UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

Progetto per il corso di Intelligenza Artificiale

Applicazione di Value Iteration e Q-learning in ambiente Pacman

Dumas Nicholas
Matteo Rossini

Indice

Capitolo 1

Introduzione

L'elaborato presentato in questa relazione è mirato a sperimentare i fondamenti del *Reinforcement Learning* in un ambiente non deterministico utilizzando a tale scopo il classico gioco arcade del Pacman. Tale progetto offre l'opportunità di applicare i concetti teorici visti durante il corso ad un problema pratico, permettendo inoltre di confrontare diverse tecniche di apprendimento per rinforzo all'interno di un contesto ben definito e di facile comprensione.

1.1 Obiettivi del Progetto

- Risolvere i quesiti presentati nel corso di Intelligenza Artificiale di Berkeley, con particolare attenzione all'implementazione di un agente basato su *Q-learning*.
- Sviluppare un agente Pacman basato sull'algoritmo *Value Iteration* nell'ambiente di gioco Pacman per confrontare i risultati ottenuti con l'algoritmo di *Q-Learning* in modo da valutare i pro e i contro di ciascun metodo.

1.2 Implementazioni chiave

- **Q-Learning**: un metodo model-free che apprende una funzione di valore azione-stato (*Q-function*) attraverso l'interazione diretta con l'ambiente. Questo approccio è particolarmente adatto per ambienti in cui il modello di transizione non è noto a priori.
- **Value Iteration**: Un algoritmo model-based che calcola la funzione di valore ottimale attraverso iterazioni successive. Questo metodo richiede una conoscenza completa del modello dell'ambiente, ma può convergere più rapidamente alla policy ottimale.

1.3 Pacman: il Gioco

Pacman è un videogioco arcade sviluppato da Namco nel 1980, divenuto un'icona della cultura pop e un interessante banco di prova per l'intelligenza artificiale.

1.3.1 Regole del Gioco

- Il giocatore controlla Pacman, rappresentato da una sfera gialla.
- L'obiettivo principale è mangiare tutto il cibo sparso sulla mappa di gioco.
- Pacman deve evitare i fantasmi presenti nella mappa.
- Se Pacman si trova nella stessa cella di un fantasma, perde una vita, e il gioco riparte da capo.
- Sulla mappa, possono essere presenti delle capsule energetiche che permettono temporaneamente a Pacman di mangiare i fantasmi, guadagnando punti extra.
- Il giocatore vince quando Pacman riesce a mangiare tutto il cibo presente nel labirinto.

Capitolo 2

Value Iteration

Il seguente capitolo presenta un'applicazione dell'algoritmo **Value Iteration** all'ambiente non stazionario del gioco Pac-Man. L'obiettivo è sviluppare un agente autonomo, capace di apprendere una **policy ottimale** per muoversi nell'ambiente di gioco al fine di massimizzare le ricompense e conseguire un elevato tasso di vittorie.

2.1 Introduzione

In questo capitolo, studiamo l'applicazione del Value Iteration (VI), un algoritmo RL model-based, al dominio Pac-Man. I nostri obiettivi principali sono:

1. Implementare e ottimizzare un agente basato su VI per l'ambiente Pac-Man.
2. Analizzare l'impatto dei parametri chiave sulle prestazioni dell'agente.
3. Individuare delle strategie per adattare VI ad ambienti non stazionari.

2.2 Formulazione del problema

2.2.1 Processo Decisionale di Markov (MDP)

Per applicare VI, modelliamo il gioco Pac-Man come un problema MDP definito dalla tupla (S, A, R, P, γ), dove:

- S: Spazio degli stati, rappresentate tutte le possibili configurazioni di gioco (posizione dei fantasmi, pac-man, cibo, pillole energetiche).
- A: Spazio delle azioni $A = \{\text{nord}, \text{sud}, \text{est}, \text{ovest}\}$

- $R : S \times A \times S \to \mathbb{R}$: Funzione di ricompensa $R(s, a, s')$ progettata per incentivare i comportamenti desiderati.
- $P : S \times A \times S \to [0, 1]$: 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.

2.2.2 Caratteristiche dell'Ambiente Pac-Man

L'ambiente Pac-Man presenta diverse caratteristiche che lo rendono particolarmente interessante e sfidante per l'applicazione di algoritmi di apprendimento per rinforzo:

1. **Stocasticità**: Le azioni dell'agente sono soggette a incertezza. Esiste una probabilità dell'80% che Pac-Man si muova nella direzione intesa, mentre c'è una probabilità del 10% di muoversi in ciascuna delle due direzioni perpendicolari. Questa caratteristica introduce un elemento di imprevedibilità che l'agente deve imparare a gestire.
2. **Osservabilità completa**: L'agente ha accesso a tutte le informazioni rilevanti dello stato del gioco in ogni momento. Questo include la posizione di Pac-Man, dei fantasmi, del cibo e delle capsule energetiche. Tuttavia, la complessità dell'ambiente rende comunque difficile l'elaborazione e l'utilizzo ottimale di queste informazioni.
3. **Non stazionarietà**: L'ambiente è intrinsecamente non stazionario a causa del movimento continuo dei fantasmi. Questo implica che le probabilità di transizione e le ricompense associate a determinate azioni possono cambiare nel tempo. Questa caratteristica avrà un impatto notevole sulle prestazioni di Value Iteration e più in generale sugli algoritmi model-base che tendono a fare affidamento su un modello statico dell'ambiente.
4. **Episodicità**: Il gioco Pac-Man è per sua natura episodico, con stati terminali ben definiti (vittoria quando tutto il cibo è stato consumato, o sconfitta quando Pac-Man viene catturato da un fantasma).
5. **Ambiente multi-obiettivo:** L'agente deve bilanciare obiettivi multipli e talvolta contrastanti, come massimizzare il punteggio, evitare i fantasmi, e completare il livello il più rapidamente possibile.
6. **Multi-agente:** L'ambiente è intrinsecamente multi-agente. I fantasmi possono essere considerati agenti avversari con i propri comportamenti e obiettivi.

7. **Dimensionalità variabile:** La complessità dell'ambiente può variare significativamente tra diverse mappe (come evidenziato dalla differenza tra *SmallGrid* e *MediumClassic*, influenzando direttamente la dimensione dello spazio degli stati e la difficoltà del problema di apprendimento.

2.3 Introduzione al Value Iteration

2.3.1 Formulazione matematica

Il Value Iteration (VI) è un algoritmo fondamentale nell'apprendimento per rinforzo, particolarmente adatto per problemi modellati come Processi Decisionali di Markov (MDP) con spazi degli stati e delle azioni discrete e finiti. Nel contesto del gioco Pac-Man, il VI offre un approccio sistematico per calcolare la funzione valore ottimale e, conseguentemente, derivare una politica ottimale per l'agente.

Il Value Iteration si basa sull'equazione di Bellman, che definisce ricorsivamente il valore di uno stato in termini del valore degli stati futuri. VI procede iterativamente, aggiornando i valori di ogni stato fino a convergere alla funzione valore ottimale $v^*(s)$, $s \in S$:

Equazione di Bellman per la funzione valore-stato

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

Dove:

- $v_\pi(s)$: valore atteso dello stato s seguendo la policy π
- $\pi(a|s)$: probabilità di scegliere l'azione a nello stato s secondo la policy π
- $P(s'|s,a)$: probabilità di transizione dallo stato s allo stato s' data l'azione a
- $R(s,a,s')$: ricompensa immediata per la transizione da s a s' che si è intrapresa con l'azione a
- γ:fattore di sconto

Equazione di Bellman per la funzione valore ottimale

L'obiettivo di VI è trovare la funzione valore ottimale $v^*(s)$, che rappresenta il massimo valore ottenibile da qualsiasi policy:

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

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

Questa equazione cattura l'idea che il valore ottimale di uno stato è il massimo valore ottenibile considerando tutte le possibili azioni e le conseguenti transizioni di stato.

Risoluzione dell'equazione di Bellman

La risoluzione diretta dell'equazione di Bellman, ad esempio mediante il metodo di eliminazione di Gauss (MEG), comporta elevati costi computazionali, in particolare per problemi di dimensioni non triviali come l'ambiente Pac-Man. In particolare, per un MDP con $|S|$ stati la risoluzione diretta del sistema di equazioni lineari richiede $O(|S|^3)$ operazioni. Nel contesto di Pac-Man dove $|S|$ può essere dell'ordine di $10^4 - 10^6$ (considerando tutte le possibili configurazioni del labirinto), questo approccio diventa computazionalmente proibitivo. I metodi diretti necessitano inoltre di $O(|S|^2)$ spazio per memorizzare la matrice completa delle transizioni.

Per superare questi limiti, Il Value Iteration, come metodo iterativo, offre diversi vantaggi significativi rispetto ai metodi diretti.

La complessità temporale di VI è $O(|S|^2|A|$, dove $|A|$ è il numero di azioni. Tuttavia, il numero di iterazioni necessarie per la convergenza è spesso indipendente da $|S|$, il risultato è una complessità effettiva inferiore rispetto ai metodi diretti per grandi spazi degli stati.

La complessità temporale di VI è solo $O(|S|)$, ovvero lo spazio per memorizzare la funzione valore corrente.

Applicazione a Pac-Man

Nel contesto di Pac-Man, gli stati rappresentano le configurazioni del labirinto, includendo la posizione di Pac-Man, dei fantasmi, del cibo e delle capsule. Le azioni sono i movimenti possibili (nord,sud,est,ovest).

Per esempio, consideriamo uno stato in cui Pac-Man è adiacente a un fantasma. Il Value Iteration assegnerà un valore basso a questo stato, poiché la probabilità di perdere (e quindi ricevere una ricompensa negativa) è alta. D'altra parte, uno stato in cui Pac-Man è vicino a molto cibo e lontano dai fantasmi avrà un valore più alto.

Regola di aggiornamento della Value Iteration

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

L'algoritmo si basa sull'equazione di Bellman, alternando fasi di valutazione e miglioramento della policy fino a convergere alla funzione valore ottimale.

1. **Inizializzazione**: Si inizializza arbitrariamente $v_0(s)$ per tutti gli stati $s \in S$.

2. **Iterazione del valore**: Per $k = 0, 1, 2, ...$, 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 $k + 1$

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.

La Value Iteration alterna fasi di valutazione della funzione valore corrente con fasi di miglioramento implicito della policy, convergendo simultaneamente verso la funzione valore ottimale e la policy ottimale.

Una caratteristica fondamentale di VI è la sua capacità di calcolare iterativamente V^* senza la necessità di memorizzare esplicitamente una policy. Questo approccio risulta vantaggioso in termini di efficienza spaziale, rendendolo adatto a problemi con spazio degli stati di notevoli dimensioni, come nel caso del dominio Pac-Man.

L'implementazione dell'algoritmo richiede la definizione di un soglia d'arresto ϵ. Nella nostra applicazione abbiamo empiricamente determinato che $\epsilon = 0.01$ offre un equilibrio ottimale tra accuratezza della soluzione e costo computazionale.

2.3.2 Definizione delle ricompense

La funzione di ricompensa R è stata progettata per incentivare comportamenti desiderati nell'agente Pac-Man attraverso un sistema di ricompense e penalità. In particolare:

2.4 Implementazione

La nostra implementazione dell'algoritmo Value Iteration (VI) per l'ambiente Pac-Man è stata realizzata attraverso la classe MDPAgent. Questa classe incapsula la logica dell'agente e implementa i metodi fondamentali per l'esecuzione dell'algoritmo VI. Di seguito, presentiamo una descrizione dettagliata dei componenti chiave:

FOOD REWARD	10
GHOST REWARD	-500
DANGER ZONE REWARD	-250
CAPSULE REWARD	-
BLANK REWARD	-0.04
THETA	0.01
DISCOUNT FACTOR	0.6
SAFETY DISTANCE	1
MAX ITERATIONS	500
GHOSTBUSTER MODE	-
NOISE	0.2

Tabella 2.1: Tabella con le ricompense

2.4.1 Inizializzazione dell'agente

```
def __init__(self):
    self.max_iterations = MAX_ITERATIONS
    self.noise = NOISE
    self.values = util.Counter()
    # ... altre inizializzazioni
```

Listato 2.1: Inizializzazione classe MDPAgent

Il parametro `max_iterations` limita il numero massimo di iterazioni per l'algoritmo VI, mentre `noise` rappresenta la stocasticità dell'ambiente. La variabile `values` è di tipo `Counter` che estende il tipo `Dictionary` di Python e viene utilizzata per memorizzare i valori della funzione valore durante l'iterazione dei valori. Di Default ogni stato ha valore pari a zero.

2.4.2 Value Iteration

Il metodo principiale dell'algoritmo VI è implementato come segue:

```
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
```

Listato 2.2: metodo value_ iteration

Questo metodo implementa l'algoritmo di Value Iteration, iterando su tutti gli stati legali e aggiornando i loro valori. L'algoritmo termina quando la variazione massima (delta) scende sotto una soglia predefinita `THETA`) o al raggiungimento del numero massimo di iterazioni `max_iterations`.

2.4.3 Calcolo della policy ottimale

Il metodo per calcolare la policy ottimale è implementato come:

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

Listato 2.3: Metodo _get_best_policy

Questo metodo determina l'azione ottimale per un dato stato, selezionando quella con il massimo Q-value.

2.4.4 Calcolo dei Q-Values

Il calcolo dei Q-values è implementato come segue:

```
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)
    )
```

Listato 2.4: Metodo __compute_q_value_from_values

Questo metodo implementa l'equazione di Bellman per calcolare il Q-value di una coppia stato-azione, considerando le probabilità di transizione e le ricompense.

2.4.5 Metodo di transizione

Il modello di transizione dell'ambiente è implementato nel metodo:

```
def __get_transition_states_and_probs(self, state, action):
        x, y = state
        dx, dy = self.move_offsets[action]
        intended_next_state = self._next_state(state, self.move_offsets
    [action])

        transitions = [(intended_next_state, 1 - self.noise)]

```

```
        perpendicular_actions = (
            [Directions.EAST, Directions.WEST]
            if action in [Directions.NORTH, Directions.SOUTH]
            else [Directions.NORTH, Directions.SOUTH]
        )
        for side_action in perpendicular_actions:
            dx, dy = self.move_offsets[side_action]
            side_state = (x + dx, y + dy)
            if side_state in self.wall_positions:
                transitions.append((state, self.noise / 2))
            else:
                transitions.append((side_state, self.noise / 2))

        return transitions
```

Listato 2.5: Metodo _get_transition_states_and_probs

Questo metodo modella la dinamica dell'ambiente, restituendo una lista di tuple (stato successivo, probabilità) data una coppia stato-azione.

2.4.6 Aggiornamento dinamico delle Ricompense

Per gestire la non stazionarietà dell'ambiente Pac-Man, abbiamo implementato un metodo di aggiornamento dinamico delle ricompense:

```
    def _update_rewards(self, game_state):
        food_positions = set(api.get_food(game_state))
        capsule_positions = set(api.get_capsules(game_state))
        ghost_positions = set(api.get_ghosts(game_state))

        danger_zones = set(
            api.get_danger_zones(
                game_state,
                api.whereAmI(game_state),
                ghost_positions,
                self.map_width,
                self.map_height,
                SAFETY_DISTANCE,
            )
        )

        blank_positions = (
            self.legal_states
            - food_positions
            - capsule_positions
            - ghost_positions
            - danger_zones
        )

```

```
25          ghost_reward, danger_reward = GHOST_REWARD, DANGER_ZONE_REWARD
26
27          if self.ghostbuster_mode:
28              ghost_reward, danger_reward = api.
    calculate_ghost_and_danger_zone_rewards(
29                  game_state, GHOST_REWARD, DANGER_ZONE_REWARD
30              )
```

Listato 2.6: Metodo _update_rewards

Ghostbuster Mode

La Ghostbuster Mode è un meccanismo adattivo che modifica il comportamento dell'agente quando i fantasmi diventano vulnerabili (edibili), per un certo lasso di tempo, dopo il consumo di una pillola energetica. Questo cambia temporaneamente la dinamica del gioco, trasformando i fantasmi da minacce in opportuna di punteggio.

La Ghostbuster Mode è implementata attraverso la seguente funzione:

```
1 def calculate_ghost_and_danger_zone_rewards(state, ghost_reward,
    danger_zone_reward):
2
3     reward_multiplier = (
4
5         -1 if all(ghostState == 1 for ghostState in
     ghost_states(state)) else 1
6
7     )
8
9     return ghost_reward * reward_multiplier,
    danger_zone_reward * reward_multiplier
```

Listato 2.7: Metodo calculate_ghost_and_danger_zone_rewards

La funzione inverte dinamicamente il segno delle ricompense associate ai fantasmi e alle zone di pericolo quanto tutti i fantasmi sono vulnerabili.

Danger Zone

La Danger Zone rappresenta un'area di rischio potenziale intorno ai fantasmi. Questo concetto estende la nozione di pericolo oltre la semplice collisione diretta con i fantasmi.

La Danger Zone è implementata attraverso il seguente metodo:

```
1 self.rewards.update({pos: ghost_reward for pos in ghost_positions})
```

Calcolo della danger zone

:

```python
def get_danger_zones(state, pacman, ghosts, map_width, map_height,
    safety_distance):
    # Returns a list of unique (x, y) pairs of positions that are
    within
    # "safe_distance" of any ghost, but only if Pacman is also within
    # "safe_distance" of the ghost.

    danger_zone = set()

    for ghost in ghosts:
        if manhattanDistance(pacman, ghost) > safety_distance:
            continue

        for dx in range(-safe_distance, safety_distance + 1):
            for dy in range(-safe_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)
```

Listato 2.8: Metodo get_danger_zones

Caratteristiche principali:

- La funzione crea una zona di pericolo intorno a ogni fantasma entro una certa `safety_distance` da Pac-Man.
- Utilizza la distanza di Manhattan per determinare la prossimità di Pac-Man rispetto ai fantasmi.

Implicazioni per il comportamento dell'agente

- **Adattabilità:** Questi meccanismi permettono all'agente Pac-Man di adattarsi dinamicamente all'ambiente non-stazionario.
- **Bilanciamento rischio-ricompensa**: La combinazione di Danger Zone e Ghostbuster Mode permette all'agente di bilanciare il rischio di avvicinarsi ai fantasmi con l'opportunità di punteggio.

2.4.7 Selezione dell'azione

Il metodo `getAction` seleziona l'azione da eseguire:

```python
  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)
```

Listato 2.9: Metodo getAction

Questo metodo aggiorna le ricompense, esegue l'iterazione del valore, e seleziona l'azione con il valore Q più alto tra quelle legali, ovvero l'azione ottimale da eseguire nello stato corrente del gioco.

2.5 Test

In questa sezione, analizziamo in modo empirico l'algoritmo Value Itertion (VI) applicato al dominio Pac-Man. L'obiettivo principale è valutare l'impatto di parametri chiave sulle prestazioni dell'agente, con particolare attenzione al fattore di sconto (), alla distanza di sicurezza, e alla modalità "ghostbuster".

2.5.1 Metodologia

Abbiamo adottato un approccio sistematico, variando un parametro alla volta mentre mantenevamo costanti gli altri. Per ogni configurazione, abbiamo condotto 1000 episodi.

Le sperimentazioni sono state condotte su due mappe distinte: smallGrid e mediumClassic, per valutare la robustezza dell'algoritmo in scenari di complessità crescente.

I dati sono stati raccolti in formato CSV e successivamente analizzati utilizzando Python, con l'ausilio delle librerie pandas per la manipolazione dei dati e matplotlib per la visualizzazione grafica.

2.5.2 Test sul Discount factor

Abbiamo fatto variare il `DISCOUNT_FACTOR` nell'intervallo [0, 1] con incrementi di 0.1, eseguendo 1000 episodi per ciascun valore in entrambe le mappe.

Test 01 - discount factor (smallGrid)

Il grafico in Figura 2.1 illustra la relazione tra il `discount_factor` e il `win_rate`. Il valore massimo si osserva a `discount_factor = 0.6`, con un `win_rate` del 59%

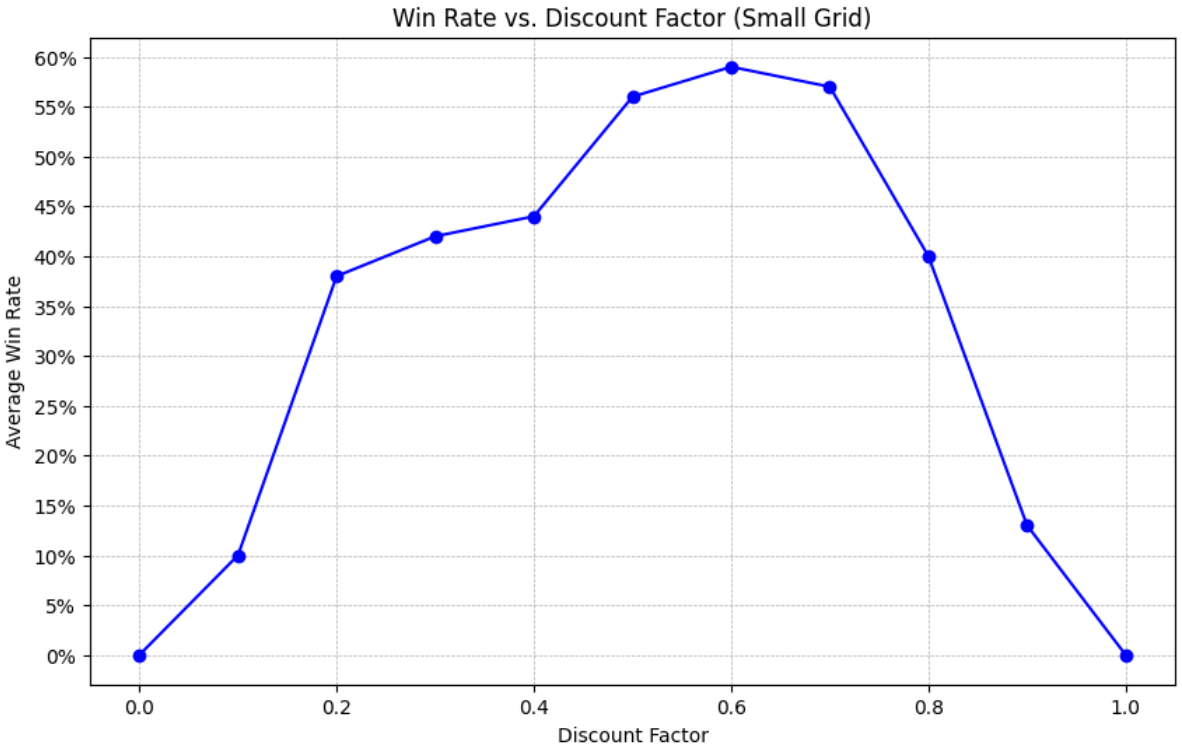


Figura 2.1: Grafico a linee per il discount factor nella smallGrid

FOOD REWARD	10
GHOST REWARD	-500
DANGER ZONE REWARD	-250
CAPSULE REWARD	-
BLANK REWARD	-0.04
THETA	0.01
DISCOUNT FACTOR	0:0.1:.1
SAFETY DISTANCE	1
MAX ITERATIONS	500
GHOSTBUSTER MODE	-
NOISE	0.2

Tabella 2.2: Tabella con i parametri di test per la smallGrid

Test 03 - discount factor (mediumClassic)

Anche nel caso dell'ambiente *Medium Classic* possiamo notare che l'algoritmo di Value Iteration è fortemente influenzato dal *discount factor*. Un *discount factor* di 0.8 sembra

ottimizzare le prestazioni dell'algoritmo, portando ad un *win rate* del 40%. Possiamo quindi dire che nel *Medium Classic* l'algoritmo tende a pensare di più le ricompense future rispetto alla *Small Grid*.

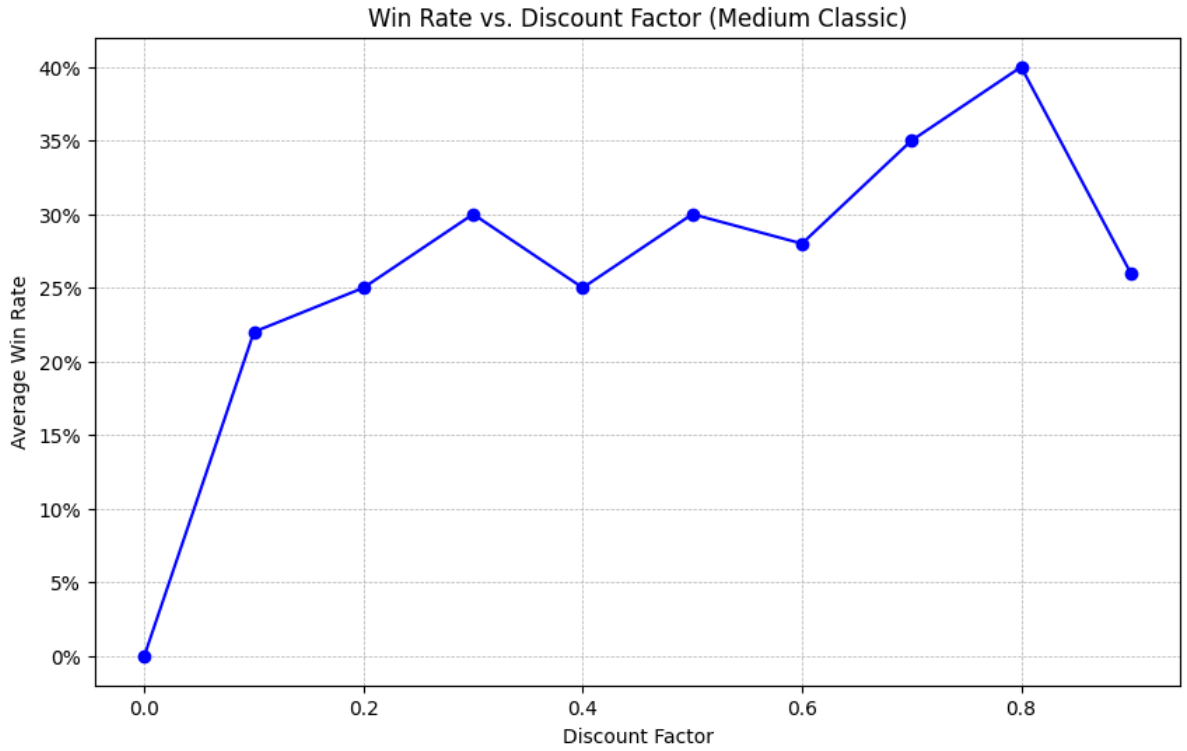


Figura 2.2: Grafico a linee per il discount factor nella mediumClassic

FOOD REWARD	10
GHOST REWARD	-500
DANGER ZONE REWARD	-250
CAPSULE REWARD	-
BLANK REWARD	-0.04
THETA	0.01
DISCOUNT FACTOR	0:0.1:.1
SAFETY DISTANCE	3
MAX ITERATIONS	500
GHOSTBUSTER MODE	TRUE
NOISE	0.2

Tabella 2.3: Tabella con i parametri di test per la mediumClassic

2.5.3 Test sulla safety distance e la ghostbuster mode

In questo test ci concentriamo sull'impatto che ha la *safety distance* e la *ghostbuster mode* sulle prestazioni dell'algoritmo di *Value Iteration*.

Test 03 - safety distance (smallGrid)

Abbiamo variato la distanza di sicurezza da 0 a 4, mantenendo gli altri parametri costanti. La modalità ghostbuster non è applicabile in questo ambiente privo di capsule energetiche.

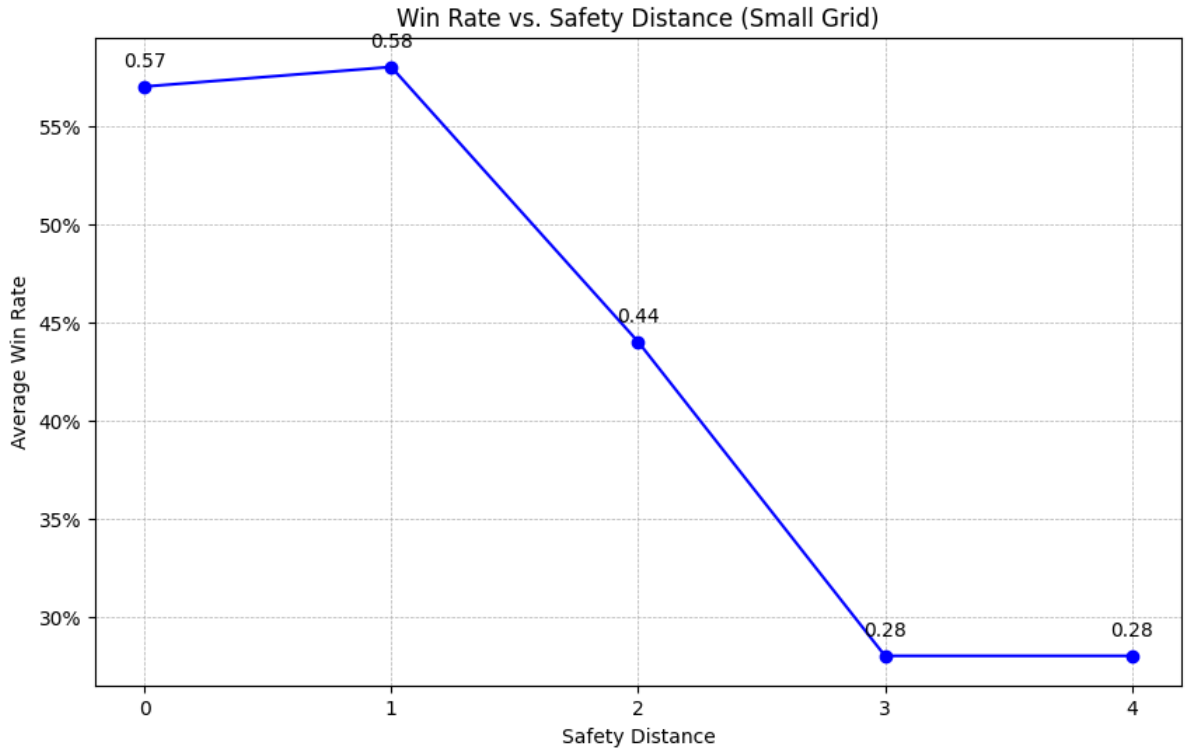


Figura 2.3: Grafico a linee per la safety distance (smallGrid)

Dai dati raccolti emerge che una *safety distance* impostata a 1 ottimizza il *win rate* a 0.58, superiore di un punto percentuale rispetto alla configurazione con *safety distance* nulla. Questo incremento, seppur modesto, suggerisce che una minima distanza di sicurezza tra il pacman e il fantasma può migliorare le performance dell'algoritmo.

Test 04 - safety distance (MediumClassic)

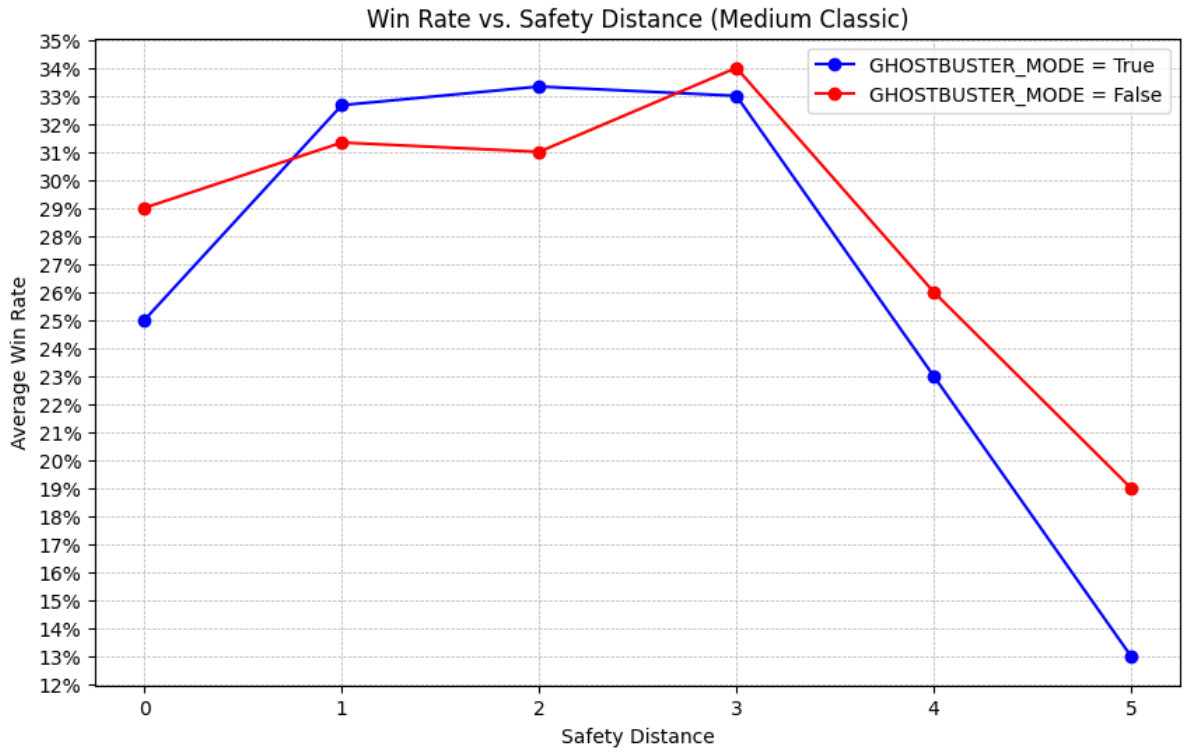


Figura 2.4: Grafico a linee per la safety distance e il ghostbuster mode (mediumClassic)

Il test nell'ambiente *medium classic* ha esplorato l'impatto della *safety distance* che è stata varietà di una unità partendo da una distanza nulla fino a distanza 4.

In particolare, si è osservato il comportamento dell'agente Pac-Man in due scenari distinti: con la *ghostbuster mode* attiva, che incentiva l'agente Pac-man a cacciare i fantasmi quando sono edibili, e quando risulta disattivata.

L'analisi dei dati raccolti non ha rilevato differenze statisticamente rilevanti tra i due scenari, suggerendo che l'attivazione della *ghostbuster mode* non influisca significativamente sulle prestazioni dell'agente. Tuttavia, si è notato che impostando la *safety distance* a 3, il *win rate* raggiunge circa il 34%.

2.6 Conclusioni

L'implementazione dell'algoritmo di Value Iteration per l'agente Pac-Man ha prodotto risultati significativi, evidenziando l'importanza di una corretta configurazione dei para-

metri per ottimizzare le prestazioni in ambienti di gioco diversi. Le nostre analisi hanno rilevato che:

- **Adattabilità dell'Algoritmo:** L'algoritmo ha richiesto una calibrazione specifica per ciascun scenario per raggiungere le massime prestazioni possibili.
- **Impatto del Discount Factor:** In ambienti più complessi, è emerso che un discount factor moderato, che attribuisce maggior peso alle ricompense future, porta a prestazioni superiori.
- **Rilevanza della Safety Distance:** Mantenere una distanza di sicurezza dai fantasmi si è rivelato vantaggioso, consentendo un equilibrio efficace tra l'evitamento dei pericoli e l'acquisizione di ricompense.
- **Efficacia Limitata della Ghostbuster Mode:** Contrariamente alle aspettative, l'attivazione della ghostbuster mode non ha portato a miglioramenti significativi delle prestazioni nell'ambiente mediumClassic, suggerendo potenziali inefficienze nell'implementazione o nella definizione delle ricompense associate.

2.6.1 Configurazioni Ottimali

Le configurazioni parametriche che hanno prodotto i risultati migliori sono riassunte nelle Tabelle 1 e 2 per gli ambienti smallClassic e mediumClassic rispettivamente.

FOOD REWARD	10
GHOST REWARD	-500
DANGER ZONE REWARD	-250
CAPSULE REWARD	-
BLANK REWARD	-0.04
THETA	0.01
DISCOUNT FACTOR	0.6
SAFETY DISTANCE	1
MAX ITERATIONS	500
NOISE	0.2
Win Rate	0.58%

Tabella 2.4: Configurazione ottimale per la smallClassic

FOOD REWARD	10
GHOST REWARD	-500
DANGER ZONE REWARD	-250
CAPSULE REWARD	50
BLANK REWARD	-0.04
THETA	0.01
DISCOUNT FACTOR	0.8
SAFETY DISTANCE	3
MAX ITERATIONS	500
GHOSTBUSTER MODE	Indifferente
NOISE	0.2
Win Rate	34%

Tabella 2.5: Configurazione ottimale per la mediumClassic

2.6.2 Limitazioni e Prospettive Future

Nonostante i risultati ottenuti, non possiamo escludere che l'implementazione ottenuta presenta dei margini di miglioramento. Le direzioni per ottimizzare l'implementazione di VI potrebbero includere:

1. **Raffinamento della Ghostbuster Mode:** Una ridefinizione delle ricompense associate alla ghostbuster mode potrebbe portare a miglioramenti significativi nelle prestazioni.

2. **Danger Zones Dinamiche:** L'implementazioni di danger zone più sofisticate, capaci di adattarsi dinamicamente alle condizioni di gioco potrebbe migliorare le prestazioni.

In conclusione, questo studio ha dimostrato l'efficacia del Value Iteration nel dominio del Pac-Man, evidenziando al contempo la complessità intrinseca nell'ottimizzazione di agenti per ambienti di gioco dinamici. Le prospettive future delineate promettono di espandere ulteriormente la nostra comprensione e l'applicabilità di questo approccio in scenari di reinforcement learning più ampi e complessi.

Capitolo 3

Q-learning

3.1 Introduzione

Il *Q-Learning* è un algoritmo del campo del reinfocement learning che si distingue per la sua capacità di apprendere strategie ottimali senza richiede un modello esplicito dell'ambiente.

La *"Q"* sta per *"Qualità"*; ovvero quanto è valida l'azione nel massimizzare i futuri reward.

3.2 Fondamenti Teorici

Il Q-learning è caratterizzato da tre attributi principali:

1. Model-free
2. Valued-based
3. Off-policy

3.2.1 Model-free vs Model-based

Model-free

- Il Q-learning opera senza una rappresentazione esplicita delle dinamiche dell'ambiente.
- apprende direttamente dalle conseguenze delle azioni, senza necessità di conoscere le funzioni di transizione e di reward.

Model-based

- Gli algoritmi model-based come ad esempio Value Iteration utilizzano un modello esplicito dell'ambiente.
- Richiedono la conoscenza delle probabilità di transizione e delle funzioni di reward.

Implicazioni Il Q-learning è più adattabile a ambienti sconosciuti o in rapido cambiamento, mentre gli approcci model-based possono essere più efficienti quando il modello dell'ambiente è noto e accurato.

3.2.2 Value-based vs Policy-based

Value-based

- Il Q-learning stima il valore delle azioni in diversi stati.
- La policy è derivata implicitamente dai valori Q stimati.

Policy-based

- Metodi policy-based lavorano direttamente sulla policy.
- Apprendono direttamente quale azione intraprendere in ogni specifico stato.

Confronto: I metodi value-based come il Q-learning possono essere più stabili nell'apprendimento, ma i metodi policy-based possono essere più efficaci in spazi di azione continui o di grandi dimensioni

3.2.3 On-policy vs Off-policy

Off-policy

- il Q-learning apprende la policy ottimale indipendentemente dalla policy di esplorazione.

On-policy

- Algoritmi come SARSA valutano e migliorano la stessa policy usata per generare il comportamento.

Vantaggi dell'approccio Off-policy: Maggiore flessibilità nell'esplorazione e capacità di apprendere da esperienze passate.

3.3 Implementazione e Funzionamento

3.3.1 Q-table e Aggiornamento

Il Q-learning mantiene una Q-table che rappresenta il valore stimato di ogni coppia stato-azione. L'aggiornamento della Q-values avviene secondo la formula:

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

dove:

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

Le Q-values rappresentano l'utilità attesa di selezionare una determinata azione partendo da uno stato specifico e seguendo la policy ottimale successivamente.

Inizialmente i valori sono inizializzati con valori arbitrari per poi essere successivamente aggiornati iterativamente sfruttando l'esperienza acquisita dall'agente.

3.3.2 Esplorazione vs Sfruttamento

Il Q-learning tipicamente utilizza strategie come ϵ-greedy per bilanciare esplorazione e sfruttamento:

- Con probabilità ϵ: scelta casuale dell'azione (esplorazione)
- Con probabilità $1 - \epsilon$: scelta dell'azione con il massimo Q-value (sfruttamento)

3.4 Algoritmo Q-learning

3.4.1 Primo step: definire l'ambiente

Dapprima, si definiscono gli stati e le azioni possibili nell'ambiente in cui l'agente dovrà lavorare.

- Stati: tutte le possibili situazioni che l'agente può incontrare
- Azioni: le mosse che l'agente può compiere in ciascun stato

3.4.2 Secondo step: creazione Q-table

Una volta definiti gli stati e le azioni è possibile creare una Q-table, tipicamente una matrice bidimensionale ($stati \times azioni$, come spiegato precedentemente).

Le Q-value in un primo momento sono inizializzate a zero oppure ad un piccolo valore random, dipendentemente dall'ambiente e dall'algoritmo specifico adottato. Alcuni ambienti infatti beneficiano di una partenza con valori random per incoraggiare una esplorazione iniziale.

3.4.3 Terzo step: aggiornamento Q-table

La Q-table è aggiornata con le interazioni dell'agente con l'ambiente mediante la formula esposta precedentemente.

3.4.4 Quarto step: 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 (sfruttamento). Esistono comunque varianti che bilanciano esplorazione e sfruttamento durante la fase di apprendimento come la strategia epsilon-greedy.

Si ripetono gli step precedenti fino a convergenza o per un numero predefinito di epsiodi.

3.5 Vantaggi

- **Approccio model-free:** non è richiesto nessun modello. Questo è particolarmente utile nel caso le dinamiche di un ambiente non siano conosciute o siano difficili da modellare

- **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

3.6 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.

Capitolo 4

Analisi comparativa tra Value Iteration e Q-learning nell'ambiente Pac-Man

Il presente capitolo confronta due approcci fondamentali del reinforcement learning, Value Iteration (VI) e Q-Learning (QL), applicati all'ambiente non stazionario Pac-Man. L'analisi si concentrata sulle prestazioni, l'adattabilità e l'efficienza computazionale di entrambi gli algoritmi.

4.1 Adattabilità in Ambienti Non Stazionari

- **Q-Learning** ha dimostrato una superiore capacità di adattamento. Essendo un algoritmo model-free, QL apprende direttamente dall'esperienza di gioco, modificando continuamente la sua strategia in risposta alle mutevoli condizioni dell'ambiente.
- **Value Iteration**, basandosi su un modello statico, ha mostrato limitazioni nell'adattarsi rapidamente ai cambiamenti dinamici del gioco. Ad ogni iterazione, VI determina la policy ottimale senza considerare pienamente le dinamiche temporali delle posizioni dei fantasmi e altri elementi variabili.

4.2 Analisi della complessità

4.2.1 Complessità Temporale

- **VI:** $O(k \cdot |S|^2 \cdot |A|)$, dove k è il numero di iterazioni , $|S|$ la cardinalità dello spazio degli stati, e $|A|$ quella delle azioni.

- **QL:** $O(n \cdot |A|)$, dove n è il numero di passi di apprendimento.

4.2.2 Complessità Spaziale

- **VI:** $O(|S|)$.
- **QL:** $O(|S| \cdot |A| \cdot |H|)$, dove $|H|$ è l'orizzonte temporale (passi di apprendimento.

4.2.3 Risultati Empirici

Q-Learning ha mostrato una convergenza più rapida verso soluzioni efficaci, particolarmente evidente nelle mappe di dimensioni ridotte. Questo vantaggio è attribuibile alla complessità temporale lineare di QL rispetto a quella quadratica di VI.

Nonostante la maggiore complessità spaziale di QL, l'impatto sulle prestazioni è stato trascurabile nelle mappe testate, probabilmente a causa delle loro dimensioni relativamente contenute.

4.2.4 Conclusioni

La superiorità di Q-Learning in questo contesto evidenzia l'importanza dell'apprendimento online in ambienti non-stazionari, come l'ambiente Pac-Man. Il Value-Iteration mantiene il suo vantaggio in termini di garanzia di convergenza alla policy ottimale in ambienti stazionari con modelli noti.

Questi risultati suggeriscono per problemi di reinfocement learning in ambienti non stazionari e con spazi degli stati di dimensione moderate, gli algoritmi model-free come Q-Learning possono offrire vantaggi sostanziali in termini di adattabilità e efficienza computazionale. E' importante notare che questi vantaggi potrebbero non generalizzarsi necessariamente a tutti i domini o a problemi con spazi degli stati di dimensioni significativamente maggiori.

Capitolo 5

Quesiti di Berkeley

In seguito sono presentati i quesiti svolti appartenenti al progetto di Berkeley, presentando prima la consegna ed in seguito l'implementazione della nostra soluzione in linguaggio Python.

5.1 Question 1 - Value Iteration

Il primo compito richiede di implementare un agente di Value Iteration, parzialmente specificato nel file *valueIterationAgents.py*. A differenza di quello che abbiamo implementato successivamente (Capitolo 2) questo agente è offline, quindi non è un vero e proprio agente di *reinforcement learning*, ma piuttosto un pianificatore.

5.1.1 Metodi Chiave Implementati

- ***computeActionFromValues(state)***: calcola l'azione più promettente *best action* in un determinato stato *state*.
- ***computeQValueFromValues(state, action)***: calcola il valore stimato che si ha nell'eseguire una specifica azione in un determinato stato.

5.1.2 Dettagli Implementativi

`computeActionFromValues` esamina tutte le possibili azioni che possono essere eseguite nello stato corrente `state` e seleziona quella con il valore più alto, ovvero quella che si stima porterà al maggior accumulo di ricompense a lungo termine.

```
1 def computeActionFromValues(self, state):
2     if self.mdp.isTerminal(state):
3         return None
4
```

```
    qValues = util.Counter()
    actions = self.mdp.getPossibleActions(state)

    for action in actions:
        qValues[action] = self.computeQValueFromValues(state, action)

    return max(qValues, key=lambda x: qValues[x])
```

Listato 5.1: Implementazione del metodo computeActionFromValues.

`computeQValueFromValues` prende in input lo stato corrente e l'azione da valutare e restituisce il valore Q `qValue` associato alla coppia (stato, azione). Tale valore indica la "bontà" di eseguire l'azione in quel particolare stato.

```
def computeQValueFromValues(self, state, action):
    qValue = 0
    for nextState, prob in self.mdp.getTransitionStatesAndProbs(state,
   action):
        reward = self.mdp.getReward(state, action, nextState)
        qValue += prob * (reward + self.discount * self.getValue(
   nextState))
    return qValue
```

Listato 5.2: Implementazione del metodo computeQValueFromValues.

`ValueIteration` converge alla funzione valore ottima, che rappresenta una soluzione all'equazione di Bellman.

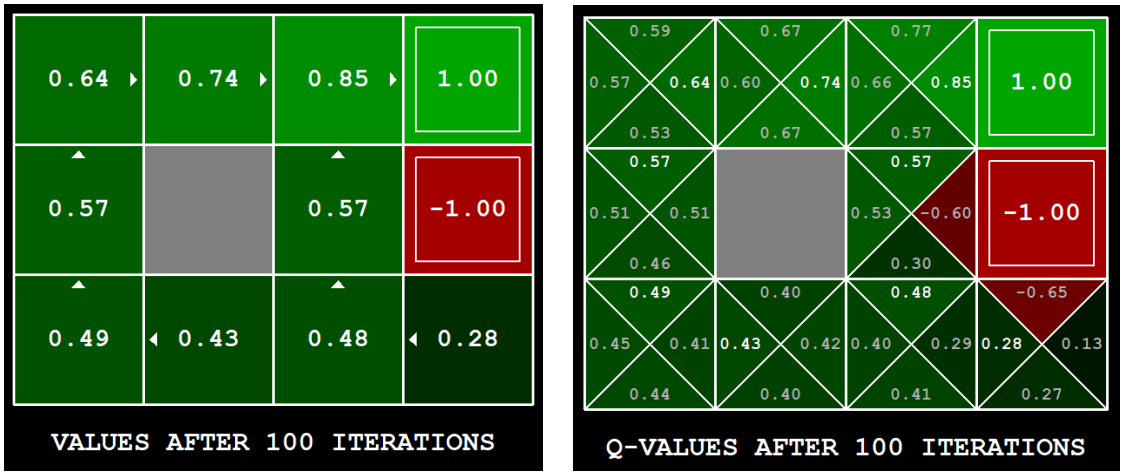
```
def valueIteration(self):
    # Initialization
    theta = 0.001
    delta = 0
    k = 0

    for k in range(self.iterations):
        newQValues = self.values.copy()
        for state in self.mdp.getStates():
            if self.mdp.isTerminal(state):
                continue
            action = self.computeActionFromValues(state)
            QValue = self.computeQValueFromValues(state, action)
            newQValues[state] = QValue
            delta = max(delta, abs(self.values[state] - QValue))
            self.values = newQValues
        if delta < theta:
            return k
```

Listato 5.3: Implementazione dell'algoritmo di ValueIteration.

5.1.3 Risultati

Abbiamo testato l'algoritmo utilizzando il comando `python gridworld.py -a value -i 100 -k 10`

I risultati sono visualizzabili nelle seguenti figure:



(a) *Values dopo 100 iterazioni.* (b) *Q-Values dopo 100 iterazioni.*

Figura 5.1: Output generato dall'algoritmo eseguito da riga di comando con la seguente stringa "*python gridworld.py -a value -i 100 -k 10*"

5.1.4 Conclusione

L'implementazione del Value Iteration si è dimostrata efficace nel calcolare i valori ottimali per gli stati in un ambiente gridworld semplificato. Tuttavia, è importante notare che Value Iteration assume una conoscenza completa del modello dell'ambiente. In scenari reali, la probabilità di transizione e le ricompense potrebbero non essere note a priori o potrebbero variare nel tempo. Pertanto, dal momento in cui Value Iteration richiede di esaminare ogni stato, ogni azione e ogni possibile stato successivo in ogni iterazione, risulta computazionalmente costoso in spazi degli stati grandi.

5.2 Question 2 - Bridge Crossing Analysis

BridgeGrid è un particolare gridworld con uno stato terminale a basso reward e uno con alto reward separati da un "ponte" stretto (da qui viene il nome "bridge"), ai lati del ponti ci sono tanti stati con alto reward negativo.
Utilizzando i valori di default per il discount e il noise (rispettivamente *0,9* e *0,2*) la policy ottimale non riesce ad attraversare il ponte.

È richiesto quindi di cambiare solo una delle due variabili alla volta affinché la policy ottimale riesca a far attraversare il ponte all'agente. Le risposte sono da inserire in *question2()* all'interno del file *analysis.py*.

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0
    return answerDiscount, answerNoise
```

Listato 5.4: Risposta question 2

La soluzione proposta imposta il valore del noise a 0, eliminando quindi qualsiasi forma di non determinismo. Eliminare il noise non è particolarmente interessante a scopi didattici, per questo motivo abbiamo trovato un altro valore che porta al risultato sperato di attraversare il ponte ovvero con noise a 0,01 (comunque prossimo a zero).

Figura 5.2: Output dopo 100 iterazioni con discount *0,9* e noise *0,2*

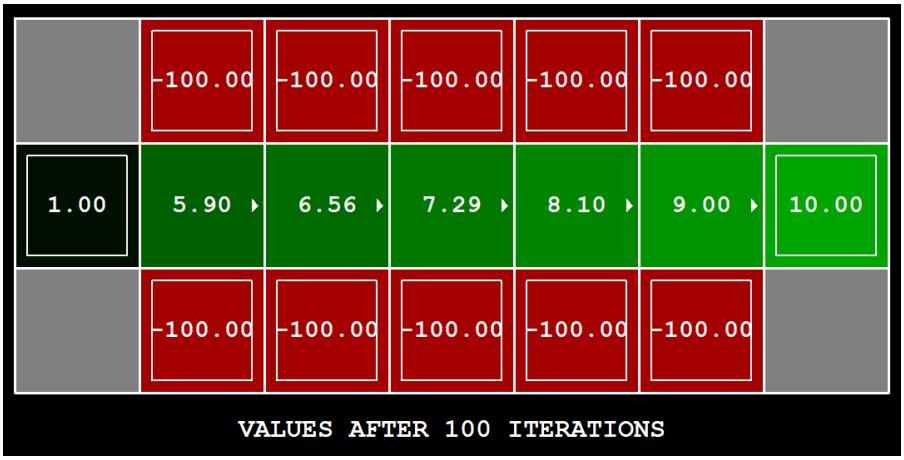


Figura 5.3: Output dopo 100 iterazioni con discount *0,9* e noise *0*

5.3 Question 3 - Policies

Il layout *DiscountGrid* (mostrato in figura 5.4) ha due stati terminali con reward positivo, uno con valore +1 e uno più distante con valore +10. Nell'ultima riga del layout sono presenti stati terminali con reward negativo di -10. Lo stato di partenza è quello evidenziato di giallo. Sono distinti due diversi tipi di *pathing*:

- quello più greedy che percorre una strada vicino all'ultima riga (ovvero quella con tutti i reward negativi), questo è più veloce e di conseguenza il più rischioso (freccia rossa)
- il secondo invece evita l'ultima riga, sarà quindi più lento ma sicuro (freccia verde)

In questo compito è richiesto di scegliere opportunamente i valori di discount, noise e living reward per produrre in questo MDP le policy ottimali qui di seguito elencate:

(a) preferenza sull'uscita più vicina (+1), rischiando la zona rossa dell'ultima riga (-10)

(b) preferenza sull'uscita più vicina (+1), **non** rischiando la zona rossa dell'ultima riga (-10)

(c) preferenza sull'uscita più lontana (+10), rischiando la zona rossa dell'ultima riga (-10)

(d) preferenza sull'uscita più lontana (+10), **non** rischiando la zona rossa dell'ultima riga (-10)

(e) evitare entrambe le uscite e la zona rossa (ovvero un episodio che non dovrebbe terminare)

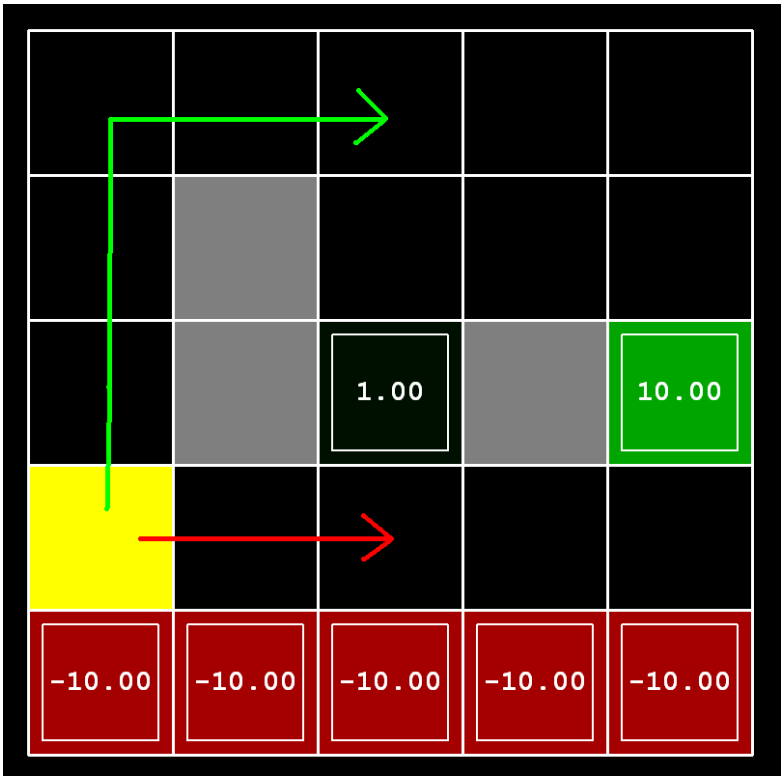


Figura 5.4: Layout *DiscountGrid*

Il value di una policy ottimale può essere definito con le seguenti:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V^*(s') \right]$$

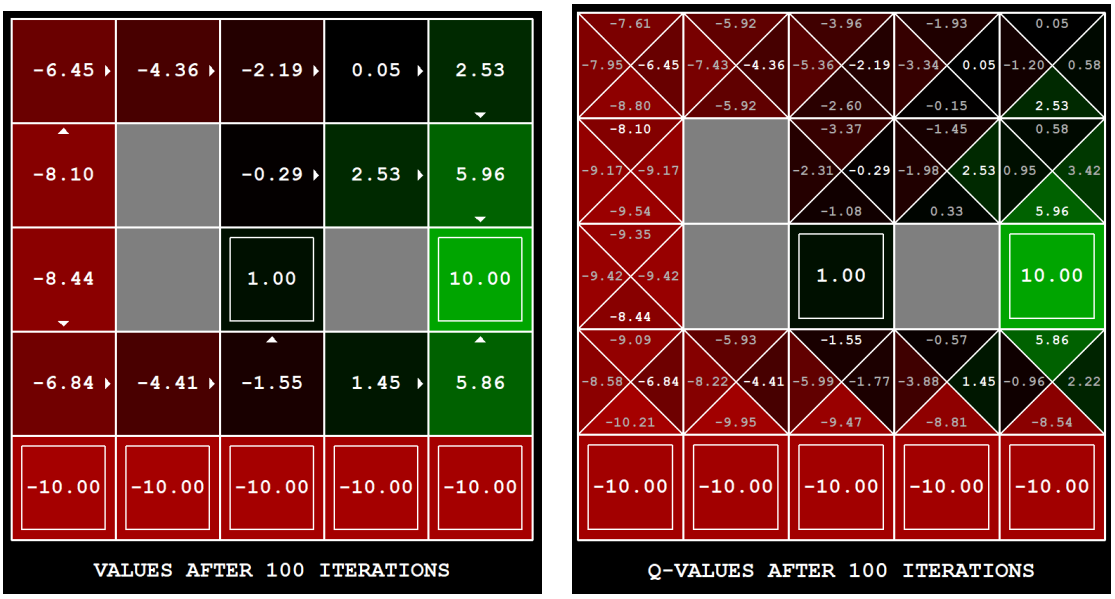
$$V^*(s) = \max_a Q^*(s,a)$$

$$\pi^*(s) = arg\max_a Q^*(s,a)$$

5.3.1 Risposta al punto 3a

```
def question3a():
    answerDiscount = 0.9
    answerNoise = 0.2
    answerLivingReward = -2
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

Listato 5.5: Risposta per: *"preferenza sull'uscita più vicina (+1), rischiando la zona rossa dell'ultima riga (-10)"*



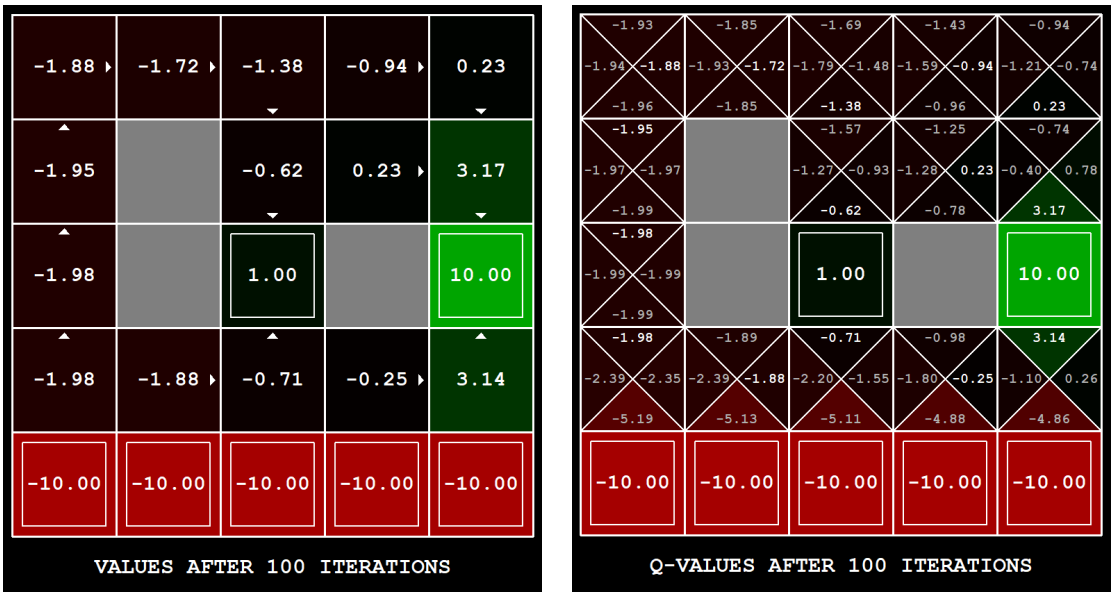
(a) *Values dopo 100 iterazioni.* (b) *Q-Values dopo 100 iterazioni.*

Figura 5.5: Output generato dall'algoritmo eseguito da riga di comando con la seguente stringa "*python gridworld.py -a value -i 100 -g DiscountGrid –discount 0.9 –noise 0.2 –livingReward -2*"

5.3.2 Risposta al punto 3b

```
def question3b():
    answerDiscount = 0.5
    answerNoise = 0.2
    answerLivingReward = -1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

```

Listato 5.6: Risposta per: *"preferenza sull'uscita più vicina (+1), **non** rischiando la zona rossa dell'ultima riga (-10)"*



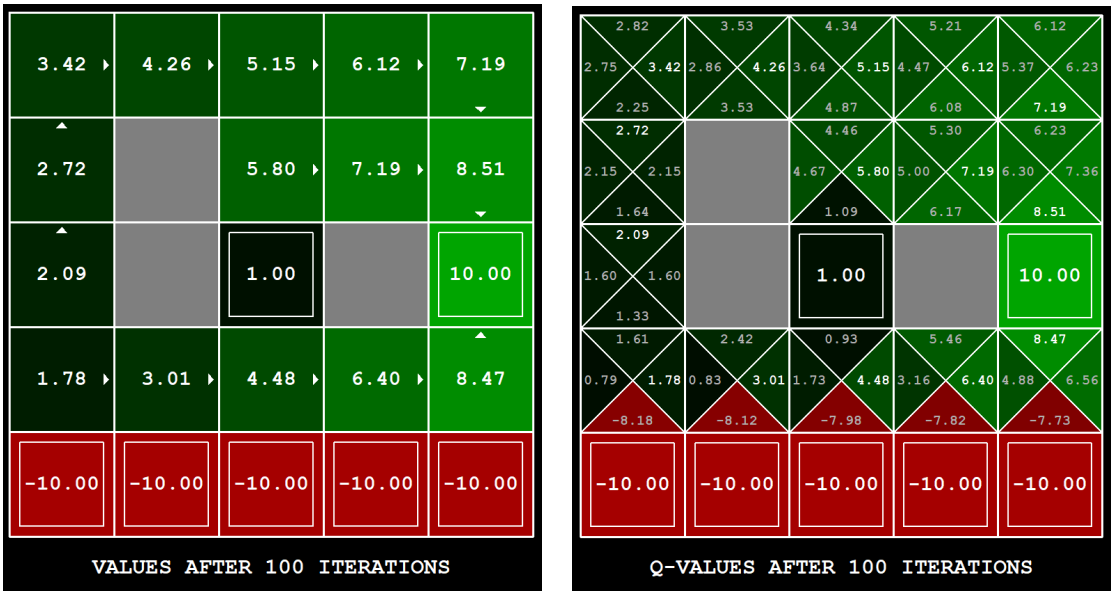
(a) *Values dopo 100 iterazioni.* (b) *Q-Values dopo 100 iterazioni.*

Figura 5.6: Output generato dall'algoritmo eseguito da riga di comando con la seguente stringa "*python gridworld.py -a value -i 100 -g DiscountGrid –discount 0.5 –noise 0.2 –livingReward -1*"

5.3.3 Risposta al punto 3c

```
def question3c():
    answerDiscount = 0.9
    answerNoise = 0.1
    answerLivingReward = -0.3
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

```

Listato 5.7: Risposta per: *"preferenza sull'uscita più lontana (+10), rischiando la zona rossa dell'ultima riga (-10)"*



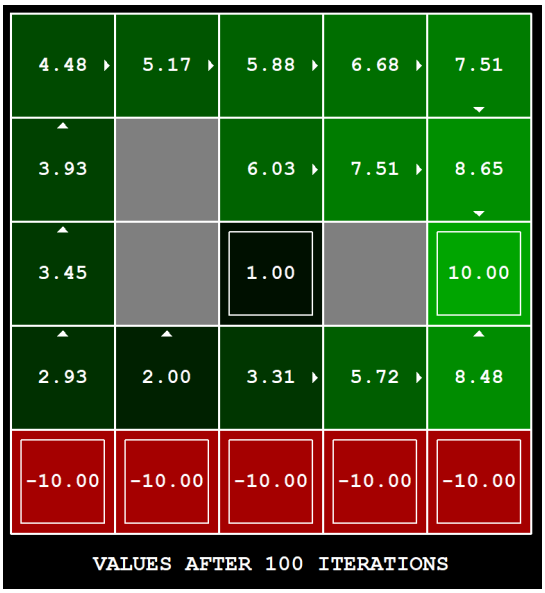
(a) *Values dopo 100 iterazioni.* (b) *Q-Values dopo 100 iterazioni.*

Figura 5.7: Output generato dall'algoritmo eseguito da riga di comando con la seguente stringa "*python gridworld.py -a value -i 100 -g DiscountGrid –discount 0.9 –noise 0.1 –livingReward -0.3*"

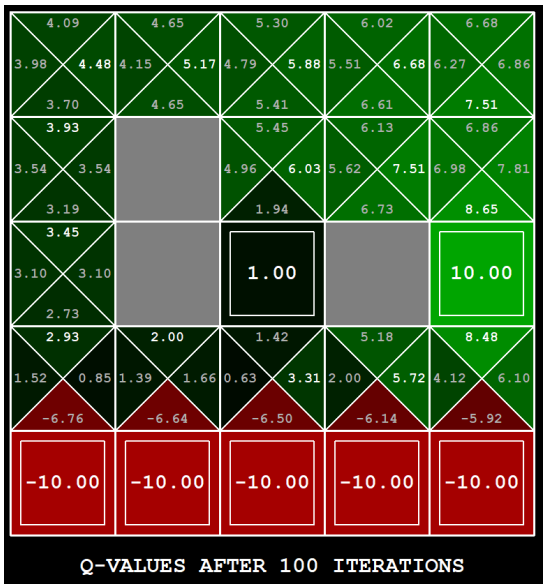
5.3.4 Risposta al punto 3d

```
def question3d():
    answerDiscount = 0.9
    answerNoise = 0.2
    answerLivingReward = 0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'

```

Listato 5.8: Risposta per: *"preferenza sull'uscita più lontana (+10),* ***non*** *rischiando la zona rossa dell'ultima riga (-10)"*



(a) *Values dopo 100 iterazioni.*



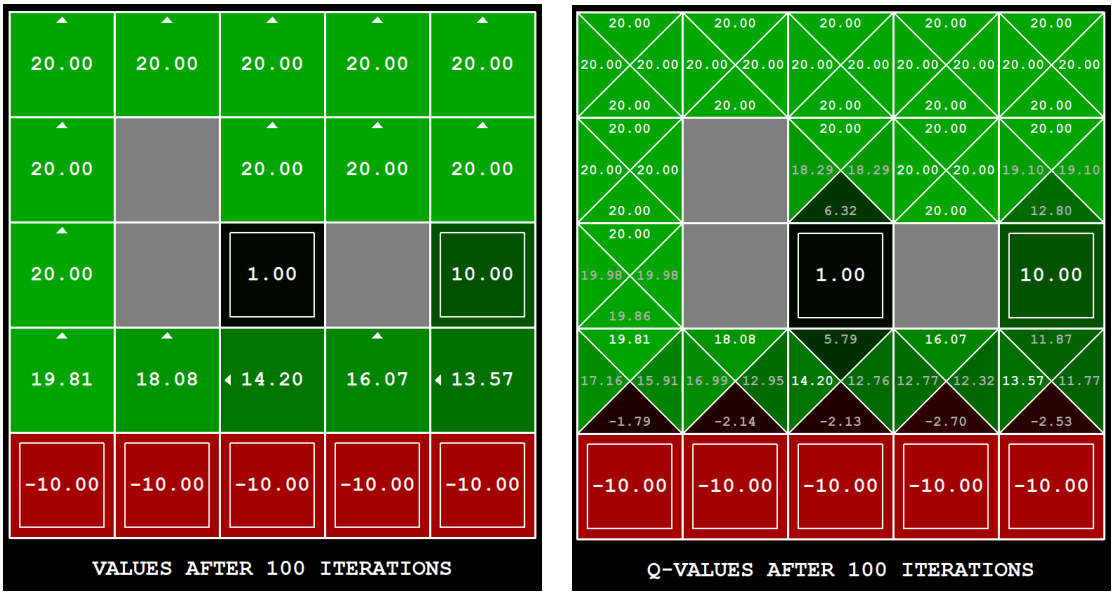
(b) *Q-Values dopo 100 iterazioni.*

Figura 5.8: Output generato dall'algoritmo eseguito da riga di comando con la seguente stringa "*python gridworld.py -a value -i 100 -g DiscountGrid –discount 0.9 –noise 0.2 –livingReward 0*"

5.3.5 Risposta al punto 3e

```
def question3e():
    answerDiscount = 0.9
    answerNoise = 0.2
    answerLivingReward = 2
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

Listato 5.9: Risposta per: "evitare entrambe le uscite e la zona rossa (ovvero un episodio che non dovrebbe terminare)"



(a) *Values dopo 100 iterazioni.* (b) *Q-Values dopo 100 iterazioni.*

Figura 5.9: Output generato dall'algoritmo eseguito da riga di comando con la seguente stringa "*python gridworld.py -a value -i 100 -g DiscountGrid –discount 0.9 –noise 0.2 –livingReward 2*"

5.4 Question 4 - Q-Learning

Il quarto compito chiede di implementare un agente secondo l'algoritmo di Q-learning, il quale rispetto all'algoritmo di value iteration apprende dall'ambiente empiricamente, ovvero tramite *"trial and error"*.

L'algoritmo di Q-learning è contraddistinto dalle seguenti equazioni:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \max_{a'} Q^*(s',a')$$

dove $Q^*(s,a)$ è il *discounted reinforcement* atteso per aver preso l'azione a nello stato s e in seguito scegliendo le azioni in modo ottimale. $V^*(s)$ è il valore di s dopo aver assunto che inizialmente è stata intrapresa l'azione migliore, quindi $V^*(s) = \max_{a'} Q^*(s,a)$. Partendo da $V^*(s) = \max_{a'} Q^*(s,a)$, abbiamo $\pi^*(s) = arg\max_a Q^*(s,a)$ come policy ottimale.

L'equazione del Q-learning è:

$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha \left[R(s,a) + \gamma \max_{a'} Q(s',a') - Q_{t-1}(s,a) \right]$$

L'implementazione dei quattro metodi richiesti, ovvero *update*, *computeValueFromQValues*, *getQValue*, e *computeActionFromQValues*, è riportata qui di seguito.

```
    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)]

    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))

    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.
```

```
25          """
26          legalActions = self.getLegalActions(state)
27          if not legalActions:
28              return None
29
30          bestQValue = float("-inf")
31          action_q = {}
32          for action in legalActions:
33              targetQValue = self.getQValue(state, action)
34              action_q[action] = targetQValue
35
36              if targetQValue > bestQValue:
37                  bestQValue = targetQValue
38
39          bestAction = [k for k, v in action_q.items() if v == bestQValue
    ]
40
41          return random.choice(bestAction)
42
43      def update(self, state, action, nextState, reward):
44          """
45          The parent class calls this to observe a
46          state = action => nextState and reward transition.
47          You should do your Q-Value update here
48
49          NOTE: You should never call this function,
50          it will be called on your behalf
51          """
52          Q_avg = self.getQValue(state, action)
53          q_update = Q_avg + self.alpha * (
54              reward + self.discount * self.getValue(nextState) - Q_avg
55          )
56          self.QValues[(state, action)] = q_update
```

Listato 5.10: Implementazione dei metodi principali richiesti per il funzionamento dell'agente con *Q-learning*

5.5 Question 5 - Epsilon Greedy

Il quinto compito richiede di completare l'agente di Q-learning implementando la variante *epsilon-greedy* in *getAction* (metodo già precedentemente implementato).

La variante *epsilon-greedy* introduce un elemento di casualità nella scelta dell'azione, bilanciando così *l'esplorazione* (*exploration*) e lo *sfruttamento* (*exploitation*).

Ad ogni passo, l'agente sceglie l'azione da eseguire in base alla seguente regola:

- Con una probabilità ϵ: sceglie un'azione casuale tra quelle disponibili
- Con una probabilità $1 - \epsilon$: sceglie l'azione che massimizza il *Q-value* attuale per lo stato corrente.

Il parametro ϵ controlla il grado di esplorazione: valori alti di ϵ portano a prediligere una maggiore esplorazione, mentre valori bassi favoriscono lo sfruttamento delle conoscenze pregresse.

5.5.1 Vantaggi

- Bilanciamento esplorazione/sfruttamento: permette all'agente di esplorare l'ambiente per scoprire nuove opportunità e allo stesso tempo di sfruttare le conoscenze acquisite per massimizzare la ricompensa a lungo termine
- Convergenza: in condizioni opportune, l'algoritmo Q-learning con epsilon-greedy converge alla policy ottimale
- Semplicità: è un algoritmo relativamente semplice da implementare e comprendere

5.5.2 Limitazioni

- Dipendenza da ϵ: la scelta del valore di ϵ è cruciale per le prestazioni dell'algoritmo. Un valore troppo alto può rallentare la convergenza, mentre un valore troppo basso può impedire all'agente di esplorare sufficientemente l'ambiente
- Dimensione dello spazio degli stati e delle azioni: per problemi con grandi spazi degli stati e delle azioni, la tabella Q può diventare molto grande, rendendo l'algoritmo computazionalmente costoso

Per testare il metodo implementato abbiamo avviato un *crawler robot* per verificare che riesca ad apprendere il movimento corretto per arrivare al proprio traguardo. Tramite linea di comando con la seguente stringa si avvia la simulazione: "*python crawler.py*"

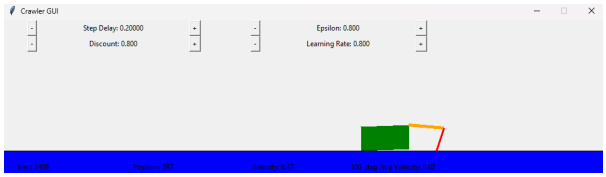


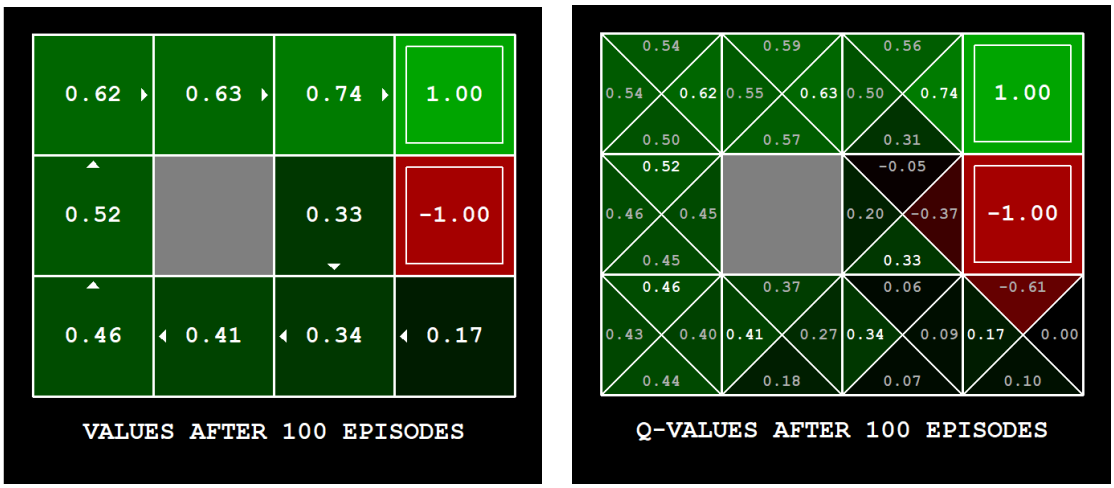
Figura 5.10: Crawler robot

5.5.3 Implementazione

```
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.

        HINT: You might want to use util.flipCoin(prob)
        HINT: To pick randomly from a list, use random.choice(list)
        """
        # 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
```

Listato 5.11: Implemtazione epsilon-greedy nel metodo getAction

5.5.4 Output generato al seguito dell'implementazione in un ambiente gridworld di prova



(a) *Values dopo 100 iterazioni.* (b) *Q-Values dopo 100 iterazioni.*

Figura 5.11: Output generato dall'algoritmo eseguito da riga di comando con la seguente stringa "*python gridworld.py -a q -k 100* "

5.6 Question 6 - Bridge Crossing Revisited

Il quesito 6 chiede inizialmente di trainare per *50* episodi un q-learner completamente randomico con un valore di default per il learning rate nel *BridgeGrid* (utilizzato precedentemente nel quesito 5.2), il quale è senza noise, ed osservare se trova l'optimal policy.

Da riga di comando bisogna digitare la seguente stringa: *python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1*

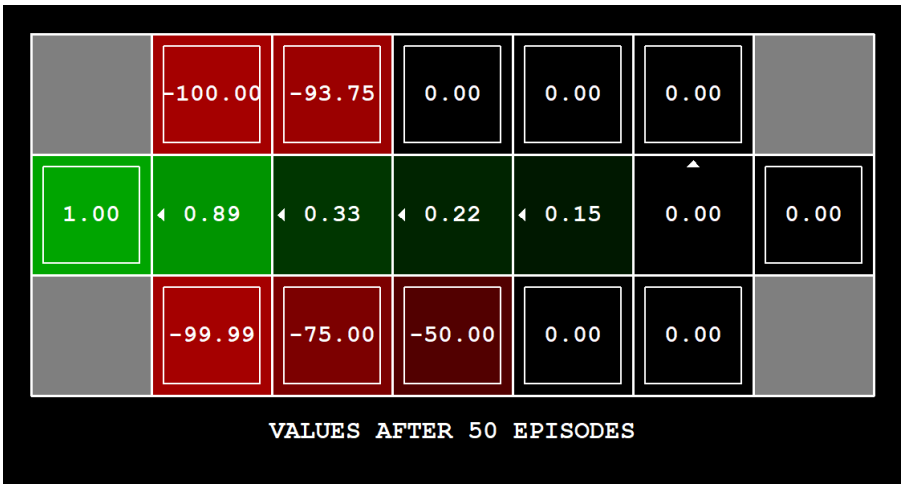


Figura 5.12: Values ottenute dopo *50* iterazioni

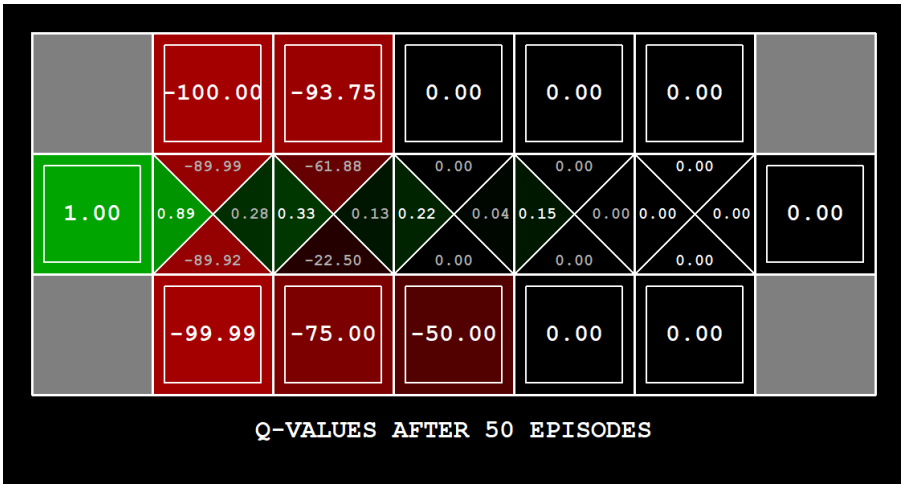


Figura 5.13: Q-values ottenute dopo *50* iterazioni

Successivamente è richiesto di provare lo stesso esperimento con un epsilon equivalente a *0*.

Il quesito da rispondere è il seguente: "Esistono dei valori da attribuire a epsilon e al learning rate in modo tale che sia altamente probabile (sopra il *99%*) che si raggiunga la policy ottimale dopo *50* iterazioni?"

La soluzione da noi fornita ed inserita come sempre in *analysis.py* è qui proposta:

```
def question6():
    answerEpsilon = None
    answerLearningRate = None
    return "NOT POSSIBLE"
    # If not possible, return 'NOT POSSIBLE'
```

Listato 5.12: Risposta fornita per la scelta dei valori di epsilon e del learning rate in modo tale che sia altamente probabile che si raggiunga la policy ottimale dopo 50 iterazioni"

5.7 Question 7 - Q-Learning and Pacman

Finalmente l'agente verrà eseguito nel mondo di Pacman. Pacman giocherà le partite in due fasi. Nella prima fase, denominata training, Pacman comincerà ad imparare circa le values delle posizioni e delle azioni. Una volta che il training è ultimato entrerà nella testing mode. Quando sta testando, i valori di epsilon e alfa sono impostati a 0,0 per permettere a Pacman di valutare la policy che ha imparato.

Per vedere i primi risultati si può utilizzare il layout più piccolo per Pacman disponibile ovvero *smallGrid*.

Avviamo tramite riga di comando l'agente per vedere i risultati utilizzando la seguente stringa: *python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid*

PacmanQAgent utilizza lo stesso codice implementato nei quesiti precedenti cambiando solo alcuni parametri:

- *epsilon = 0,05*
- *alpha = 0,02*
- *gamma = 0,8*

Questi sono i risultati che otteniamo eseguendo il codice:

```
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 502
Pacman emerges victorious! Score: 499
Average Score: 499.7
Scores:        499.0, 503.0, 503.0, 495.0, 499.0, 503.0, 499.0, 495.0, 502.0, 499.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

Figura 5.14: Win rate finale ottenute dopo una fase di training di 2000 episodi e 10 episodi di testing

5.8 Question 8 - Approximate Q-Learning

L'ultimo compito richiede di implementare un agente con algoritmo di Q-learning approssimato il quale impara i pesi per le features degli stati, dove molti stati possono condividere le stesse feature. La Q-function approssimata è la seguente:

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i$$

dove ogni peso w_i è associato con una particolare feature $f_i(s,a)$.

Abbiamo implementato i pesi dei vettori come funzioni di mappatura del dizionario sul valore del peso che sono stati restituiti dagli estrattori di funzionalità.

I vettori dei pesi saranno aggiornati similmente a come si aggiornano le Q-values:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$

$$difference = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a)$$

```
    def __init__(self, extractor="IdentityExtractor", **args):
        self.featExtractor = util.lookup(extractor, globals())()
        PacmanQAgent.__init__(self, **args)
        self.weights = util.Counter()

    def getWeights(self):
        return self.weights

    def getQValue(self, state, action):
        """
        Should return Q(state,action) = w * featureVector
        where * is the dotProduct operator
        """
        featureVector = self.featExtractor.getFeatures(state, action)
        return featureVector * self.weights

    def update(self, state, action, nextState, reward):
        """
        Should update your weights based on transition
        """
        featureVector = self.featExtractor.getFeatures(state, action)
        oldValue = self.getQValue(state, action)
        newQvalue = self.getValue(nextState)
        difference = reward + self.discount * newQvalue - oldValue
        for feature in featureVector:
            weight = self.weights[feature]
            self.weights[feature] = (
```

```
28                  weight + self.alpha * difference * featureVector[
    feature]
29              )
```

Listato 5.13: Implemtazione approximate q-learning

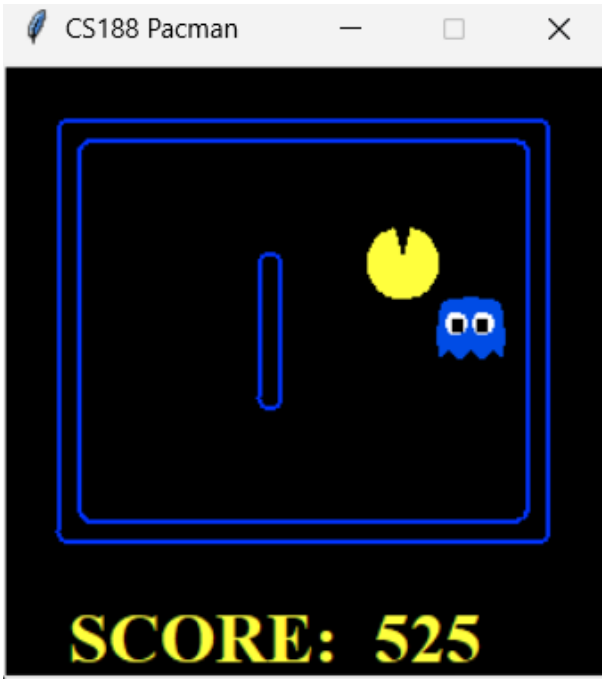


Figura 5.15: Agente approximate q-learning in mediumGrid

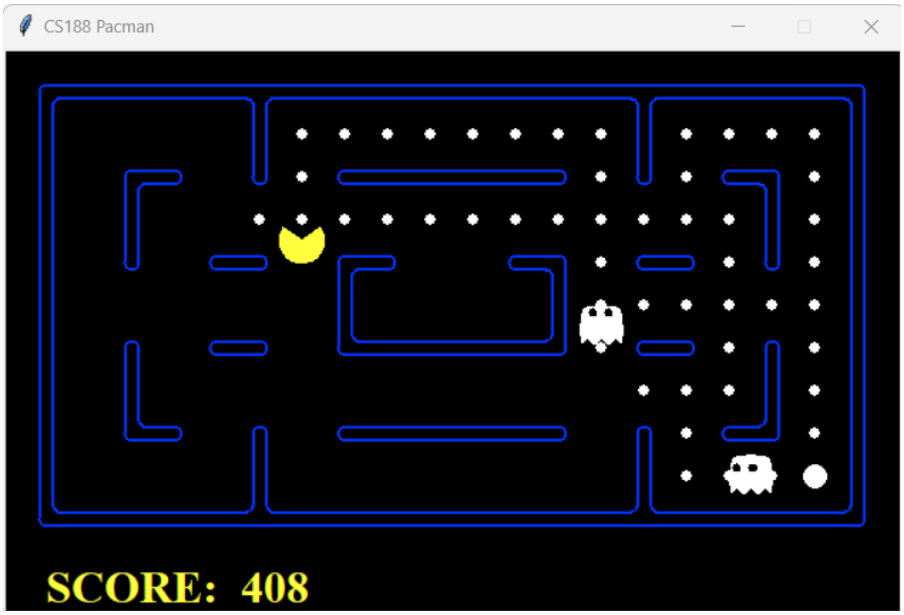


Figura 5.16: Agente approximate q-learning in mediumClassic

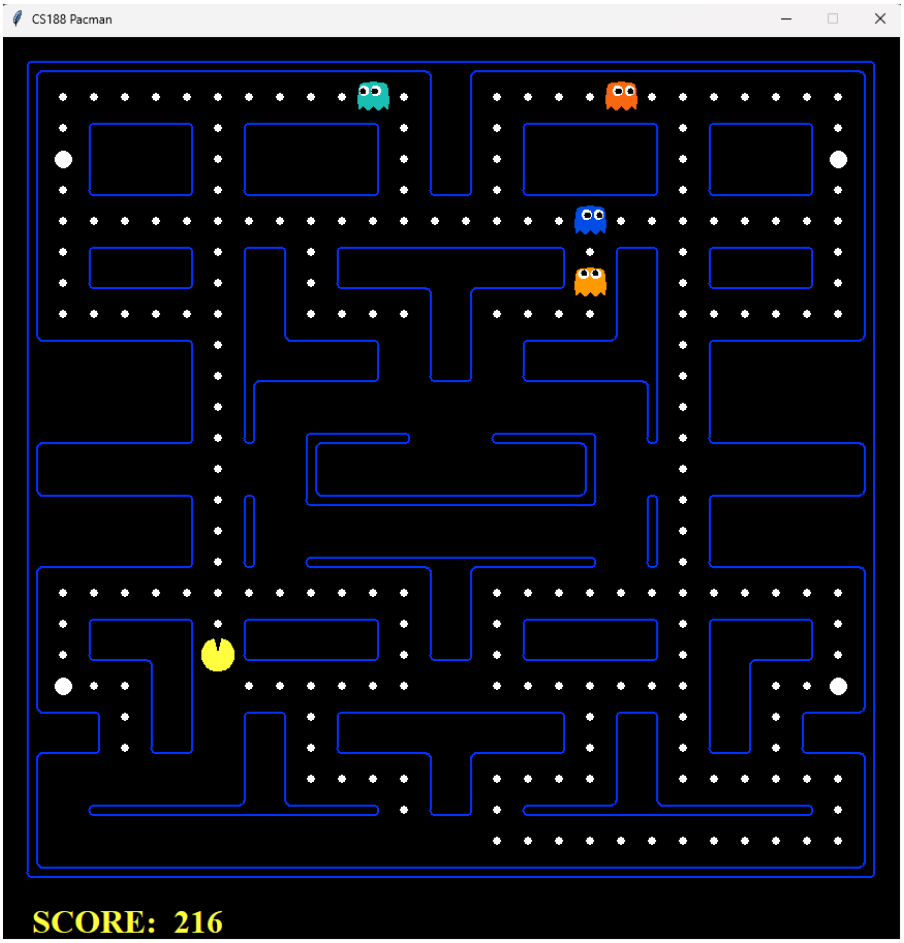


Figura 5.17: Agente approximate q-learning in originalClassic

smallGrid	***smallClassic***	***mediumGrid***	***mediumClassic***	***originalClassic***
10/10	10/10	10/10	9/10	9/10

Tabella 5.1: Win rate di *ApproximateQAgent* su diversi layout disponibili nel progetto (utilizzando sempre 50 episodi di training con *learning rate* di 0,2