Appunti di Reinforcement Learning

Gabriele Marino

June 2024

Contents

1 Markov Decision Process

È un processo stocastico che descrive un ambiente in cui un *Agente* può prendere decisioni. Esso può essere rappresentato da un grafo orientato, dove i nodi sono gli *Stati* e gli archi sono le *Azioni*. La probabilità di passare da uno stato ad un altro dipende da tutte le azioni prese fino a quel momento:

$$p(s_t = \hat{s}|s_1, ..., s_{t-1})$$

Ma noi faremo riferimento alla **Markov Property**:

$$p(s_t = \hat{s}|s_1, ..., s_{t-1}) = p(s_t = \hat{s}|s_{t-1})$$

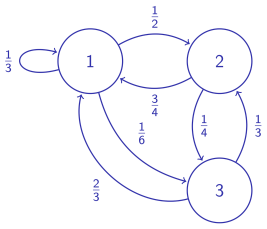
Come passare da uno stato all'altro è deciso da una distribuzione di probabilità detta **Transition Kernel**:

$$p(s_t = \hat{s}|\hat{s_{t-1}})$$

Non è detto a priori che io mi trovi in uno stato con probabilità certa, anzi, la probabilità di trovarmi in uno stato è data da una distribuzione di probabilità, e quindi, se prendiamo uno MDP con 3 stati, la probabilità di trovarmi in uno stato è data da un vettore di probabilità grande 3, dove ongni entry è la probabilità di trovarmi in uno stato: $\pi = [\pi_1, \pi_2, \pi_3]$.

Fra gli elementi che compongono un MDP troviamo:

- S: Spazio degli stati
- A: Spazio delle azioni
- $P \in \mathbb{R}^{|S|\times|S|}$: Transition Matrix
- $s^{(0)} \in \wp(S)$: Stato iniziale preso da una distribuzione di probabilità iniziale



(a) Grafo MDP

$$P = \begin{bmatrix} \frac{1}{3} & \frac{3}{4} & \frac{2}{3} \\ \frac{1}{2} & 0 & \frac{1}{3} \\ \frac{1}{6} & \frac{1}{4} & 0 \end{bmatrix}$$

(b) Transition Matrix

$$s^* = \left[\frac{66}{127}, \frac{40}{127}, \frac{21}{127}\right]^T$$

(c) Stato finale

Un Markov Process definisce lo stato $s^{(t+1)}$ come dipendente dalla Transition Matrix P, e $\pi^{(t)}$, ovvero la distribuzione di probabilità dei vari stati:

$$s^{(t+1)} = P \cdot \pi^{(t)} = P^t \cdot s^{(0)}$$

Abbiamo una distribuzione Stazionaria, il che significa che la distribuzione di probabilità non cambia nel tempo, e quindi, per $t \to \infty$, abbiamo una saturazione della distribuzione di probabilità, che arriva ad un punto di equilibrio:

$$s^* = \lim_{t \to \infty} P^{(t)} = P \cdot s^{(0)}$$

2 Agent, Environment e Markov hypothesis

L'agente è colui che si muove in un ambiente, attraverso delle azioni che lo portano in stati diversi. L'ambiente è tutto ciò che circonda l'agente, e che può essere influenzato dalle azioni dell'agente, e che può influenzare l'agente. L'ambiente (o Environment) è rappresentato da un MDP: $< S, A, P, R, \gamma >$, dove:

- S: Spazio degli stati, che è un insime di interi rappresentati i vari stati disponibili
- A: Spazio delle azioni, che è un insieme di interi rappresentati le azioni disponibili
- $P : S \times A \to S$ (Oppure $P : S \times A \to \wp(S)$): detta state transition function, che rappresenta la probabilità di passare da uno stato ad un altro
- $R : S \times A \to \mathbb{R}$: detta reward function, che rappresenta la ricompensa che l'agente riceve per essere passato da uno stato ad un altro
- γ: Discount Factor, utile per evitare che la somma delle ricompense sia infinita, e quindi evitare i loop nel MDP

2.1 Agent

L'agente è colui che prende le decisioni, e che si muove all'interno dell'ambiente. Esso dipende da una **Policy** π, che è una funzione che mappa uno stato ad un'azione:

$$\pi : S \to A$$

Oppure:

$$\pi : S \to \wp(A)$$

Essa serve per valutare le due funzioni:

- **Value Function**: $V_\pi : S \to \mathbb{R}$: che rappresenta il valore atteso del ritorno cumulativo a partire dallo stato s
- **Q-function**: $Q_\pi : S \times A \to \mathbb{R}$: che rappresenta il valore atteso del ritorno cumulativo a partire dallo stato s e prendendo l'azione a

2.2 Definizioni Analitiche

- **Policy**: $\pi(a|s) = \mathbb{P}(a_t = a|a_t = s)$
- **Reward**: $R_s^a = \mathbb{E}[r_{t+1}|s_t = s, a_t = a]$
- $R_s^\pi = \sum_{a \in A} \pi(a|s) \cdot R_s^a = \mathbb{E}[R_s^a]$
- $P(s'|s, a) = P_{ss'}^a = \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a)$
- $P_{ss'}^\pi = \sum_{a \in A} \pi(a|s) \cdot P_{ss'}^a = \mathbb{E}[P_{ss'}^a]$
- **Return**: $G_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1}$
- **Value Function**: $V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$
- **Q-function**: $Q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$

2.3 Policy

Policy π è la strategia che l'agente segue per prendere le decisioni.

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$$

2.4 Reward

Reward R_s^a è la ricompensa che l'agente riceve per essere passato dallo stato s all'azione a.

$$R_s^a = \mathbb{E}[r_{t+1}|s_t = s, a_t = a]$$

2.5 Return

Il Return G_t è la somma delle ricompense future a partire dal tempo t, pesate dal Discount Factor γ.

$$G_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1}$$

Il Discount Factor γ è un valore compreso tra 0 e 1, che serve per evitare che la somma delle ricompense sia infinita, e quindi evitare di preferire i loop nel MDP.

2.6 Value Function

Value function $V(s)$ è la funzione che assegna ad ogni stato s il valore atteso del ritorno cumulativo a partire da s. In altre parole, $V(s)$ è il valore atteso della somma delle ricompense future a partire da s:

$$V(s) = \mathbb{E}[G_t|s_t = s]$$

dove s_t è lo stato al tempo t.

$$G_t = R_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

dove γ è il fattore di Discount, r_t è la ricompensa al tempo t e s_t è lo stato al tempo t. La value function $V(s)_\pi$ è la value function che tiene conto della policy π, ovvero il la somma delle ricompense a partire da uno stato s, seguendo la policy π.

2.7 Q-function

Q-function $Q(s, a)$ è la funzione che assegna ad ogni stato-azione pair (s, a) il valore atteso del ritorno cumulativo a partire da s e prendendo l'azione a:

$$Q(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a]$$

La Q-function $Q(s, a)_\pi$ è la Q-function che tiene conto della policy π, ovvero il la somma delle ricompense a partire da uno stato s, e seguendo l'azione a al primo passo, e la policy π negli step successivi.

2.8 Bellman Equation

La Bellman Equation è un'equazione che lega il valore di uno stato al valore dei suoi successori, facendoci capire che il valore di uno stato è legato al valore dei suoi successori.

$$G_t = r_{t+1} + \gamma G_{t+1}$$

$$V_\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma V_\pi(s_{t+1}) | s_t = s]$$

$$Q_\pi(s, a) = \mathbb{E}_\pi[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a]$$

Dimostrazione: Consideriamo la definizione di $G_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1}$ e togliamo fuori dalla sommatoria il primo termine:

$$G_t = \gamma^0 r_{t+1} + \sum_{k=0}^{\infty} \gamma^{k+1} \cdot r_{t+k+2} = r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+2} = r_{t+1} + \gamma G_{t+1}$$

Dove possiamo mettere in evidenza il termine di Discount γ, rimandendo il valore del suo esponente all'interno della sommatoria pari a k, perchè tanto $k = 1, ..., \infty$.

Similmente possiamo fare per $V_\pi(s)$ e $Q_\pi(s, a)$:

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1} | s_t = s] =$$

$$\mathbb{E}_\pi[r_{t+1} + \gamma V_\pi(s_{t+1}) | s_t = s]$$

2.9 Corollaio delle equazioni di Bellman

Dalle equazioni di Bellman possiamo ricavare una serie di corollari, che ci permettono di capire come si comportano le funzioni $V_\pi(s)$ e $Q_\pi(s,a)$:

- $V_\pi(s) = \sum_{a\in A} \pi(a|s) \cdot (R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a \cdot V_\pi(s'))$
- $Q_\pi(s,a) = R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a \sum_{a'\in A} \pi(a'|s') \cdot Q_\pi(s',a')$

Dimostrazione: Per dimostrarlo, sviluppiamo l'equazione di Bellman per $V_\pi(s)$, risolvendo le aspettazioni matematiche:

$$V_\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma V_\pi(s_{t+1})|s_t = s]$$
$$= \sum_{a\in A} \pi(a|s) \cdot \mathbb{E}[r_{t+1} + \gamma V_\pi(s_{t+1})|s_t = s, a_t = a]$$

Da qui l'aspettazione matematica $\mathbb{E}[r_{t+1}] = R_s^a$ e $\mathbb{E}[\gamma V_\pi(s_{t+1})] = \gamma \sum_{s'\in S} P_{ss'}^a \cdot V_\pi(s')$ da cui ricaviamo la prima equazione:

$$V_\pi(s) = \sum_{a\in A} \pi(a|s) \cdot (R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a \cdot V_\pi(s'))$$

2.10 Forma Chiusa delle Equazioni di Bellman

Le equazioni di Bellman possono essere scritte in forma chiusa, ovvero in forma matriciale, dove la matrice V_π è un vettore di dimensione $|S|$, e la matrice R_s^a è una matrice di dimensione $|S|\times|A|$, e la matrice $P_{ss'}^a$ è una matrice di dimensione $|S| \times |S|$.

$$V_\pi = (I - \gamma P^\pi)^{-1} \cdot R^\pi$$

Dimostrazione: Per dimostrarlo, partiamo dal corollario della equazione di Bellman per $V_\pi(s)$, e scriviamolo in forma matriciale:

$$V_\pi(s) = R^\pi + \gamma P^\pi \cdot V_\pi$$

Portiamo $\gamma P^\pi \cdot V_\pi$ a sinistra:

$$V_\pi - \gamma P^\pi \cdot V_\pi = R^\pi$$

Mettiamo in evidenza V_π:

$$(I - \gamma P^\pi) \cdot V_\pi = R^\pi$$

E infine isoliamo V_π:

$$V_\pi = (I - \gamma P^\pi)^{-1} \cdot R^\pi$$

Notiamo che R^π è un vettore le cui entrate sono R_s^π, e che P^π è una matrice le cui entrate sono $P_{ss'}^\pi$. La notazione chiusa è difatti un sistema lineare, che diventa impraticabile da risolvere in contesti reali, con un numero di stati e azioni molto elevato.

2.11 Definizione di Optimal Policy:

Per il motivo descritto sopra, si preferisce utilizzare algoritmi di approssimazione, dove un Agente ottimo è dato da una **Ottima Policy** π^* che massimizza il valore atteso del ritorno cumulativo, per ogni stato s di partenza:

$$\pi^* = \arg\max_{\pi} \mathbb{E}[G_1 | s_1 \sim \wp(S)] = \arg\max_{\pi} \mathbb{E}_{\pi}[V_{\pi}(s_1) | s_1 \sim \wp(S)]$$

2.12 Definizione di Optimal Value Function:

L'Optimal Value Function $V^*(s)$ è la funzione che assegna ad ogni stato s il valore atteso del ritorno cumulativo a partire da s, seguendo la policy ottima π^*:

$$V^*(s) = \max_{\pi} V_{\pi}(s)$$

2.13 Definizione di Optimal Q-function:

L'Optimal Q-function $Q^*(s, a)$ è la funzione che assegna ad ogni stato-azione pair (s, a) il valore atteso del ritorno cumulativo a partire da s e prendendo l'azione a, seguendo la policy ottima π^*:

$$Q^*(s, a) = \max_{\pi} Q_{\pi}(s, a)$$

2.14 Teorema

Per ogni MDP esiste almeno una policy ottima π^* che è deterministica. In oltre vale che:

$$\pi^*(a|s) = \begin{cases} 1 & \text{se } a = \arg\max_{a \in A} Q^*(s, a) \\ 0 & \text{altrimenti} \end{cases} \tag{1}$$

Alla fine il nostro obiettivo è quello di trovare l'ottima Value Function $V^*(s)$ o l'ottima Q-function $Q^*(s, a)$, che ci permettono di trovare la policy ottima π^*. Ma esiste un problema: esse dipendono dalla Policy stessa, per cui non si conoscono a priori.

3 Bellman Optimality Equation

Dato che la forma chiusa delle equazioni di Bellman non è praticabile, si preferisce utilizzare le Bellman Optimality Equation, che sono delle equazioni che legano il valore di uno stato all'azione ottima, e il valore dei suoi successori:

$$V^*(s) = \max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \cdot V^*(s')$$

$$Q^*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \cdot \max_{a' \in A} Q^*(s', a')$$

Dimostrazione: Per definizione possiamo scrivere $Q^*(s)$ come:

$$Q^*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \cdot V^*(s')$$

E sostituendo $V^*(s')$ con il massimo valore possibile, $V^*(s) = \max_{a \in A} Q^*(s,a)$, otteniamo la seconda equazione. sostituendo invece al contrario, l'estressione di $Q^*(s',a')$ in quella di $V^*(s')$, otteniamo la prima equazione.

Nota: Le Bellman Optimality Equation sono delle equazioni non lineari, e quindi non sono facilmente risolvibili, e più in generale non esiste nemmeno una soluzione chiusa per le Bellman Optimality Equation. Per risolverle si preferisce utilizzare algoritmi di approssimazione che permettono di approssimare il valore di $Q^*(s,a)$.

Considerazioni:

- Nei classici MDP (Markov Decision Processes), si assume che l'agente possa osservare completamente lo stato dell'ambiente.
- Potremmo avere a che fare con dei Partial Observable Markov Decision Processes (POMDP), dove l'agente non può osservare completamente lo stato dell'ambiente. Questi ambienti sono definiti:

  $$POMDP =< S, A, O, P, R, Y, \gamma >$$

  , dove:

  - S: Spazio degli stati
  - A: Spazio delle azioni
  - O: Spazio delle osservazioni
  - $P : S \times A \to \wp(S)$: Transition Matrix
  - $R : S \times A \to \mathbb{R}$: Reward Function
  - $Y : S \to O$ oppure $Y : S \to \wp(O)$: Observation Function
  - γ: Discount Factor

  In questo contesto l'agente è descritto come: $< b, \pi, V_\pi >$ o $< b, \pi, Q_\pi >$, dove:

  - b: Belief Function, che rappresenta la distribuzione di probabilità sui possibili stati dell'ambiente
  - π: Policy, che rappresenta la strategia dell'agente
  - V_π: Value Function, che rappresenta il valore atteso del ritorno cumulativo a partire da uno stato

4 Policy Evaluation Paradigm

Nel contesto generale del Reinforcement Learnig, abbiamo a che fare con due tipi di problemi di ottimizzazione:

- **Policy Evaluation**: Data una policy π, calcolare la Value Function V_π o la Q-function Q_π
- **Policy Optimization**: Trovare la policy π^* che massimizza la Value Function V_π o la Q-function Q_π

Ora prendiamo in considerazione il primo problema, ovvero il Policy Evaluation.

Il Policy Evaluation Paradigm è un paradigma che si basa sull'idea di valutare la policy π data, calcolando la Value Function V_π o la Q-function Q_π. L'idea del valutare una policy è quella di capire quanto essa sia valida.

In questa fase la value function V_π o la Q-function Q_π sono inizialmente sconosciute, e vanno valutate esplorando l'ambiente tramite la policy π, tenendo traccia delle ricompense ottenute nel tempo.

Alla fine si usa la Value Function V_π o la Q-function Q_π per valutare la policy π, e capire se essa è buona o meno.

La convergenza è garantita sotto ipotesi blande.

Inizialmente la policy π è scelta in modo casuale, e poi viene migliorata iterativamente. Essa però ci permette di esplorare l'ambiente, e di capire come esso si comporta. Successivamente la policy π viene migliorata iterativamente, fino a trovare la policy ottima π^*, attraverso l'uso di algoritmi, come può essere l'algoritmo di *greedy*. Ogni iterazione è detta Generalized Policy Iteration (GPI).

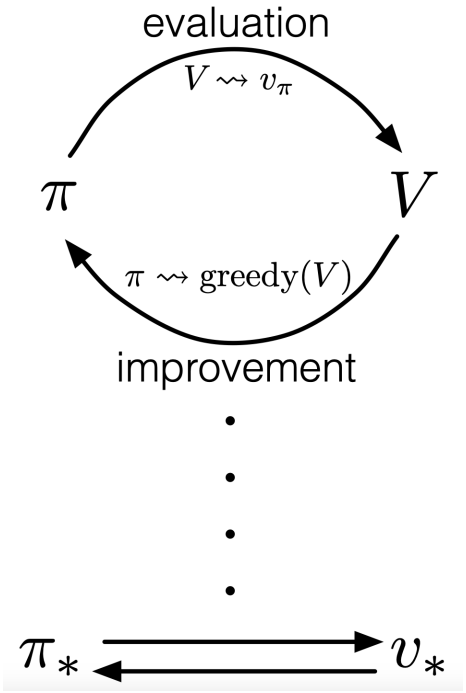


Figure 2: Policy Evaluation Paradigm

4.1 Dynamic Programming

Dynamic Programming è una tecnica che permette di risolvere problemi di ottimizzazione, suddividendo il problema in sotto-problemi più piccoli. Da non confondere col "Divide et Impera", che è una tecnica che permette di risolvere un problema suddividendolo in sotto-problemi più piccoli, ma senza che ci sia una relazione tra i vari sotto-problemi, mentre la Dynamic Programming permette di risolvere un problema suddividendolo in sotto-problemi più piccoli, ma con una relazione tra i vari sotto-problemi.

I MDP possono essere risolti con il Dynamic Programming, con la decomposizione delle equazioni di Bellman in modo ricorsivo.

In questa fase, supponiamo di conoscere l'intero MDP, e di avere a disposizione la Transition Matrix P, la Reward Function R, e il Discount Factor γ.

Algorithm 1 Valutazione della Policy tramite Dynamic Programming

Input: $S, A, R, P, \pi, \epsilon$
$V_0(s) = 0 \forall s \in S$
$k = 0$
while k == 0 or $||V_k - V_{k-1}|| > \epsilon$ **do**
    $V_{k+1}(s) = \sum_{a \in A} \pi(a|s) \cdot \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \cdot V_k(s')\right)$
    $k = k + 1$
end while
return V_k

Per aggiornare la Policy:

$$\pi(s) \leftarrow \arg\max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \cdot V_k(s')$$

Ovvero mettendo dentro arg max l'equazione di Bellman per $V_\pi(s)$, otteniamo la Policy π ottima.

4.2 Teorema - Policy Improvement

Data una value function V_π, calcoliamo la policy $\pi' = greedy(V_\pi)$ con $\pi' \neq \pi$. Calcolando la policy π', con l'algoritmo *greedy*, otteniamo una policy migliore di π, ovvero:

$$V_{\pi'}(s) > V_\pi(s)$$

Dimostrazione: Per **costruzione** la Q-function valutata usando la policy π' è maggiore uguale della Q-function valutata usando la policy π:

$$V_{\pi'}(s) = Q_\pi(s, \pi'(s)) = \max_{a \in A} Q_\pi(s, a) \geq Q_\pi(s, \pi(s)) = V_\pi(s)$$

Il caso in cui vale l'uguaglianza, è quello in cui siamo arrivati alla policy ottima π^*, ovvero: $V_\pi(s) = \max_{a \in A} Q_\pi(s, a)$, e quindi non possiamo migliorare ulteriormente la policy, per cui:

$$V_{\pi'}(s) = V_\pi(s) = V_{\pi'}^*(s)$$

Il che dimostra che se miglioriamo la policy con una politica greedy, otteniamo una policy migliore di quella iniziale fino ad arrivare alla policy ottima.

4.3 Vantaggi e limiti del Dynamic Programming

Il Dynamic Programming è un algoritmo che permette di risolvere i MDP, ma ha dei vantaggi e dei limiti:

Vantaggi:

- Riduce il costo di aggiornamento della Value Function attraverso **backup asincroni**, e nella stessa locazione di memoria (**in-place**): $V_{k+1}(s) = update(V_k(s))$
- **Prioritized**: Aggiorna prima gli stati più importanti, ovvero quelli che hanno il più alto errore sulle Bellman Optimality Equation
- **Real-time**: vengono considerati solo gli stati che sono stati visitati, e non tutti. Ma questo necessita di un'implementazione efficiente della Policy esplorativa π.
- **Early Stopping**: Non attente la convergenza

Limiti: Non è **scalabile** con il numero di stati e azioni, e quindi non è adatto per MDP con un numero elevato di stati e azioni. Infatti a ogni iterazione vengono considerati tutti gli stati e azioni, e quindi il costo computazionale cresce in modo esponenziale.

Per superare questo limite, esistono due soluzioni, basate sul campionamento:

- **Model-free**
- **Model-based**

5 Model-free Sample-based Approach

In un contesto di Model-free Learning, l'agente non conosce l'ambiente, ovvero non conosce la Transition Matrix P e la Reward Function R, per questo dovrà prima esplorarlo, e poi imparare da esso.

Sample-based perché non considero tutti gli episodi possibili, ma solo un sottoinsieme di essi.

Due tipi di algorithmi per fare Reinforcement Learning, in un contesto di Model-free Sample-based Approach (Ma non solo, possono essere usati anche in contesti diversi):

- **Monte Carlo Methods**
- **Temporal Difference Learning**

5.1 Monte Carlo Methods

Monte Carlo Methods è un algoritmo che permette di stimare la Value Function V_π o la Q-function Q_π, prendendo in considerazione un intero episodio (per episodio si intende una sequenza di stati, azioni e ricompense fino a raggiungere uno stato terminale). Questo metodo non ha baias nello stimare l'ambiente, dato che si usano tantissimi episodi e si vede quali sono i migliori. Soffre però di una varianza molto alta, per via del fatto che considera tutti gli episodi esplorati, e abbiamo una probabilità maggiore che qualche episodio sia molto diverso dagli altri in termini di ricompense.

Questo approccio è definito **offline learning**, ovvero l'agente impara da un insieme di episodi, e non in tempo reale.

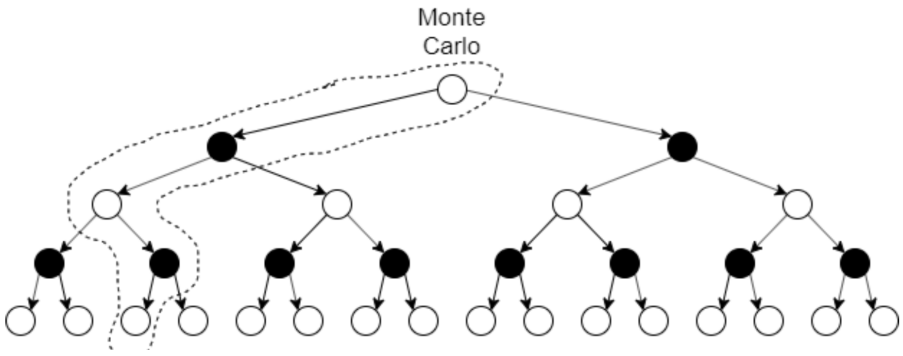


Figure 3: Monte Carlo Methods

Inizialmente l'agente **esplora** l'ambiente utilizzando la policy π, e raccoglie episodi, ovvero una sequenza di stati, azioni e ricompense: $s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots, s_T$. Colleziona tuple nel formato $< S, A, R, S' >$. Un'insime di queste tuple formano un episodio.

Bisogna poi calcolare la Value Function V_π o la Q-function Q_π, utilizzando gli episodi raccolti:

$$V_\pi(s) = \mathbb{E}[G_t | s_t = s]$$

$$G_t^T = \sum_{k=t}^{T-1} \gamma^{k-t} \cdot r_{k+1}$$

Dove T è il massimo numero di azioni possibili, se non si raggiunge uno stato terminale prima della fine di un episodio.

Per valutare la Policy π in Monte Carlo, posso utilizzare due tipi di implementazioni:

- **First-Encounter**: Calcola V_π considerando solo la prima volta che si visita uno stato.
- **Every-Encounter**: Calcola V_π considerando ogni volta che si visita uno stato.

Algorithm 2 Every-Encounter Monte Carlo Policy Evaluation

Input: E
$N(s) = 0, \forall s \in S$ // Numero di volte che si visita uno stato
$G(s) = 0, \forall s \in S$ // Valore approssimato per uno stato
$V(s) = 0, \forall s \in S$ // Valore approssimato della Value Function
for each episode $e \in E$ **do**
  for each step $t = 1, ..., T-1$ **do**
    $N(s_t) = N(s_t) + 1$
    $G(s_t) = G(s_t) + G_t$
  end for
end for
$V(s_i) = G(s_i)/N(s_i)$
return V

Da notare che:

$$\lim_{N(s)\to\infty} V(s) = V_\pi(s)$$

Viene usata la tecnica di **Incremental Mean**, che permette di calcolare la media di un insieme di valori, aggiungendo un valore alla volta:

$$\mu_k = \frac{1}{k}\sum_{i=1}^{k} x_i = \mu_{k-1} + \frac{1}{k}\left(x_k - \mu_{k-1}\right)$$

Così facendo non abbiamo bisogno si memorizzare tutti i valori di $G(s)$, e possiamo aggiornare il valore di $V(s)$ dopo ogni episodio, col risultato di memoria risparmiata.

Utilizzando l' **Incremental Mean**, possiamo calcolare la Value Function V_π come:

$$V(s_t) = V(s_t) + \frac{1}{N(s_t)}\left(G_t - V(s_t)\right)$$

Nel caso di MDP non stazionari, abbiamo bisogno di dimendicare i vecchi episodi, e dare più peso ai nuovi, perchè l'ambiente è in continua evoluzione, e ciò che si esplorato nel passato potrebbe non essere più valido. Questo viene fatto attraverso un *exponential decay* α:

$$V(s_t) = V(s_t) + \alpha\left(G_t - V(s_t)\right), \text{ con } \alpha \in [0, 1]$$

5.2 Temporal Difference Learning

Temporal Difference Learning è un algoritmo che permette di stimare la Value Function V_π o la Q-function Q_π, prendendo in considerazione un singolo passo. Questo metodo ha una varianza più bassa rispetto al Monte Carlo Methods, perché considera un singolo passo alla volta. Questo però introduce un bias, dovuto al fatto che non può avere la precisione del Monte Carlo Methods, il quale considera tutti gli episodi disponibili.

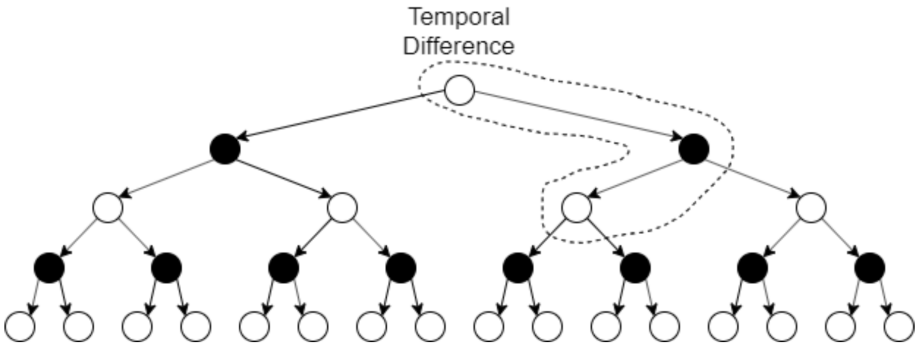


Figure 4: Temporal Difference Learning

Inizialmente l'agente **esplora** l'ambiente utilizzando la policy π, e raccoglie episodi, ovvero tuple nel formato $< S, A, R, S' >$. Si parla in questo caso di **online learning**, ovvero l'agente impara in tempo reale, e non da un insieme di episodi.

Incremental Learnig: l'agente aggiorna la Value Function V_π o la Q-function Q_π dopo ogni passo. Non c'è bisogno di assumere che gli episodi terminano in uno stato terminale, e quindi non c'è bisogno di considerare il massimo numero di azioni possibili.

Per calcolare la Value Function V_π si utilizza la seguente formula:

$$V(s_t) = V(s_t) + \alpha \left(G_t - V(s_t)\right)$$

$$G_t = r_{t+1} + \gamma V(s_{t+1})$$

Dove α è il **learning rate**, che permette di dare più peso ai nuovi valori, rispetto ai vecchi, mentre γ è il **discount factor**, che permette di evitare che la somma delle ricompense sia infinita, e quindi evitare i loop nel MDP.

Questa particolare scelta di G_t etichetta l'algoritmo come **TD(0)**, ovvero Temporal Difference con un passo.

Nota: Temporal Difference converge più velocemente del Monte Carlo Methods, ma ha il bias di cambiare la Value Function a seconda delle tuple che incontra.

Esempio - MC vs TD: Nell'esempio in figura 5 abbiamo un semplice MDP con due stati A e B, e due stati finali interpretati dai quadrati grigi. Le tuple del dataset sono quelle indicate a fianco, del tipo $< S, A, R, S' >$.

Nel caso di Monte Carlo, non importa in quale ordine vengono usate le tuple, perchè considera tutti gli episodi possibili, e quindi non avrà alcun bias. Alla fine il risultato sarà sempre lo stesso. Usando le formule di Monte Carlo, otteniamo che $V(A) = 0$ perchè passo una sola volta per A e ottengo reward

nullo, mentre $V(B) = 0.75$ perchè passo 6 volte per B e ottenendo reward 1, e 2 volte ottenendo reward 0, quindi $V(B) = \frac{6}{8} = 0.75$.

Nel caso di Temporal Difference, l'ordine delle tuple è importante, perchè considera un solo passo alla volta, e quindi il risultato finale sarà diverso a seconda dell'ordine delle tuple. Se seguo l'ordine dell'esempio in figura 5, otterrò sempre che $V(A) = 0$, ma se mescolo le tuple, otterò un risultato diverso (Es: $V(B) = 0.75, V(A) = 0.99 \cdot 0.75 = 0.74$).

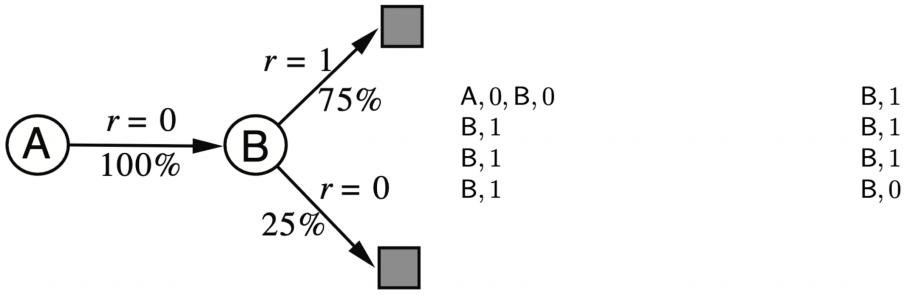


Figure 5: MC vs TD

5.3 Monte Carlo vs Temporal Difference

Monte Carlo

- Converge al minimo errore quadratico, perchè considera tutti gli episodi possibili ma ha una varianza molto alta.

$$V_\pi = \arg\min_V \sum_{e=1}^{E} \sum_{t=1}^{T_e - 1} (G_r - V(s_t))^2$$

- Tende a adattarsi meglio ai Returns osservati.
- **Non rispetta** la Markov Property.

Temporal Difference

- Converge più velocemente alla soluzione più probabile (maximum likelihood), ma ha un bias.
- È più preciso perchè considera un solo passo alla volta, e quindi ha una varianza più bassa.
- Rispetta la Markov Property.

$$\hat{P}^a_{ss'} = \frac{1}{N(s,a)} \sum_{e=1}^{E} \sum_{t=1}^{T_e - 1} \mathbb{1}(s_t^{(e)} = s, a_t^{(e)} = a, s_{t+1}^{(e)} = s')$$

$$\hat{R}^a_s = \frac{1}{N(s,a)} \sum_{e=1}^{E} \sum_{t=1}^{T_e - 1} \mathbb{1}(s_t^{(e)} = s, a_t^{(e)} = a) \cdot r_t^{(e)}$$

dove $\mathbb{1}$ è la funzione che restituisce 1 se la condizione al suo interno è vera, 0 altrimenti.

5.4 TD(n) e TD(λ)

L'idea di $TD(n)$ è quella di considerare gli n passi successivi, e non solo il passo successivo, per stimare la Value Function V_π.

$$V(s_t) = V(s_t) + \alpha \left(G_t^{(n)} - V(s_t) \right)$$

$$G_t^{(n)} = \sum_{k=1}^{n} \gamma^k r_{t+k} + \gamma^{n+1} V(s_{t+n+1})$$

Mentre in $TD(\lambda)$, l'idea è quella di considerare la media pesata delle $G_t^{(n)}$ (Quella descritta sopra per $TD(n)$):

$$V(s_t) = V(s_t) + \alpha \left(G_t^{(\lambda)} - V(s_t) \right)$$

$$G_t^{(\lambda)} = (1 - \lambda) \sum_{n=1}^{N} \lambda^{n-1} G_t^{(n)}$$

dove N è grande a piacimento, e $\lambda \in [0, 1]$, $\sum_{n=1}^{\infty} (1 - \lambda)\lambda^{n-1} = 1$.

Entrambe le $G_t^{(n)}$ e $G_t^{(\lambda)}$ possono essere aggiornate in modo incrementale. Entrambi i due approcci sono **offline learning**, infatti imparano da un insieme di episodi, e non in tempo reale.

5.5 Eligibility Traces and Backward TD(λ)

L'obiettivo è implementare questo algoritmo utilizzando episodi parziali, ovvero segmenti di esperienze di apprendimento anziché episodi completi. Per farlo si utilizzano l'**Eligibility Traces**. Essa è una funzione $E : S \to \mathbb{R}^+$, che assegna un valore positivo ad ogni stato utile per pesare quanto un particolare stato è "responsabile" delle azioni future compiute dall'agente.

Uno dei **problemi** centrali nel Reinforcement Learnig è determinare in che misura uno stato o un'azione hanno contribuito a un risultato che si verifica molto tempo dopo. Ad esempio, se un agente vince una partita, bisogna capire quali azioni durante la partita sono state determinanti per quella vittoria, e la

funzione di Eligibility Traces permette di fare proprio questo, misura quanto lontano nel futuro uno stato rimane importante:

$$E_0(s) = 0$$

$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbb{1}(s = s_t)$$

Per quanto riguarda l'aggiornamento della Value Function $V(s)$, si utilizza la seguente formula:

$$V(s) = V(s) + \alpha\delta_t E_t(s)$$

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Dove appunto si da meno importanza ai valori più vecchi, e più importanza ai valori più recenti.

Esempio: Nell'esesempio in figura 6, possiamo vedere come l'Eligibility evolve nel tempo, e come la Value Function $V(s)$ viene aggiornata in base all'Eligibility. Supponiamo di partire con:
$V(s) = 0$, $\gamma = 1$, $\lambda = 0.5$, $\alpha = 0.99$, $r_{t+1} = 1$, $E_0(s) = 0$

$V(S)$		$E(S)$	δ	$V(S)$
$\begin{bmatrix}0.00\\0.00\\0.00\\0.00\end{bmatrix}$	$s_1 \to s_2$	$\begin{bmatrix}1.00\\0.00\\0.00\\0.00\end{bmatrix}$	1.00	$\begin{bmatrix}0.99\\0.00\\0.00\\0.00\end{bmatrix}$
$\begin{bmatrix}0.99\\0.00\\0.00\\0.00\end{bmatrix}$	$s_2 \to s_3$	$\begin{bmatrix}0.50\\1.00\\0.00\\0.00\end{bmatrix}$	1.00	$\begin{bmatrix}1.48\\0.99\\0.00\\0.00\end{bmatrix}$
$\begin{bmatrix}1.48\\0.99\\0.00\\0.00\end{bmatrix}$	$s_3 \to s_1$	$\begin{bmatrix}0.25\\0.50\\1.00\\0.00\end{bmatrix}$	2.48	$\begin{bmatrix}2.09\\1.60\\0.61\\0.00\end{bmatrix}$
$\begin{bmatrix}2.09\\1.60\\0.61\\0.00\end{bmatrix}$	$s_1 \to s_4$	$\begin{bmatrix}1.13\\0.25\\0.50\\1.00\end{bmatrix}$	-1.09	$\begin{bmatrix}0.87\\1.33\\0.09\\-1.08\end{bmatrix}$

Figure 6: Model-Free On-Policy Optimization Esempio

6 Model-Free on-policy Optimization: ϵ-greedy

L'obiettivo è quello di trovare la policy ottima π^*, partendo da una policy casuale π, e migliorandola iterativamente. Per farlo si utilizzano due tipi di approcci:

- **On-policy**: si migliora la policy π iterativamente, e si esplora l'ambiente utilizzando la policy π stessa.

- **Off-policy**: si migliora la policy π iterativamente, ma si esplora l'ambiente utilizzando una policy diversa da μ.

La prima idea per un approccio on-policy è la **Generalized Policy Iteration** (GPI), che si basa sull'idea di alternare fra le seguenti fasi:

1. **Policy Evaluation**: Calcolare la Value Function V_π o la Q-function Q_π, utilizzando la policy π: $\pi_k \to V_{\pi_k}$
2. **Policy Improvement**: Migliorare la policy π utilizzando la Value Function V_π o la Q-function Q_π: $\pi_{k+1} \leftarrow greedy(V_{\pi_k})$
3. $k = k + 1$ e si ripete il processo.

Problema 1: Vogliamo migliorere la policy π in modo greedy:

$$\pi_{k+1}(a|s) = \arg\max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \cdot V_{\pi_k}(s')$$

Ma non conosciamo l'ambiente, e di conseguenza i valori di R_s^a e $P_{ss'}^a$. Per questo si utilizza il Q-Learnig, che permette di stimare la Q-function Q_π, e di migliorare la policy π in modo greedy:

$$\pi_{k+1}(a|s) = \arg\max_{a \in A} Q_{\pi_k}(s, a)$$

.

Problema 2: La prima policy π_0 è casuale, e quindi esplora l'ambiente in modo casuale, non prendendo decisioni ottimali. Le policy successive, quelle π_k con $k > 0$ sono deterministiche, nel senso che si basano su un approccio ben definito, per esempio l'aproccio greedy. Questo porta al problema che le policy deterministiche non faranno più esplorazione, ma preferiranno sempre seguire le azioni che, per quel che hanno potuto vedere, sono le migliori, anche perchè non è detto che la miglior azione al passo t mi garantisca che al passo $t + 1$ abbia un reward migliore. Di conseguenza non convergiamo alla policy ottima π^*.

6.1 ϵ-greedy

Per superare questo problema, si utilizza la tecnica **ϵ-greedy**, che permette di esplorare l'ambiente con probabilità ϵ, e di sfruttare la policy π con probabilità $1 - \epsilon$:

$$\pi_k(a|s) = \begin{cases} \frac{\epsilon}{m} + (1 - \epsilon) & \text{se } a = \arg\max_{a \in A} Q_\pi(s, a) \\ \frac{\epsilon}{m} & \text{altrimenti} \end{cases} \tag{2}$$

Con m è il numero di azioni fatte dall'agente per arrivare allo stato s.
La convergenza è ancora garantita? Si, c'è una preposizione ci aiuta.

Proposizione: miglioramento della policy π con ϵ-greedy. L'aggiornamento della policy π_k con ϵ-greedy ci porta a una policy migliore π_{k+1} tale per cui $V_{\pi_{k+1}}(s) > V_{\pi_k}(s)$.

Dimostrazione: Per dimostrarlo, consideriamo $Q_{\pi_k}(s, \pi_{k+1}(s))$, stiamo valutando la Q-function Q_{π_k} nello stato s e considerando l'azione suggerita dalla nuova policy π_{k+1}. Per determinare il valore atteso di questa Q-function, dobbiamo prendere in considerazione tutte le possibili azioni a che la nuova policy π_{k+1} potrebbe suggerire nello stato s e pesare ciascuna azione con la probabilità che la policy π_{k+1} assegna a quella azione. Questo ci porta alla seguente espressione:

$$Q_{\pi_k}(s, \pi_{k+1}(s)) = \sum_{a \in A} \pi_{k+1}(a|s) \cdot Q_{\pi_k}(s, a)$$

Isoliamo fuori dalla sommatoria l'azione $a^* = \arg\max_{a \in A} Q_{\pi_k}(s, a)$, e indichiamo con $Q(s, a^*)$ il valore massimo per la Q-function:

$$Q_{\pi_k}(s, a) = \sum_{a \in A - a^*} \pi_{k+1}(a|s) \cdot Q_{\pi_k}(s, a) + \pi_{k+1}(a^*|s) \cdot Q_{\pi_k}(s, a^*)$$

Sostituendo la definizione di π_k con ϵ-greedy (quella indicata dal sistema di sopra), otteniamo:

$$Q_{\pi_k}(s, a) = \frac{\epsilon}{m} \sum_{a \in A - a^*} Q_{\pi_k}(s, a) + \left[\frac{\epsilon}{m} + (1 - \epsilon)\right] \cdot Q_{\pi_k}(s, a)$$

$$Q_{\pi_k}(s, a) = \frac{\epsilon}{m} \sum_{a \in A - a^*} Q_{\pi_k}(s, a) + \frac{\epsilon}{m} \cdot Q_{\pi_k}(s, a^*) + (1 - \epsilon) \cdot Q_{\pi_k}(s, a^*)$$

$$Q_{\pi_k}(s, a) = \frac{\epsilon}{m} \sum_{a \in A} Q_{\pi_k}(s, a) + (1 - \epsilon) \cdot Q_{\pi_k}(s, a^*)$$

Possiamo maggiorare quest'ultima espressione, dividendo e moltiplicando per $\frac{\pi_k(a|s) - \frac{\epsilon}{m}}{1 - \epsilon}$:

$$\geq \frac{\epsilon}{m} \sum_{a \in A} Q_{\pi_k}(s, a) + (1 - \epsilon) \frac{\pi_k(a|s) - \frac{\epsilon}{m}}{1 - \epsilon} \cdot Q_{\pi_k}(s, a)$$

Da notare che l'espressione $\frac{\pi_k(a|s) - \frac{\epsilon}{m}}{1 - \epsilon} \rightarrow \frac{\frac{\epsilon}{m} + (1 - \epsilon) - \frac{\epsilon}{m}}{1 - \epsilon} \rightarrow 1$ se sostiutiamo la policy π_k con l'ottimo per l'ϵ-greedy, e quindi vale l'ugualianza

Il nostro obiettivo è quelllo di dimostrare che prendendo l'azione a^*, otteniamo un valore maggiore di $Q_{\pi_k}(s, a)$, rispetto al prendere una qualsiasi altra azione non ottima. Se invece sostituiamo la $\pi(a|s) = \frac{\epsilon}{m}$, ovvero quando non è ottima, otteniamo:

$$= \frac{\epsilon}{m} \sum_{a \in A} Q_{\pi_k}(s, a) + (1 - \epsilon) \cdot \frac{\frac{\epsilon}{m} - \frac{\epsilon}{m}}{1 - \epsilon} \cdot Q_{\pi_k}(s, a^*)$$

$$= \frac{\epsilon}{m} \sum_{a \in A} Q_{\pi_k}(s,a) + 0 \cdot Q_{\pi_k}(s,a^*)$$

Ma questa non è altro che la definizione di $V_{\pi_k}(s)$, risostituendo $\pi_k(a|s) = \frac{\epsilon}{m}$, ovvero la Value Function per la policy π_k:

$$= \sum_{a \in A} \pi_k(a|s) \cdot Q_{\pi_k}(s,a) = V_{\pi_k}(s)$$

Che per il Teorema di Policy Improvement (Sia $\pi' := greedy(V_\pi)$ con $\pi' \neq \pi$ allora $V_{\pi'} > V_\pi$), sappiamo che $V_{\pi_{k+1}} > V_{\pi_k}$, e quindi la tesi è dimostrata.

Problema: Le Optimality Bellman Equation hanno bisogno di una policy deterministica, e non di una policy stocastica, come quella ottenuta con ϵ-greedy. Ma la convergenza all'ottimalità è garantita, grazie alla definizione di **Greedy in the Limit with Infinite Exploration** (GLIE) $\rightarrow$ una ϵ-greedy policy è definita GLIE se e solo se:

1. $\lim_{k \to \infty} N_k(s,a) = \infty$ per ogni stato s e azione a esplorete un numero infinito di volte.
2. $\lim_{k \to \infty} \pi_k(a|s) = \mathbb{1}(a = \arg\max_{a' \in A} Q_{\pi_k}(s,a'))$. Ovvero che la policy converge alla policy greedy (la policy greedy è quella che assegna probabilità 1 all'azione che massimizza $Q(a,s)$).
3. $\epsilon_k \in \mathcal{O}(\frac{1}{k})$ con ϵ_k che è la probabilità di esplorazione al passo k.

Tutti gli algoritmi visti fin ora (MC, TD, TD(λ), ϵ-greedy) rispettano la proprietà GLIE.

MC: Nell'algorimto di MC abbiamo che le iterazioni di ottimizzazione della policy vengono fatte:

- $N_{k+1}(s,a) = N_k(s,a) + 1$
- $Q_{k+1}(s,a) = Q_k(s,a) + \frac{1}{N_{k+1}(s,a)} (G_t - Q_k(s,a))$
- $\epsilon \leftarrow \frac{1}{k}$
- $\pi_{k+1}(a|s) = \epsilon/m + (1-\epsilon) \cdot \mathbb{1}(a = \arg\max_{a' \in A} Q_k(s,a'))$

TD: Nell'algoritmo di TD abbiamo diverse varianti:

- **SARSA**: State-Action-Reward-State-Action. L'idea è la stessa del semplice TD, ma si stima la Q-function. I campioni di un episodio sono nel formato $< S, A, R, S', A' >$. La regola di aggiornamento è:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma Q(s',a') - Q(s,a)\right)$$

  Con la condizione Robbins-Monro:

$$\sum_{k=1}^{\infty} \alpha_k = \infty \quad \text{e} \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty$$

- **Expected SARSA**: Considera l'algoritmo di apprendimento che è proprio come il Q-learning tranne per il fatto che invece del massimo per le successive coppie stato-azione utilizza il valore atteso, tenendo conto della probabilità di ciascuna azione per la policy corrente:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \mathbb{E}_{\pi}[Q(s',a')|s'] - Q(s,a)\right) =$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \sum_{a' \in A} \pi(a'|s') \cdot Q(s',a') - Q(s,a)\right)$$

  Expected SARSA è più costoso computazionalmente rispetto a SARSA, ma soffre meno la varianza grazie alla scelta randomica di a'. Queto tipo di algoritmo è complesso da esegure quando il numero di azioni da considerare è molto alto (per via della sommatoria sulle azioni).

- **SARSA(n)**: Utilizza la tecnica di n-step. L'idea è quella di considerare n passi successivi, e non solo il passo successivo.

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha \left(G_{t \to t+n} - Q_t(s,a)\right)$$

$$G_{t \to t+n} = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n Q_t(s_n, a_n)$$

- **SARSA(λ)**:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma Q_t^{(\lambda)} - Q(s,a)\right)$$

$$Q_t^{(\lambda)} = (1-\lambda) \sum_{n=1}^{N} \lambda^{n-1} Q_t^{(n)}$$

  Con N grande a piacimento, e $\lambda \in [0,1]$: $\sum_{n=1}^{\infty} (1-\lambda)\lambda^{n-1} = 1$.

- **Backward (Expected) SARSA(λ)**:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \delta_t E_t(s,a)$$

$$\delta_t = r_{t+1} + \gamma \sum_{a' \in A} \pi(a'|s') \cdot Q(s',a') - Q(s,a)$$

$$E_t(s,a) = \gamma \lambda E_{t-1}(s,a) + \mathbb{1}(s = s_t, a = a_t)$$

  con $E_0(s,a) = 0$.

6.2 Off-policy Learning

L'idea è quella di esplorare l'ambiente utilizzando una policy μ, e di migliorare la policy π in modo greedy. Questo è importante perchè:

- **Imparare per imitazione:** In molti casi, possiamo avere accesso a un esperto umano o a una policy predefinita che esegue molto bene in un determinato ambiente. L'off-policy learning permette di apprendere una policy imitativa da questi esempi, anche se la policy corrente dell'agente è differente.
- **Riutilizzo delle esperienze:** Molte esperienze possono essere raccolte mentre l'agente segue diverse policy nel tempo. L'off-policy learning consente di riutilizzare queste esperienze passate per migliorare la policy corrente, invece di dover generare nuovi dati ogni volta che la policy cambia.
- **Esplorazione e sfruttamento:** Durante l'apprendimento, è spesso utile seguire una policy che esplora l'ambiente (ad esempio, una policy ϵ-greedy). L'off-policy learning permette di utilizzare i dati raccolti da questa policy esplorativa per apprendere una policy più sfruttativa e ottimale.
- **Multi-policy learning:** È possibile apprendere più policy contemporaneamente utilizzando i dati raccolti da una singola policy di comportamento. Questo è utile, ad esempio, per apprendere sia una policy ottimale che una policy sicura in parallelo.
- **Evitare test nel mondo reale:** Nel mondo reale, generare nuove esperienze può essere molto costoso e talvolta pericoloso (ad esempio, nei robot o nei veicoli autonomi). L'off-policy learning permette di apprendere policy migliori senza dover eseguire continuamente nuove esplorazioni nell'ambiente reale.

Importance Sampling L'Importance Sampling consente di stimare le aspettative di una funzione (come la funzione valore) rispetto a una distribuzione target utilizzando campioni raccolti da una distribuzione diversa (quella della policy di comportamento). Questo è cruciale per poter riutilizzare esperienze generate da policy diverse senza introdurre errori di stima significativi.

Formalmente, l'Importance Sampling è definito come:

$$\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x)f(x) = \sum_x Q(x)\frac{P(x)}{Q(x)}f(x) = \mathbb{E}_{x\sim Q}\left[\frac{P(x)}{Q(x)}f(x)\right]$$

Dove P è la distribuzione target, Q è la distribuzione di comportamento, f è la funzione da stimare, e x è la variabile casuale.

6.3 Off-policy Monte Carlo

L'idea è usare due policy: una policy target π e una policy di comportamento μ. La policy target è quella che vogliamo migliorare, mentre la policy di comportamento è quella che esplora l'ambiente. Dato che abbiamo due policy, dobbiamo

considerare due tipi di Return: G_t^π e G_t^μ, che sono rispettivamente il Return per la policy target e il Return per la policy di comportamento. Per calcolare il Return G_t^π:

$$G_t^\pi = \prod_{k=t}^{T} \frac{\pi(a_k|s_k)}{\mu(a_k|s_k)} \cdot G_t^\mu$$

Usiamo poi la solita update rule per aggiornare la Q-function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(G_t^\pi - Q(s_t, a_t)\right)$$

Problema: l'Importance Sampling fa drammaticamente aumentare la varianza perchè se $\pi(a_k|s_k) \approx 1$ e $\mu(a_k|s_k) \approx 0$, allora il loro rapporto sarà molto grande. Questo fa eslpodere la varianza del Return G_t^π. Per questo motivo è necessario che π e μ siano il più simili possibile.

6.4 Off-policy Temporal Difference Learning

L'idea è la stessa di prima, ma si utilizza la tecnica di Temporal Difference Learning. L'update rule per la Q-function è la seguente:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[\frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t)\right]$$

Dove, questa volta, non abbiamo la produttoira di tutti i rapporti delle probabilità, ma solo il rapporto della probabilità dell'azione scelta dalla policy target e la probabilità dell'azione scelta dalla policy di comportamento. Dato che ho solo un caso da gestire, la varianza è più sotto controllo rispetto all'Off-policy di Monte Carlo, ma il bias è più alto.

6.5 Off-policy - Q-Learning

Nel Q-Learning, l'idea è quella di stimare la Q-function Q_π, e di migliorare la policy π in modo greedy. Per farlo vengono scelte due policy: una policy target π e una policy di comportamento μ:

- **Policy target**: $\pi(s) = \arg\max_{a \in A} Q(s, a)$ che è la policy greedy
- **Policy di comportamento**: $\mu(s) = \epsilon - \text{greedy}(Q(s, a))$

In questo caso, non facciamo uso dell'Importance Sampling, perchè la policy target non è più inferita dalla policy di comportamento. L'update rule per il Q-Learning è la seguente:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1}^\mu + \gamma Q(s_{t+1}, a_{t+1}^\pi) - Q(s_t, a_t)\right)$$

dove r_{t+1}^μ è la ricompensa ottenuta dall'azione a_t nello stato s_t secondo la policy di comportamento μ, e a_{t+1}^π è l'azione suggerita dalla policy target π nello stato s_{t+1}.

Semplificando il learning target, otteniamo:

$$r^{\mu}_{t+1} + \gamma Q(s_{t+1}, a^{\pi}_{t+1}) = r^{\mu}_{t+1} + \gamma Q(s_{t+1}, \arg\max_{a' \in A} Q(s_{t+1}, a'))$$

$$= r^{\mu}_{t+1} + \gamma \max_{a \in A} Q(s_{t+1}, a)$$

7 Value Function Approximation

Fin ora abbiamo visto come funzionano i metodi tabulari, ossia quando abbiamo un numero discreto di stati e azioni. Ma cosa succede quando abbiamo un numero continuo di stati e azioni? Introduciamo la continuità nello spazio degli stati e delle azioni, anche per affrontare il caso in cui abbiamo due stati molto simili, ma non uguali, che però dovrebbero avere comportamenti simili.

Si parla di **State Encoding**, che permette di codificare gli stati, per cui stati simili saranno codificati allo stesso modo, così che, se arriviamo in uno stato mai visto, ma che riusciamo ad associare a qualcosa si simile già esplorato, possiamo comportarci come ci saremmo comportati nel caso conosciuto.

In questo caso, dobbiamo utilizzare la **Value Function Approximation** o la **Q-function Approximation**, che permettono di approssimare V_π o Q_π utilizzando una funzione parametrica:

$$\hat{V} : S \times \Theta \to \mathbb{R}$$

$$\hat{Q} : S \times \Theta \to \mathbb{R}^{|A|}$$

Sono difatti Rete Neurali, che permettono di approssimare la Value Function o la Q-function. In particolare la Q-function è definita così (e non $Q : S \times A \times \Theta \to \mathbb{R}$) perchè altrimenti dovremmo usare la NN che la approssima per ogni azione, invece abbiamo che in usicta a questa NN abbiamo un valore per ogni azione possibile (un vettore in output).

Usando una Rete Neurale, abbiamo la possibilità di predirre il valore di V_π o Q_π per uno stato s e una azione a qualsiasi, anche mai visti prima.

Problema 1: i dati non sono stazionari La policy π cambia nel tempo, e anche l'ambiente potrebbe cambiare.

Problema 2: i dati non sono indipendenti e identicamente distribuiti (i.i.d), quindi è necessario fare un campionamento per superare questo problema.

7.1 Incremental Learning

Possiamo utilizzare l'approccio di **Incremental Learning**, che permette di aggiornare la Value Function o la Q-function dopo ogni passo, attraverso l'utilizzo di una loss function.

Un approccio possibile è quello di utilizzare lo **Stochastic Gradient Descent** (SGD), minimizzando il Mean Squared Error (MSE):

$$\mathcal{L}(\theta) = \mathbb{E}\left[(V_\pi(s) - \hat{V}(s,\theta))^2\right]$$

dove θ sono i parametri della Rete Neurale che approssima la Value Function, e $\hat{V}(s,\theta)$ è la Value Function approssimata, e $V_\pi(s)$ è la Value Function reale, che però non conosciamo a priori.

Linear Value Approximation: Approssimiamo la Value Function con una funzione lineare: $\hat{V}(s, \theta) = \Phi(s)^T \theta$, dove $\Phi(s)$ è il vettore che codifica lo spazio degli stati, e θ è il vettore dei pesi.

Questa approssimazione lineare è sufficente, questo è garantito dal Representer Theorem, che afferma che la soluzione ottima per un problema di approssimazione della Value Function è una combinazione lineare delle feature.

Poiché stiamo utilizzando una approssimazione lineare, la funzione di perdita $L(\theta)$ diventa una funzione quadratica rispetto ai parametri θ. Questo è vantaggioso perché le funzioni quadratiche sono più facili da minimizzare utilizzando tecniche di ottimizzazione come il Gradient Descent.

Il gradiente della loss rispetto ai parametri θ è dato da:

$$\nabla_\theta \mathcal{L}(\theta) = \nu(V_\pi(s) - \hat{V}(s, \theta))\phi(s)$$

Usando lo Stocastic Gradient Descent, aggiorniamo i parametri θ in direzione del gradiente per ridurre la perdita.

Problema: Ancora non conosciamo la Value Function reale $V_\pi(s)$.

Per superare questo problema, possiamo rimpiazzarla con il Return G_t, calcolato empiricamente. Questo ci porta a distinguere fra G_t per il Monte Carlo e per il Temporal Difference:

- **Monte Carlo**: G_t è non biased, e bisogna fare apprendimento supervisionato sui campioni (s_t, G_t). Abbiamo però una convergenza migliore.
- **Temporal Difference**: G_t è biased, e bisogna fare apprendimento supervisionato sui campioni $(s_t, r_{t+1} + \gamma \hat{V}(s_{t+1}, \theta))$. Abbiamo però una convergenza più veloce ma meno precisa.
- **TD(λ)**: TD(λ) estende TD(0) aggiungendo i trace di eleggibilità per includere una parte delle ricompense future. Utilizza una media ponderata di più aggiornamenti TD con differenti intervalli temporali.

  $$E_t = \gamma \lambda E_{t-1} + x(s_t)$$

  Dove $x(s_t)$ è una funzione che dipende dallo stato s_t, questo perché abbiamo bisogno di aumentare l'eleggibilità per gli stati che sono stati visitati più di recente. Bisogna però considerare anche gli stati vicini a s_t, perchè dovrebbero comportarsi in modo simile. La funzione $x(s_t)$ da un valore allo stato rispetto a quando siamo vicini allo stato s_t, quindi è massimima quando siamo in s_t.

  L'Eligibility da una sovrapposizione delle codifiche degli stati in questo contesto di approssimazione.

7.2 Batch Learning

Un altro approccio per l'approssimazione della Value Function è il **Batch Learning**, che permette di aggiornare la Value Function utilizzando un insieme di

episodi raccolti in precedenza, appunto un batch di episodi. Questo ci permette di avere dei gradienti più stabili. Così facendo risolviamo anche il problema della non stazionarietà dei dati, perchè i dati venono raccolti e salvati in un database, e poi vengono campionati in batch. Questo fa si che Non-IID non sia più un problema, perchè i dati sono raccolti in modo casuale.

In ogni caso, non imapro direttamente da ciò che sperimento, ma prima di tutto raccolgo i dati, poi faccio campionamento, e poi apprendo.

7.3 DNN based Approach - Deep Q Network (DQN)

Vogliamo approssimare la Q-function con una Deep Neural Network, che permette di approssimare funzioni complesse, e cerca di prevedere i valori di $Q(s, a)$ per ogni coppia (s, a). Questa rete non solo apprende la relazione tra stati e valori Q, ma anche la corretta codifica degli stati.

Viene utilizzato un buffer di replay, che permette di memorizzare i dati raccolti in precedenza, e di fare campionamento in modo casuale.

Viene poi utilizzata una Q-target, usata per stabilizzare l'apprendimento, ovvero una copia della Q-function che viene aggiornata meno frequentemente rispetto alla Q-function principale.

Il problema principale nell'uso di un DQN è l'instabilità nel training nel momento in cui, per calcolare la loss mi serve usare la Q attuale per aggiornare la Q attuale. Per evitare ciò, si utilizza una Q-target, ovvero la Q-function dei passi precedenti (Quella con i parametri θ^-). La **Simple DQN Loss** è la seguente:

$$\mathcal{L}(\theta) = \mathbb{E}_D \left[(r + \gamma \max_{a'} \hat{Q}(s', a'; \theta^-) - Q(s, a; \theta))^2 \right]$$

In S-DQN l'idea principale è utilizzare la Q-target per stabilizzare l'apprendimento, e per evitare l'instabilità del training. Si utilizza un ground truth dipendente da ciò che la rete conosceva nel passato, e non quello che conosce ora.

Ma se nel passato la rete sbagliava? In questo caso si parla di maximum bias, perchè la rete approssima il massimo valore di Q, e quindi se sbaglia continua a sbagliare, e non riesce a correggersi. Nell'esempio in figura 7, possiamo vedere come, se durante la prima esplorazione la rete va verso sinistra, e ha la fortuna di ricevere un reward positivo (dato dalla distribuzione Normale $\mathcal{N}(-0.1, 1)$), allora sarà incentivata ad andare a sinistra, anche se non è la scelta migliore.

Per risolvere questo problema, si utilizza la **Double DQN**, che permette di ridurre il maximum bias.

Mentre la **Double DQN Loss** è la seguente:

$$\mathcal{L}(\theta) = \mathbb{E}_D \left[(r + \gamma \hat{Q}(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-) - Q(s, a; \theta))^2 \right]$$

In questo caso viene usato il Q-target con i parametri θ^-, ma dipendente da uno stato che massimizza l'attuale Q-function, quella con i parametri θ.

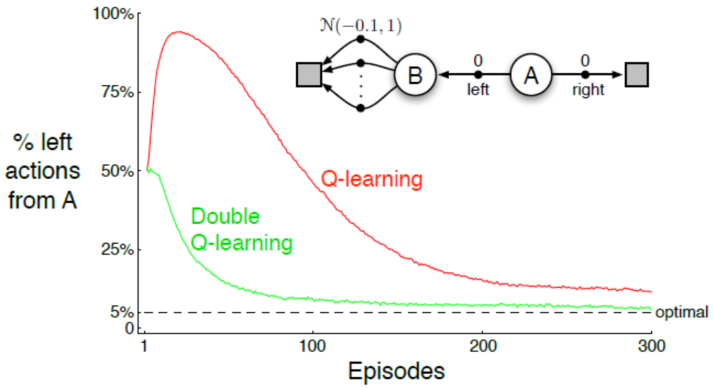


Figure 7: Maximum Bias

Nella figura si può vedere come nel contesto del Simple DQN, la rete all'inizio tende a sbagliare e andare a sinistra molte più volte di quanto fa nel caso del Double DQN. Questo porta il Double DQN a convergere più velocemente e in modo più preciso rispetto al Simple DQN.

Duelling DQN: Un'altra tecnica per migliorare il DQN è il Duelling DQN, nella quale la Q-function viene divisa in due parti:

- **Value Function**: $\hat{V}(s, \theta_V)$ che approssima il valore del reward atteso per uno stato specifico.
- **Advantage Function**: $\hat{A}(a, s, \theta_A)$ che stima il vantaggio di scegliere l'azione a quando si è nello stato s.

In formule abbiamo che:

$$Q(s, a, \theta) = \hat{V}(s, \theta_V) + \left(\hat{A}(s, a, \theta_A) - \max_{a' \in A} \hat{A}(s, a', \theta_A) \right)$$

8 Policy Gradient Optimization

Fin ora abbiamo visto come approssimare la Value Function o la Q-function per poi valutare la policy sulla base di queste approssimazioni. Però in realtà la policy converge più velocemente se la approssimiamo direttamente, molto più velocemente rispetto alla Value Function. Quindi approssimiamo direttamente la policy, e non la Value Function.

Ora quindi cerchiamo di approssimare i parametri della policy: $\pi(a|s,\theta)$. Così facendo, la policy diventa una funzione parametrica che restituisce una distribuzione di probabilità sugli stati, per cui è continua e non discreta.

Pro di questo approccio:

- Converge più velocemente rispetto alla Value Function.
- Può gestire spazi di azioni continui e non banali.
- Possiamo imparare policy in modo stocastico.

Mentre i contro sono:

- Alta varianza.
- Non è garantita la convergenza all'ottimalità.
- Valutazione inefficiente.
- La policy π deve essere differenziabile rispetto ai parametri θ.
- Molto sensibile alla variazione dei parametri θ, infatti piccoli cambiamenti possono portare a policy molto diverse.

Quindi, come valutare la policy senza usare la Value Function o la Q-function?

Per farlo bisogna definire una loss alla basse dell'apprendimento.

Monte Carlo: Nel caso di Monte Carlo, viene usato il valore atteso a partire dallo stato iniziale, questo perchè con l'approccio off-line, conosciamo tutti gli episodi futuri. La loss function è la seguente:

- **Deterministic Case**: $\mathcal{L}(\theta) = \mathbb{E}_{\hat{\pi}}\left[V(s_1)\right]$
- **Stocastich Case**: $\mathcal{L}(\theta) = \mathbb{E}_{\hat{\pi}}\left[V(s_1)|s_1 \sim \wp(S)\right]$

Temporal Difference: Nel caso di Temporal Difference, viene usato il valore atteso a partire dallo stato corrente, per stimare la value function media o il reward atteso da un certo stato. La loss function è la seguente:

- **Avarage Value**: $\mathcal{L}(\theta) = \sum_{s \in S} d_{\hat{\pi}}(s) \cdot V_{\hat{\pi}}(s) = \mathbb{E}_{\hat{\pi}} \left[V(s) | s \sim d_{\hat{\pi}}(S) \right]$
- **Stocastich Case**: $\mathcal{L}(\theta) = \sum_{s \in S} d_{\hat{\pi}}(s) \cdot \sum_{a \in A} \hat{\pi}(a|s, \theta) \cdot R_s^a$

dove $d_{\hat{\pi}}(s)$ è la distribuzione stazionaria della policy $\hat{\pi}$, ovvero la probabilità di visitare lo stato s sotto la policy $\hat{\pi}$, e R_s^a è il reward atteso per lo stato s e l'azione a.

8.1 Policy Gradient Theorem

Il Policy Gradient Theorem ci permette di calcolare il gradiente della policy rispetto ai parametri θ:

Sia $\hat{\pi} \in C^1(\mathbb{R}^n)$ (policy differenziabile). In ogni contesto di ottimizzazione della policy, dove abbiamo una loss function $\mathcal{L}(\theta) \propto \mathbb{E}_{\hat{\pi}} \left[V(s) \right]$, il gradiente di quest'ultima rispetto ai parametri θ è:

$$\nabla_\theta \mathcal{L}(\theta) \propto \mathbb{E}_{\hat{\pi}} \left[Q_{\hat{\pi}}(s, a) \cdot \nabla_\theta \log \hat{\pi}(a|s, \theta) | d_{\hat{\pi}}(S, A) \right]$$

Dimostrazione: Partendo dalla definizione di aspettazione matematica della loss, per come l'abbiamo definita, abbiamo:

$$\mathcal{L}(\theta) \propto \mathbb{E}_{\hat{\pi}} \left[V(s) \right] = \sum_{s \in S} d_{\hat{\pi}}(s) \cdot \sum_{a \in A} \hat{\pi}(a|s, \theta) \cdot Q_{\hat{\pi}}(s, a)$$

Facendone il gradiente rispetto ai parametri θ, otteniamo:

$$\nabla_\theta \mathcal{L}(\theta) = \sum_{s \in S} d_{\hat{\pi}}(s) \cdot \sum_{a \in A} \nabla_\theta \hat{\pi}(a|s, \theta) \cdot Q_{\hat{\pi}}(s, a)$$

Questa espressione può essere riscritta come, l'aspettazione matematica rispetto alla policy $\hat{\pi}$:

$$\nabla_\theta \mathcal{L}(\theta) = \mathbb{E}_{\hat{\pi}} \left[\sum_{a \in A} Q_{\hat{\pi}}(s, a) \cdot \nabla_\theta \hat{\pi}(a|s, \theta) | d_{\hat{\pi}}(S, A) \right]$$

Ora, utilizzando il trucco del logaritmo che permette di semplificare il gradiente di una funzione composta:

$$\nabla_\theta \hat{\pi}(a|s, \theta) = \hat{\pi}(a|s, \theta) \cdot \nabla_\theta \log \hat{\pi}(a|s, \theta)$$

$$\nabla_\theta \log \hat{\pi}(a|s, \theta) = \frac{\nabla_\theta \hat{\pi}(a|s, \theta)}{\hat{\pi}(a|s, \theta)}$$

sostituendo nella formula precedente, otteniamo la formula del Policy Gradient Theorem:

$$\nabla_\theta \mathcal{L}(\theta) \propto \mathbb{E}_{\hat{\pi}} \left[Q_{\hat{\pi}}(s, a) \cdot \nabla_\theta \log \hat{\pi}(a|s, \theta) | d_{\hat{\pi}}(S, A) \right]$$

Il che vale $\forall a \in A$ e $\forall s \in S$.

Questo teorema è fondamentale perchè ci permette di approssimare il gradiente della loss function rispetto ai parametri θ, senza dover conoscere esplicitamente la forma della loss function.

Monte Carlo Policy Gradient: Come sempre nel caso di Monte Carlo, abbiamo che il Return Value G_t è non biased, perchè basiamo l'apprendimento su tutti gli episodi futuri.

Abbiamo un approccio **On-Policy**, dove la loss function è approssimata, utilizzando il teorema del Policy Gradient:

$$\mathcal{L}(\theta) \propto \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \hat{\pi}(a_t^i | s_t^i, \theta) \cdot G_t^i$$

dove N è il numero di episodi, e T è il numero di passi in ogni episodio.

La prima parte della formula ($\frac{1}{N} \sum_{t=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \hat{\pi}(a_t^i | s_t^i, \theta)$) è il Maximum Likelihood Gradient, che è un problema supervisionato dalla conoscenza della policy corrente, mentre la seconda parte (G_t^i) è il Return Value. Quest'ultimo incoraggia la policy a intraprendere azioni che seguno traiettorie più performanti.

Se vogliamo usare l'approccio **Off-Policy**, dobbiamo usare l'Importance Sampling, che permette di stimare l'aspettazione della loss function rispetto alla policy target π utilizzando campioni raccolti dalla policy di comportamento μ. Questo processo presenta molto rumore.

Actor-Critic Policy Gradient: Il problema principale del Monte Carlo Policy Gradient è la varianza, che può essere molto alta come abbiamo già detto, quindi il nostro obiettivo è cercare di ridurla il più possibile. L'idea è quella di usare sia l'approccio di Policy Learnig che quello di Value Learning, ovvero approssimare $\hat{Q}(s, a, \omega) \approx \hat{Q}_\pi(s, a)$, come abbiamo già visto negli approcci di MC, TD e TD(λ), ecc.

Nel caso in cui $\hat{Q}(s, a, \omega)$ è una Deep Neural Network, allora bisogna distinguere due casi:

- $\omega \neq \theta$: dove abbiamo due DNN separate, una per la policy e una per la Q-function. Questo approccio è più semplice e stabile perché i parametri delle due reti non interferiscono tra loro.
- $\omega = \theta$: dove abbiamo una sola DNN per entrambe le funzioni. In questo caso abbiamo una condivisione dei parametri fra le due funzioni, e quindi la policy è influenzata dalla Q-function, e viceversa. Questo approccio è più complesso e instabile per via dei conflitti dei gradienti tra le due funzioni (vogliamo massimizzare la Q-function e minimizzare il gradiente della policy).

Introduciamo due entità:

- **Actor**: $\hat{\pi}$ che genera l'esperieza da cui basare le direzioni di aggiornamento.
- **Critic**: $\hat{Q}$ che valuta l'esperienza e fornisce un feedback all'Actor.

Compatible Function Approximation Theorem: Questo teorema ci dice che se si verificano le segueenti condizioni:

1. **Compatiblity**: $\nabla_\omega \hat{Q}(s,a,\omega) = \nabla_\theta \log \hat{\pi}(a|s,\theta)$, il che vuol dire che la Q-function e la policy sono compatibili tra loro, e quindi hanno il gradiente rispetto ai loro paramtri (rispettivamente ω e θ) uguale.
2. **Unbiased Property**: $\mathbb{E}_{\hat{\pi}}\left[(Q_\pi - \hat{Q})^2\right] \xrightarrow{t\to\infty} 0$, il che vuol dire che la Q-function approssimata converge alla Q-function reale.

allora il Policy Gradient Theorem è ancora valido:

$$\nabla_\theta \mathcal{L}(\theta) \propto \mathbb{E}_{\hat{\pi}}\left[\hat{Q}(S,A,\omega) \cdot \nabla_\theta \log \hat{\pi}(A|S,\theta) | d_{\hat{\pi}}(S,A)\right]$$

La **varianza** può essere essere ulteriormente ridotta utilizzando la **Baseline**,

che è una funzione che approssima il valore atteso della Q-function, ovvero approssima empiricamente il modello dell'ambiente:

$$\nabla_\theta \mathcal{L}(\theta) \propto \mathbb{E}_{\hat{\pi}}\left[\sum_{a\in A} (Q_{\hat{\pi}}(s,a) - B(s)) \cdot \nabla_\theta \hat{\pi} | d_{\hat{\pi}}(S,A)\right]$$

Distribuendo il prodotto della policy e della Q-function, otteniamo:

$$\nabla_\theta \mathcal{L}(\theta) \propto \mathbb{E}_{\hat{\pi}}\left[Q_{\hat{\pi}}(s,a) \cdot \nabla_\theta \hat{\pi} | d_{\hat{\pi}}(S,A)\right] - \mathbb{E}_{\hat{\pi}}\left[B(s) \cdot \nabla_\theta \hat{\pi} | d_{\hat{\pi}}(S,A)\right]$$

Da qui possiamo notare che la Baseline $B(s)$ dipende solo dallo stato s, e non dall'azione a, questo fa si che il gradiente della policy non ne sia influenzato:

$$\mathbb{E}_{\hat{\pi}}\left[\sum_{a\in A} B(S) \nabla_\theta \hat{\pi}\right] = \mathbb{E}_{\hat{\pi}}\left[B(S) \sum_{a\in A} \nabla_\theta \hat{\pi}\right] = \mathbb{E}_{\hat{\pi}}\left(B(S) \cdot \nabla_\theta 1\right) = 0$$

Un ottima scelta per la Baseline è la Value Function, perchè approssima il valore atteso della Q-function, e quindi riduce la varianza:

$$B(s) = V_{\hat{\pi}}(s)$$

Definiamo ora la funzione di **Advantage**:

$$A_{\hat{\pi}}(s,a) = Q_{\hat{\pi}}(s,a) - V_{\hat{\pi}}(s)$$

che misura quanto valore aggiunge l'azione a nello stato s.

Usando la funzione di Advantage come Baseline, otteniamo la seguente formula per il gradiente della policy:

$$\nabla_\theta \mathcal{L}(\theta) \propto \mathbb{E}_{\hat{\pi}} \left[A_{\hat{\pi}}(s, a, \omega) \cdot \nabla_\theta \log \hat{\pi}(a|s, \theta) | d_{\hat{\pi}}(S, A) \right]$$

In un contesto Actor-Critic, la funzione di Advantage complica molto le cose, perchè abbiamo bisogno di approssimare ben tre set di parametri diversi:

$$\hat{A}(s, a, \omega, \rho) = Q_{\hat{\pi}}(s, a, \omega) - V_{\hat{\pi}}(s, \rho)$$

- θ: parametri della policy.
- ω: parametri della Q-function.
- ρ: parametri della Value Function.

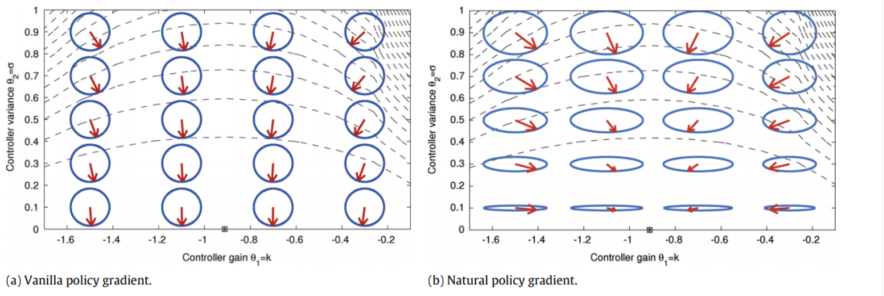
In ogni caso solo il set di parametri ρ è strettamente necessario, infatti possimo scrivere:

$$Q_\pi(s, a) = \mathbb{E}_{s' \in S} \left[r + \gamma V_\pi(s') \right]$$
$$A_\pi(s, a) = \mathbb{E}_{s' \in S} \left[r + \gamma V_\pi(s') - V_\pi(s) | s, a \right]$$

8.2 Natural Policy Gradient

Una politica rumorosa (noisy policy) è una politica che ha alta variabilità e incertezza nelle sue decisioni. Ciò genera dati rumorosi, cioè dati con alta varianza che possono portare ad aggiornamenti poco affidabili e instabili durante il processo di apprendimento. Per questo motivo i metodi di Policy Gradient performano maluccio, perché cercano di aggiornare direttamente i parametri della policy usando il gradiente della loss function che include la policy rumorosa stessa.

In oltre gli spazi in cui opera la policy posso avere dei **plateau**, ovvero delle zone piatte in cui il gradiente è molto piccolo, o delle **discontinuità** che portano nel primo caso a un apprendimento molto lento, e nel secondo a un apprendimento molto instabile, così da non riuscire a raggiunge un minimo globale accettabile, come si può vedere in figura 8a, in cui il normale Policy Gradient non riesce a raggiungere il minimo globale in basso al centro.



(a) Vanilla policy gradient. (b) Natural policy gradient.

Figure 8: Policy Gradient vs Natural Policy Gradient

Una soluzione proposta è quella di fare cambiamenti ai parametri θ in modo proporzionale alla variazione della politica. Questo significa calibrare l'entità dei cambiamenti ai parametri in base a quanto la politica cambia effettivamente, per evitare grandi balzi o passi inutilmente piccoli. Per fare ciò si utilizza la **KL-Divergence**, per calcolare la loss nel caso in cui la KL-Divergence tra la policy attuale e la policy aggiornata sia pari a un certo valore ϵ:

$$KL(\pi_\theta||\pi_{\theta+\Delta\theta}) = \int_{S\times A} \pi_\theta(s,a) \cdot \ln\left(\frac{\hat{\pi}(s|a,\theta_t)}{\hat{\pi}(s|a,\theta_{t-1})}\right) d(s,a)$$

Così facendo, riformuliamo il gradiente della loss function come:

$$\nabla_\theta \mathcal{L}(\theta) \leftarrow \nabla_\theta \mathcal{L}(\theta) \text{ s.t. } KL(\pi_\theta||\pi_{\theta+\Delta\theta}) = \epsilon$$

dove ϵ è un iperparametro che regola la variazione della policy.

Come si può vedere in figura 8b, il Natural Policy Gradient altera lo spazio di ricerca dei parametri localmente, in modo da attenuare i plateau e le discontinuità, a differenza del Policy Gradient che invece da equi-importanza a tutti i paramtri (a tutte le direzioni) nello spazio di ricerca.

9 Model-Based Reinforcement Learning

Fin ora abbiamo visto come funzionano i metodi di Reinforcement Learning che approssimano la Value Function o la Q-function, e che permettono di valutare la policy sulla base di queste approssimazioni. Ora vediamo come funzionano i metodi di Reinforcement Learning che approssimano il modello dell'ambiente, e che permettono di pianificare la policy sulla base di queste approssimazioni.

Quindi in questo nuovo paradigma, prima viene approssimato il modello dell'ambiente dall'esperienza raccolta (e questo è un problema supervisionato), e poi viengono approssimate la Value Function, la Q-function e la policy sulla base di questo modello.

L'incertezza del modello è considerata come una prospettiva di ragionamento aggiuntiva. Nel model-based reinforcement learning, è importante riconoscere che il modello dell'ambiente potrebbe non essere sempre perfetto e potrebbe introdurre incertezze. Questo implica che bisogna considerare e gestire l'incertezza del modello nell'ottimizzazione delle decisioni.

Utiliziamo due sorgenti di approssimazione dell'errore:

- P_η: approssimazione della matrice di transizione.
- R_η: approssimazione della funzione di reward.

Da cui quindi abbiamo l'ambiente approssimato:

$$M_\eta = \langle P_\eta, R_\eta \rangle$$

dove η sono i parametri del modello approssimato.

Table Lookup Model: Questo è il metodo più semplice, dove si costruisce una tabella che mappa coppie stato-azione a risultati futuri. È un metodo tabulare che funziona bene con spazi di stato relativamente piccoli e discreti. In questo caso, la matrice di transizione e la funzione di reward sono:

- $P^a_{ss'} = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbb{1}(s, a, s')$
- $R^a_s = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbb{1}(s, a) \cdot r_t$

Per calcolare la matrice di transizione $P^a_{ss'}$ e la funzione di reward R^a_s, si utilizza l'esperienza raccolta durante l'addestramento, e poi il modello viene usato per generare nuove esperienze.

Il modello appreso può essere utilizzato per pianificare basandosi su campioni (sample-based planning), ovvero simulare molteplici traiettorie possibili e utilizzare queste informazioni per ottimizzare le decisioni.

Problema 1: Il modello approssimato potrebbe non essere sempre perfetto, e potrebbe introdurre incertezze. L'incertezza del modello porta a una policy che non è ottimale, e quindi bisogna considerare e gestire l'incertezza del modello nell'ottimizzazione delle decisioni.

Quando l'incertezza del modello è alta, possiamo:

- Tornare all'approccio di Model-Free Reinforcement Learning.
- Introdurre esplicitamente l'incertezza del modello per poi usarla per ottimizzare le decisioni.
- Una via di mezzo tra i due approcci, ovvero utilizzando due sorgenti di esperienza: il modello approssimato e l'esperienza reale. (**Integrated Approaches**).

Un vantaggio nell'usare un approccio Integrated è che possiamo utilizzare l'esperienza reale per ridurre l'incertezza del modello, e poi usare il modello per generare nuove esperienze. Quest'ultima cosa è molto conveniente quando abbiamo un ambiente che è costoso o pericoloso da esplorare.

9.1 Dyna-Q Algorithm

L'algoritmo di Dyna-Q è un esempio di Integrated Approach, che combina l'apprendimento diretto dall'esperienza reale con la simulazione dell'ambiente. L'algorimto è composto da due fasi:

- **Singletone Experience**: apprendimento diretto dall'esperienza reale.
- **Past-Multiple Experience**: simulazione dell'ambiente per generare nuove esperienze.

L'algorimto di Dyna-Q è in grado di adattarsi automaticamente alle variazioni nel modello dell'ambiente. Se il modello viene aggiornato in base a nuove esperienze, Dyna-Q è capace di recuperare e migliorare le sue stime di Q utilizzando queste nuove informazioni.

Algorithm 3 Dyna-Q Algorithm

Input: S, A, γ
Inizialize $Q(s,a)$ and $M(s,a)$
// Singletone Experience Phase
while true **do**
  Choose $s \in S$
  $a \leftarrow greedy(Q(s,))$
  Collect r and s'
  $M(s,a) \leftarrow \langle r, s' \rangle$
  Use $TD(0)$ with M to update $Q(s,a)$
  // Past-Multiple Experience Phase
  for $i = 1$ to N **do**
    Choose $s \in S$ from past experience H
    Choose a from A from past experience $H(s)$
    $\langle r, s' \rangle \leftarrow M(s,a)$
    Use $TD(0)$ with M to update $Q(s,a)$
  end for
end while
return V_k

1. **Initialize Q and M**
   - Inizializzare le stime di Q (funzione di valore stato-azione) e M (modello dell'ambiente).
   - M = model: M rappresenta il modello dell'ambiente, che viene aggiornato con le nuove esperienze.
2. **while true do**
   - Ciclo principale dell'algoritmo.
   - **Singleton phase**:
     (a) **Choose** $s \in S$
       – Scegliere uno stato s dall'insieme di stati S.
     (b) $a \leftarrow$ **ϵ-greedy(**$Q(s, \cdot)$**)**
       – Scegliere un'azione a utilizzando una politica ϵ-greedy basata sulla funzione Q corrente. Questo comporta scegliere l'azione ottimale la maggior parte delle volte, ma occasionalmente esplorare nuove azioni.
     (c) **Collect** r and s'
       – Raccogliere la ricompensa r e il nuovo stato s' risultante dall'esecuzione dell'azione a nello stato s.
     (d) **Model update:** $M(s, a) \leftarrow \langle s', r \rangle$
       – Aggiornare il modello M con la nuova transizione osservata $\langle s, a, r, s' \rangle$.
   - **Past multiple repetition**:
     (a) **for** n **times do**
       – Ciclo per eseguire più volte aggiornamenti basati su simulazioni dal modello.
       – **Choose** $s \in S$ **from** H = history
         * Scegliere uno stato s dalla storia H, cioè un insieme di stati storici memorizzati.
       – **Choose** $a \in A$ **from** $H(s)$
         * Scegliere un'azione a dall'insieme di azioni storiche associate allo stato s.
       – $\langle r, s' \rangle \leftarrow M(s, a)$
         * Utilizzare il modello M per simulare la transizione $\langle r, s' \rangle$ risultante dall'azione a nello stato s.
       – **TD(0) update of** $Q(s, a)$
         * Applicare l'aggiornamento del TD(0) alla funzione $Q(s, a)$:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

     (b) **end for**
3. **end while**

9.2 Forward Search

In questo approccio l'Agent decide quale azione intraprendere simulando le possibili azioni e i possibili stati futuri, semplicemente guardando avanti. Per fare ciò, viene costruito un albero di ricerca, in cui ogni nodo rappresenta uno stato, e ogni arco rappresenta un'azione, e dove la radice è lo stato attuale. Questo approccio ricorda la Breadth-Search nella programmazione dinamica, infatti nell'albero di ricerca vengono esplorati tutti i possibili stati futuri, e poi viene scelta l'azione che porta allo stato migliore. Ci sono però delle differenze:

- **Sample-Based**: Ci basiamo su campioni per costruire l'albero di ricerca, e non su una conoscenza esatta dell'ambiente. Ovvero simuliamo l'ambiente per generare nuove esperienze.
- **Planning:** Usiamo l'albero per pianificare (prevedere e scegliere azioni basate su simulazioni) piuttosto che sull'apprendimento (adattare la politica basandosi su esperienze passate). I dati utilizzati possono essere imperfetti o parziali.

Naive Monte Carlo Search: Questo approccio non migliora la policy π e per lunghi run diventa inefficiente.

Algorithm 4 Naive Monte Carlo Search

1: **Input:** M, S, A, s, π, k
2: **for** all $a \in A$ **do**
3: Simulate k steps from s
4: $Q(s,a) \leftarrow \frac{1}{k} \sum_{t=1}^{k} G_t$
5: **end for**
6: **return** $a = \arg\max_{a \in A} Q(s,a)$

1. **Input:** M, S, A, s, π, k
   - M: modello dell'ambiente.
   - S: insieme di stati.
   - A: insieme di azioni.
   - s: stato attuale.
   - π: politica corrente.
   - k: numero di episodi da simulare.
2. **for** all $a \in A$ **do**
   - Per ogni azione a disponibile, eseguire i seguenti passaggi.
   - **Simulate** k **steps from** s: Simulare k episodi a partire dallo stato s a uno stato finale.

- $Q(s,a) \leftarrow \frac{1}{k}\sum_{t=1}^{k} G_t$: Calcolare il valore $Q(s,a)$ come la media dei ritorni G_t ottenuti dalle simulazioni.

3. **return** $a = \arg\max_{a \in A} Q(s,a)$: Restituire l'azione a che massimizza il valore $Q(s,a)$.

Monte Carlo Tree Search: Questo approccio migliora il Naive Monte Carlo Search, e si basa su quattro fasi:

1. **Selection:** Scegliere il nodo da cui iniziare la simulazione. Esso diventerà la radice dell'albero di ricerca.

2. **Expansion:** Una volta selezionato un nodo, si espande l'albero aggiungendo uno o più figli, rappresentando le possibili azioni e stati risultanti nel tree di ricerca.

3. **Simulation:** Si esegue una simulazione "grezza" partendo dal nodo selezionato per esplorare una sequenza di azioni fino a raggiungere una condizione terminale o un numero massimo di mosse. Durante questa fase si raccolgono le ricompense ottenute.

4. **Backpropagation:** I risultati della simulazione vengono propagati indietro lungo il tree di ricerca, aggiornando le statistiche (come valore e contatore di visite) di ogni nodo attraversato fino alla radice.

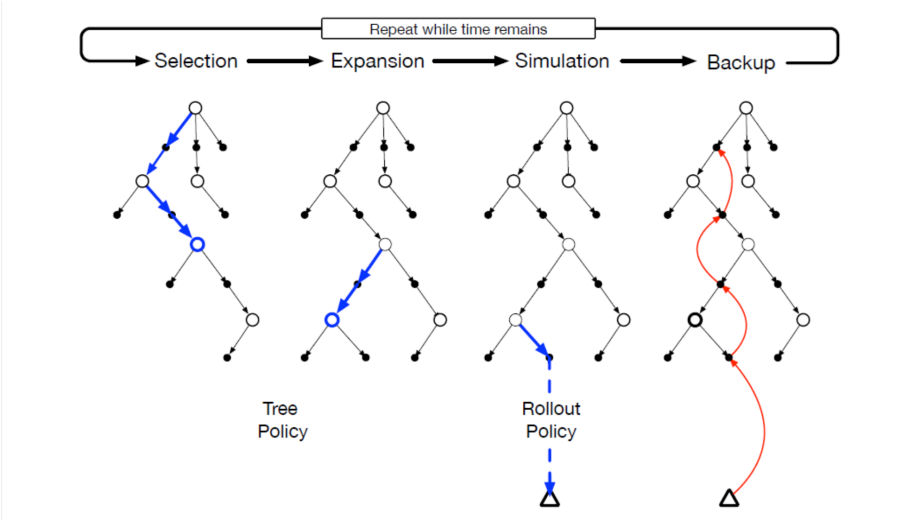


Figure 9: Monte Carlo Tree Search

La Policy π viene valutata in base al valore $Q(s,a)$, che è la media dei ritorni

ottenuti dalle simulazioni:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T} \mathbb{1}(s_t^k, a_t^k) G_t^k$$

Dove:

- $N(s,a)$: numero di volte che l'azione a è stata eseguita nello stato s.
- K: numero di episodi simulati.
- T: numero di passi per ciascun episodio.
- $\mathbb{1}(s_t^k, a_t^k)$: Indicatore che vale 1 se in t-esimo passo dell'episodio k, ci troviamo nello stato s ed è stata eseguita l'azione a.
- G_t^k: ritorno ottenuto al passo t della simulazione k.

Per migliorare la policy si utilizza l'$\epsilon - greedy(Q)$, quindi si sceglie l'azione ottima (quella che in quel momento massimizza Q) per il $1 - \epsilon$ delle volte, e un'azione casuale per il ϵ delle volte.

I vantaggi di questo approccio sono:

- **Best-First Search**: L'algoritmo si concentra sulle azioni che portano a stati più promettenti.
- **Parallelizable**: L'algoritmo può essere facilmente parallelizzato, in quanto ogni simulazione è indipendente dalle altre.
- **No-Exhaustive Search**: Fare una ricerca esaustiva nello spazio delle azioni è computazionalmente proibitivo, quindi l'algoritmo si basa su simulazioni per esplorare le possibili azioni.

9.3 TD Search

Come abbiamo già visto l'approccio Temporal Difference (TD) opera in modalità online. Utilizzando $\langle S, A, R, S, A \rangle$ per aggiornare la Q-function. La policy viene valutata sempre con l'algorimto $\epsilon - greedy(Q)$, e l'azione viene scelta in base alla policy.

9.4 Dyna2 Algorithm

L'idea è sempre la stessa di *Dyna*, mescolando esperienza reale e simulata, ma in questo caso si utilizzano due Q-function:

- **Persistent** Q^P: viene aggiornata con l'esperienza reale.
- **Transient** Q^T: viene aggiornata con l'esperienza simulata, e poi dopo ogni simulazione viene azzzerata.

Le azioni vengono scelte in base alla policy $\epsilon - greedy(Q^P + Q^T)$, così da tener conto sia dell'esperienza reale che di quella simulata.

9.5 Exploration vs Exploitation: Trade-Off

Letteralmente **esplorazione** vs **sfruttamento**. Questo trade-off è fondamentale in Reinforcement Learning, perchè se da una parte vogliamo sfruttare al massimo le informazioni che abbiamo, dall'altra parte dobbiamo esplorare nuove azioni per scoprire nuove informazioni.

Il Forward Search è un approccio di tipo **Exploration**, che necessita di una buona esplorazione per performare bene. Ma fin ora abbiamo usato tecniche di esplorazione pressochè randomiche.

Per bilanciare esplorazione e sfruttamento, spesso si introduce rumore nelle decisioni della politica greedy. Questo perché una politica completamente greedy potrebbe non esplorare abbastanza, e quindi non raccogliere abbastanza informazioni per migliorare la policy.

Vengono proposti due paradigmi aggiuntivi che possono migliorare l'esplorazione in modo più sistematico che non semplicemente l'esplorazione casuale:

- **Optimistic**: Stima dell'incertezza del valore stato-azione. Questo si riferisce al tenere traccia dell'incertezza associata alle stime dei valori delle azioni. Se una coppia stato-azione è molto incerta (meno conosciuta), è saggio esplorarla ulteriormente per raccogliere dati e ridurre l'incertezza. Si cerca di esplorare meglio le coppie stato-azione meno conosciute o più incerte.

- **Information-State**: Lo spazio stato-azione viene arricchito con informazioni derivanti dall'esperienza passata. Vengono usate queste informazioni per guidare l'esplorazione (Look-Ahead Search).

Multi-Armed Bandit Scenario: Il problema del "multi-armed bandit" (letteralmente "bandito con più braccia") è un classico problema di esplorazione e sfruttamento (exploration-exploitation tradeoff) nel machine learning e nel reinforcement learning. Immagina di trovarti davanti a una serie di slot machine (o "banditi" con più braccia) e devi decidere quale macchina giocare per massimizzare la tua ricompensa totale. Ogni slot machine ha una distribuzione di ritorno diversa, e il compito è scoprire quale slot machine offre la miglior media del ritorno.

In questa situazione l'Agent parte sempre nello stesso stato, deterministico. Poi l'Agent sceglie un'azione fra tante disponibili, associate allo stesso stato (Ad esempio, scegliere tra diverse slot machine da giocare), ogniuna delle quali porta a uno stato terminale diverso. Questo significa che, tirando una delle braccia di una slot machine, si raggiunge una diversa distribuzione di ritorni rispetto a un'altra braccia.

Non c'è uno spazio degli stati dinamico come nei problemi di reinforcement learning tipici. L'agente non transita fra diversi stati nel tempo, ma prende una decisione singola in termini di quale azione eseguire ogni volta.

L'obiettivo è apprendere R^a, cioè la funzione di ricompensa associata a ciascuna azione a. In altre parole, determinare la ricompensa media (o attesa) di ciascuna delle azioni disponibili.

9.6 Regret

Come misurare la qualità dell'esplorazione? La qualità dell'esplorazione può essere misurata osservando quanto efficacemente un algoritmo bilancia il tradeoff tra esplorazione e sfruttamento. Il **regret** viene spesso utilizzato come metrica per questa qualità.

Il **regret asintotico** rappresenta la perdita rispetto all'agente ottimale (quello che conosce sempre la migliore azione da prendere). Misura la differenza accumulata tra la ricompensa dell'agente ottimale e la ricompensa effettivamente raccolta dall'agente attuale, per cui più è basso il regret, migliore è l'agente.

Minimizzare il regret è equivalente a massimizzare la ricompensa cumulativa attesa. Entrambi gli obiettivi portano a una performance migliore dell'agente nel lungo periodo.

Abbiamo però il problema di non conoscere l'agente ottimale, quindi non possiamo calcolare il regret esatto. Per risolvere questo problema si utilizza:

- **Greedy Algorithm**: che selezionano sempre l'azione con il miglior valore stimato corrente senza esplorare nuove azioni, hanno un regret che cresce linearmente con il numero di passi temporali. Questo significa che perdono opportunità di miglioramento a lungo termine perché non esplorano sufficientemente.

- **ϵ-greedy**: che selezionano l'azione con il miglior valore stimato con probabilità $1-\epsilon$, e un'azione casuale con probabilità ϵ. Questo approccio ha un regret che cresce con il tempo, ma in modo sublineare rispetto al tempo.

L'obiettivo è implementare una strategia di esplorazione che minimizzi il regret in modo sublineare, senza la necessità di un oracolo che conosca a priori le ricompense ottimali delle azioni.

Per fare ciò si utilizza **Hoeffding's Inequality**, che fornisce un limite superiore alla probabilità che la media empirica di un campione di variabili casuali I.I.D. sia lontana dal suo valore atteso (media della popolazione) di più di una certa quantità. È uno strumento statistico importante per garantire che le stime delle ricompense siano vicine ai loro veri valori attesi con alta probabilità:

Proposizione - Hoeffding's Inequality: Sia $X_1, X_2, ..., X_t$ un campione di variabili casuali I.I.D. con $0 \leq X_i \leq 1$ per ogni i, e sia $\bar{X} = \frac{1}{t}\sum_{i=1}^{t} X_i$ la media empirica. Allora:

$$P(\mathbb{E}[x] - \bar{x}_t \geq u) = P(\mathbb{E}[x] \geq \bar{x}_t + u) \leq e^{-2tu^2}$$

9.7 Optimistm

In questo caso si sceglie l'azione sulla base di due fattori:

- **Expected Reward**: La ricompensa attesa, che rappresenta la stima attuale del valore di un'azione. È basata sulla media delle ricompense osservate per questa azione.

- **Variance**: La varianza, che rappresenta l'incertezza associata alla stima del valore di un'azione.

Il bound superiore di confidenza (Upper Confidence Bound) per un'azione indica un intervallo che rappresenta una stima ottimistica di quanto può essere buona un'azione, prendendo in considerazione sia la ricompensa stimata che l'incertezza. La confidenza segue il seguente ordine di grandezza:

$$U_t(a) \propto \frac{1}{\sqrt{N_t(a)}}$$

dove $N_t(a)$ è il numero di volte che l'azione a è stata eseguita fino al tempo t. Quindi più l'zione è rara e più è incerta.

Si considerano solo le azioni il a tale che:

$$Q_t(a^*) \leq Q_t(a) + U_t(a)$$

Questo per assicurarsi di non ignorare potenziali azioni buone che sono state esplorate poco. E si sceglie quindi l'azione che massimizza: $Q_t(a) + U_t(a)$:

$$a_t = \arg\max_{a \in A} Q_t(a) + U_t(a)$$

L'algorimto Upper Confidence Bound (UCB): Per prima cosa riscriviamo la disuguaglianza di Hoeffding in modo che sia una disuguaglianza in termini di $N_t(a)$:

$$\mathbb{P}(Q_t(a^*) \geq Q_t(a) + U_t(a)) \leq e^{-2N_t(a)U_t(a)^2}$$

Dove:

- $Q_t(a^*)$: Valore atteso ottimale corrente per l'azione migliore conosciuta a^*.
- $Q_t(a)$: Valore stimato per l'azione a a tempo t.
- $U_t(a)$: Bound superiore di confidenza per l'azione a.
- $N_t(a)$: Numero di volte che l'azione a è stata eseguita fino al tempo t.

Questa formuala ci dice che la probabilità che il valore ottimo $Q_t(a^*)$ sia maggiore del valore stimato $Q_t(a)$ più il bound superiore di confidenza $U_t(a)$ è limitata dall'esponezialmente.

Il passo successivo è stabilire un livello di incertezza p per l'azione che stiamo prendendo in considerazione:

$$e^{-2N_t(a)U_t(a)^2} = p$$

Risolvendo poi per $U_t(a)$ otteniamo:

$$U_t(a) = \sqrt{\frac{-\ln p}{2 \cdot N_t(a)}}$$

Il livello di incertezza decade nel tempo t dato che abbiamo il numero di volte che l'azione a è stata eseguita fino a quel momento al denominatore. Il che vuol dire che diventiamo sempre più sicuri man mano che passiamo per l'azione a.

Applicando l'algoritmo UCB, il regret dell'agente cresce sublinearmente nel tempo. Questo significa che anche se il regret totale aumenta, aumenta a un tasso più lento, portando a decisioni migliori e più ottimali nel tempo. In oltre grazie alla formula UCB e alla disuguaglianza di Hoeffding, possiamo garantire che il nostro agente, col passare del tempo e l'accumularsi delle esperienze, converge asintoticamente verso l'azione ottimale, minimizzando il regret.

10 Formulario

10.1 Agent, Environment, State, Action, Reward

- **Policy**: $\pi(a|s) = \mathbb{P}(a_t = a|a_t = s)$
- **Reward**: $R_s^a = \mathbb{E}[r_{t+1}|s_t = s, a_t = a]$
- $R_s^\pi = \sum_{a\in A} \pi(a|s) \cdot R_s^a = \mathbb{E}[R_s^a]$
- $P(s'|s,a) = P_{ss'}^a = \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a)$
- $P_{ss'}^\pi = \sum_{a\in A} \pi(a|s) \cdot P_{ss'}^a = \mathbb{E}[P_{ss'}^a]$
- **Return**: $G_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1}$
- **Value Function**: $V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$
- **Q-function**: $Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$

10.2 Bellman Equation

- **Bellman Expectation Equation**:

$$V_\pi(s) = R_s^\pi + \gamma \sum_{s'\in S} P_{ss'}^\pi \cdot V_\pi(s')$$

- **Bellman Expectation Equation for Q-function**:

$$Q_\pi(s,a) = R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a \cdot V_\pi(s')$$

10.3 Bellman Optimality Equation

- **Bellman Optimality Equation**:

$$V_*(s) = \max_{a\in A} R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a \cdot V_*(s')$$

- **Bellman Optimality Equation for Q-function**:

$$Q_*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \cdot \max_{a' \in A} Q_*(s', a')$$

10.4 Policy Iteration

- $\pi'(s) = \arg\max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \cdot V_\pi(s')$

10.5 Monte Carlo

- $V_\pi(s) = \mathbf{E}_\pi[G_t | S_t = s]$
- $G_t^T = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$
- **Incremental Mean**: $V(s) \leftarrow V(s) + \frac{1}{N(s)}(G_t - V(s))$
- **Mean Squared Error**: $V_\pi = \arg\min_V \mathbb{E}_\pi[(G_t - V(s))^2]$

10.6 Temporal Difference

- $V(s) \leftarrow V(s) + \alpha(G_t - V(s))$
- $G_t = r_{t+1} + \gamma V(s_{t+1})$
- $\hat{P}_{ss'}^a = \frac{1}{N(s,a)} \sum_{e=1}^{E} \sum_{t=1}^{T} \mathbb{1}(s, a, s')$
- $\hat{R}_s^a = \frac{1}{N(s,a)} \sum_{e=1}^{E} \sum_{t=1}^{T} \mathbb{1}(s, a) \cdot r_t^{(e)}$

10.7 TD(n)

- $V(s) \leftarrow V(s) + \alpha(G_t^{(n)} - V(s))$
- $G_t^{(n)} = \sum_{k=1}^{n} \gamma^{k-1} r_{t+k} + \gamma^n V(s_{t+n+1})$

10.8 TD(λ)

- $V(s) \leftarrow V(s) + \alpha(G_t^{(\lambda)} - V(s))$
- $G_t = (1-\lambda) \sum_{n=1}^{N} \lambda^{n-1} G_t^{(n)}$
- $\sum_{n=1}^{\infty} (1-\lambda)\lambda^{n-1} = 1$

10.9 TD(λ) - Eligibility Traces

- $V(s) \leftarrow V(s) + \alpha \delta_t E_t(s)$
- $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$
- $E_0(s) = 0$
- $E_t(s) = \gamma \lambda E_{t-1}(s) + \mathbb{1}(s_t = s)$

10.10 ϵ-greedy

- $\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + (1-\epsilon) & \text{se } a = \arg\max_{a \in A} Q(s,a) \\ \frac{\epsilon}{m} & \text{altrimenti} \end{cases}$

10.11 GLIE - Greedy in the Limit with Infinite Exploration

- $\lim_{k \to \infty} N_k(s,a) = \infty$
- $\lim_{k \to \infty} \epsilon_k = 0$
- $\sum_{k=1}^{\infty} \epsilon_k = \infty$

10.12 SARSA

- $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$

10.13 Expected SARSA

- $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \sum_{a' \in A} \pi(a'|s') Q(s',a') - Q(s,a))$

10.14 SARSA(n)

- $Q(s,a) \leftarrow Q(s,a) + \alpha(G_t^{(n)} - Q(s,a))$
- $G_t^{(n)} = \sum_{k=1}^{n} \gamma^{k-1} r_{t+k} + \gamma^n Q(s_{t+n}, a_{t+n})$

10.15 SARSA(λ)

- $Q(s,a) \leftarrow Q(s,a) + \alpha(G_t^{(\lambda)} - Q(s,a))$
- $G_t = (1-\lambda) \sum_{n=1}^{N} \lambda^{n-1} G_t^{(n)}$

10.16 Backward (Expected) SARSA(λ)

- $Q(s,a) \leftarrow Q(s,a) + \alpha \delta_t E_t(s,a)$
- $\delta_t = r_{t+1} + \gamma \sum_{a' \in A} \pi(a'|s') Q(s',a') - Q(s,a)$
- $E_0(s,a) = 0$
- $E_t(s,a) = \gamma \lambda E_{t-1}(s,a) + \mathbb{1}(s_t = s, a_t = a)$

10.17 Off-Policy - Importance Sampling

- $\mathbf{E}_{x\sim p}[f(x)] = \sum_x p(x)f(x) = \sum_x q(x)\frac{p(x)}{q(x)}f(x) = \mathbf{E}_{x\sim q}[\frac{p(x)}{q(x)}f(x)]$
- Monte Carlo:

$$G_t^\pi = \frac{\pi(a_t|s_t)}{b(a_t|s_t)}G_t^b$$

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(G_t^\pi - Q(s_t,a_t))$$

- TD:

$$Q(s_t,a_t) \leftarrow \alpha\left[\frac{\pi(a_t|s_t)}{b(a_t|s_t)}(r_{t+1} + \gamma Q(s_{t+1},a_{t+1})) - Q(s_t,a_t)\right]$$

10.18 Q-Learning

- $Q(s,a) \leftarrow Q(s,a) + \alpha(r_{t+1}^\mu + \gamma Q(s_{t+1}, a_{t+1}^\pi) - Q(s,a))$
- $r_{t+1}^\mu + \gamma Q(s_{t+1}, a_{t+1}^\pi) = r_{t+1}^\mu + \gamma Q(s_{t+1}, \arg\max_{a'\in A} Q(s_{t+1},a'))$
- $r_{t+1}^\mu + \gamma \max_{a\in A} Q(s_{t+1},a)$

10.19 Deep Q Network (DQN)

- **Simple DQN** $L(\theta) \propto \mathbb{E}_D[(r + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta))^2]$
- **Double DQN** $L(\theta) \propto \mathbb{E}_D[(r+\gamma Q(s', \arg\max_{a'} Q(s',a';\theta);\theta^-) - Q(s,a;\theta))^2]$

10.20 Policy Gradient

- **Policy Gradient Theorem**: $\nabla L(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a) | s \sim d_\pi(S,A)]$
- **Monte Carlo**: $L(\theta) \propto \sum_{i=1}^N \sum_{t=1}^T \log \pi_\theta(s_t^i, a_t^i) G_t^i$
- **Actor-Critic: Teorema di compatibilità**:
  1. **Compatiblity**: $\nabla_\omega \hat{Q}(s,a,\omega) = \nabla_\theta \log \hat{\pi}(a|s,\theta)$
  2. **Unbiased Property**: $\mathbb{E}_{\hat{\pi}}\left[(Q_\pi - \hat{Q})^2\right] \xrightarrow{t\to\infty} 0$