



UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

Applicazione di Value Iteration e Q-learning in ambiente Pacman

.....

Progetto per il corso di Intelligenza Artificiale

Nicholas Dumas, Matteo Rossini

Obiettivi del progetto

Value Iteration applicato a Pacman

- Rispondere ad alcuni quesiti relativi al **Reinforcement Learning**.
- Implementare l'algoritmo Value Iteration per l'ambiente Pac-Man
- Condurre un'analisi comparativa tra Value Iteration e Q-Learning
- Investigare l'impatto di alcuni iperparametri sulle prestazioni dell'agente.

Ambiente Pacman

smallGrid

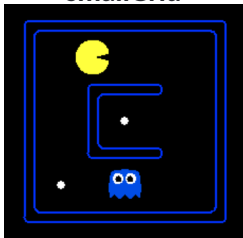


Figura: Dimensioni 7x7

mediumClassic

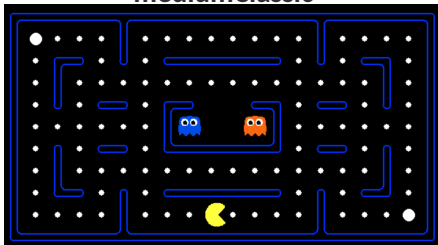


Figura: Dimensioni 11x20

Formalizzazione MDP

Definition

MDP := (S, A, R, P, γ)

- S : Spazio degli stati, rappresentate tutte le possibili configurazioni di gioco (posizione dei fantasmi, pac-man, cibo, pillole energetiche).
- A : Spazio delle azioni $A = \{\text{nord, sud, est, ovest}\}$
- $R : S \times A \times S \rightarrow \mathbb{R}$: Funzione di ricompensa $R(s, a, s')$ progettata per incentivare i comportamenti desiderati.
- $P : S \times A \times S \rightarrow [0, 1]$: Funzione di transizione $P(s'|s, a)$ che modella la stocasticità dell'ambiente.
- $\gamma \in [0, 1]$: Fattore di sconto, che bilancia le ricompense immediate e future.

Caratteristiche Ambiente Pacman

- **Stocasticità:** Le azioni dell'agente sono soggette a incertezza. (80% dir. intrapresa, 20% dir. perpendicolare)
- **Osservabilità completa:** L'agente ha accesso a tutte le informazioni rilevanti dello stato del gioco in ogni momento.
- **Non stazionarietà:** Dinamiche dell'ambiente in evoluzione
- **Episodicità:** Stati terminali ben definiti.
- **Ambiente multi-obiettivo:** massimizzare il punteggio, evitare i fantasmi e completare il gioco il più rapidamente possibile.
- **Multi-agente:** I fantasmi possono essere considerati avversari con propri comportamenti e obiettivi.
- **Dimensionalità variabile:** la complessità dell'ambiente può variare significativamente tra diverse mappe.

Ottimizzazione di Value Iteration per Pacman

- Aggiornamento dinamico delle ricompense per gestire la non-stazionarietà
- Implementazione di "danger zones" per modellare le minacce dei fantasmi
- Modalità "Ghostbuster" per sfruttare le capsule energetiche e guadagnare più punti
- Ottimizzazione delle strutture dati per ridurre la complessità computazionale

Parametri Critici di Value Iteration

- Fattore di sconto γ
- Distanza di sicurezza dai fantasmi
- Peso delle ricompense per cibo, fantasmi e capsule
- Numero massimo di iterazioni

Fondamenti del Value Iteration

Equazione di Bellman per la Funzione Valore-Stato

$$v_{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P(s'|s, a) [R(s, a, s') + \gamma v_{\pi}(s')]$$

- $v_{\pi}(s)$: valore atteso dello stato s seguendo la policy π
- $\pi(a|s)$: probabilità di scegliere l'azione a in s secondo π
- $P(s'|s, a)$: probabilità di transizione da s a s' data l'azione a
- $R(s, a, s')$: ricompensa immediata per la transizione $s \rightarrow s'$ con azione a
- γ : fattore di sconto (importanza delle ricompense future)

Fondamenti del Value Iteration (2)

Equazione di Bellman per la Funzione Valore-Stato Ottimale

$$v_*(s) = \max_{\pi} v_{\pi}(s)$$

$$v_*(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') [R(s, a, s') + \gamma v_*(s')]$$

Costi Computazionali

- **Problema:** Risolvere il sistema di equazioni di Bellman (lineare e non lineare)
- **Obiettivo:** Trovare V_* tale che $v_*(s) = f(v_*(s))$ (eq. punto fisso)
- **Metodo diretto:** MEG ha complessità temporale $O(S^3)$ e complessità spaziale $O(|S|^2)$.
- **Metodo iterativo:** Value Iteration ha complessità temporale $so(|S|^2 \cdot |A|)$ e complessità spaziale $O(S)$.

Value Iteration

Framework

1. **Inizializzazione:** Si inizializza arbitrariamente $v_0(s)$ per tutti gli stati $s \in S$.
2. **Iterazione del valore:** Per $k = 0, 1, 2, \dots$, si calcola per un singolo step di miglioramento:

$$v_{k+1}(s) = \max_{a \in A} \sum_{s' \in S} P(s, a, s') [R(s, a, s') + \gamma v_k(s')]$$

Dove:

- $v_k(s)$ è la stima del valore dello stato s all'iterazione k
- $v_{k+1}(s)$ è la stima aggiornata del valore dello stato s per l'iterazione $k + 1$

Value Iteration

Framework (2)

3. **Convergenza:** L'algoritmo termina quando $\max_{s \in S} |v_{k+1}(s) - v_k(s)| < \epsilon$, dove ϵ è una soglia di tolleranza predefinita, o quando viene eseguito un numero predeterminato di iterazioni.

Rewards e parametri del Value Iteration

Configurazione per l'ambiente Pacman

Parametro	Valore
FOOD_REWARD	10
GHOST_REWARD	-500
DANGER_ZONE	-250
CAPSULE_REWARD	50
BLANK_REWARD	-0.04

Tabella: Rewards

Parametro	Valore
THETA	0.01
DISCOUNT_FACTOR	0.6
SAFETY_DISTANCE	1
MAX_ITERATIONS	500
GHOSTBUSTER_MODE	True/False
NOISE	0.2

Tabella: Parametri

Implementazione Value Iteration

Classe MDPAgent

- Metodi chiave:
 - `__init__`: Inizializzazione dell'agente
 - `value_iteration`: Implementa l'algoritmo di Value Iteration
 - `_get_best_policy`: Calcola la policy ottimale
 - `__compute_q_value_from_values`: Calcola i Q-values
 - `__get_transition_states_and_probs`: Modella la dinamica dell'ambiente
 - `_update_rewards`: Aggiorna dinamicamente le ricompense
 - `getAction`: Seleziona l'azione ottimale

Implementazione del Value Iteration

```
def value_iteration(self):  
    for i in range(self.max_iterations):  
        delta = 0  
        for cell in self.legal_states:  
            cell_value = self.values[cell]  
            self.values[cell] = self._get_best_policy(cell)  
            delta = max(delta, abs(cell_value - self.values[cell]))  
        if delta < THETA:  
            return i
```

- Itera su tutti gli stati legali.
- Aggiorna i valori degli stati
- Termina se la variazione massima tra i valori degli stati in due interazioni successive è inferiore alla soglia predeterminata THETA, altrimenti quanto si raggiunge il max_iterations

Calcolo della Policy Ottimale

```
def _get_best_policy(self, cell):  
    return max(  
        self.__compute_q_value_from_values(state, action)  
        for action in self.move_offsets  
    )
```

- Seleziona l'azione con il massimo Q-value per un dato stato
- Utilizza `__compute_q_value_from_values` per calcolare i Q-values

Calcolo dei Q-Values

```
def __compute_q_value_from_values(self, state, action):  
    return sum(  
        prob * (self.rewards[next_state] + DISCOUNT_FACTOR  
                * self.values[next_state])  
        for next_state, prob in self.  
            __get_transition_states_and_probs(state,  
            action)  
    )
```

- Implementa l'equazione di Bellman per calcolare il Q-value di una coppia stato-azione.

Funzione di transizione

Metodo `__get_transition_states_and_probs`

- Calcola le probabilità di transizione considerando la stocasticità dell'ambiente Pacman
- **Input:** Stato attuale e azione da intraprendere
- **Output:** List di tuple (stato successore, probabilità di transizione)

Collisioni con i Muri

Se un movimento perpendicolare porta a collidere con il muro, Pacman rimane fermo e la sua probabilità di rimanere fermo aumenta. (esempio: se i movimenti perpendicolari falliscono la probabilità di rimanere fermo è del $10\% + 10\% = 20\%$)

Aggiornamento dinamico delle ricompense

Metodo _update_rewards

- Aggiorna le ricompense basandosi sullo stato corrente del gioco
- Gestisce la natura non stazionaria dell'ambiente
- Considera:
 - Posizioni del cibo
 - Posizioni delle capsule
 - Posizioni dei fantasmi
 - Danger Zone
- Implementa la "Ghostbuster Mode" per gestire fantasmi vulnerabili

Ghostbuster Mode

Sfrutta le capsule energetiche incentivando il Pacman a seguire i fantasmi quando sono edibili andando ad invertire il segno della ricompensa.

```
def calculate_ghost_and_danger_zone_rewards(state,
    ghost_reward, danger_zone_reward):
    reward_multiplier = -1 if all(ghostState == 1 for
        ghostState in ghost_states(state)) else 1
    return ghost_reward * reward_multiplier,
        danger_zone_reward * reward_multiplier
```

Calcolo della Danger Zones

Modella la minaccia dei fantasmi

```
def get_danger_zones(state, pacman, ghosts, map_width, map_height, safety_distance):  
    danger_zone = set()  
    for ghost in ghosts:  
        if manhattanDistance(pacman, ghost) > safety_distance:  
            continue  
        for dx in range(-safety_distance, safety_distance + 1):  
            for dy in range(-safety_distance, safety_distance + 1):  
                x, y = int(ghost[0] + dx), int(ghost[1] + dy)  
                if 0 <= x < map_width and 0 <= y < map_height:  
                    danger_zone.add((x, y))  
    return list(danger_zone)
```

- Crea una "zona di pericolo" intorno ai fantasmi
- Considera solo i fantasmi entro una certa distanza da Pac-Man
- La `safety_distance` determina l'ampiezza delle danger zones

Selezione dell'azione

```
def getAction(self, game_state):
    self._update_rewards(game_state)
    self.value_iteration()
    pacman_pos = api.whereAmI(game_state)
    legal_actions = api.legalActions(game_state)
    legal_actions.remove(Directions.STOP)

    best_action = max(
        legal_actions,
        key=lambda action: self.__compute_q_value_from_values(pacman_pos, action),
    )
    return api.makeMove(best_action, legal_actions)
```

- Aggiorna le ricompense
- Esegue l'iterazione del valore
- Seleziona l'azione con il Q-value più alto

Fase di Test

Metodologia

- Variazione sistematica di un parametro alla volta

Setup degli esperimenti:

- Ambienti: SmallGrid (7x7) e MediumClassic (11x20)
- Numero di episodi per configurazione: 1000
- Parametri variati: `discount_factor`, `safety_distance`, `ghostbuster_mode`
- Metriche: `win_rate`
- Strumenti: Python con pandas per analisi dati, matplotlib per i grafici

Test sul Discount Factor

Test 1

Parametro variato: Discount Factor da 0 a 1 con passo 0.1. (1000 episodi).

smallGrid:

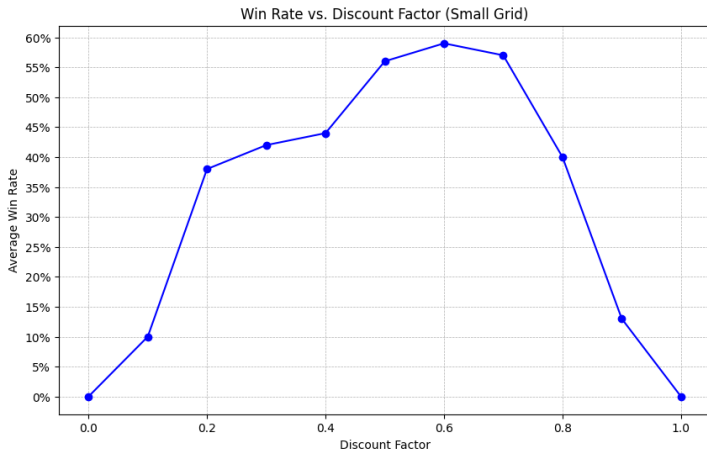
- Discount Factor ottimo = 0.6
- Win rate: 59%

mediumClassic:

- Discount Factor ottimo = 0.8
- Win rate: 40%

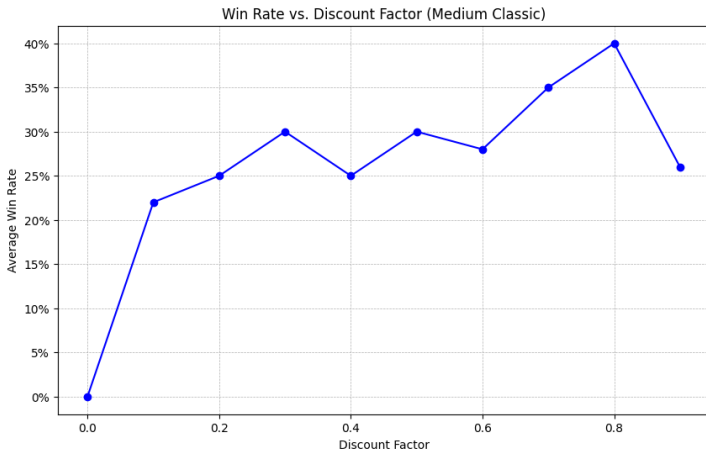
Test sul Discount Factor

Small Grid



Test sul Discount Factor

Medium Classic



Safety Distance e Ghostbuster Moder

Test 2

Parametri variati:

- **Small Grid:** `safety_distance = 0..4`
- **Medium Classic:** `safety_distance = 0..5` e `ghostbuster_mode = True/False`.

mediumClassic:

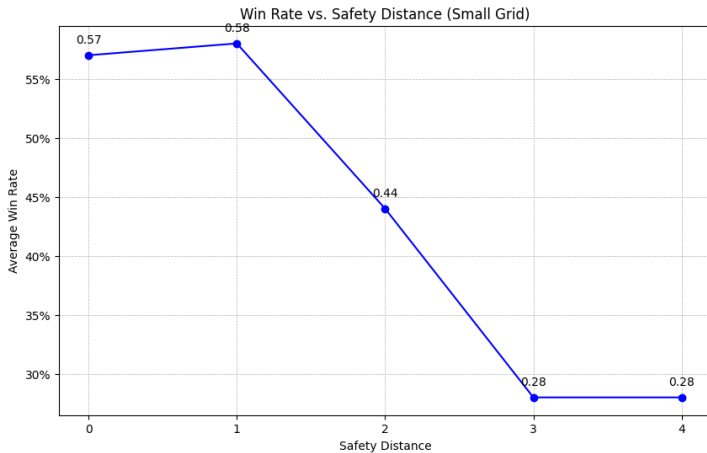
smallGrid:

- `safety_distance` ottima: 1
- Win rate: 58%

- `safety_distance` ottima: 3
- `ghostbuster_mode =`
indifferente
- Win rate: 34%

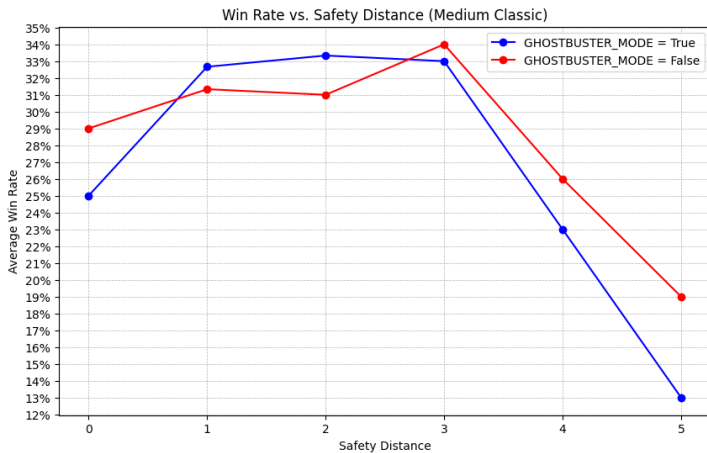
Test Safety Distance

Small Grid



Test Safety Distance e Ghostbuster Mode

Medium Classic



Configurazioni Ottimali

FOOD REWARD	10
GHOST REWARD	-500
DANGER ZONE REWARD	-250
CAPSULE REWARD	-
BLANK REWARD	-0.04
THETA	0.01
DISCOUNT FACTOR	0.8
SAFETY DISTANCE	1
MAX ITERATIONS	500
NOISE	0.2
Win Rate	58%

Tabella: smallGrid

FOOD REWARD	10
GHOST REWARD	-500
DANGER ZONE REWARD	-250
CAPSULE REWARD	50
BLANK REWARD	-0.04
THETA	0.01
DISCOUNT FACTOR	0.6
SAFETY DISTANCE	3
MAX ITERATIONS	500
GHOSTBUSTER MODE	Indifferente
NOISE	0.2
Win Rate	34%

Tabella: mediumClassic

Conclusioni

Risultati

- Value Iteration: Adattabile a diversi ambienti Pacman
- Importanza critica della configurazione dei parametri
- Benefici del discount factor moderato in ambienti complessi
- Vantaggio strategico nel mantenere distanza dai fantasmi
- Limitata efficacia della Ghostbuster mode

Osservazioni

- Ambienti più complessi richiedono un discount factor più alto (0.8 vs 0.6)
- La safety distance ottimale aumenta con la complessità dell'ambiente (3 vs 1)
- Il win rate diminuisce significativamente in ambienti più complessi (58% vs 34%)

Q-Learning in ambiente Pac-Man

Fondamenti del Q-learning

Equazione di Bellman per l'aggiornamento della Q-value

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

- S : stato corrente
- A : azione selezionata
- R : reward ricevuto dopo aver effettuato l'azione A partendo dallo stato S
- S' : nuovo stato raggiunto in seguito all'azione
- a : una qualsiasi azione eseguibile dallo stato S'
- α : *learning rate* dove $0 < \alpha \leq 1$
- γ : *discount factor* dove $0 \leq \gamma < 1$

Q-learning

Framework

1. **Definire l'ambiente:** si definiscono gli stati e le azioni possibili nell'ambiente in cui l'agente dovrà lavorare.
2. **Creazione Q-table:** si può creare ora la Q-table, tipicamente una matrice bidimensionale (*stati* \times *azioni*). Le Q-value in un primo momento sono inizializzate a zero oppure ad un piccolo valore random dipendentemente dall'ambiente.
3. **Aggiornamento Q-table:** la Q-table è aggiornata con le interazioni dell'agente con l'ambiente mediante l'equazione di Bellman.
4. **Scelta azioni tramite Q-table:** una volta che la Q-table è aggiornata essa può scegliere le azioni. Tipicamente l'azione scelta in uno stato è quella con il valore più alto di Q-values.

Vantaggi

- **Off-policy:** Q-learning è un algoritmo off-policy, questo permette all'agente di imparare da azioni esplorative, le quali non fanno necessariamente parte della policy corrente
- **Semplice implementazione:** richiede sostanzialmente il mantenimento delle Q-table e la gestione del loro aggiornamento mediale l'equazione di Bellman
- **Convergenza:** la convergenza alla policy ottimale è garantita se ogni coppia stato azione viene visitata tante volte
- **Robustezza:** il Q-learning può gestire ambienti con transizioni e reward stocastici

Svantaggi

- **Dimensioni Q-table:** con l'aumentare del numero di stati o di azioni, le dimensioni della Q-table possono crescere esponenzialmente. Questo può diventare insostenibile per ambienti con un grande numero di stati/azioni a causa della necessità di avere un'enorme disponibilità di memoria
- In ambienti con molti stati e azioni, la tabella Q può richiedere molto tempo per convergere ai valori ottimali, soprattutto perché ogni coppia stato-azione deve essere campionata sufficientemente per ottenere stime affidabili
- **Iperparametri:** le performance dipendono significativamente dalla scelta degli iperparametri. Una scelta errata comporta una convergenza lenta.

Aggiornamento valori

```
def update(self, state, action, nextState, reward):  
    """  
    The parent class calls this to observe a  
    state = action => nextState and reward transition.  
    You should do your Q-Value update here  
    """  
    Q_avg = self.getQValue(state, action)  
    q_update = Q_avg + self.alpha * (  
        reward + self.discount * self.getValue(nextState) - Q_avg  
    )  
    self.QValues[(state, action)] = q_update
```

Il metodo proposto aggiorna il valore della Q-value relativa a state e action secondo l'equazione di aggiornamento di Bellman

Calcolo azione da effettuare nello stato corrente (1)

```
def getAction(self, state):  
    """  
    Compute the action to take in the current state. With  
    probability self.epsilon, we should take a random action and  
    take the best policy action otherwise. Note that if there are  
    no legal actions, which is the case at the terminal state, you  
    should choose None as the action.  
    """  
  
    # Pick Action  
    legalActions = self.getLegalActions(state)  
    if not legalActions:  
        return None  
  
    action = None  
    return (  
        random.choice(legalActions)  
        if util.flipCoin(self.epsilon)  
        else self.getPolicy(state)  
    )  
  
    return action
```

Il metodo getAction funziona come segue:

1. Ottiene tutte le azioni possibili per lo stato corrente.

Calcolo azione da effettuare nello stato corrente (2)

2. Implementa una strategia epsilon-greedy per scegliere l'azione:
 - Con una probabilità ϵ , sceglie una azione casuale (esplorazione).
 - Con una probabilità $1 - \epsilon$, sceglie l'azione con il valore Q più alto (sfruttamento).
3. Restituisce l'azione scelta

Calcolo della migliore azione (1)

```
def computeActionFromQValues(self, state):  
    """  
    Compute the best action to take in a state. Note that if there  
    are no legal actions, which is the case at the terminal state,  
    you should return None.  
    """  
  
    legalActions = self.getLegalActions(state)  
    if not legalActions:  
        return None  
  
    bestQValue = float("-inf")  
    action_q = {}  
    for action in legalActions:  
        targetQValue = self.getQValue(state, action)  
        action_q[action] = targetQValue  
  
        if targetQValue > bestQValue:  
            bestQValue = targetQValue  
  
    bestAction = [k for k, v in action_q.items() if v == bestQValue]  
  
    return random.choice(bestAction)
```

Calcolo della migliore azione (2)

`computeActionFromQValues` calcola la migliore azione da intraprendere in uno stato dato utilizzando i valori `Q`. Iter del metodo:

1. **Ottenere Azioni Legali:** Il metodo inizia ottenendo tutte le azioni legali per lo stato corrente. Questo è importante perché non tutte le azioni possono essere valide in ogni stato (ad esempio, in uno stato terminale non ci sono azioni legali).
2. **Inizializzazione:**
 - `bestQValue` è inizializzato a meno infinito per garantire che qualsiasi valore `Q` trovato sarà maggiore di questo valore iniziale.
 - `action_q` è un dizionario che mapperà ogni azione al suo valore `Q` corrispondente.

Calcolo della migliore azione (3)

3. Calcolo dei Valori Q:

- Per ogni azione legale, il metodo calcola il valore Q chiamando `self.getQValue(state, action)`.
- Aggiorna il dizionario `action_q` con il valore Q calcolato.
- Se il valore Q calcolato è maggiore di `bestQValue`, aggiorna `bestQValue`.

4. Determinazione della Migliore Azione:

- Dopo aver iterato su tutte le azioni legali, il metodo trova tutte le azioni che hanno il valore Q uguale a `bestQValue`.
- Se ci sono più azioni con lo stesso valore Q massimo, ne sceglie una casualmente usando `random.choice`.

Questo metodo è fondamentale per determinare la politica ottimale in Q-learning, la quale massimizza il valore atteso a lungo termine.

Calcolo della state-value (1)

```
def computeValueFromQValues(self, state):  
    """  
    Returns max_action Q(state,action)  
    where the max is over legal actions. Note that if  
    there are no legal actions, which is the case at the  
    terminal state, you should return a value of 0.0.  
    """  
  
    if not self.getLegalActions(state):  
        return 0.0  
    return self.getQValue(state, self.getPolicy(state))
```

- **Controllo delle Azioni Legali:** il metodo inizia controllando se ci sono azioni legali disponibili per lo stato dato utilizzando `self.getLegalActions(state)`. Se non ci sono azioni legali (ad esempio, se lo stato è terminale), restituisce un valore di 0.0.

Calcolo della state-value (2)

- **Calcolo del Valore Q Massimo:** se ci sono azioni legali disponibili, il metodo procede a calcolare il valore Q massimo tra tutte le azioni legali. Questo viene fatto in due passaggi:
 1. **Determinazione della Politica Ottimale:** utilizza `self.getPolicy(state)` per ottenere l'azione con il valore Q massimo (la politica ottimale) per lo stato dato.
 2. **Restituzione del Valore Q:** utilizza `self.getQValue(state, action)` per ottenere il valore Q corrispondente a quella azione e lo restituisce.

Nel contesto dell'algoritmo di Q-learning, questo metodo è cruciale per determinare il valore atteso di uno stato, che è il massimo valore Q tra tutte le azioni possibili in quello stato. Questo valore viene utilizzato per aggiornare le stime dei valori Q durante il processo di apprendimento, aiutando l'agente a prendere decisioni ottimali basate sulle esperienze passate.

Q-value associato ad una specifica tupla stato-azione

```
def getQValue(self, state, action):  
    """  
    Returns Q(state,action)  
    Should return 0.0 if we have never seen a state  
    or the Q node value otherwise  
    """  
    return self.QValues[(state, action)]
```

Restituisce il valore $Q(s, a)$ per una data coppia stato-azione. Se non abbiamo mai visto lo stato, restituisce 0.0.

Approximate Q-learning

Framework

L'algoritmo Approximate Q-learning è una variante dell'algoritmo di Q-learning che utilizza funzioni di approssimazione per rappresentare i valori Q. Questo è particolarmente utile quando lo spazio degli stati è molto grande o continuo, rendendo impraticabile l'uso di una tabella Q per memorizzare i valori Q per tutte le coppie stato-azione.

1. Rappresentazione dei Valori Q:

- Invece di memorizzare i valori Q in una tabella, Approximate Q-learning utilizza una funzione di approssimazione, come una combinazione lineare di caratteristiche (features) dello stato e dell'azione.
- La funzione di approssimazione può essere rappresentata come:
$$Q(s, a) = \sum_i w_i \cdot f_i(s, a)$$
dove $(f_i(s, a))$ sono le caratteristiche (features) dello stato e dell'azione, e (w_i) sono i pesi associati a queste caratteristiche.

Approximate Q-learning (2)

2. Inizializzazione:

- I pesi w_i sono inizializzati a valori casuali o a zero.
- Le caratteristiche $f_i(s, a)$ sono definite in base al problema specifico.

3. Aggiornamento dei Valori Q: Quando l'agente esegue un'azione (a) in uno stato (s) e osserva la ricompensa (r) e il nuovo stato (s'), aggiorna i pesi (w_i) utilizzando la seguente formula:

$w_i \leftarrow w_i + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)) \cdot f_i(s, a)$ dove:

- α è il tasso di apprendimento.
- γ è il fattore di sconto.
- $r + \gamma \cdot \max_{a'} Q(s', a')$ è il target di apprendimento.
- $Q(s, a)$ è il valore Q corrente.

Approximate Q-learning (3)

4. **Selezione delle Azioni:** L'agente utilizza una strategia epsilon-greedy per selezionare le azioni. Con una probabilità ϵ , sceglie un'azione casuale (esplorazione), e con una probabilità $1 - \epsilon$, sceglie l'azione con il valore Q massimo (sfruttamento).

Value Iteration vs Q-learning

Analisi temporale e spaziale

	Value Iteration	Q-learning
Temporale	$O(k \cdot S ^2 \cdot A)$	$O(n \cdot A)$
Spaziale	$O(S)$	$O(S \cdot A \cdot H)$

- k : numero iterazioni, $|S|$: cardinalità stati, $|A|$: numero azioni
- n : passi apprendimento, $|H|$: orizzonte temporale
- $|S| \cdot |A|$: memorizza in ogni stato le azioni possibili