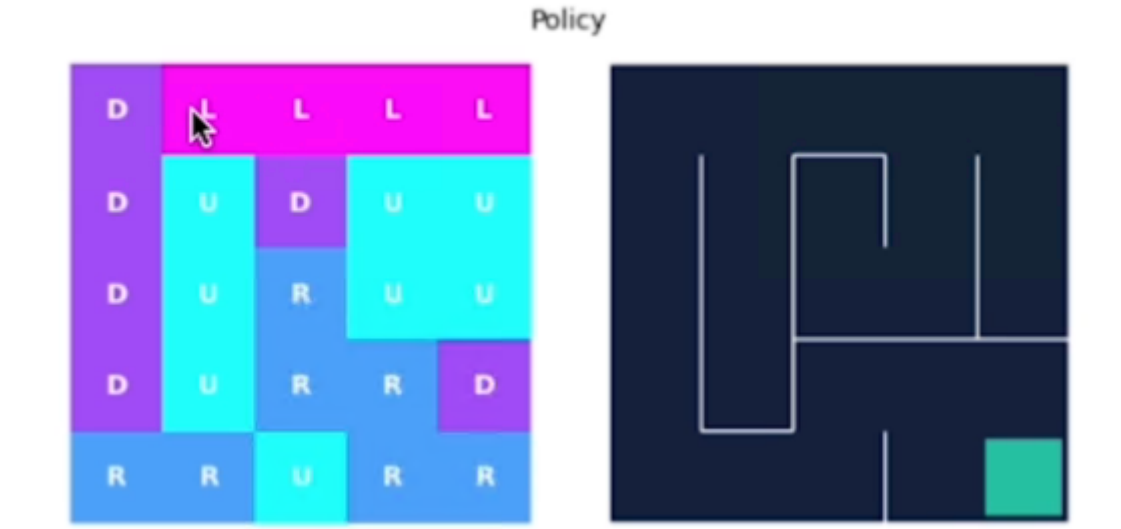
Viene calcolato il delta che se per tutti gli stati e tutte le azioni è inferiore ad una soglia "teta", ovvero che per tutti gli stati-valore non ci sono grandi scostamenti, allora il ciclo termina con valori e azioni ottimali.

Alla fine questo è il risultato dell'algoritmo:



Show resulting policy $\pi(\cdot|s)$

```
]: plot_policy(policy_probs, frame)
```



La policy ci porta quindi direttamente al goal.

Iterazione della Policy (Policy iteration)

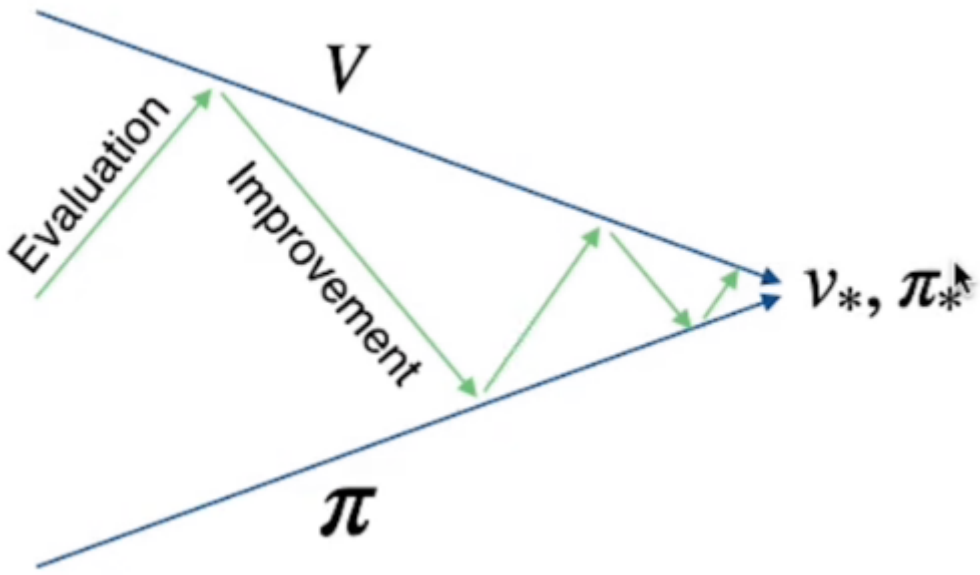
Questo algoritmo evolve il precedente (value iteration) e, dal punto di vista funzionale, funge da base per futuri algoritmi utilizzati nell'ambito del Reinforcemente Learning.

I valori degli **stati** e della **policy** vengono inizializzati con dei valori arbitrari. La policy può essere per esempio definita come la probabilità equamente distribuita di effettuare una azione tra quelle disponibili.

Con la policy inizializzata vengono calcolati i valori degli stati, poi, nella seconda parte, la policy viene migliorata utilizzando i valori degli stati calcolati dalla prima parte dell'algoritmo. Dopo di questo viene ripetuto il miglioramento dei valori degli stati fino a che non si giunge ai valori e alla

policy ottimale.

Queste due ottimizzazioni sono in concorrenza ma nella pratica uno va ad ottimizzare l'altro alternandosi sino a giungere all'ottimale.



Di seguito il pseudo codice dove si possono notare le due fasi di ottimizzazione:

Algorithm 2 Policy Iteration

Input: $\theta > 0$ tolerance parameter, γ discount factor
Initialize $V(s)$ and $\pi(a|s)$ arbitrarily
while policy-stable $= false$ **do**

    Policy Evaluation:
    while $\Delta > \theta$ **do**
        $\Delta \leftarrow 0$
        for $s \in S$ **do**
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
        end for
    end while

    Policy Improvement:
    policy-stable $= true$
    for $s \in S$ **do**
        old-action $\leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_{a \in A(s)} \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
        if old-action $\neq \pi(s)$ **then**
            policy-stable $\leftarrow false$
        end if
    end for

end while
Output: Optimal policy $\pi(a|s)$ and state values $V(s)$

Nella prima parte che calcola il valore degli stati, il loop viene eseguito fino a che non si giunge a dei valori che risultano essere ottimali per l'attuale policy. (loop fintanto che Delta > Theta)

Da notare che, **a differenza dell'algoritmo di value iteration,** dove la regola di aggiornamento dello stato era di aggiornare il valore scegliendo l'azione che massimizzava il ritorno, nella *policy iteration* invece, il valore dello stato viene aggiornato sulla base delle probabilità che la policy assegna ad ogni azione. Dopo ogni "sweep" degli stati, le stime saranno sempre più vicine all'ottimale. V0 -> V1 -> V2 -> ... -> V**π**

Calcolati i volori degli stati, segue l'ottimizzazione delle policy che consiste, per ogni stato dell'ambiente, nell'eseguire tutte le azioni possibili nel singolo stato. Per ogni azione viene applicata l'equazione di Bellman che preleva i valori dagli stati. (ricordo che il valore degli stati era stato calcolato precedentemente applicando la policy **π0** che adesso stiamo aggiornando a **π1**)

A questo punto viene selezionata e salvata nella tabella delle probabilità associate alla policy (sempre per ciascun stato) l'azione che massimizza il ritorno secodo l'equazione di Bellman. Se dopo lo sweep di **tutti** gli stati, le azioni non sono variate rispetto allo sweep precedente, allora

abbiamo trovato la policy ottimale e l'algoritmo si interrompe, altrimenti si rapassa all'ottimizzazione dei valori degli stati con la nuova policy appena calcolata.

Da notare che la policy evaluation continua a looppare fino a che non trova il valore ottimale degli stati con la policy attiva, mentre la policy improvement effettua una sola passata.

Ora vediamo l'implementazione in Python:

anche qui, come nell'value iteration andiamo ad inizalizziare la policy e il valore degli stati:

Create the $V(s)$ table

```
: state_values = np.zeros(shape=(5,5))
```

il valore di default per gli stati sarà zero.

Define the policy $\pi(\cdot|s)$

Create the policy $\pi(\cdot|s)$

```
In [5]: policy_probs = np.full((5, 5, 4), 0.25)  # 25 states with 4 possible actions [0.25, 0.25, 0.25, 0.25]

In [6]: def policy(state):
            return policy_probs[state]
```

Test the policy with state (0, 0)

```
In [7]: action_probabilities = policy((0, 0))
        for action, prob in zip(range(4), action_probabilities):
            print(f"Probability of taking action {action}: {prob}")

Probability of taking action 0: 0.25
Probability of taking action 1: 0.25
Probability of taking action 2: 0.25
Probability of taking action 3: 0.25
```

La probabilità di ogni azione per ciascuno stato sarà di defautl uniforme, ovvero del 25%.

```python
def policy_evaluation(policy_probs, state_values, theta=1e-6, gamma=0.99):
    delta = float("inf")

    while delta > theta:
        delta = 0

        for row in range(5):
            for col in range(5):
                old_value = state_values[(row, col)]
                new_value = 0.
                action_probabilities = policy_probs[(row, col)]

                for action, prob in enumerate(action_probabilities):
                    next_state, reward, _, _ = env.simulate_step((row, col), action)
                    new_value += prob * (reward + gamma * state_values[next_state])

                state_values[(row, col)] = new_value

                delta = max(delta, abs(old_value - new_value))
```

Nella funzione di "Policy evaluation", viene fatta una sweep di tutti gli stati dell'ambiente, dove per ciascun stato vengono eseguite **tutte** le azioni che la policy mette a disposizione sommando nella varibile "new_value" i valori ottenti applicando la formula di Bellman. Si otterrà quindi una sommatoria di valori per ciascin stato, dove gli elementi della sommatoria sono le *azioni dello stato applicate con Bellman*.

Il ciclo si ripete fino a che, lo sweep N ha dei valori (per tutti gli stati) che differiscono di poco rispetto allo sweep N-1 (delta > theta)

In questo caso si passa all'ottimizzazione della policy appena utilizzata, ovvero:

```python
def policy_improvement(policy_probs, state_values, gamma=0.99):

    policy_stable = True

    for row in range(5):
        for col in range(5):
            old_action = policy_probs[(row, col)].argmax()

            new_action = None
            max_qsa = float("-inf")

            for action in range(4):
                next_state, reward, _, _ = env.simulate_step((row, col), action)
                qsa = reward + gamma * state_values[next_state]

                if qsa > max_qsa:
                    new_action = action
                    max_qsa = qsa

            action_probs = np.zeros(4)
            action_probs[new_action] = 1.
            policy_probs[(row, col)] = action_probs

            if new_action != old_action:
                policy_stable = False

    return policy_stable
```

L'ottimizzazione della polcy prevede il classico sweep di tutti gli stati dove per ogni stato viene salvata l'azione della policy che si sta cercando di ottimizzare (che potremmo definire vecchia)

Sempre per ogni stato vengono eseguite tutte le azioni previste dalla policy precedente e tra queste scelta quella che, tramite l'equazione di Bellman, ritorna un valore più alto.

Se dopo aver effettuato uno sweep tutte le azioni definite in tutti gli stati sono identiche allo sweep precedente, allora l'algoritmo ha trovato la policy ottimale e quindi si interrompe, diversamente si passa alla prima parte ovvero la policy evaluation.

Questa è la parte che messe insieme la valutazione della policy e la successiva ottimizzazione.

```python
def policy_iteration(policy_probs, state_values, theta=1e-6, gamma=0.99):
    policy_stable = False

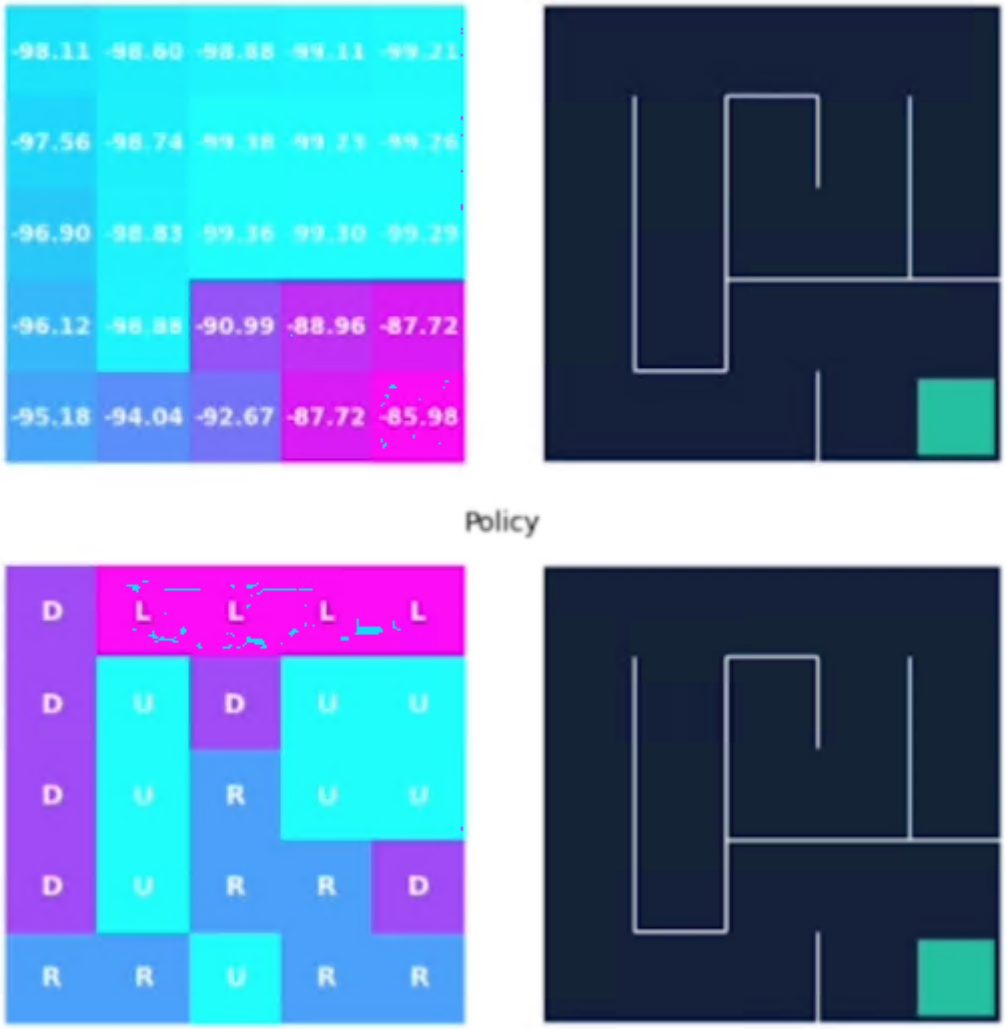
    while not policy_stable:

        policy_evaluation(policy_probs, state_values, theta, gamma)

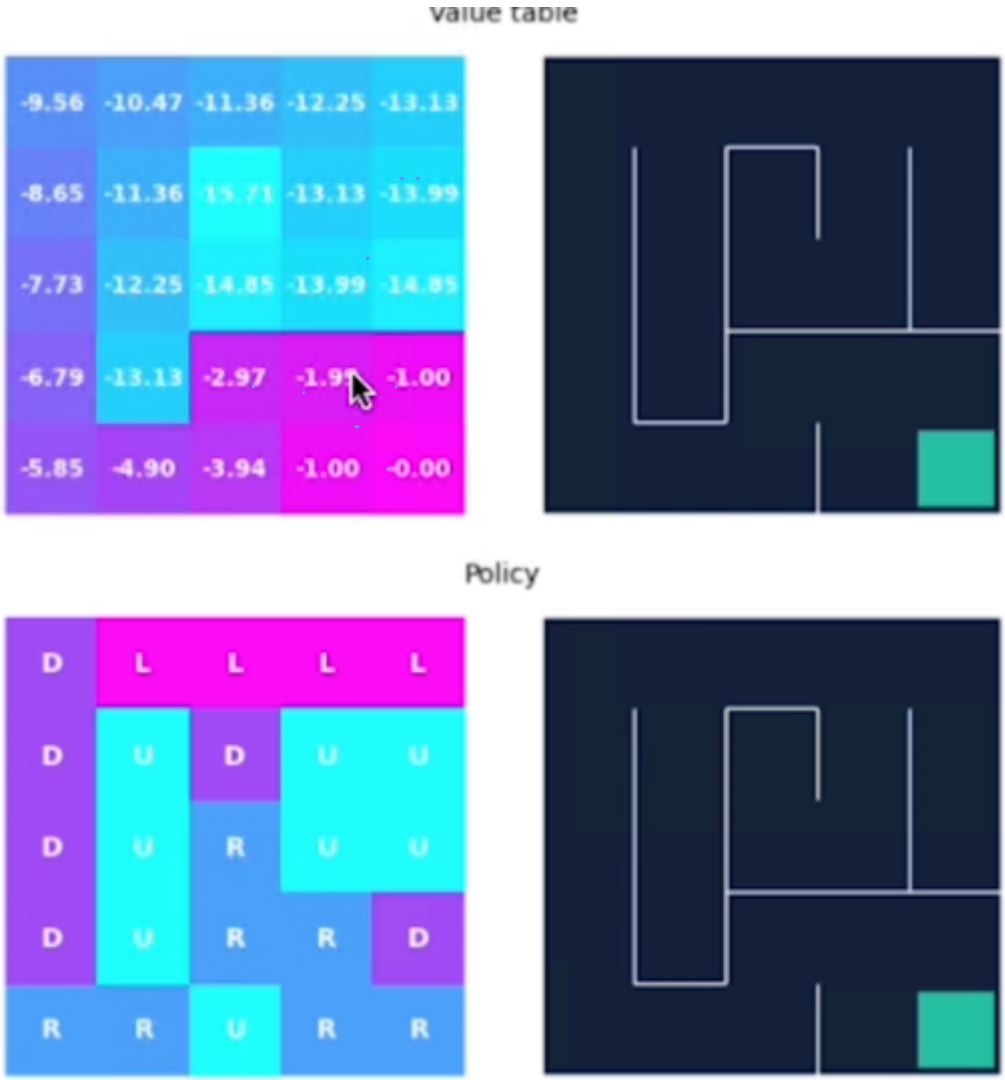
        policy_stable = policy_improvement(policy_probs, state_values, gamma)
```

Di seguito vengono visualizzati i valori degli stati dopo il primo giro, si può notare come tali valori siano particolarmente alti in quanto la policy iniziale impostata, sbaglia molto non essedo ottimizzata. Il valore alto infatti indica il numero di passi che ci sono voluti per la policy a trovare lo stato di uscita.

Bisogna altresì dire che, nonostante i valori alti degli stati, le azioni sono già quelle ottimali, in quanto seppur non ottilame il goal vien raggiunto.



Quelli di seuito sono già valori che si ottengon alla seconda passata con una policy che ha subito un primo processo di ottimizzazione.

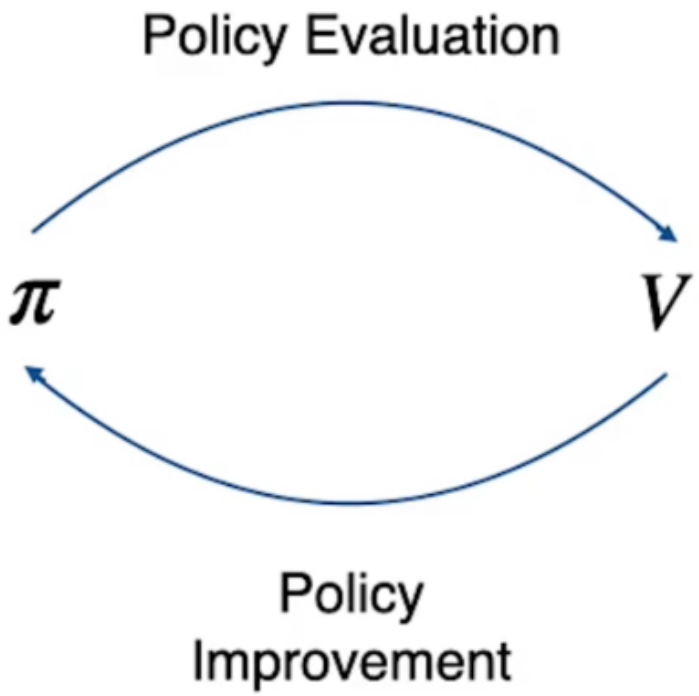


Generalize Policy Evaluation

L'algoritmo di **policy iterarion** ci insegna che questa logica può essere mutuata come un template in molti degli algoritmi di RL attraverso la **valutazione** e il **miglioramento** della *policy* stessa.

Nei prossimi capitoli verranno studiati degli algoritmi più aderenti alla reatà in quanto i modelli non saranno noti (perfetti) a priori come invece avviene nella programmazione dinamica che risulta utile più ai fini didattici che pratici, senza considerate che la DP ha un alto costo computazionale. I problemi reali hanno un vasto se non infinito numero di stati, per cui questa tecnica non è praticabile.

Policy iteration results in the following iterative process:



Sessione 3 (Metodo Montecarlo)

Il Metodo Montecarlo (MC) migliora la policy iteragendo con l'ambiente e ottenendo dei ritorni (scontati da gamma) di cui viene calcolata la media. Per la legge dei grandi numeri più osservazioni (e quindi ritorni) otteniamo più ci avviciniamo al valore ottimale atteso **Vπ(S)**.

$$P\left(\lim_{n\to\infty} \bar{G}_s = v_\pi(s)\right) = 1$$

Dalla formula si evince che il limite per n che tende a infinto dove n è il numero di misurazioni, genera una stima dei valori degli stati con probabilià 100% di essere ottimale.

Il MC ha dei vantaggi rispetto al DP (dynamic programmig spiegato nella sessione 2):

- la stima dei valori degli stati non dipende dagli altri
- il costo per stimare il valore di uno stato è indipendente dal numero toale degli stati

Nel caso del DP l'algoritmo ***bootstrappa*** il valore degli altri stati in modo da utilizzare una stima per produrne un'altra. Per questo la complessità di un algoritmo cresce esponenzialmente con il numero di stati.

Cosa molto importante MC non necessita del modello, la dinamica dell'ambiete sarà implicita nella nostra stima.

Per risolvere l'algorimo del MC verrà utilizzato il metodo, visto in precedenza anche con il DP detto "*Generalized Policy Iteration*".

Generalized Policy Iteration

La GPI si basa sulla valutazione della policy (partendo da una che può essere randomica o arbitraria) per poi migliorarla e looppando fino a quando non si arriva a quella ottiamale.

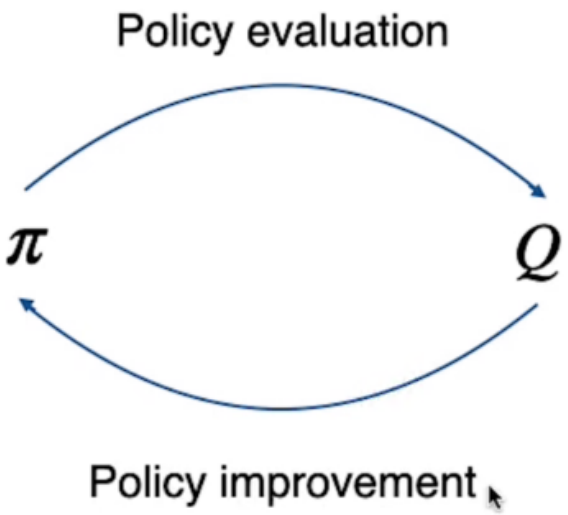
MC si elabora generando un episodio la cui traiettoria parte dallo stato iniziale sino allo stato finale durante il quale vengono raccolte e sommate tutte le ricompense scontate gamma, *per ogni stato*

dell'enviroment.

Bisogna quindi trovare la policy che sceglie l'azione con il Q-Value più alto, nella pratica dovremo tenere traccia nella tabella dei valori, non più i valori degli stati V(s) come nel DP, ma salveremo i valori delle azioni Q(s,a), ovvero dell'azione che massimizza il valore.

$$\pi'(s) = \arg\max_a q_\pi(s, a)$$

Il processo per MC diventerà quindi:



$$\pi_0 \to Q_{\pi_0} \to \pi_1 \to Q_{\pi_1} \to \cdots \to Q_{\pi_*} \to \pi_*$$

Dove la policy calcola i Q-Value che viene a sua volta utilizzato per migliorare la policy in un ciclo continuo fino ai valori ottimali **Q*** e **π***.

Exploration

Quindi la policy migliora sulla base dell'esperienza che agente effettua mentre interagisce con l'ambiente. L'esperienza che l'agente raccoglie dipende dalle azioni che effettua, e le azioni dipendono dalla policy utilizzata in quel momento. Per questo motivo avremo una policy **π'** che seleziona l'azione sulla base delle stime **Q(s,a)**. E queste stime saranno sempre più accurate soprattutto mentre ci avviciniamo alle fasi finali dell'apprendimento. (a differenza delle fase iniziale dove invece sono inaccurate)

Immaginiamo il caso in cui l'azione è ottimale ma la sua stima Q(s,a) è pessima, allora la policy non la sceglierà in quanto la stima del valore è molto bassa. Si rende quindi necessario che *tutte le azioni vengano scelte ogni tanto* in maniera **casuale** per "**esplorare**" l'ambiente con scelte che normalmente non verrebbero effettuate, ma che possono migliorare la policy anche se apparentemente non nell'immediato. Tutto questo per scoprire eventuali azioni ottimali non considerate dalla policy in uso.

> Quindi come mantenere l'esplorazione?

Beh, nella pratica abbiamo due opzioni:

1. La prima si chiama "**exploring starts**" e prevede che l'agente inizi la sua esplorazione in uno stato casuale dell'ambiente ed effettui un'azione iniziale casuale. Purtroppo non è una modalità molto realistica in quanto ci sono molti task che semplicamente non hanno questa possibilità. (soprattutto se parliamo del mondo reale)
2. La seconda si chiama "**stocastic policies**" ovvero vengono considerate le azioni che hanno una probabilità maggiore di zero, in questo modo ogni tanto vengono prese delle azioni "a caso" che potrebbero aiutare la policy a migliorare grazie al caso. (una sorta di evoluzione naturale della specie □□) Queta seconda ipotesti è più realista e più facilmente implementabil.

Le policy stocastiche si suddivino in due tipologie:

- On-policy learning strategies: la quale genera l'esperienza basandosi nulla stessa policy che stiamo ottimizzando
- Off-policy learning strategies: la quale utilizza due policy distinte, una per esplorare l'ambiente e un'altra da ottimizzare

On-policy

Questo metodo segue una strategia che ogni tanto (randomicamente) effettua un'azione a caso, questa policy è chiamata epsilon-greedy. **(ϵ-greedy)**

In questa policy ogni azione ha la probabilità di essere eseguita maggiore di zero, ogni volta che bisogna scegliere un'azione ne scegliamo una casuale tra quelle disponibili nello stato con probabilità **ϵ** (che quindi deve essere abbastanza bassa) mentre nelle restanti probabilità **1-ϵ** andiamo a scegliere l'azione con probabilità più alta, ovvero:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \epsilon_r & a = a^* \\ \epsilon_r & a \neq a^* \end{cases} \qquad \epsilon_r = \frac{\epsilon}{|A|}$$

come si può vedere dalla formula la probabilità di scegliere l'azione ottimale **a*** (quindi con la stima q-value più alta) è **1-ϵ** sommato alla probabilità di scegliere un'azione casualmente, mentre, per contro, abbiamo la probabilità **ϵ** di sceglire una azione che **potrebbe** non essere ottiamale. **ϵr** rappresenta la probabilità di scegliare un'azione tra tutte le azioeni **A** disponibili.

Facciamo un esempio:

Abbiamo 4 possibili azioni e un **ϵ** del 20%, come sotto riportato:

$$|A| = 4, \quad \epsilon = 0.2$$

$$\pi(a|s) = \begin{cases} 1 - 0.2 + 0.05 = 0.85 & a = a^* \\ 0.05 & a \neq a^* \end{cases} \qquad \epsilon_r = \frac{0.2}{4} = 0.05$$

La probabilità di sceglire l'azione con il valore **Q(s,a)** più alto è 80%.

Quando scegliamo un'azione a caso **ϵr**, la probabilità di scegliare cmq un'azione ottimale tra le 4 possibili è del 0,2/4 che sono le azioni possibili, ovvero del 5%. (o 0,05)

Per questo la probabilità di scegliere un'azione ottimale è del 0,85 (85%) perchè tra le 4 azioni c'è quella migliore (delle 4) che comunque va sommata a **1-ϵ.**

Di seguito il pseudo codice che descrive l'alagoritmo:

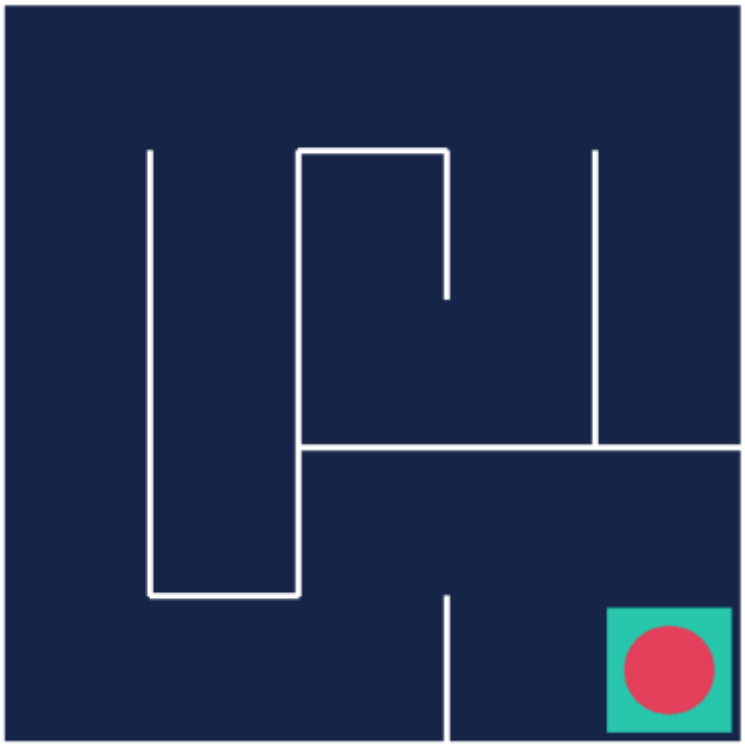
Algorithm 1 On-policy Monte Carlo Control

Input: ϵ random action probability, γ discount factor
$\pi \leftarrow$ e-greedy policy w.r.t $Q(s,a)$
Initialize $Q(s,a)$ arbitrarily, with $Q(terminal, \cdot) = 0$
$G(s,a) \leftarrow [\,]$
for episode $\in 1..N$ **do**
    Generate episode following $\pi : S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    for $t \in T - 1..0$ **do**
        $G \leftarrow R_{t+1} + \gamma G$
        Append G to $G(S_t, A_t)$
        $Q(s,a) \leftarrow average(G(S_t, A_t))$
    end for
end for
Output: Near optimal policy π and action values $Q(s,a)$

In input viene passato epsilon e gamma (che ricordo rappresentare il fattore di sconto)

Vediamolo in codice Python.

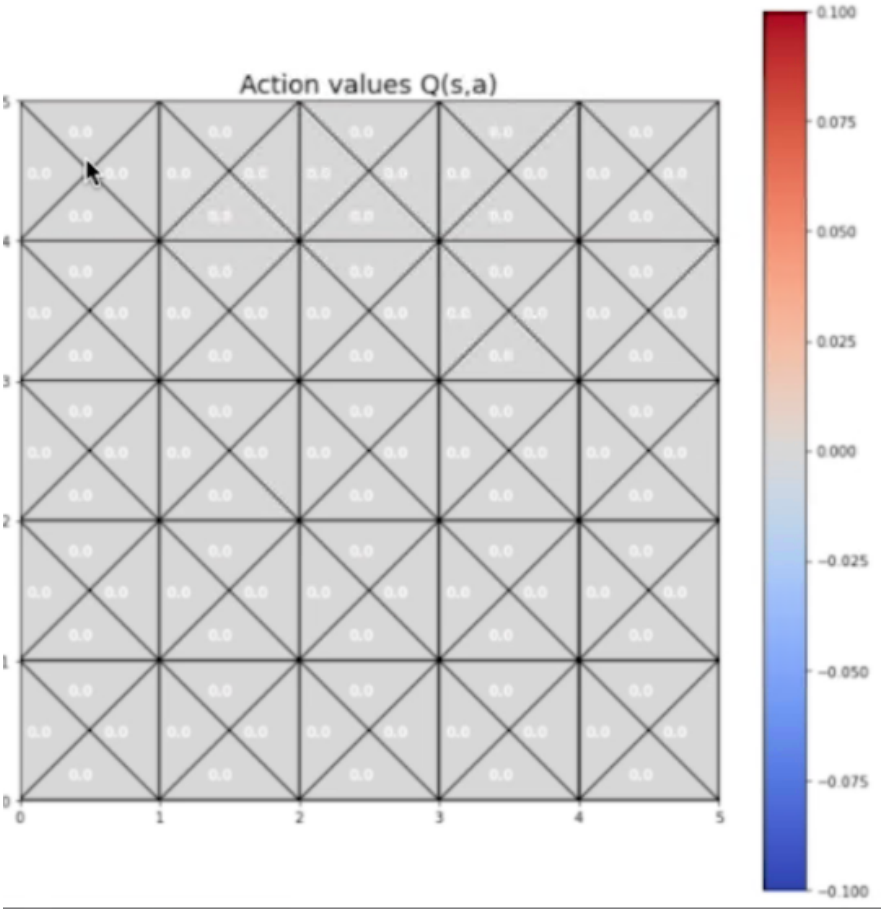
L'esempio di utilizzo è il classico labirinto 5x5, dove l'agente inizia nell'angolo in alto a sx e finisce il suo percorso in basso a dx.



Per prima inizializziamo la matrice che contiene le azioni effettuabili nello stato con valori zero il cui shape è 5x5x4 dove 5x5 sono gli stati mentre 4 sono le azioni effettuabili in ciascun stato. Il valore per inizializzare scelto è zero ma avrebbe potuto essere un qualsiasi valore arbitrario che il processo di apprendimento avrebbe comunque ottimizzato.

```
action_values = np.zeros((5, 5, 4))
```

e ora plottiamo la rappresentazione dei Q-values associati a ciascuno stato:



Si possomo vedere i valori per 4 movimenti effettuabili in ciascono.

Creiamo una policy che scelga una azione dato uno stato e la probabilità di scegliere un'azione casuale. (**ϵ**)

```
def policy(state, epsilon=0.2):
    if np.random.random() < epsilon:
        return np.random.choice(4)
    else:
        av = action_values[state]
        return np.random.choice(np.flatnonzero(av == av.max()))
```

La funzione che sceglie la policy, effettua un'azione a **caso** tra le 4 disponibili se un numero compreso tra zero e uno, generato randomicamente, è inferiore al epsilon. (prima riga della funzione)

Nel caso invece nel quale si effettui l'azione migliore (**1-ϵ**) allora vengono in prima battuta estratti i 4 valori Q(s,a) associati allo stato passato in input. Di questi 4 valori viene scelto quello con il Q(s,a) più alto e, nel caso i valori più alti siano indentici tra loro, allora viene effettuata una scelta random tra questi. (vedi ultima riga della funzione)

NOTA: la funzione np.flatnonzero ritorna gli indici di un array che possiedono un valore diverso da zero.

giusto per cuiosità ho estrapolato la parte di codice che esegue l'azione con il Q-value più alto per far vedere come vengono gesiti i casi particolari come più valori identici massimi:

```
action_values = np.zeros((5,5,4))

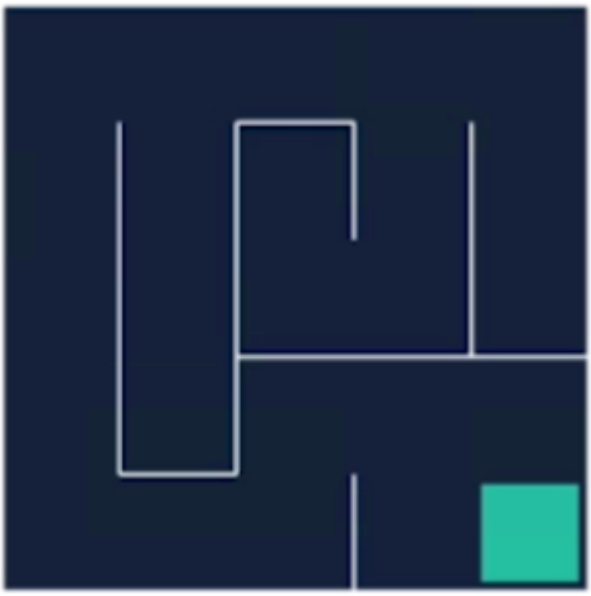
def policy(state, epsilon=0.01):
    av= action_values[state]
    print (av.max())
    print (av == av.max())
    print ( np.flatnonzero(av == av.max()))
    return np.random.choice(np.flatnonzero(av == av.max()))


print (policy((0,0)))

0.0
[ True  True  True  True]
[0 1 2 3]
3
```

Visualizziamo la policy con i valori inizializzati, ovviamente essendo inizializzati l'azione di default è univoca per tutti gli stati.





ora definiamo l'algoritmo principale

```
def on_policy_mc_control(policy, action_values, episodes, gamma=0.99)
"""
Algoritmo di Montecarlo nella modalità on-policy
policy: funzione policy che scaglie le azioni, spiegata prima e che
        attinge dalla tabella degli stati (action_values)
```

```
action_values: tabella contente tutte le azioni effettuabili per tutti gli stati dell'env
episodes: numero di episodi utili per far si che la policy migliori
gamma: fattore di sconto
"""

  # dizionario dove verranno salvati i valori associati alle coppie stato-azione
  sa_return={}

  # main loop
  for episode in range (1, episodes +1)

    # ricavo lo stato iniziale
    state = env.reset()

    # flag che determina la fine dell'episodio
    done = False

    # lista alla quale appendere i valori ritornanti dall'ambiente a fronte in una zione
    transtions = []

    # loop che gira finchè l'agente trova l'uscita e quindi termina il task
    while not done:

      # effettuo l'azione utilizzando la policy che abbiamo definito (cone random actions)
      action = policy(state, epsilon)

      # salvo le risposte dell'ambiente
      next_state, reward, done, _ = env.step(action)

      # salvo lo stao-azione e la reward ottenuta
      transtions.append ([state,action,reward])

      # salvo il prossimo stato da eseguire
      state = next_state

    # inizializzo il rtorno
    G = 0

    # calcolo il ritorno in modalità "backword" ovvero dall'ultimo al primo
    for state_t, action_t, reward_t in reverse(transtions)
```

```
        G = reward_t + gamma*G

        # se l'elemento non esiste nel dictionary allora lo creo con per la coppia stato-azione
        if not (state_t, action_t) in sa_retutns:
          sa_returns[(state_t, action_r)] = []

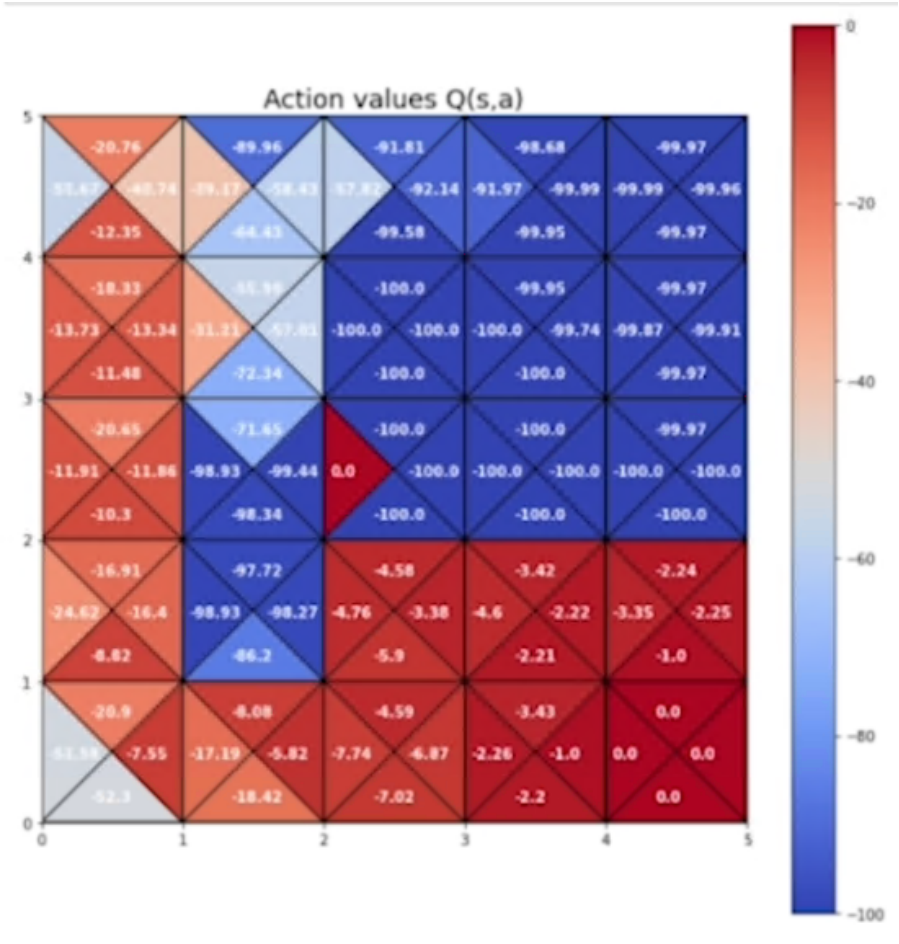
        # aggiungo il ritorno
        sa_returns[(state_t, action_r)].append(G)

        # salvo nella stato/azione la media dei ritorni per lo stato-azione
        action_vales[state][action] = np.mean (sa_returns[(state_t, action_t)])



# test
on_policy_mc_control(policy, action_values, episodes = 10000)
```

di seguito la tabella delle azioni ottimali negli stati:



di seguito la mappa delle azioni migliori che portano alla fine del tastk





Oltre al set delle azioni migliori si vede come negli stati non ottimali (in alto a dx) l'agente le evita tranne per il fatto che ogni le esplora casualmente.

Ottimizzazione On-Policy

L'ottimizzazione consiste nel modificare l'algoritmo per aggiornare i valori degli stati in maniera più efficiente semplificando l'algoritmo, mantenendo allo stesso tempo la sua effeficacia.

Le differenze si sostanziano in:

1. La prima differenza consiste nel fatto che non teniamo traccia dei ritorni osservati dall'agente
2. La seconda invece nel "pushare lentamente" il valore della stima di una percentuale alpha **(α)** che moltipica la differenza tra il ritorno appena calcolato e il precedente valore stato-azione, ovvero: **Q(s,a) = Q(s,a) + α[G - Q(s,a)]**

Vediamo il pseudo-codice

Algorithm 2 On-policy Monte Carlo Control

Input: ϵ random action probability, γ discount factor
$\pi \leftarrow$ e-greedy policy w.r.t $Q(s,a)$
Initialize $Q(s,a)$ arbitrarily, with $Q(terminal,\cdot) = 0$
for episode $\in 1..N$ **do**
    Generate episode following $\pi : S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    for $t \in T-1..0$ **do**
        $G \leftarrow R_{t+1} + \gamma G$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\,[G - Q(s,a)]$
    end for
end for
Output: Near optimal policy π and action values $Q(s,a)$

l'implementazione

```
def on_policy_mc_control(policy, action_values, episodes, gamma=0.99, alpha=0.2)
"""
Algoritmo di Montecarlo nella modalità on-policy
policy: funzione policy che sceglie le azioni, spiegata prima e che
        attinge dalla tabella degli stati (action_values)
action_values: tabella contente tutte le azioni effettuabili per tutti gli  stati dell'env
episodes: numero di episodi utili per far si che la policy migliori
gamma: fattore di sconto
alpha= parametro che muove lo stato valore di una percentuale che va in direzione del ritorno
appena osservato
"""


  # main loop
  for episode in range (1, episodes +1)

    # ricavo lo stato iniziale
    state = env.reset()

    # flag che determina la fine dell'episodio
    done = False

    # lista alla quale appendere i valori ritornanti dall'ambiente a fronte in una zione
    transtions = []
```

```
    # loop che gira finchè l'agente trova l'uscita e quindi termoina il task
    while not done:

      # effettuo l'azione utilizzando la policy che abbiamo definito (cone random actions)
      action = policy(state, epsilon)

      # salvo le risposte dell'ambiente
      next_state, reward, done, _ = env.step(action)

      # salvo lo stao-azione e la reward ottenuta
      transtions.append ([state,action,reward])

      # salvo il prossimo stato da eseguire
      state = next_state

    # inizializzo il rtorno
    G = 0

    # calcolo il ritorno in modalità "backword" ovvero dall'ultimo al primo
    for state_t, action_t, reward_t in reverse(transtions)
      G = reward_t + gamma*G

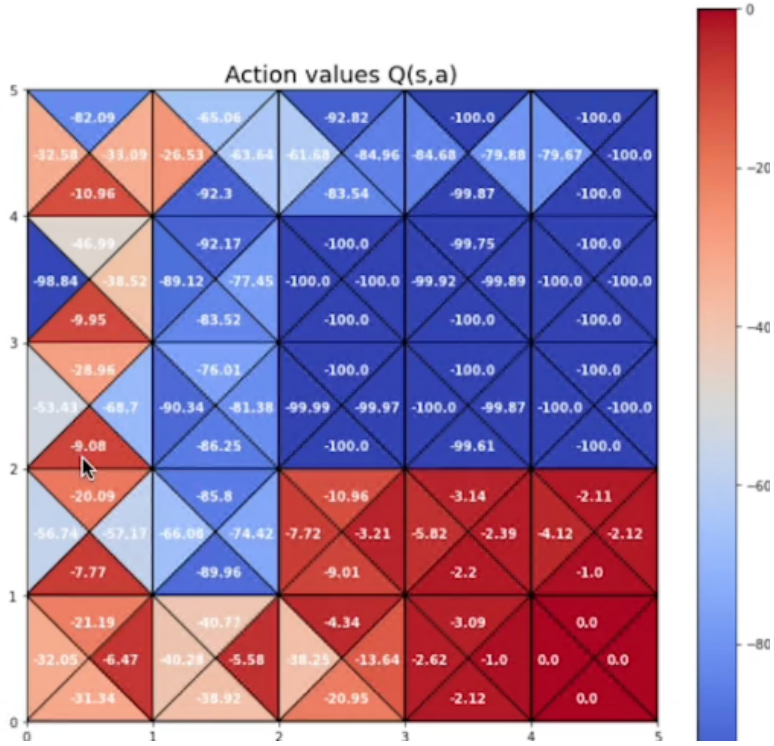
      # ottengo il q-value dalla tabella dei valori
      qsa =  action_vales[state][action]

      # ottimizzazione, mi muovo nella direzione del valor appena calcolato
      action_vales[state][action] += alpha * ( G - qsa)



# test
on_policy_mc_control(policy, action_values, episodes = 10000)
```

Visualizzare la nuova tabella stati valore



Off-Policy

E' la seconda strategia (la prima è l'on-policy) utile per mantenere il miglioramento e l'esplorazione. La logica è sempre quella di effettuare, ogni tanto, un'azione "sub-ottimale" e per questo motivo vengono utilizzate due policy separate.



Exploratory policy

$$b(a \mid s)$$

Target policy

$$\pi(a \mid s)$$

Nell'apprendimento MC-Off Policy, andiamo quindi a separare la "Exploratory policy" **b(a|s)** dalla "Target policy" **π(s,a)** (quella da ottimizzare)

La policy di esplorazione effettuerà quindi una traiettoria explorativa la cui esperienza verrà utilizzata dalla target per essere migliorata. **π(s,a) <- arg max Q(s,a)**

Nella pratica i valori della target policy verranno aggiornati sui campioni raccolti dalla exploration policy. Per funzionare, la exploratory policy deve raccogliere tutte le azioni che la target policy può effettuare. Quindi se la target ha una probabilità di effettuare un'azione > 0 allora anche l'exploratory deve avere questa probabilità.

Ovvero: if **π(s,a) > 0** allora **b(s,a) >0** altrimenti ci potrebbero essere azioni che la target sceglie mentre la exploratory no, il che non farebbe migliorare la target.

Nella pratica viene calcolato il ritorno medio utilizzano la policy di eplorazione andando ad approssimare il valore Q(s,a) non della target policy. Per migliorare la target policy bisogna quindi utilizzare una tecnica chiamata "*importance sampling*" la quale va a stimare i valori attesi di una distribuzione (la target), lavorando con i campioni di un'altra (l'exploratory).

Importance sampling

> Il metodo utilzzato per legare statisticamente le due policy è chiamato "**Importance sampling**" aka IS, consta nel moltiplicare il *ritorno* al tempo "*t*" per un valore chiamato "**Wt**" il cui valore è dato dal rapporto tra: **tutte** le probabilità generate allo stato **k** dalla target policy nell'effettuare le azioni, **divisa** dalla proprietà generata dalla exploration policy anch'essa nel scegliere le azioni.

$$W_t = \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}$$

E[**Wt** x **Gt** | St = s] = Vπ(s)

> In questo modo moltiplicando l'**IS** per il ritorno **Gt** andiamo ad approssimare il valore ottimale della target policy **Vπ(s)**

La regola di aggiornamento dei valori ottimali Q-values (s,a) è quella di tenere traccia della lista di tutti i ritorni osservabili, poi quando c'è da aggiornare i valori Q si fa la media di tutti i ritorni G precedenti. Il ricalcolo però è inefficiente perchè necessita di un grosso quantitativo di memoria per salvare tutti i ritoni G.

Utilizzeremo quindi il metodo già visto nel *MC On-Policy ottimizzato*, che va a "spostare" lentamente lo stato valore verso il nuovo valore osservato:

Update rule for $Q(s,a)$:

$$Q(s,a) \leftarrow Q(s,a) + \frac{W_t}{C(s,a)} \left[G - Q(s,a)\right]$$

where:

$$C(s,a) = \sum_{k=1}^{N} W_k$$

solo che questa volta **non** utilizzeremo un valore alpha discrezionale invece verrà utilizzato l'" **Importance sampling**" precedente descritto, normalizzato con la sommatoria C(s,a) di tutti i valori "importance samples" precedenti.

Per evitare distorsioni dovute ai valori di IS, lo si va a normalizzare divedendolo per tutti gli IS osservati per lo stato azione elaborati. (vedi immagine sopra) Questo manterrà gli aggiornamenti tra zero e uno.

Di seguito il pseudo codice:

Algorithm 2 Off-policy Monte Carlo Control

Input: γ discount factor
$\pi \leftarrow$ greedy policy w.r.t $Q(s,a)$
$b \leftarrow$ arbitrary policy with coverage of π
$C(s,a) \leftarrow 0$
Initialize $Q(s,a)$ arbitrarily
for episode $\in 1..N$ **do**
    Generate episode following $b : S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    for $t \in T-1..0$ **do**
        $G \leftarrow R_{t+1} + \gamma G$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} \left[G - Q(S_t, A_t)\right]$
        if **then** $A_t \neq \pi(S_t)$
            Break the loop, move to next episode.
        end if
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
    end for
end for
Output: Optimal π and action values $Q(s,a)$

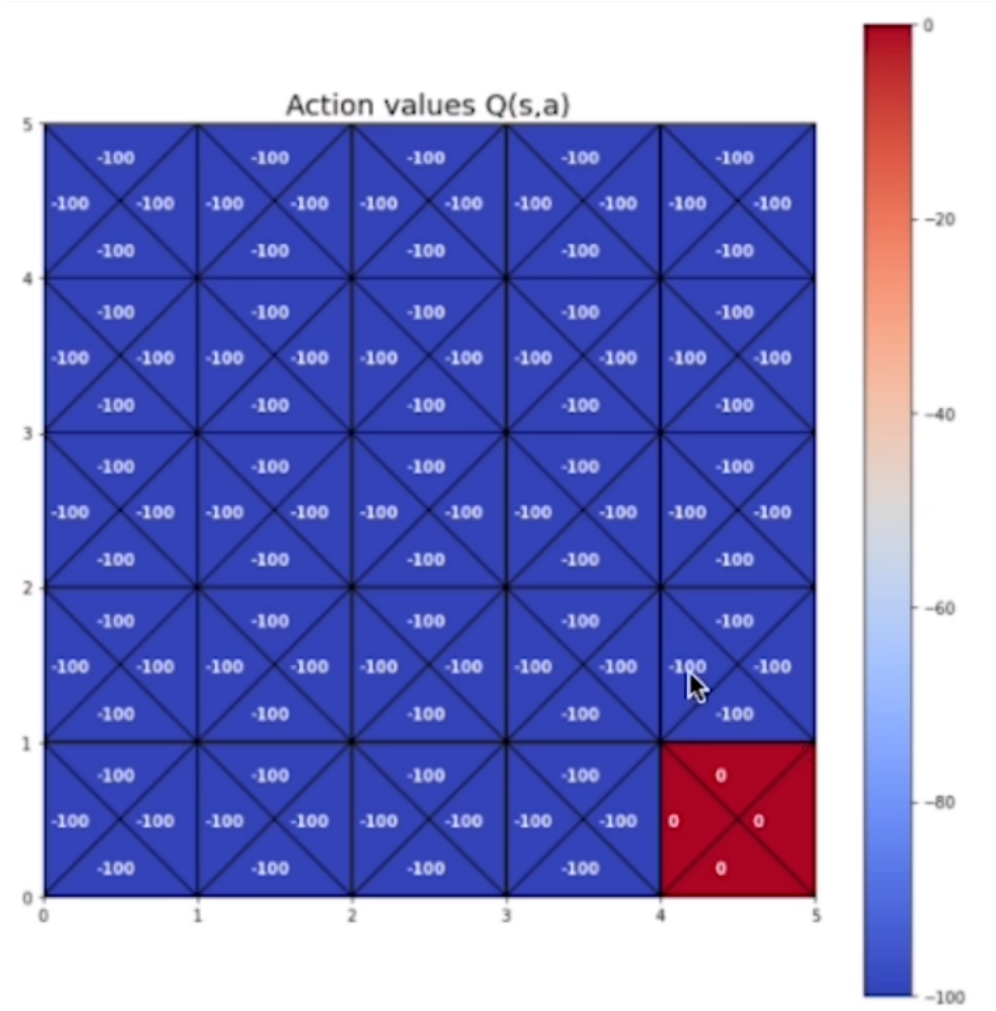
E' simile all'online MC ma utilizza due policy e tiene un totalizzatore di "important sampling ratio C(s,a)" per tutti gli stati-azioni.

Passiamo ora all'implementazione

```
# inizializzazione della tabella Q-value (s,a) con valori arbitrari
action_values = np.full((5,5,4),-100)

# setto il valore del goal a zero
action_values [4,4,:] = 0.
```

e visualizziamo la tabella con i valori:




```
# creiamo la policy
# scegliere un valore a caso tra quelli pià alti
def target_policy(state):
    av = action_values[state]
```

```
    return np.random.choice(np.flatnonzero(av == av.max()))

# creiamo una policy esplorativa
def exploratory_policy (state, espsilon=0.2):
  """
  state: id del valore azione
  espsilon: probabilità di intraprendere un'azione casuale
  """
    # definiaimo che ogni tanto casualmetne effettua un'azione casuale
    if np.random.random() < espsilon:
        return np.random.choice(4)
    else:
        # in caso in cui sceglie l'azione migliore
        return target_policy(state)



# implementiamo l'algoritmo MC-off policy
def off_policy_mc_control(action_values, target_policy, exploratory_policy, episodes,
gamma=.99, espsilon=.2):
  """
  action_values: tabella stati azioni Q(s,a)
  target_policy: policy da ottimizzare
  exploratory_policy: policy che esplora casuale ogni tanto (espilon)
  episodes: numero di episodi
  gamma: fattore di sconto
  espsilon: probabilità di scelta di un'azione casuale
  """

    # creiamo una matrice dove verranno salvate le somme dei rapporti associati agli IS
inizializzandola a zeroes
    # la matrice contiene un valore per ogni combinazioni di stato-azione 5x5 stati x4 valori
a stato
    csa = np.zeroes ((5,5,4))

    # ciclo per tutti gli episodi passati in input
    for episode in rage (1, episodes +1):

      # inizializzo il ritorno a zero
      G = 0
```

```
      # inizializzo l'importance sampling (rapporto tra le due policy)
      W=1

      # inizializzo le variabili del task
      state = env.reset()
      done = False
      transition = [] # array dove vengono salvate le ossercazioni ritornate dell'env

      # loop che si ripete fino alla fine dell'episodio
      while not done:

        # usimo la exploratory policy
        action = exploratory_policy(state, espsilon)

        # eseguiamo l'azione "esplorativa"
        next_state, reward, done, _ = env.step(action)

        # salvo l'osservazione ritornata dall'env
        transition.append([state,action,reward])

        # salvo lo stato per il prossimo giro
        state = next_state

      # utilizziamo l'esperienza ottenuta dall'esplorazione per migliorare la policy target
      # parto dalla fine del task e calcolo il ritorno G scontato da gamma
      # L'ordine è inverso per facilitare il calcolo dei ritorno
      # Nella pratica faccio passare tutti gli stati-azioni che sono stati ritornati durante
      # la fase di esplorazione. Per ciascuno di essi calcolo:
      # 1) il rapporto IS
      # 2) uso il rapporto IS per "spostare" proporzionalmente il valore Q(s,a) verso il
ritorno appena calcolato
      #    e lo sommo alla tabella stato-azione
      for state_t, action_t, reward_t in reversed(transition):

        # calcolo il ritorno scontato da gamma
        G = reward_t + gamma * G

        # calcolo l'important sampling W
        csa [state_t,action_t] += W
```

```
        # salvo il vecchio valore associato allo stato-azione
        qsa = action_values[state_t][action_t]

        # aggiorniamo i q-values spostando il valore
        action_values[state_t][action_t] += (W/csa[state_t][action_t]) * (G -qsa)

        # dopo il ricalcolo degli stati azioni riprovo la target policy e verifico  se
l'azione è diversa da quella precedente
        # se così allora interrompo il miglioramento e riprendo con l'esplorazione con un
altro episodio
        if action_t != target_policy(state_t):
          break



        # ricalcolo l'IS
        W = W * 1 / (1-epsilon + epsilon/4 )
```

Sessione 4 (Temportal Difference)