Università di Perugia

Dipartimento di Matematica e Informatica

LAUREA MAGISTRALE IN INFORMATICA
CORSO DI ARTIFICIAL INTELLIGENT SYSTEM

Relazione progetto d'esame

Reinforcement Learning
Cat vs Mouse

Professore
Prof. Alfredo Milani

Studente
Cristian Cosci
(Matricola: 350863)

Anno Accademico 2021-2022

Indice

Capitolo 1

Introduzione e descrizione del progetto

Il lavoro svolto ha l'obiettivo di realizzare un progetto che rappresenta il classico problema del gatto contro il topo. In questa istanza vi sono 3 entità principali:

- il Gatto;
- il Topo;
- il Formaggio.

In particolare i due agenti principali sono il Gatto e il Topo i quali sono in **competizione l'uno contro l'altro**.

- **L'obiettivo del Gatto** è quello di catturare il Topo prima che quest'ultimo raggiunga il Formaggio. Nel caso in cui l'obiettivo viene raggiunto, il Gatto vince la partita.
- **L'obiettivo del Topo** è invece quello di mangiare il formaggio per vincere la partita.

Il problema è rappresentato mediante un'istanza di **Reinforcement Learning**.

Definizioni di recap:

- **Reinforcement Learning**: è un campo dell'intelligenza artificiale (fa parte del machine learning) in cui un agente sceglie le azioni basandosi su un premio numerico che ottiene dall'ambiente. Nel reinforcement learning:
  - l'agente ha uno spazio degli stati che esplora
  - l'agente conosce le azioni che può eseguire
  - c'è una serie di azioni che l'agente può intraprendere in qualsiasi stato particolare
  - c'è una ricompensa associata allo stato successivo in cui l'agente atterra

  L'obiettivo dell'apprendimento per rinforzo è progettare la policy (dato uno stato lui conosce le azioni che deve eseguire) ottimale o quasi ottimale basata sui premi ricevuti. Nell'apprendimento di rinforzo non c'è supervisione, ma ci sono delle ricompense (reward). Il feedback è in ritardo, non immediato. Il tempo conta e le azioni sono sequenziali. Le azioni dell'agente influenzano i dati successivi (modificano l'ambiente quindi gli stati successivi) che esso riceve.

- **Policies**: è una regola utilizzata dagli agenti per decidere quale azione compiere dato un determinato stato.

I protagonisti del RL sono l'agente e l'ambiente.
L'ambiente è il mondo in cui l'agente vive e con cui interagisce. Ad ogni passo dell'interazione, l'agente vede un'osservazione (possibilmente parziale) dello stato del mondo, e poi decide l'azione da intraprendere. L'ambiente cambia quando l'agente agisce su di esso, ma può anche cambiare da solo.
L'agente percepisce anche un segnale di ricompensa immediato dall'ambiente, un numero che gli dice quanto sia buono o cattivo l'attuale stato del mondo. L'obiettivo dell'agente è massimizzare la sua ricompensa cumulativa, chiamata rendimento.

In un ambiente di rl, l'agente si muove per prove ed errori e apprende attraverso la ripetuta azione delle stesse operazioni e tiene conto del bilancio finale di ogni serie di azioni. Non è importante in realtà il massimizzare il rendimento (reward) immediato ma quello a lungo termine (cumulativo totale).

Nell'implementazione ho deciso di creare molteplici istanze con diverse caratteristiche, in modo da avere uno studio molto più approfondito del problema. Possiamo contraddistinguere le istanze in 3 principali categorie, in ordine di complessità del problema:

1. Il Topo è un agente intelligente che deve essere allenato, mentre il Gatto fa parte dell'ambiente e si muove in maniera basica, svolgendo il ruolo di semplice sentinella di guardia per il Formaggio.

2. Il topo è un agente intelligente che deve essere allenato, mentre vi sono due Gatti che fanno parte dell'ambiente con il ruolo di sentinella di guardia per il Formaggio.

3. Sia il Gatto che il Topo sono due agenti intelligenti in competizione l'uno contro l'altro.

Ho inoltre implementato una visualizzazione grafica del problema per rendere il tutto più di semplice visione (vedi Figura 1.1).

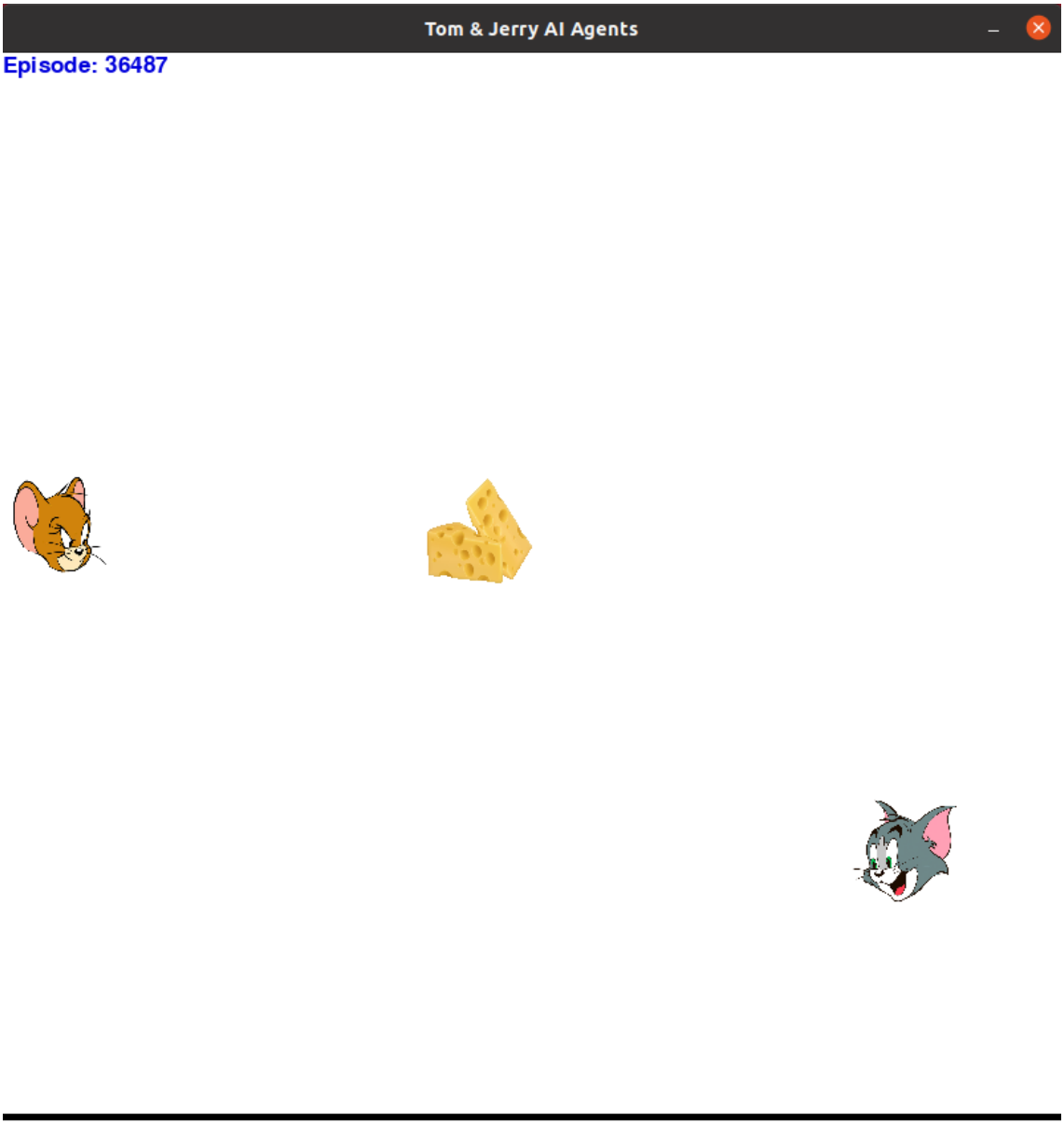

Figura 1.1: Esempio interfaccia grafica del problema.

Capitolo 2

Implementazione del problema

In questo capitolo viene descritta la relativa implementazione del progetto, le specifiche utilizzate e il codice.
Come riportato precedentemente ho deciso di implementare 3 situazioni principali del sistema e per ognuna di esse è necessario ridefinire opportuni metodi e caratteristiche.
In tutte le implementazioni le 4 principali componenti del programma sono le stesse:

- **Environment**;
- **Agente**;
- **l'Apprendimento**;
- **l'Esecuzione**.

La struttura del progetto si basa quindi su 4 file (per ogni implementazione) contenenti le precedenti componenti (vedi Figura 2.1). Qui di seguito verranno descritte in modo generale le varie parti e verranno specificate anche le differenze a seconda dell'implementazione.
Il tutto è stato implementato utilizzando il linguaggio di programmazione *python* e alcune sue librerie:

- **pygame**
- **matplotlib**
- **numpy**

Il codice è disponibile in questa repository GitHub:
`https://github.com/CristianCosci/Mouse_vs_Cat_AI_Reinforcement_Learning.git`

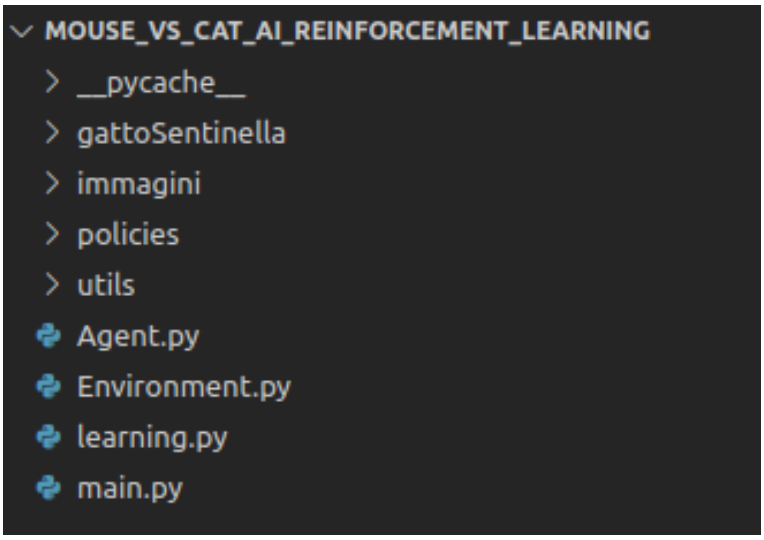

Figura 2.1: Struttura del progetto.

2.1 Rappresentazione del problema

Per quanto riguarda la rappresentazione del problema ho deciso di utilizzare una matrice quadrata di dimensione n x n, in cui gli agenti si muovono eseguendo un passo ad ogni step in una delle direzioni possibili. Gli agenti, ovviamente, non possono superare i bordi esterni della mappa in quanto quest'ultimi fungono da muro (vedi implementazione ambiente successiva). Inoltre, possono essere presenti nella mappa degli ostacoli che danno una penalità nel caso in cui gli agenti gli vadano incontro.
Per quanto riguarda **lo stato percepito**, gli agenti ricevono dall'ambiente:

- **le distanze**: dal formaggio, dal gatto o dal topo (a seconda dell'agente). Come misura di distanza è stata utilizzata la **distanza di Manhattan** in quanto essa rappresenta effettivamente il numero di passi da dover compiere per raggiungere l'altro agente o formaggio. In questo modo si presume che il topo proceda cercando di **minimizzare** la distanza nei confronti del formaggio e cercando di **massimizzare** quella nei confronti del gatto. La stessa cosa vale per il gatto,

in quanto quest'ultimo cercherà di minimizzare la sua distanza nei confronti del topo.

- **informazioni sugli oggetti (solo ostacoli o muri)** che si trovano nell'intorno **UP, DOWN, LEFT, RIGHT**. Ovviamente sono state mappate tutte le possibili combinazioni in cui l'agente può trovarsi. In questo caso i valori ricevuti sono dati da un numero intero che varia da 1 a 14 a seconda della combinazione.

La posizione iniziale degli agenti e del formaggio viene generata in modo casuale, seguendo dei criteri a seconda dell'implementazione.
Riassumendo:

- **Azioni possibili dell'agente**: *UP, DOWN, RIGHT, LEFT*
- **I bordi esterni della mappa** fungono da muro (non è presente l'*effetto pacman*).
- **Gli agenti sono posizionati in modo casuale** ad ogni epoca seguendo dei criteri in base al problema in essere.
- Gli agenti hanno anche un **numero limitato di mosse** a disposizione per ogni epoca.
- Il topo vince la partita se mangia il formaggio, mentre subisce una sconfitta se viene mangiato dal topo o finisce le mosse a disposizione.
- Sono fornite delle reward ad ogni step dell'algoritmo che verranno approfondite successivamente.
- Gli agenti ricevono come stato (dall'ambiente) le informazioni relative agli altri agenti e al formaggio (come distanza di Manhattan) e informazioni sugli ostacoli nel loro vicinato.

Informazioni sui reward.
Ad ogni step vengono ovviamente restituiti agli agenti dei reward che variano in base all'azione eseguita e dallo stato in cui si trovano. Il tutto è qui di seguito riassunto:

- se l'agente (gatto o topo) va a **sbattere su un muro o un ostacolo** riceve un **reward di -20**;
- se il **topo mangia il formaggio** riceve un **reward di 200**;
- se il **gatto mangia il topo** riceve un **reward di 200**. Il topo riceve invece un **reward di -200**;
- **ad ogni mossa** l'agente riceve un **reward di -1**.

2.2 Environment

L'Environment o **ambiente** è una delle componenti principali di ogni problema di Reinforcement Learning. L'ambiente ha il compito di gestire gli oggetti presenti, le informazioni di quest'ultimi e di ritornare lo stato agli agenti, i quali in base ad esso dovranno decidere come agire.
La struttura di base dell'ambiente è principalmente la stessa per tutte le implementazioni, vi sono comunque differenze e accortezze specifiche a seconda del problema. In questa sezione verrà descritta l'istanza in cui sia il gatto che il topo sono agenti intelligenti, mentre nelle sottosezioni successive sono presentate le principali differenze con le altre due modalità. Ho deciso quindi di creare una classe **Environment**, la quale conterrà i principali metodi per:

- gestione degli ostacoli;
- ritorno dello stato;
- reset delle informazioni e delle posizioni degli elementi ad ogni epoca;
- gestione dello spostamento degli agenti e dei vari reward;
- render per la visualizzazione grafica.

Qui di seguito è riportata la descrizione generale per i vari metodi, e nelle sottosezioni successive sono descritte le differenze di ogni singola implementazione.

Costruttuore della classe Env.

```
class Env():
    def __init__(self, display, matrix, cat_mode, map_mode):
        self.HEIGHT = matrix.ROWS
        self.WIDTH = matrix.COLUMNS
        self.PCT_OBS = matrix.PCT_OBSTACLES
        self.CAT_MODE = cat_mode
        self.MAP_MODE = map_mode

        # Pygame setting
        self.DISPLAY = display
        displayWidth, displayHeight = display.get_size()
        displayHeight -= 100  # To have additional space to show other information
        self.BLOCK_WIDTH = int(displayWidth/self.WIDTH)
        self.BLOCK_HEIGHT = int(displayHeight/self.HEIGHT)

        # Agents
        self.CAT = Cat(self.DISPLAY, self.BLOCK_WIDTH, self.BLOCK_HEIGHT)
        self.MOUSE = Mouse(self.DISPLAY, self.BLOCK_WIDTH, self.BLOCK_HEIGHT)
        self.MOVES = {'mouse':100,'cat':100}

        # Obstacles
        self.OBSTACLES = self.load_obstacles(matrix.OBSTACLES, self.PCT_OBS)

        # Cheese
        self.CHEESE_IMG = pygame.transform.scale(pygame.image.load('immagini/cheese.
    png'),(self.BLOCK_WIDTH, self.BLOCK_HEIGHT))
```

Listing 2.1: Costruttore classe Env

Il costruttore prende in input:

- il parametro **display** necessario alla visualizzazione grafica con pygame
- la variabile **matrix** che rappresenta un oggetto della classe **Matrix**. Quest'ultima è una classe di convenienza per rappresentare la griglia (mappa) e ha il solo scopo di contenere informazioni sulla dimensione e sugli ostacoli.

```
# Convenience class to represent the grid (map)
class Matrix:
    def __init__(self, rows=5, columns=5, max_pct_obstacles = 0):
        self.ROWS = rows
        self.COLUMNS = columns
        self.PCT_OBSTACLES = max_pct_obstacles
        if max_pct_obstacles > 0:
            self.OBSTACLES = self.createObstacles(self.ROWS, (self.ROWS-1), 0)
        else:
            self.OBSTACLES = []

```

```
12
13      def createObstacles(self, n, cat_axis, mouse_axis):
14          possible_obstacles = []
15          for x in range(n):
16              if x == mouse_axis or x == cat_axis:
17                  pass
18              else:
19                  for y in range(n):
20                      possible_obstacles.append([x, y])
21
22          possible_cheese_positions = [4,4], [4,5], [5,4], [5,5]
23          possible_obstacles_new = possible_obstacles.copy()
24
25          for obs in possible_obstacles:
26              if obs in possible_cheese_positions:
27                  possible_obstacles_new.remove(obs)
28
29          return tuple(possible_obstacles_new)
```

Listing 2.2: Classe Matrix

Questa classe mediante il metodo **createObstacles()** genera tutte le posizioni possibili in cui possono essere presenti gli ostacoli nella mappa. Questa funzione differisce a seconda dell'istanza del problema, e nelle sottosezioni successive sono descritte le altre due varianti.

- le variabili **cat_mode** e **map_mode** che contengono informazioni relative alla modalità di gioco (gli esempi di modalità sono spiegati nel Capitolo 3).

I metodi di questa classe sono:

```
1       def load_obstacles(self, possible_obstacles, pct_obstacles):
2           '''
3               Used to random choose n (pct_obstacles*100) obstacles from the possble
    obstacles
4           '''
5
6       def set_obstacles(self, obstacles): # Used to change the obstacles in the map
7
8       def get_state(self):
9           '''
10              Return the state for the agent
11          '''
12
13      def reset(self):
14          '''
15              Used to reset all elements position in the environment
16          '''
```

```
17
18      def render(self, i_episode = -1):
19
20      def step(self, mouse_action, cat_action):
21
22      def check_towards_obstacle(self, action, agent):
23
24      def check_out_of_bounds(self, action, agent):
25
26      def update_positions(self, mouse_action, cat_action, mouse_action_null,
        cat_action_null):
27
28      def get_changes(self, action, action_null):
29
30      def checkWall(self, agent):
31
32      def checkDoubleObstacles(self, wall_position, agent):
```

Listing 2.3: Metodi classe Env

loadObstacle()

Il metodo **loadObstacle()** permette di selezionare casualmente un numero n di ostacoli, il quale indica la percentuale di riempimento che si vuole nella mappa. Il metodo viene richiamato ad ogni epoca, in questo modo si hanno sempre ostacoli diversi, generati in modo casuale.

```
1       def load_obstacles(self, possible_obstacles, pct_obstacles):
2           '''
3               Used to random choose n (pct_obstacles*100) obstacles from the possble
        obstacles
4           '''
5           n = int(len(possible_obstacles) * pct_obstacles)
6           obstacle_list = list()
7           numbers = random.sample(range(len(possible_obstacles)), n)
8           for i in numbers:
9               obstacle_list.append(possible_obstacles[i])
10
11          return tuple(obstacle_list)
```

get_state()

Uno dei metodi principali è invece **get_state()**. Quest'ultimo ritorna le informazioni necessarie agli agenti per costruire e popolare la **Q-table**.

```
1       def get_state(self):
2           '''
3               Return the state for the agent
4           '''
```

```
        wall_mouse = self.checkWall('mouse')
        obstacles_mouse_first = self.checkDoubleObstacles(wall_mouse, 'mouse')
        if wall_mouse != obstacles_mouse_first:  # Used to avoid worthless operation in case there are not obstacles near the agent
            obstacles_mouse_second = self.checkDoubleObstacles(obstacles_mouse_first, 'mouse')
            if obstacles_mouse_first != obstacles_mouse_second:  # Used to avoid worthless operation in case there are not more than 1 obstacle near the agent
                obstacles_mouse = self.checkTripleObstacles(obstacles_mouse_second, 'mouse')
            else:
                obstacles_mouse = obstacles_mouse_second
        else:
            obstacles_mouse = obstacles_mouse_first

        wall_cat = self.checkWall('cat')
        obstacles_cat_first = self.checkDoubleObstacles(wall_cat, 'cat')
        if wall_cat != obstacles_cat_first:  # Used to avoid worthless operation in case there are not obstacles near the agent
            obstacles_cat_second = self.checkDoubleObstacles(obstacles_cat_first, 'cat')
            if obstacles_cat_first != obstacles_cat_second:  # Used to avoid worthless operation in case there are not more than 1 obstacle near the agent
                obstacles_cat = self.checkTripleObstacles(obstacles_cat_second, 'cat')
            else:
                obstacles_cat = obstacles_cat_second
        else:
            obstacles_cat = obstacles_cat_first

        distanzaManhattan = (self.MOUSE_X - self.CAT_X) + (self.MOUSE_Y - self.CAT_Y)
        if self.CAT_MODE == 'classico':
            self.STATE = {'mouse':(distanzaManhattan, (self.MOUSE_X - self.CHEESE_X) + (self.MOUSE_Y -  self.CHEESE_Y), obstacles_mouse),
                          'cat':(distanzaManhattan, obstacles_cat)}
        elif self.CAT_MODE == 'knowCheese':
            self.STATE = {'mouse':(distanzaManhattan, (self.MOUSE_X - self.CHEESE_X) + (self.MOUSE_Y -  self.CHEESE_Y), obstacles_mouse),
                          'cat':(distanzaManhattan, (self.CAT_X - self.CHEESE_X) + (self.CAT_Y -  self.CHEESE_Y), obstacles_cat)}

        return self.STATE
```

Il metodo qui sopra riportato è relativo all'istanza in cui sia il gatto che il topo sono agenti intelligenti. Come si può vedere, lo stato ritornato al topo è dato dalla tripla:

- *distanza di Manhattan tra topo e gatto*
- *distanza di Manhattan tra topo e formaggio*

- *informazioni relative agli ostacoli e muri*

mentre per il gatto la tupla:

- *distanza di Manhattan tra topo e gatto*
- *informazioni relative agli ostacoli e muri*
- ed eventualmente *la distanza di Manhattan tra gatto e formaggio*

reset()

Il metodo **reset()** permette di riposizionare gli agenti e il formaggio (in modo casuale e secondo dei criteri), infatti i valori assegnati corrisponderanno alle posizioni iniziali ad ogni epoca. Inoltre sono anche resettate le informazioni relative al numero di mosse disponibili per gli agenti. La funzione termina richiamando **get_state()**.

```
    def reset(self):
        '''
            Used to reset all elements position in the environment
        '''
        self.MOUSE_X, self.MOUSE_Y = (0, np.random.randint(0, self.HEIGHT-1))
        self.CAT_X, self.CAT_Y = (self.WIDTH-1, np.random.randint(0, self.HEIGHT-1))
        self.CHEESE_X, self.CHEESE_Y = (np.random.randint((self.WIDTH // 3) + 1, (
self.WIDTH // 3 * 2)), np.random.randint((self.HEIGHT // 3) + 1, (self.HEIGHT //
3 * 2)))

        self.MOVES['mouse'] = 100
        self.MOVES['cat'] = 100

        return self.get_state()
```

step()

È uno dei metodi principali della classe ambiente. Qui sono gestite tutte le informazioni relative alle azioni degli agenti e per i reward.

```
    def step(self, mouse_action, cat_action):
        '''
            Principal method in wich all needed controls are do
        '''
        done = False
        mouse_action_null = False
        cat_action_null = False
        mouse_out_of_bounds = False
        cat_out_of_bounds = False
```

```
        reward = {'mouse': -1, 'cat': -1}
        toccate_ostacolo_mouse = 0
        toccate_ostacolo_cat = 0
        toccate_muro_mouse = 0
        toccate_muro_cat = 0
        info = {
            'cheese_eaten': False,
            'mouse_caught': False,
            'x': -1, 'y': -1,\
            'width': self.BLOCK_WIDTH,
            'height': self.BLOCK_HEIGHT
        }

        self.MOVES['cat'] -= 1
        self.MOVES['mouse'] -= 1
        # done if moves = 0
        if self.MOVES['cat'] == 0 or self.MOVES['mouse'] == 0:
            done = True

        mouse_towards_obstacle = self.check_towards_obstacle(mouse_action, agent='
    mouse')
        cat_towards_obstacle = self.check_towards_obstacle(cat_action, agent='cat')
        if mouse_towards_obstacle:
            toccate_ostacolo_mouse +=1
            reward['mouse'] = -20
            mouse_action_null = True
        if cat_towards_obstacle:
            toccate_ostacolo_cat +=1
            reward['cat'] = -20
            cat_action_null = True

        mouse_out_of_bounds = self.check_out_of_bounds(mouse_action, agent='mouse')
        cat_out_of_bounds = self.check_out_of_bounds(cat_action, agent='cat')
        if mouse_out_of_bounds:
            reward['mouse'] = -20
            toccate_muro_mouse +=1
            mouse_action_null = True
        if cat_out_of_bounds:
            reward['cat'] = -20
            toccate_muro_cat +=1
            cat_action_null = True

        self.update_positions(mouse_action, cat_action, mouse_action_null,
    cat_action_null)

        # Mouse ha mangiato il formaggio
        if self.MOUSE_X == self.CHEESE_X and self.MOUSE_Y == self.CHEESE_Y:
            done = True
            reward['mouse'] = 200
            reward['cat'] = -200
```

```
58              info['cheese_eaten'], info['x'], info['y'] = True,  self.MOUSE_X, self.
    MOUSE_Y
59
60          # Cat ha mangiato mouse
61          if self.CAT_X == self.MOUSE_X and self.CAT_Y == self.MOUSE_Y:
62              done = True
63              reward['cat'] = 200
64              reward['mouse'] = -200
65              info['mouse_caught'], info['x'], info['y'] = True,  self.MOUSE_X, self.
    MOUSE_Y
66
67          return self.get_state(), reward, done, info, toccate_muro_mouse,
    toccate_muro_cat, toccate_ostacolo_mouse, toccate_ostacolo_cat
```

In questo metodo viene controllata e gestita ogni possibile azione degli agenti. Ad esempio viene controllato se l'agente si muove nella direzione di un ostacolo o di un muro esterno e, se ciò avviene, gli viene tornato un reward negativo. In questo metodo sono anche aggiornate opportunamente le posizioni degli agenti. Inoltre, viene controllato se il gatto mangia il topo oppure se il topo mangia il formaggio (situazioni di terminazione della partita), e nel caso in cui una delle due situazioni si verifica sono restituiti gli opportuni reward.

Tutti gli altri metodi di questa classe sono di comodo e servono a:

- controllare se gli agenti sono in prossimità di un ostacolo o di un muro esterno (informazioni relative allo stato)
- controllare se gli agenti si muovono verso un muro o un ostacolo (informazioni relative allo spostamento e al reward)
- modificare e aggiornare le posizioni degli agenti

Per quanto riguarda gli altri metodi:

- **check_towards_obstacle()**: controlla se l'agente si è mosso verso un ostacolo. In questo caso l'agente in realtà non cambia di posizione ma riceve un reward negativo. Questo perchè l'obiettivo è fargli apprendere anche il fatto di evitare gli ostacoli.
- **check_out_of_bounds()**: controlla se l'agente si è mosso verso un muro esterno, tentando di uscire dalla mappa. Anche in questo caso la posizio-

ne dell'agente non cambia ma riceve un reward negativo. L'obiettivo è fargli apprendere che non deve uscire dai bordi della mappa.

- **update_positions()**: aggiorna le posizioni degli agenti in base all'azione eseguita.
- **get_changes()**: ritorna i valori di spostamento in base all'azione eseguita.
- **checkWall()**: controlla se l'agente si trova in prossimità di uno o due muri esterni, ovvero in una delle quattro celle dell'intorno **UP, DOWN, LEFT, RIGHT**.
- **checkDoubleObstacles()**: controlla se l'agente si trova in prossimità di uno o due ostacoli, tra le quattro celle dell'intorno **UP, DOWN, LEFT, RIGHT**.
- **checkTripleObstacles()**: controlla se l'agente si trova in prossimità di un terzo ostacolo, tra le quattro celle dell'intorno **UP, DOWN, LEFT, RIGHT**.

2.2.1 Gatto Sentinella

In questa implementazione le principali differenze sono:

- **createObstacles()**
- **get_state()**
- **reset()**
- **step()**
- **checkRegularPosition()**

createObstacles()
Il gatto sentinella in questa istanza si muove lungo l'asse verticale, esattamente a metà della mappa. Di conseguenza gli ostacoli possono essere posizionati in qualunque altro punto della mappa, al di fuori dell'asse su cui il gatto si muove.

```
    def createObstacles(self, n, cat_axis):
        possible_obstacles = []
        for x in range(n):
            if x == cat_axis:
                pass
            else:
                for y in range(n):
                    possible_obstacles.append((x, y))

        return tuple(possible_obstacles)
```

get_state()

Essendo solo il topo l'agente intelligente, in questa implementazione lo stato tornato è dato solo dalla tripla *<Distanza di Manhattan tra topo e formaggio, Distanza di Manhattan tra topo e gatto, informazioni sugli ostacoli>*

```
    def get_state(self):
        wall = self.checkWall()
        obstacles_first = self.checkDoubleObstacles(wall)
        if wall != obstacles_first:  # Used to avoid worthless operation in case
    there are not obstacles near the agent
            obstacles_second = self.checkDoubleObstacles(obstacles_first)
            if obstacles_first != obstacles_second:  # Used to avoid worthless
    operation in case there are not more than 1 obstacle near the agent
                obstacles = self.checkTripleObstacles(obstacles_second)
            else:
                obstacles = obstacles_second
        else:
            obstacles = obstacles_first

        self.STATE = {'mouse':((self.MOUSE_X - self.CAT_X) + (self.MOUSE_Y - self.
    CAT_Y),
            (self.MOUSE_X - self.CHEESE_X) + (self.MOUSE_Y -  self.CHEESE_Y),
            obstacles)}

        return self.STATE
```

reset()

Il posizionamento iniziale degli elementi nella mappa all'inizio di ogni epoca rispetta i seguenti criteri:

- il topo viene posizionato in modo casuale nella parte sinistra della mappa
- il gatto viene posizionato ad un'altezza casuale sull'asse verticale di metà mappa
- il formaggio viene posizionato in modo casuale nella parte destra della mappa

```
    def reset(self):
        self.MOUSE_X, self.MOUSE_Y = (np.random.randint(0, (self.WIDTH // 3 )-1), np.
    random.randint(0,9))
        self.CAT_X, self.CAT_Y = ((self.WIDTH // 2) -1 ,np.random.randint(0, 9))
        self.CHEESE_X, self.CHEESE_Y = (np.random.randint((self.WIDTH // 3 * 2)+1, 9)
    , np.random.randint(0, 9))

        self.checkRegularPosition()
        self.MOVES['mouse'] = 100
        return self.get_state()
```

step()

Per quanto riguarda questo metodo le principali accortezze sono state fatte riguardo il controllo e i movimenti del gatto sentinella. Il gatto infatti ha necessità di muoversi sempre in una direzione fino al raggiungimento di uno dei bordi esterni e, solo a quel punto di cambiare direzione (in quella opposta). In particolare il gatto si muove solo verso l'alto o verso il basso.

```
    def step(self, mouse_action, cat1_direction, cat2_direction):
        done = False
        mouse_action_null = False
        mouse_out_of_bounds = False
        cat1_out_of_bounds = False
        cat2_out_of_bounds = False
        reward = {'mouse': -1}
        toccate_ostacolo = 0
        toccate_muro = 0
        info = {    # Dict that contains all the information to keep during the
    process
            'cheese_eaten': False,
            'mouse_caught': False,
            'x': -1, 'y': -1,\
            'width': self.BLOCK_WIDTH,
            'height': self.BLOCK_HEIGHT
        }

        self.MOVES['mouse'] -= 1
        # done if moves = 0
        if self.MOVES['mouse'] == 0:
            done = True

        mouse_towards_obstacle = self.check_towards_obstacle(mouse_action, agent='
    mouse')
        if mouse_towards_obstacle:
            toccate_ostacolo +=1
            reward['mouse'] = -20
            mouse_action_null = True
```

```
28
29          mouse_out_of_bounds = self.check_out_of_bounds(mouse_action, agent='mouse')
30
31          cat1_out_of_bounds = self.check_out_of_bounds(cat1_direction, agent='cat1')
32          cat2_out_of_bounds = self.check_out_of_bounds(cat2_direction, agent='cat2')
33
34          if mouse_out_of_bounds:
35              reward['mouse'] = -20
36              toccate_muro +=1
37              mouse_action_null = True
38
39          if cat1_out_of_bounds:
40              if cat1_direction == 2:
41                  cat1_direction = 3
42              else:
43                  cat1_direction = 2
44
45          if self.CAT2_MODE == 'verticale':
46              #Gatto doppio verticale
47              if cat2_out_of_bounds:
48                  if cat2_direction == 2:
49                      cat2_direction = 3
50                  else:
51                      cat2_direction = 2
52          elif self.CAT2_MODE == 'misto':
53              if cat2_out_of_bounds:
54                  if cat2_direction == 0:
55                      cat2_direction = 1
56                  else:
57                      cat2_direction = 0
58
59          self.update_positions(mouse_action, cat1_direction, cat2_direction,
    mouse_action_null)
60
61          #mouse reached the cheese
62          if self.MOUSE_X == self.CHEESE_X and self.MOUSE_Y == self.CHEESE_Y:
63              done = True
64              reward['mouse'] = 200
65              info['cheese_eaten'], info['x'], info['y'] = True,  self.MOUSE_X, self.
    MOUSE_Y
66
67          #cat caught the mouse
68          if self.CAT1_X == self.MOUSE_X and self.CAT1_Y == self.MOUSE_Y:
69              done = True
70              reward['mouse'] = -200
71              info['mouse_caught'], info['x'], info['y'] = True,  self.MOUSE_X, self.
    MOUSE_Y
72
73          if self.CAT2_X == self.MOUSE_X and self.CAT2_Y == self.MOUSE_Y:
74              done = True
```

```
75              reward['mouse'] = -200
76              info['mouse_caught'], info['x'], info['y'] = True,  self.MOUSE_X, self.
    MOUSE_Y
77
78          return self.get_state(), reward, done, info, cat1_direction, cat2_direction,
    toccate_muro, toccate_ostacolo
```

checkRegularPosition()

Questo metodo serve a controllare che tra gli ostacoli, selezionati per essere utilizzati nell'epoca corrente, non ce ne siano alcuni che vadano ad occupare la stessa posizione in cui il topo o il formaggio vengono posizionati all'avvio. Nel caso in cui la posizione iniziale ritornata da **reset()** sia sopra la posizione di un ostacolo, il topo o il formaggio vengono riposizionati (e si ricontrolla se la posizione è regolare).

```
1   def checkRegularPosition(self):
2       for obs in self.OBSTACLES:
3           if self.CHEESE_X == obs[0] and self.CHEESE_Y == obs[1]:
4               self.CHEESE_X, self.CHEESE_Y = (np.random.randint((self.WIDTH // 3 *
    2)+1, 9), np.random.randint(0, 9))
5               self.checkRegularPosition()
6
7       for obs in self.OBSTACLES:
8           if self.MOUSE_X == obs[0] and self.MOUSE_Y == obs[1]:
9               self.MOUSE_X, self.MOUSE_Y = (np.random.randint(0, (self.WIDTH // 3 )
    -1), np.random.randint(0,9))
10              self.checkRegularPosition()
```

2.2.2 Gatto Sentinella Doppio

È possibile utilizzare questa istanza del problema in due modalità: due gatti sentinella che si muovono in verticale, oppure due gatti sentinella in cui uno si muove in verticale e l'altro per orizzontale.
In questa implementazione le principali differenze sono:

- **createObstacles()**: ha lo stesso scopo del metodo per le altre istanze. La differenza sta nella generazione dei possibili ostacoli e dipende soprattutto dalla modalità con cui lo si utilizza (*"verticale" o "misto"*)

```
1   def createObstacles(self, n, cat1_axis, cat2_axis, cat2_mode):
2       possible_obstacles = []
3       if cat2_mode == 'verticale':
4           for x in range(n):
5               if x == cat1_axis or x == cat2_axis:
```

```
                    pass
                else:
                    for y in range(n):
                        possible_obstacles.append((x, y))
        elif cat2_mode == 'misto':
            for x in range(n):
                if x == cat1_axis:
                    pass
                else:
                    for y in range(n):
                        if y == cat2_axis:
                            pass
                        else:
                            possible_obstacles.append((x, y))

        return tuple(possible_obstacles)

```

- **get_state()**: anche qui l'unica differenza sta nel fatto che essendoci un gatto in più lo stato ricevuto dal topo non è più dato da una tripla ma da una quadrupla: *<distanza di Manhattan dal gatto1, distanza di Manhattan dal gatto2, distanza di Manhattan dal formaggio, informazioni sui muri e ostacoli>*

- **reset()**. Le posizioni iniziali degli elementi costituenti sono state così suddivise, a seconda della modalità utilizzata:

  - **Verticale**:

    * topo: posizionamento casuale nella parte sinistra della mappa (prima dei due gatti).
    * gatto1: in una posizione casuale lungo l'asse verticale a 1/3 della mappa.
    * gatto2: in una posizione casuale lungo l'asse verticale a 2/3 della mappa.
    * formaggio: nella parte destra della mappa (nello spazio rimanente dopo i due gatti).

  - **Misto**:

    * I due gatti in questa modalità suddividono la mappa in 4 quadranti.
    * topo: posizionamento casuale nel quadrante in alto a sinistra della mappa.

    * gatto1: in una posizione casuale lungo l'asse verticale a metà mappa.
    * gatto2: in una posizione casuale lungo l'asse orizzontale a metà mappa.
    * formaggio: nel quadrante in basso a destra.

2.3 Agente

La classe **Agente** si occupa di gestire:

- scelta dell'azione in fase di training → **get_action()**
- formula di aggiornamento della Q-table → **Q_learn()**
- settare la policy ottimale a fine training → **set_policy()**
- salvare la policy per le esecuzioni successive → **save_policy()**
- caricare la policy per l'esecuzione → **load_policy()**
- scelta dell'azione dalla policy → **take_action()**

Per tutte le implementazioni si utilizza la stessa classe e metodi, infatti quest'ultima è indipendente dalle caratteristiche di gioco.

get_action()
Questo metodo è utilizzato dall'agente in fase di allenamento. In base al valore **epsilon** di input l'azione da eseguire è scelta in modo casuale, oppure scegliendo quella con il q-value massimo in base allo stato corrente sulla Q-table. Ciò rappresenta uno dei punti cruciali dell'allenamento, in cui avviene quello che è comunemente chiamato **rapporto tra Exploitation ed Exploration**. Per Exploitation si intende il procedere utilizzando le informazioni già acquisite (basandosi sull'esperienza), in questo modo l'agente sceglie l'azione basandosi sulla Q-table. Mediante l'Exploration invece l'agente si muove verso situazioni in cui potrebbe non essersi mai trovato. L'obiettivo di quest'ultima è quello di trovare nuovi percorsi migliori e continuare l'apprendimento.
Riassumendo: un agente interagisce con l'ambiente in uno di due possibili modi

- Il primo consiste nell'usare la q-table come riferimento e visualizzare tutte le azioni possibili per un determinato stato. L'agente seleziona quindi l'azione in base al valore massimo di tali azioni. Questo è noto come **exploitation** poiché utilizziamo le informazioni che abbiamo a nostra disposizione per prendere una decisione.

- Il secondo modo per agire è farlo in modo casuale. Questo si chiama **exploration**. Invece di selezionare le azioni in base alla ricompensa futura massima, selezioniamo un'azione a caso. Agire in modo casuale è importante perché consente all'agente di esplorare e scoprire nuovi stati che altrimenti potrebbero non essere selezionati durante il processo di sfruttamento.

Quindi, la mossa migliore viene scelta dalla Q-table con una probabilità di 1-epsilon altrimenti viene scelta casualmente.
In questo caso ho scelto di utilizzare un **epsilon greedy**. In particolare il valore **epsilon** di input cambia nel tempo, partendo da un valore alto e abbassandosi nel tempo con un tasso di decadimento variabile in modo da stoppare il decadimento al 75% della fase di allenamento. Il valore minimo (una volta interrotto il decadimento) per epsilon è di 0.05, in questo modo solo il 5% delle azioni (dopo il 75% di allenamento) saranno scelte in modo casuale. Questo ci permette di avere una fase di **raffinamento** nell'ultima parte dell'allenamento.

```
    def get_action(self, state, epsilon):
        bias = random.random()
        if bias > epsilon:
            return np.argmax(self.Q[state])
        else:
            return np.random.choice(np.arange(self.possibleActions))
```

Q_learn().
Questo metodo contiene la formula di aggiornamento della Q-table:

$$\underbrace{\text{New}Q(s,a)}_{\text{New Q-Value}} = Q(s,a) + \underbrace{\alpha}_{\text{New Q-Value}}[\underbrace{R(s,a)}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount rate}} \overbrace{\max Q'(s',a')}^{\text{Maximum predicted reward, given new state and all possible actions}} - Q(s,a)]$$

in base ai parametri **gamma e alpha**(descritti nella Sezione 2.4).

```
    def Q_learn(self, state, action, reward, next_state):
        self.Q[state][action] += self.alpha*(reward + self.gamma*np.max(self.Q[
    next_state]) - self.Q[state][action])
```

set_policy()

Questo metodo viene richiamato alla fine della fase di allenamento. Permette di salvare le azioni migliori (con il q-value maggiore) in base allo stato. In questo modo si ricava la policy ottimale dalla Q-table.

```
    def set_policy(self, saveQtable, dir):
        if saveQtable:
            self.saveQtableToCsv(dir)

        policy = defaultdict(lambda: 0)
        for state, action in self.Q.items():
            policy[state] = np.argmax(action)
        self.policy = policy
```

I metodi **save_policy()** e **load_policy()** permettono rispettivamente di salvare in formato *pickle*(utilizzando l'apposito modulo *python*) la policy e di caricarla da un path.

take_action()

Questo metodo, preso in input lo stato dell'agente, ritorna la migliore azione da eseguire secondo la policy

```
    def take_action(self,state):
        return self.policy[state]
```

2.4 Learning

Nel file **learning.py** viene eseguito l'allenamento degli agenti. In esso sono definiti i rispettivi parametri di allenamento e le modalità.
Ad esempio viene definito il set di parametri per l'algoritmo di Q-learning e il numero di epoche di allenamento. Inoltre è possibile impostare le informazioni relative alla percentuale di riempimento di ostacoli, la modalità dei gatti sentinella e l'aggiunta di altre caratteristiche in base all'istanza di gioco.

Il punto fondamentale del file è il loop in cui viene eseguito il Q-learning e aggiornata la Q-table. Ad ogni epoca:

- si carica un nuovo set di ostacoli, si riposizionano gli agenti e il formaggio
- si entra in un ciclo che dura n (numero di step) iterazioni oppure finchè uno dei due agenti vince la partita
- ad ogni iterazione si cambia lo stato dell'ambiente a seconda delle azioni eseguite e vengono dati i reward agli agenti

Qui di seguito è riportato il file relativo all'apprendimento dell'istanza in cui sia il gatto che il topo sono agenti intelligenti.

```
# env, grid and agent definitions
map = Matrix(rows=10, columns=10, max_pct_obstacles=pct_obstacles)
env = Env(display, map, cat_mode, map_mode)
mouse = Agent(env, possibleActions=4, alpha = 0.1, gamma = 0.85)
cat = Agent(env, possibleActions=4, alpha = 0.1, gamma = 0.85)

# Qlearning params
epsilon, eps_decay, eps_min = 1.0, 0.99992, 0.05

# Train epoch
num_episodes = 50000

for i_episode in range(1, num_episodes+1):
    env.set_obstacles(env.load_obstacles(map.OBSTACLES,pct_obstacles)) # Load
    different obstacles at each epoch)

    epsilon = max(epsilon*eps_decay, eps_min)

    state = env.reset()
    action_mouse = mouse.get_action(state['mouse'], epsilon)
    action_cat = cat.get_action(state['cat'], epsilon)

    # Render the environment
    env.render(i_episode)

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()

        next_state, reward, done, info, toccate_muro_mouse, toccate_muro_cat,
    toccate_ostacolo_mouse, toccate_ostacolo_cat  = env.step(action_mouse, action_cat
    )
```

```
32
33          mouse.Q_learn(state['mouse'], action_mouse, reward['mouse'], next_state['
    mouse'])
34          cat.Q_learn(state['cat'], action_cat, reward['cat'], next_state['cat'])
35
36          # Render the environment
37          display.fill(WHITE)
38          env.render(i_episode)
39          show_stats(cheese_eaten, mouse_caught)
40
41          if done:
42              if info['cheese_eaten']:
43                  cheese_eaten += 1
44                  draw_stats_pannel(GREEN, info['x'], info['y'], info['width'], info['
    height'])
45
46              if info['mouse_caught']:
47                  mouse_caught += 1
48                  draw_stats_pannel(RED, info['x'], info['y'], info['width'], info['
    height'])
49              # Break episode
50              break
51
52          # Update state and action
53          state = next_state
54          action_mouse = mouse.get_action(state['mouse'], epsilon)
55          action_cat = cat.get_action(state['cat'], epsilon)
```

Inoltre sono utilizzate numerose variabili per misurare e analizzare l'apprendimento, le quali saranno successivamente descritte (vedi Capitolo 3). La parte principale dell'algoritmo di apprendimento è comunque la stessa per tutte e tre le istanze del problema. Le uniche differenze sono relative ai valori dei parametri di allenamento e delle modalità di gioco.
Qui sono anche definiti i valori per **alpha e gamma**:

- **Tasso di apprendimento (lr o learning rate)**: spesso indicato come α, può essere semplicemente definito come quanta influenza ha il nuovo valore rispetto al vecchio valore.
- **Gamma** o γ: è un fattore di sconto. È usato per bilanciare la ricompensa immediata rispetto a quella futura. In genere questo valore può variare da 0,8 a 0,99.

I valori utilizzati per gamme ed alpha sono stati selezionati dopo numerosi test di ottimizzazione, e si è giunti ad un valore di 0.85 per **gamma** e ad un valore di 0.1 per

alpha. Il fatto che gamma non abbia un valore vicino al limite superiore è dovuto al fatto che l'agente deve anche apprendere a non dirigersi verso muri e ostacoli, questo può portare ad una o più mosse in più per circumnavigarli. Allo stesso tempo il topo ad esempio potrebbe preferire l'allontanarsi dal gatto, piuttosto che dirigersi a capofitto verso il formaggio in alcune situazioni.

- **Gatto sentinella singolo**: implementazione sia senza ostacoli che con ostacoli
  - **epsilon, eps_decay (tasso di decadimento di epsilon), eps_min** = 1.0, 0.9996, 0.05
  - **alpha** = 0.1
  - **gamma** = 0.85
  - **epoche di allenamento** = 10000 (nella versione senza ostacoli) e 20000 (nella versione con ostacoli)
  - **Riempimento ostacoli** = 0% (senza ostacoli) e 5%
- **Gatto sentinella doppio**: implementazione con doppio gatto verticale e misto (un gatto in orizzontale e uno in verticale), entrambe con gli stessi parametri di train
  - **epsilon, eps_decay, eps_min** = 1.0, 0.9999, 0.05
  - **alpha** = 0.1
  - **gamma** = 0.85
  - **epoche di allenamento** = 40000
  - **Riempimento ostacoli** = 4%
- **Gatto intelligente**: vi sono due modalità differenti, una in cui il gatto conosce anche la distanza tra lui e il formaggio nell'altra no. In entrambi i casi i valori per i parametri sono gli stessi. Vi è infine la possibilità di aggiungere degli ostacoli fissi intorno al formaggio per aumentare la difficoltà del problema.
  - **epsilon, eps_decay, eps_min** = 1.0, 0.99992, 0.05
  - **epoche di allenamento** = 50000

- **alpha** = 0.1
- **gamma** = 0.85
- **Riempimento ostacoli** = 7% se non si ha l'aumento di difficolà, altrimenti 4%

Per quanto riguarda i risultati e le informazioni ricavabili dal training, saranno analizzate nel Capitolo 3

2.5 Main

Il file **main.py** viene invece utilizzato per effettuare il test del programma, ovvero di testare le varie **policies** ottenute dalle fasi di allenamento. Per fare ciò è necessario creare un'istanza per l'ambiente, indicando la percentuale di riempimento di ostacoli e eventuali opzioni aggiuntive per indicare la modalità da utilizzare. Inoltre, è necessario caricare la corrispettiva policy ad ogni agente, le azioni saranno quindi gestite dall'apposito metodo **take_action()** precedentemente descritto (vedi Sezione 2.3). Qui di seguito è riportato il codice principale del file:

```
map = Matrix(rows=10, columns=10, max_pct_obstacles=pct_obstacles)
env = Env(gameDisplay, map, cat_mode, map_mode)
cat = Agent(env, possibleActions = 4)
mouse = Agent(env, possibleActions = 4)

# Numero di epoche
num_episodes = 10000

# Load the policies
dir = 'policies/gattoIntelligente/'
dir += (cat_mode + '/')
mouse.load_policy(dir+'mouse.pickle')
cat.load_policy(dir+'cat.pickle')

# loop over episodes
for i_episode in range(1, num_episodes+1):
    env.set_obstacles(env.load_obstacles(map.OBSTACLES,pct_obstacles)) # Load
    different obstacles at each epoch
    state = env.reset()

    action_mouse = mouse.take_action(state['mouse'])
    action_cat = cat.take_action(state['cat'])

    # Render the environment
```

```
    env.render(i_episode)
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()   # Close window
                quit()

        next_state, reward, done, info = env.step(action_mouse, action_cat)

        # Render the environment
        gameDisplay.fill(WHITE)
        env.render(i_episode)
        show_info(total_cheese_eaten, total_mouse_caught)

        # Updating the display
        pygame.display.update()
        clock.tick(12)

        if done:
            if info['cheese_eaten']:
                total_cheese_eaten += 1
                draw_rect(GREEN, info['x'], info['y'], info['width'], info['height'])

            if info['mouse_caught']:
                total_mouse_caught += 1
                draw_rect(RED, info['x'], info['y'], info['width'], info['height'])
            break

        # Update state and action
        state = next_state
        action_mouse = mouse.take_action(state['mouse'])
        action_cat = cat.take_action(state['cat'])

pygame.quit()
```

Il codice è molto simile a quello per l'apprendimento, la differenza sta nel fatto che le azioni sono prese seguendo la policy e non ci sono aggiornamenti relativi alla Q-table. I risultati di test, per verificare se effettivamente è stata appresa una policy ottimale e funzionante sono descritti nel Capitolo 3.

Capitolo 3

Analisi dei risultati di Train e Test

In questo capitolo sono riportati e analizzati i risultati ottenuti nelle fasi di allenamento e di test. Per ogni modalità verranno descritti i parametri utilizzati e i grafici ottenuti. In tutte le implementazioni, per motivi di tempistiche di allenamento la matrice avente il ruolo di mappa è stata istanziata di *dimensione 10 x 10*.

3.1 Gatto Sentinella singolo

L'implementazione con un singolo gatto avente il ruolo di sentinella è la prima da me studiata. Questo perchè, essendo la più semplice anche a livello di apprendimento, permette meglio di studiare eventuali problematiche riscontrate. Risulta quindi estremamente semplice capire se l'agente intelligente, ovvero il topo, è stato in grado di apprendere una policy ottimale che gli permette di evitare di farsi prendere dal gatto e di raggiungere il suo obiettivo (mangiare il formaggio).

3.1.1 Senza Ostacoli

In questa implementazione non sono neanche presenti gli ostacoli, per semplificare il più possibile il problema.
I parametri di allenamento utilizzati (come i parametri di Q-learning) sono stati descritti precedentemente, ma per una migliore visione sono nuovamente riportati.

- **Epoche**: 10000

- **Step**: 100
- **Alpha**: 0.1
- **Gamma**: 0.85
- **epsilon**: da 1 a 0.05 con un tasso di decadimento dello 0.9996

Qui di seguito sono riportati i risultati ottenuti dall'allenamento e successivamente i risultati ottenuti in fase di train.

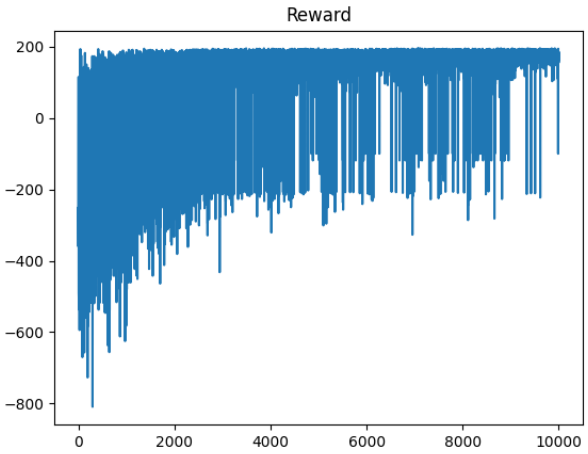

Figura 3.1: Reward ottenuti dall'agente con l'avanzare delle epoche di allenamento.

Risulta chiaro come il topo con il passare delle epoche, riesce ad apprendere un comportamento corretto in grado di fargli massimizzare il reward atteso. Inizialmente l'agente si muoverà in modo casuale, questo lo porta a farsi mangiare dal gatto sentinella molte volte ma grazie ai reward negativi, ottenuti in queste situazioni, il topo impara ad evitarlo. Allo stesso tempo il topo impara anche a raggiungere il formaggio e, come si può vedere in figura, il reward cumulativo di ogni epoca è costantemente vicino al valore di 200 (reward che si ottiene mangiando il formaggio). Ovviamente anche se non sono presenti ostacoli in questa istanza, il topo deve comunque apprendere il comportamento di evitare i muri esterni. Per questo è stato monitorato anche il conteggio di volte (per ogni epoca) in cui l'agente va a sbattere su di essi.

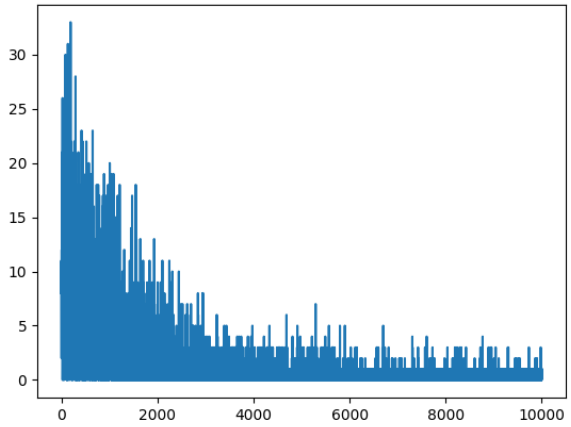

Figura 3.2: Conteggio di volte in cui l'agente va a sbattere su un muro esterno (per ogni epoca).

Anche qui risulta chiaro come l'obiettivo sia stato raggiunto. Il topo inizialmente tende spesso ad andare verso un muro esterno, mentre con il passare delle epoche questo conteggio tende progressivamente a diminuire (il fatto che non si annulli completamente è dovuto al fatto che anche nelle epoche finali c'è un 5% di possibilità di fare una mossa casuale).
Per monitorare ancora meglio ciò che accade nella fase di allenamento, nella figura qui di seguito sono riportati i conteggi di vittorie del topo rispetto al gatto con il passare delle epoche.
Si può notare come nella fase iniziale il gatto sovraperformi le performance del topo ma successivamente la linea del topo segue completamente la diagonale che taglia il grafico (questo indica che il topo vince ad ogni epoca). Questo è un chiaro esempio di un apprendimento eseguito correttamente.

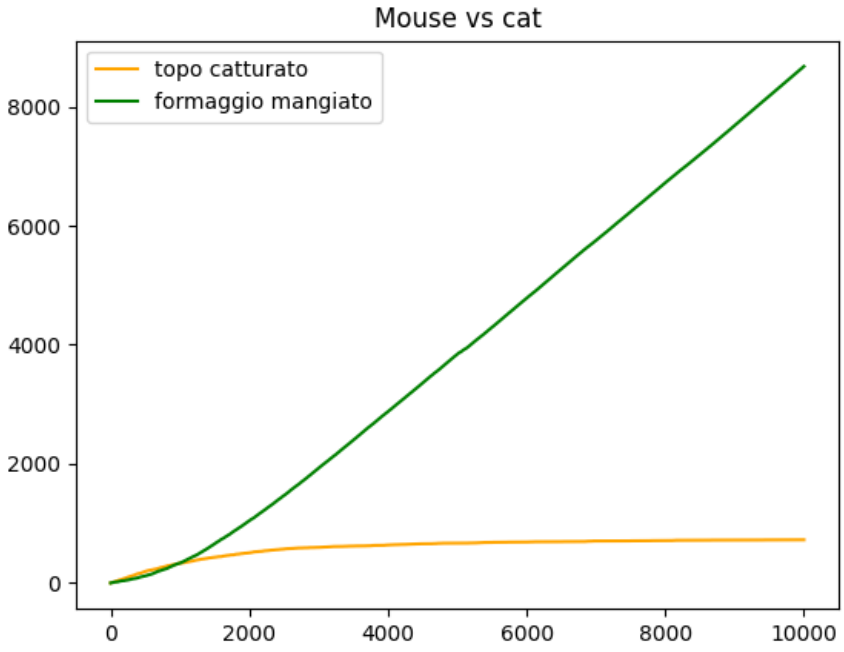

Figura 3.3: Vittorie Gatto vs Vittorie Topo in fase di allenamento.

Risulta ovviamente necessario anche valutare le performance in una fase di test successiva all'allenamento. Per fare ciò si utilizza la policy appena costruita e si testa l'agente.

I parametri di test necessari sono soltanto il numero di epoche e di step. Ovviamente gli step saranno gli stessi utilizzati per l'allenamento. Per quanto riguarda le epoche ho deciso di utilizzare, per questa implementazione e tutte le successive, 10000 epoche.

I risultati ottenuti sono:

- vittorie gatto: 0
- vittorie topo: 9970
- numero di volte in cui il topo sbatte nei muri esterni: 4313

Come si può vedere il gatto non vince neanche una volta e più del 99% delle volte il topo raggiunge il suo obiettivo. Inoltre, anche il numero di volte in cui il topo sbatte

su un muro è basso, infatti mediamente il topo tocca un muro 0.5 volte per epoca (data da un massimo di 100 step).

3.1.2 Con Ostacoli

Visti i buoni risultati ho deciso di aggiungere degli ostacoli posizionati casualmente secondo una percentuale di riempimento da me scelta. Ho inoltre ritenuto necessario aumentare il numero di epoche di allenamento in quanto si è introdotto un nuovo elemento che coinvolge direttamente l'apprendimento. Ho quindi utilizzato **20000 epoche di allenamento e una percentuale di riempimento di ostacoli del 5%**.

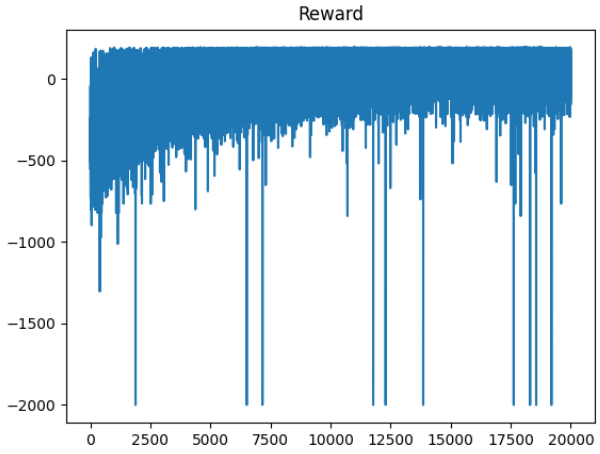

Figura 3.4: Reward ottenuti dall'agente con l'avanzare delle epoche di allenamento.

Anche in questo caso è evidente l'accrescere dei reward accumulati ad ogni epoca. Qui di seguito sono invece riportate le informazioni per il numero di volte in cui l'agente si muove verso un muro esterno, ma anche verso gli ostacoli. Chiaramente il topo impara ad evitare i muri e, come si può notare, anche il numero di ostacoli toccati diminuisce col progredire dell'allenamento, nonostante questi siano generati casualmente in posizioni diverse ad ogni epoca.

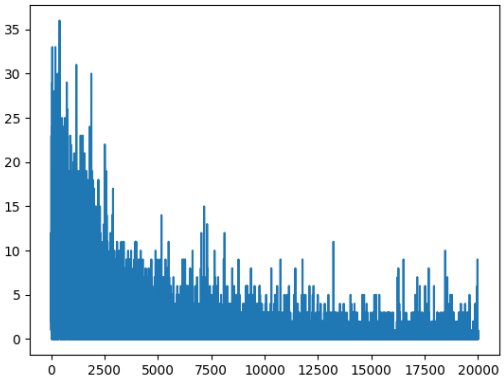

Figura 3.5: Conteggio muri esterni.

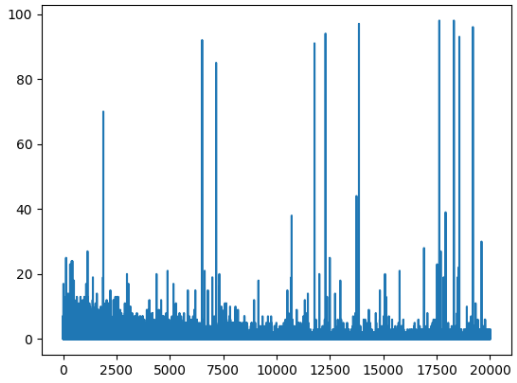

Figura 3.6: Conteggio ostacoli.

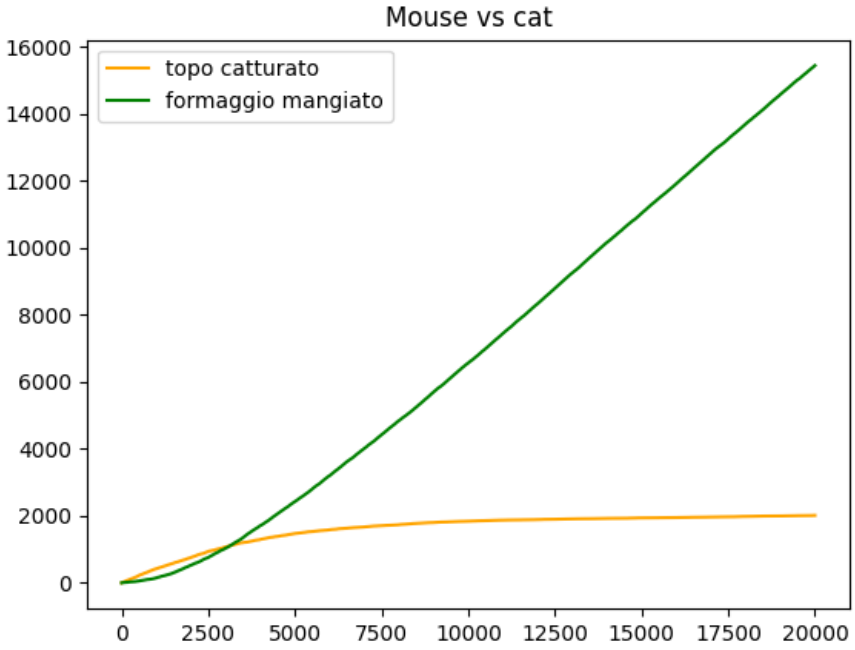

Figura 3.7: Vittorie Gatto vs Vittorie Topo in fase di allenamento.

Ovviamente sono stati fatti anche dei test per vedere come i punteggi per i reward vanno ad influenzare l'apprendimento. Qui di seguito sono riportati i risultati dando un **reward di -5 per ogni ostacolo toccato, invece di -20**.

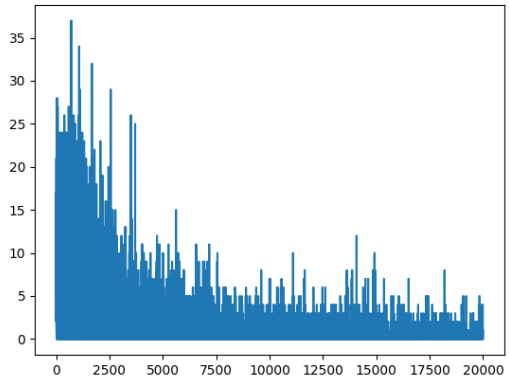

Figura 3.9: Conteggio muri esterni.

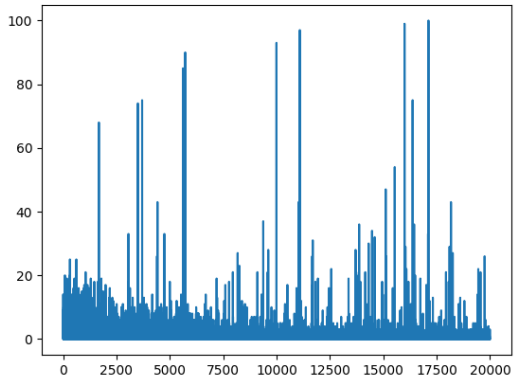

Figura 3.10: Conteggio ostacoli.

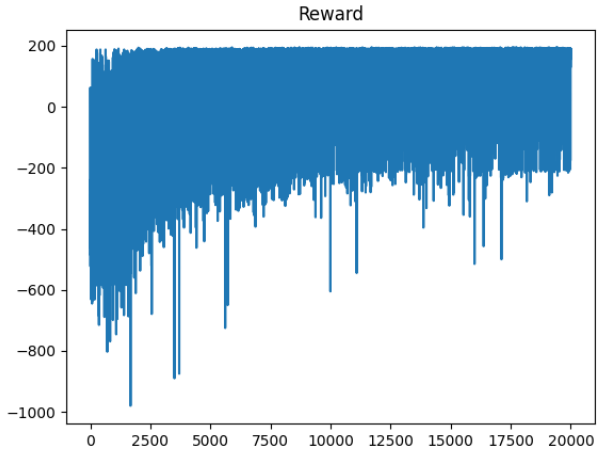

Figura 3.8: Reward ottenuti dall'agente con l'avanzare delle epoche di allenamento.

Dai grafici di allenamento non si notano particolari differenze rispetto alla prova precedente. Analizziamo ora i risultati in fase di test (10000 epoche di test).

	Test con reward di -20	**Test con reward di -5**
Vittorie gatto	48	10
Vittorie topo	8136	8412
topo contro muri esterni:	3865	1462
topo contro ostacoli:	4201	5656

Come si può notare l'influenza di questo parametro è relativamente bassa. Ci sono dei piccoli miglioramenti per quanto riguarda le vittorie ottenute dal topo, i muri toccati e le vittorie del gatto. Tuttavia, riducendo la penalità per ogni ostacolo toccato, si ha un apprendimento minore relativamente agli ostacoli toccati (toccare un ostacolo diventa meno grave). Nonostante non ci siano particolari differenze, le successive prove sono state effettuate con una penalità di -20 per ogni ostacolo toccato. Nulla vieta di utilizzare un valore di -5 o modificare i parametri come meglio si crede, per dare più influenza ad un concetto piuttosto che ad un altro. Nel mio caso ho deciso di mantenere bilanciata l'influenza dei muri e degli ostacoli in rapporto alla sconfitta (topo mangiato dal gatto) e alla vittoria (topo mangia il formaggio), in una proporzione di 1:10.

3.2 Gatto Sentinella doppio

In questa implementazione sono presenti due gatti, e in base alla modalità, il secondo si muove in verticale come il primo, oppure per orizzontale. La difficoltà del problema di apprendimento è maggiore anche in quanto vi è un elemento in più nello stato ricevuto dall'ambiente (distanza di Manhattan tra il topo e il secondo gatto). Questo va ad aggiungere una colonna nella Q-table, ma anche ad aumentare le possibili combinazioni dei valori dello stato, aumentando quindi la dimensione della Q-table e della policy.
In entrambe le implementazioni ho utilizzato **40000 epoche di allenamento e una percentuale di riempimento degli ostacoli del 4%**.

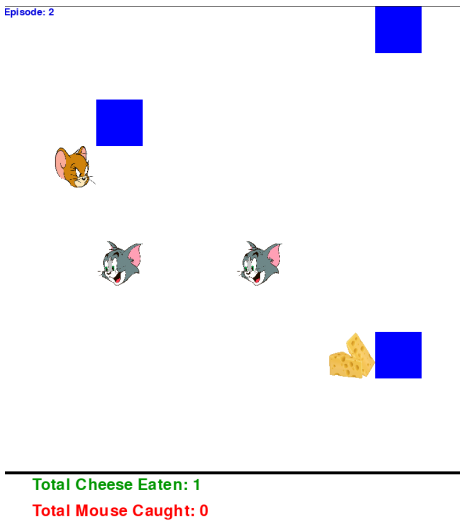

Figura 3.11: Modalità con doppio gatto Misto.

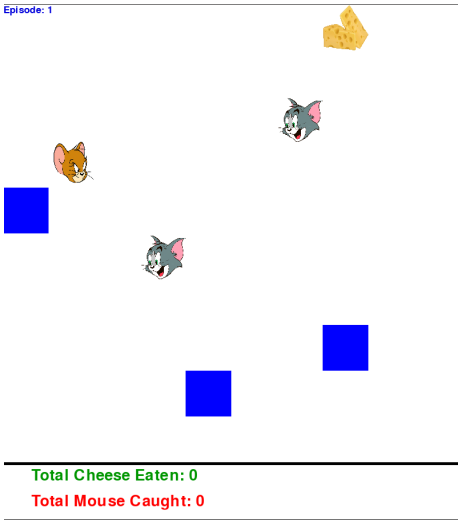

Figura 3.12: Modalità con doppio gatto Verticale.

3.2.1 Verticale

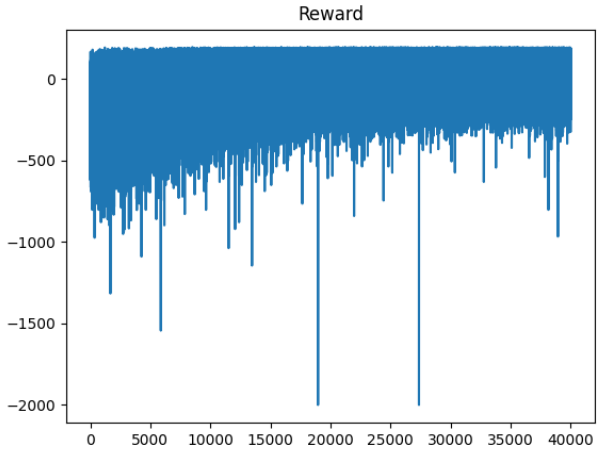

Figura 3.13: Reward ottenuti dall'agente con l'avanzare delle epoche di allenamento.

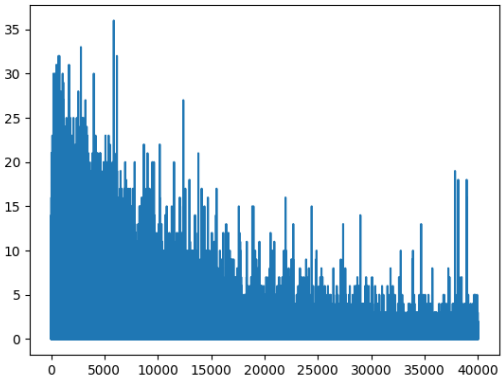

Figura 3.14: Conteggio muri esterni.

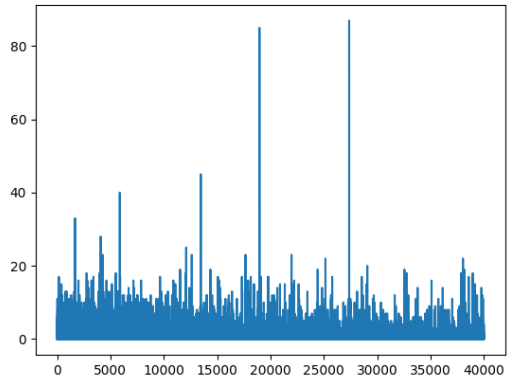

Figura 3.15: Conteggio ostacoli.

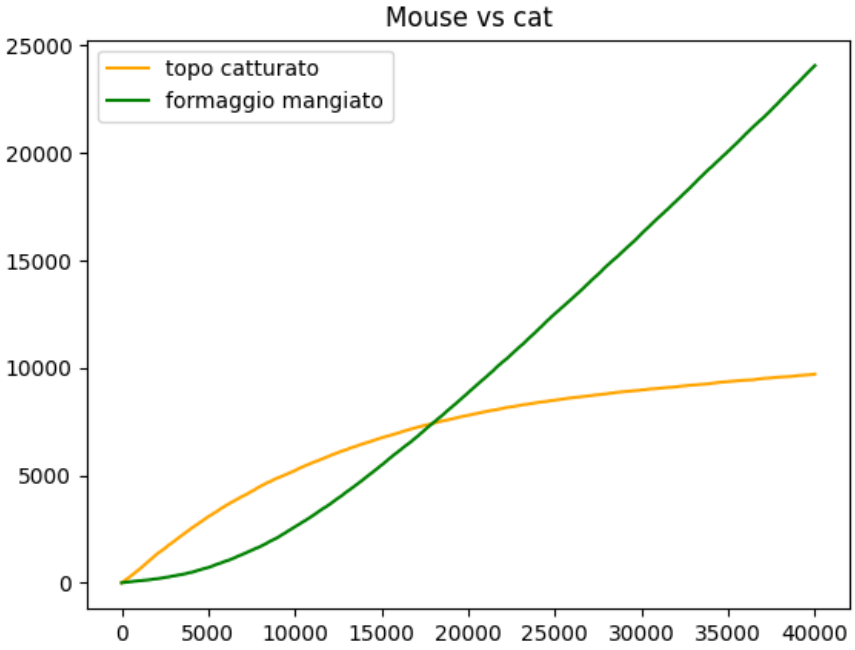

Figura 3.16: Vittorie Gatto vs Vittorie Topo in fase di allenamento.

3.2.2 Misto

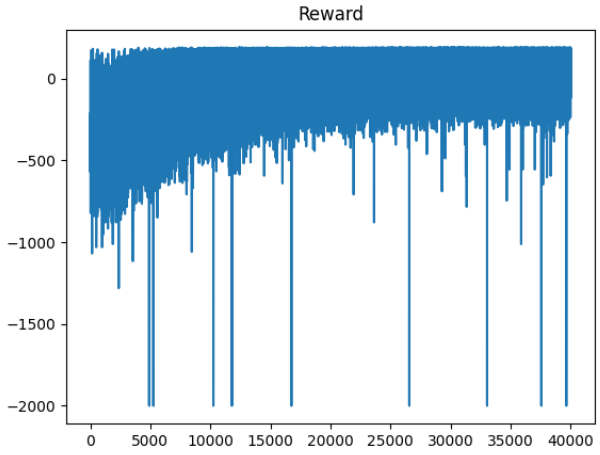

Figura 3.17: Reward ottenuti dall'agente con l'avanzare delle epoche di allenamento.

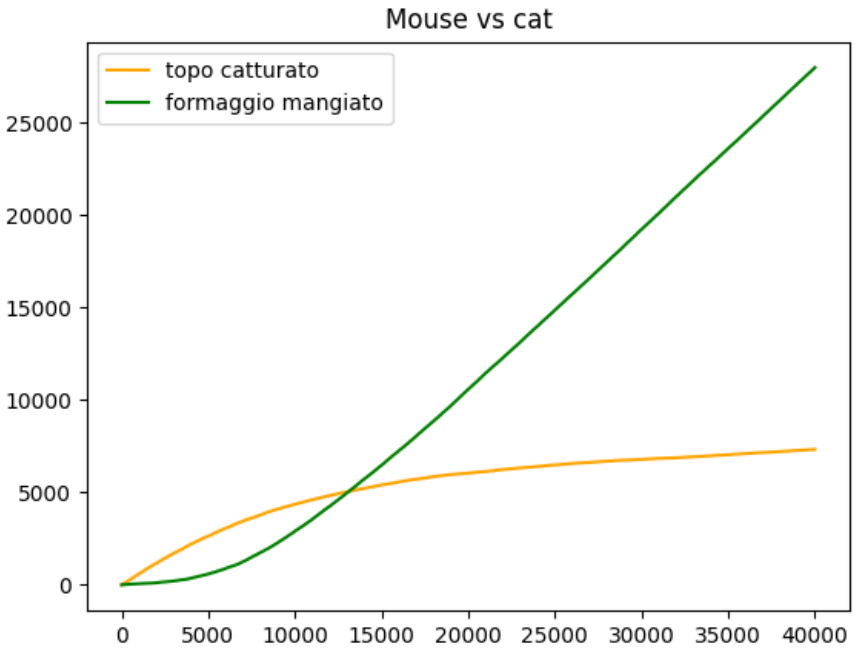

Figura 3.18: Vittorie Gatto vs Vittorie Topo in fase di allenamento.

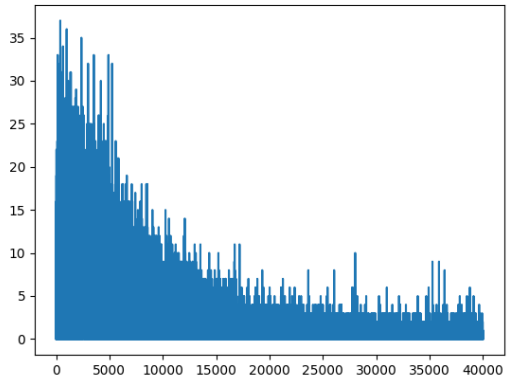

Figura 3.19: Conteggio muri esterni.

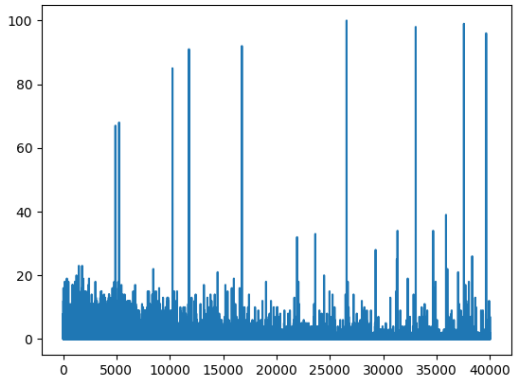

Figura 3.20: Conteggio ostacoli.

3.2.3 Analisi dei risultati

Come si può evincere dai grafici riportati precedentemente, per entrambe le istanze è stato possibile ottenere un buon apprendimento. Dai plot delle vittorie del gatto contro quelle del topo è possibile però capire quale dei due problemi è stato più difficile da allenare. Per quanto riguarda la modalità mista, il superamento del topo verso il gatto avviene circa all'epoca numero 15000, mentre nella modalità con entrambi i gatti in verticale avviene alla numero 20000.
Un'ulteriore conferma la si può avere dai risultati di test su 10000 epoche

	Modalità Mista	**Modalità verticale**
Vittorie gatto	239	263
Vittorie topo	8027	7053
topo contro muri esterni:	949	2527
topo contro ostacoli:	2197	2567

Si può vedere come le performance sono migliori in tutti i parametri per l'istanza in cui un gatto si muove per orizzontale e uno per verticale. Si ha una percentuale di vittorie di circa l'80% rispetto al 70% dell'altra modalità. Anche per quanto riguarda l'evitare i muri e gli ostacoli i risultati migliori sono prodotti nella modalità mista. In entrambe le situazioni si nota il fatto che una percentuale superiore al 15/20% di epoche terminano senza la vittoria di uno dei due agenti. In queste epoche si verifi-

cano infatti delle situazioni di **stallo**, in cui il topo non riesce a ricavare dalla policy un'azione per dirigersi verso il formaggio, senza farsi prendere da uno dei due gatti. Avviene quindi che il topo continua a muoversi in un piccolo intorno di celle senza muoversi in una direzione utile, fino al terminare degli step. Sono varie le cause possibili allo stallo, partendo da un numero di epoche non sufficienti per l'allenamento, fino ai parametri dell'algoritmo di Q-learning.
Sono stati testati diversi valori per gamma, portandolo fino ad 1, pensando che lo stallo fosse causato dal fatto che il topo venga portato in questa situazione a causa del suo essere conservativo nel rischio di farsi prendere dal gatto. Per valori inferiori ad 1 per γ, il topo ragiona cercando di massimizzare il reward totale atteso ma dando peso maggiore a quello immediato, potrebbe così preferire non rischiare un'azione (preferendo una penalità di -1) rispetto al rischio di farsi prendere dal gatto (penalità di -200). Questo potrebbe essere modificato aumentando il reward per aver mangiato il formaggio, ma in questo caso si è notato uno sbilanciamento di importanza e il topo si fa prendere più spesso dal gatto, gettandosi a capofitto verso il formaggio.
Per questo motivo, essendo l'ottimizzazione seguendo una griglia (per ottimizzare i valori dei parametri) molto costosa computazionalmente (un singolo allenamento richiede circa 2 ore), ho deciso di eseguire le altre implementazioni con questi parametri. Probabilmente si potrebbero ottenere risultati migliori utilizzando un algoritmo di Deep Q-learning. Tuttavia questi risultati sono comunque positivi, ho deciso di continuare aumentando la difficoltà del problema per vedere fin dove è possibile utilizzare l'algoritmo di Q-learning.
In aggiunta, in Figura 3.21 sono riportate delle righe di esempio della Q-table ricavata in queste istanze.

1	(-4, -1, -10, 0)	[-4.07868241 -3.64415162 -4.17806404 -2.79167902]
2	(-2, -1, -9, 0)	[-2.08127096 -59.76014159 -3.83280187 -17.3670528]
3	(0, -1, -8, 0)	[-4.06508434 -14.51617436 -1.94182358 13.46278201]
4	(2, -1, -7, 4)	[-0.59377538 8.95208943 -0.21292841 -5.42]
5	(0, -3, -8, 0)	[-6.27559325 1.21844381 -2.78242588 13.01800692]
6	(0, -3, -7, 4)	[-17.38403896 2.24008155 -0.69740788 -3.8]
7	(-1, -4, -7, 4)	[-1.46500212 -1.38212188 -1.54970736 -2.]
8	(-2, -5, -7, 4)	[-2.07616112 -20.46778042 -2.12587611 -3.84332353]
9	(-4, -7, -8, 0)	[-3.2775841 -1.76827233 -3.41132238 -3.45704009]
10	(-4, -7, -7, 4)	[-0.289523 -0.32188729 -0.3367462 -2.10082656]
11	(-4, -5, -6, 0)	[-2.80225655 1.042931 -2.80836755 -2.22855184]
12	(-4, -3, -5, 1)	[-6.94726282 -0.69754909 -0.91590427 -0.81604]
13	(-4, -1, -4, 0)	[-2.22761092 -2.21391855 -2.22447844 14.77985031]
14	(-4, -1, -5, 0)	[-2.42465671 -1.89518814 -2.7312316 0.93522194]
15	(-2, 1, -4, 0)	[-3.14024247 -6.8024781 -0.81657932 -18.34029754]
16	(-2, 1, -5, 0)	[-3.09170934 -22.52745499 -2.02924851 -7.45815557]
17	(-2, 1, -6, 0)	[-2.69477016 -3.30714239 -3.48959477 -2.94865821]
18	(0, 3, -5, 0)	[-2.83866419 -38.68630494 -2.48644903 18.01695568]

Figura 3.21: Alcune righe della Q-table nella modalità con due gatti sentinella.

3.3 Gatto Intelligente

Come ultima implementazione ho deciso di studiare l'istanza in cui entrambi gli agenti sono intelligenti, con l'obiettivo di analizzarne il comportamento. Come già descritto ognuno dei due agenti si costruisce una propria policy, entrambi però vengono allenati contemporaneamente. Per vedere se ci sono differenze, ho creato un'istanza in cui il gatto si basa solo sulla distanza con il topo e un'altra in cui conosce anche la posizione del formaggio. L'obiettivo è vedere se riesce a trarne vantaggio per battere il topo. Infine ho deciso di aggiungere un ulteriore difficoltà inserendo un confine di muri (una sorta di stanza) con due soli ingressi e vi ho messo il formaggio all'interno (vedi Figura 3.3).

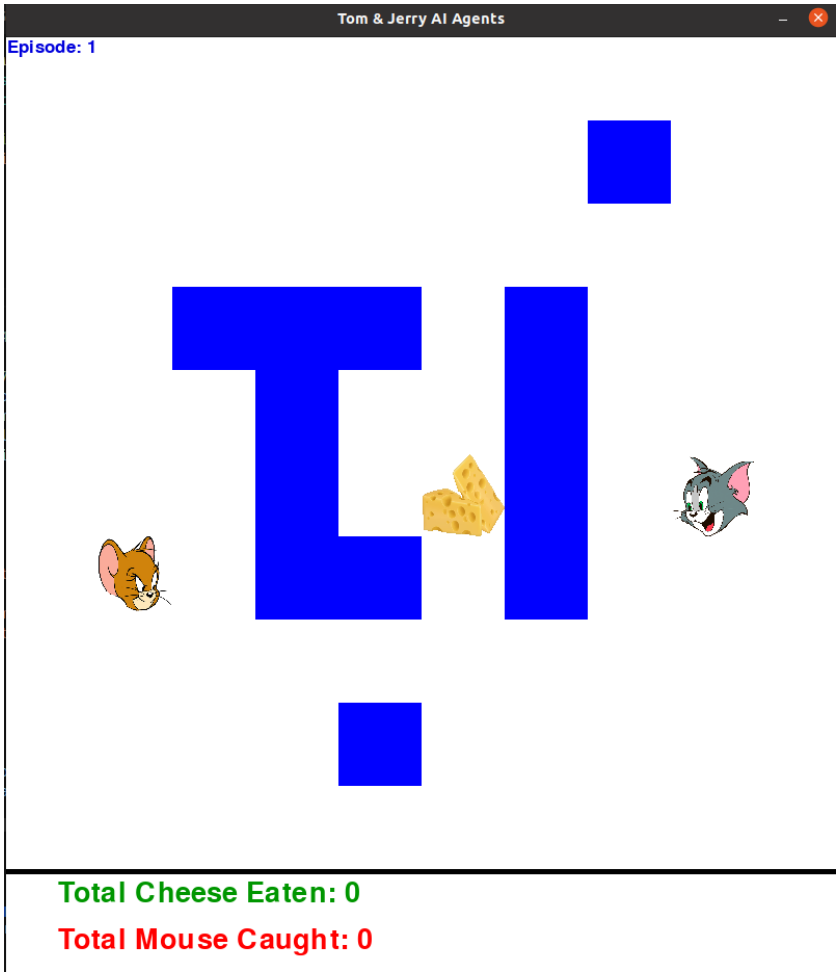

Figura 3.22: Modalità con stanza in cui rinchiudere il formaggio.

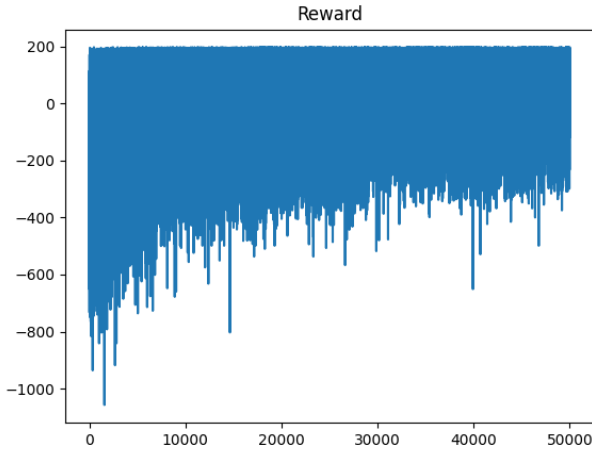

Figura 3.23: Reward topo modalità classica.

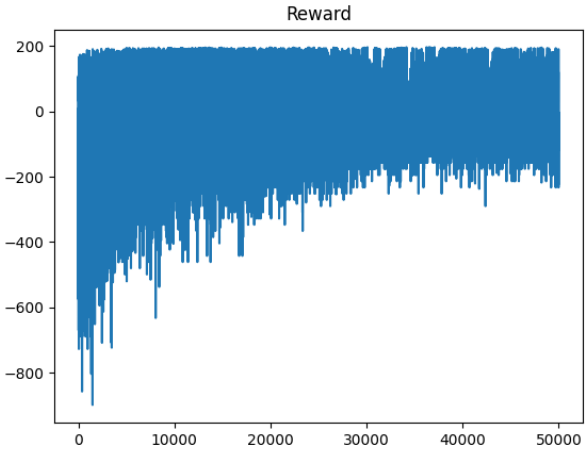

Figura 3.24: Reward gatto modalità classica.

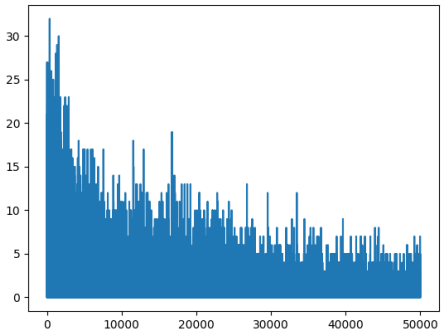

Figura 3.25: Conteggio muri esterni topo.

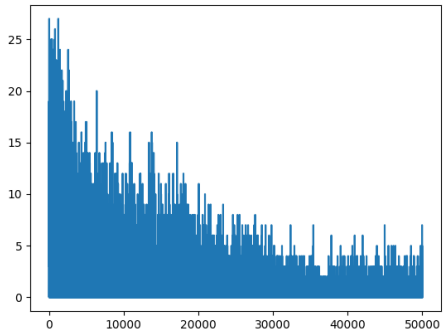

Figura 3.26: Conteggio muri esterni gatto.

3.3.1 Classico

In questa modalità il topo ha come input la tripla <*Distanza di Manhattan vero il gatto, distanza di Manhattan vero il gatto, informazioni su muro e ostacoli*>, mentre il gatto la tupla <*distanza di Manhattan verso il topo, informazioni sui muri e gli ostacoli*>.
L'allenamento per questa modalità e tutte le altre è stato fatto su **50000 epoche**, il riempimento per gli ostacoli è del 7%.

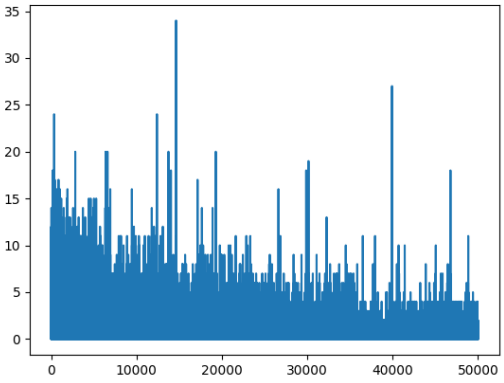

Figura 3.27: Conteggio ostacoli topo.

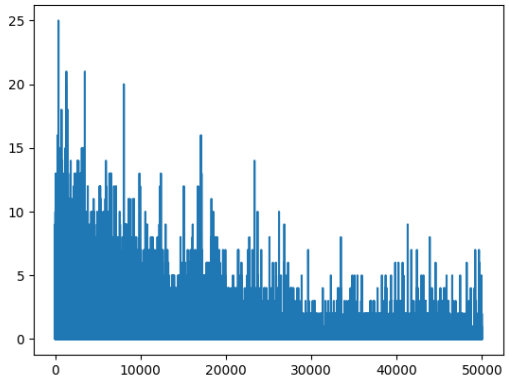

Figura 3.28: Conteggio ostacoli gatto.

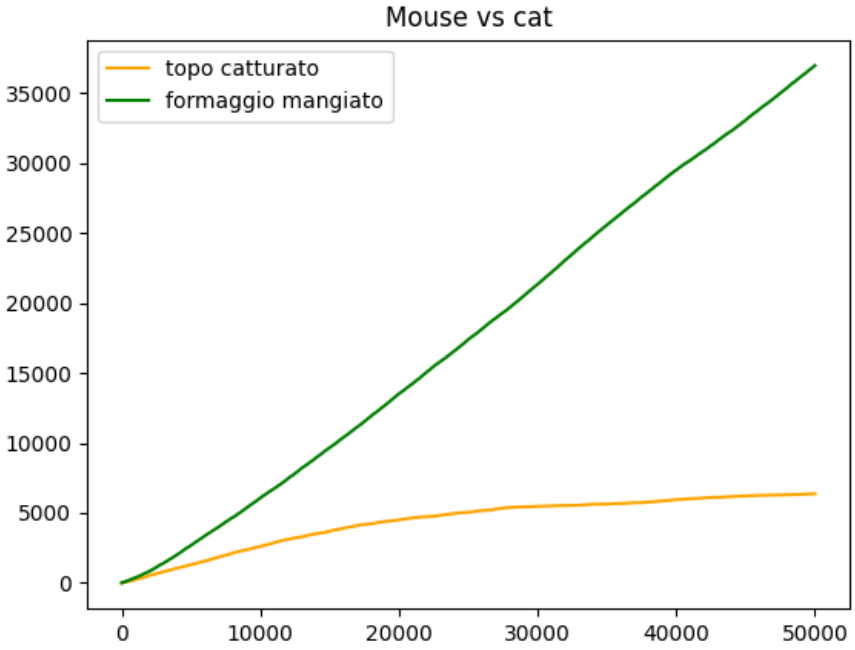

Figura 3.29: Vittorie Gatto vs Vittorie Topo in fase di allenamento.

Si può vedere come l'apprendimento avviene in modo corretto per quanto riguarda i muri e gli ostacoli. Per quanto riguarda l'obiettivo principale dei due agenti, il topo fin da subito riesce a sovraperformare il gatto (a differenza delle implementazioni in

cui i gatti erano sentinelle). Ciò è dovuto al fatto che inizialmente anche il gatto si muove senza logica e quindi risulta più difficile apprendere velocemente, soprattutto se il topo capisce prima quale è il suo obiettivo, rendendo ancora più difficile il compito del gatto. In aggiunta, i due agenti partono in una situazione di apparente parità, in cui nessuno dei due parte favorito, quindi è proprio la semplicità dell'obiettivo del topo che gli permette di sovraperformare il gatto. Infatti come si può vedere in figura l'apprendimento del gatto è molto più lento.

3.3.2 KnowCheese

Per provare ad aiutare quello che è l'apprendimento del gatto ho deciso di dare come informazione ricevuta nello stato del gatto, anche la posizione del formaggio, in modo da vedere se effettivamente questo lo aiuta.
Purtroppo, come chiaramente visibile in Figura 3.34 e in Figura 3.35, il problema precedente si ripete e il gatto non riesce a migliorare le sue performance nei confronti del topo.
Ho inoltre provato ad **aumentare sia il numero di epoche di allenamento, portandole a 80000 sia il numero di step, portandolo a 200. Ho inoltre provato ad allenare il gatto restituendogli una penalità di -200 ogni volta che il topo mangia il formaggio**. In entrambi i casi non si hanno risultati migliori, anzi si hanno dei peggioramenti nelle prestazioni. Si può notare soltanto un piccolo miglioramento nell'implementazione in cui si da la penalità al gatto (vedi Figura 3.35).

3.3.3 Difficoltà aggiuntiva

Non riscontrando buoni risultati per il gatto ho deciso di aggiungere un'ulteriore difficoltà nei confronti del topo. Ho quindi deciso di racchiudere in un perimetro (con due soli accessi) il formaggio. In questo modo il topo deve fare più fatica per raggiungere il formaggio, e di conseguenza il gatto ha più probabilità di prenderlo. Come nell'istanza precedente il gatto conosce anche la posizione del formaggio. Come si può vedere dalla Figura 3.36 in questo caso si ottengono i risultati desiderati: il gatto sovraperforma il topo per quasi tutto l'allenamento, solo nella fase finale viene raggiunto e superato. **Questo indica come per il topo è comunque più**

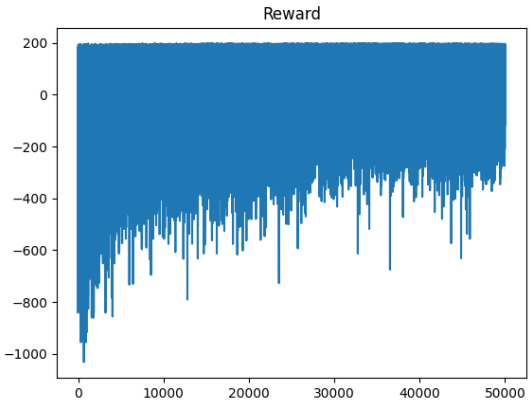

Figura 3.30: Reward topo modalità know-Cheese.

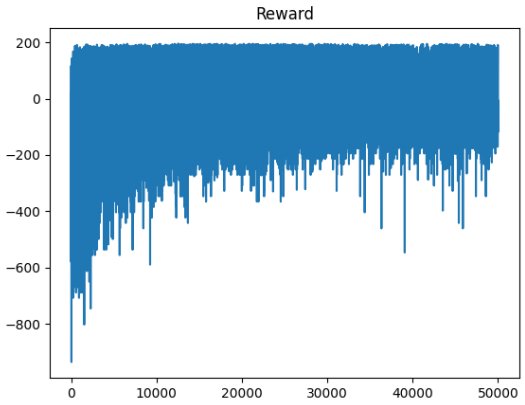

Figura 3.31: Reward gatto modalità know-Cheese.

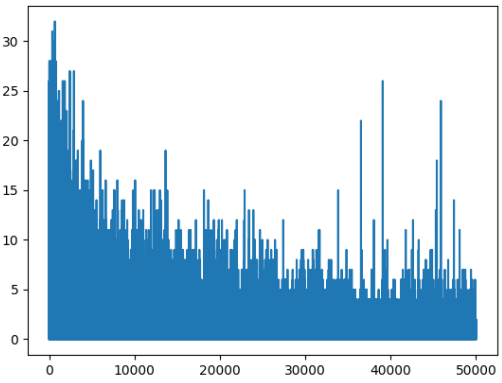

Figura 3.32: Conteggio muri esterni topo knowCheese.

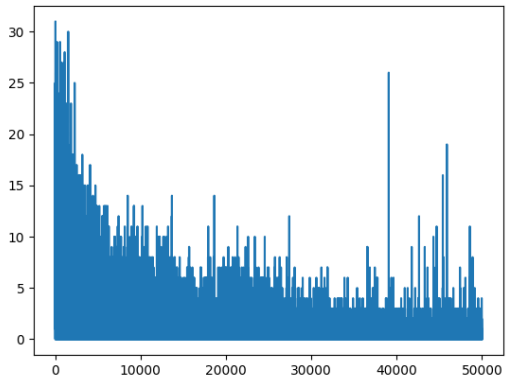

Figura 3.33: Conteggio muri esterni gatto knowCheese.

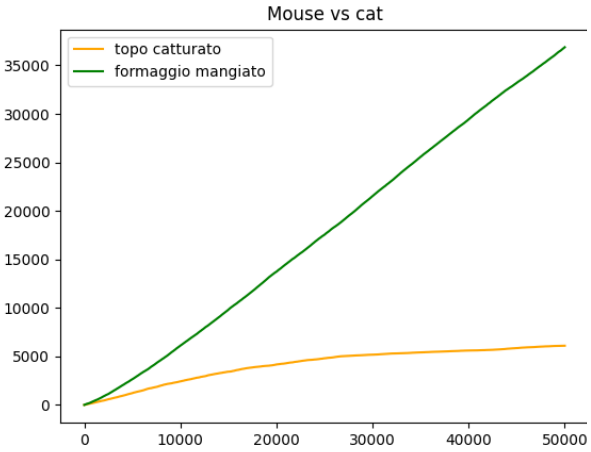

Figura 3.34: KnowCheese normale.

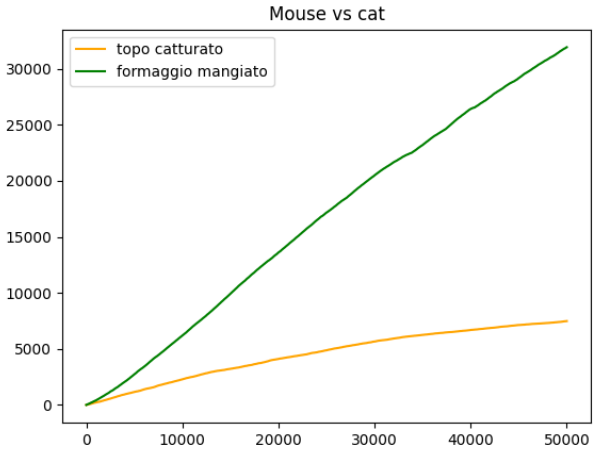

Figura 3.35: Penalità aggiuntiva.

semplice apprendere una policy e raggiungere il suo obiettivo, in quanto il suo obiettivo non si muove, a differenza di quanto fa il topo (obiettivo del gatto).

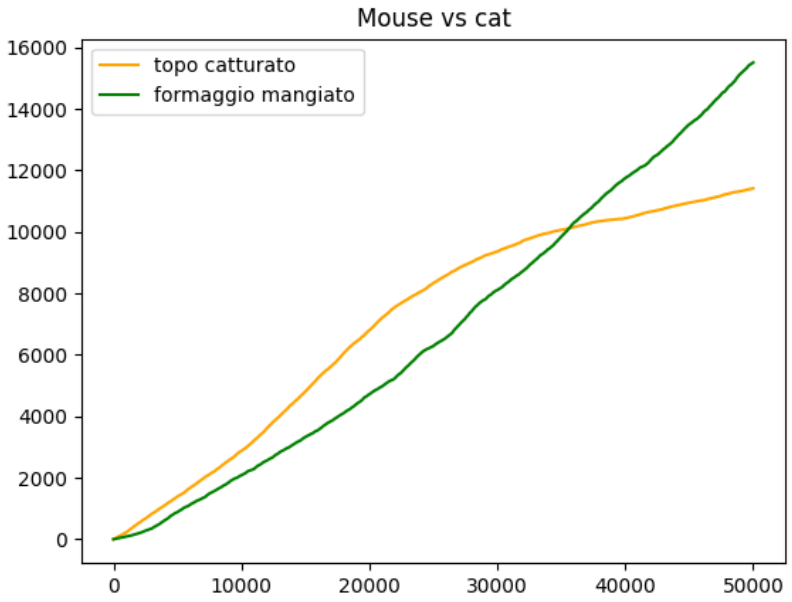

Figura 3.36: Vittorie Gatto vs Vittorie Topo in fase di allenamento.

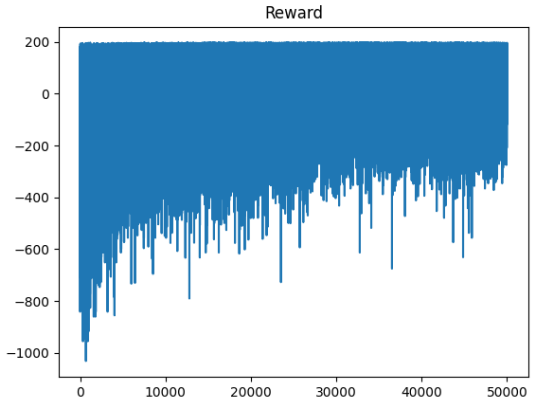

Figura 3.37: Reward topo.

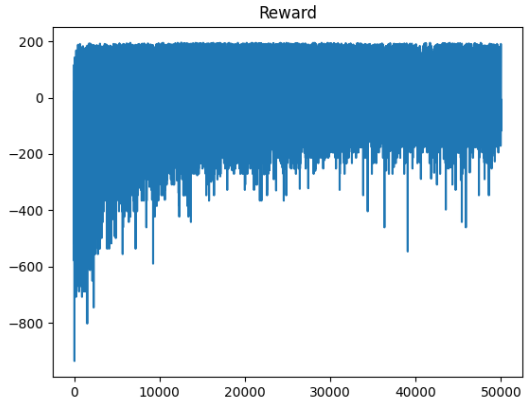

Figura 3.38: Reward gatto.

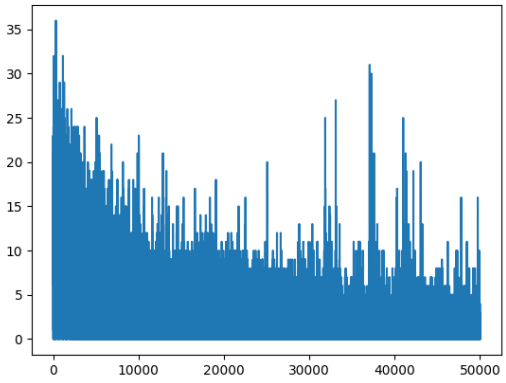

Figura 3.39: Conteggio muri topo.

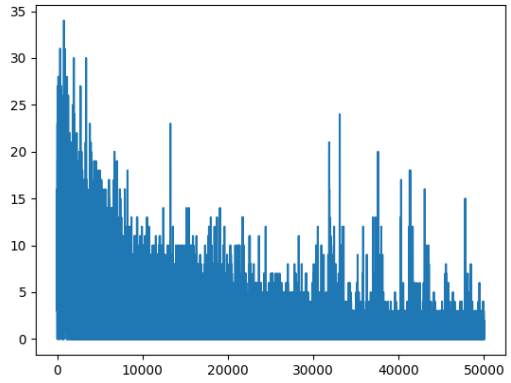

Figura 3.40: Conteggio muri gatto.

3.3.4 Analisi dei risultati

	Classico	**knowCheese**	**knowCheese + walls**
Vittorie gatto	0	25	12
Vittorie topo	4428	3557	75
topo contro muri esterni:	2958	3043	2256
topo contro ostacoli:	420	466	995
gatto contro muri esterni:	0	28	75
gatto contro ostacoli:	37	144	215

I risultati mostrano come all'aumentare della difficoltà il gatto riesce a tenere testa sempre di più al topo. Allo stesso tempo però, più riusciamo a mettere in difficoltà il topo e ad allenare il gatto, più aumentano le situazioni di stallo. Ci sono numerose epoche (oltre il 50%) in cui si verifica uno stallo e terminano in parità. Basti guardare la situazione in cui il formaggio è chiuso dentro le mura: il topo preferisce non farsi prendere piuttosto che tentare di andare a prendere il formaggio. Per questo motivo ho implementato un sistema che, ogni qualvolta in cui si verifica uno stallo porta gli agenti ad eseguire un'azione casuale, nella speranza di farli uscire dallo stallo.

	Classico	**knowCheese**	**knowCheese + walls**
Vittorie gatto	91	192	364
Vittorie topo	8253	7862	815

Capitolo 4

Conclusioni

In seguito all'analisi dei risultati, si può concludere che, utilizzando il Q-learning è possibile implementare il problema del gatto contro il topo.
È stato dimostrato come il topo riesca ad apprendere una policy in grado di portarlo a battere il gatto in ogni implementazione in cui quest'ultimo ha il ruolo di semplice sentinella (anche due gatti). Il problema aumenta di gran lunga la sua complessità nel momento in cui si tenta di implementare un'istanza dove entrambi gli agenti sono intelligenti, ovvero devono essere allenati per apprendere una policy. In questa situazione, se il gatto e il topo giocano alla pari, il topo ha comunque la meglio in quanto il suo obiettivo non si muove nel tempo e resta fisso. Al contrario l'obiettivo del gatto (mangiare il topo) è molto più difficile perchè quest'ultimo sa di doverlo evitare. In fase di test è stato possibile notare come il topo riesca a battere il gatto ma il numero di situazioni di stallo è notevolmente aumentato (più della metà delle epoche finiscono in stallo). Questo è dovuto al fatto che entrambi gli agenti sono intelligenti. Provando a dare un ulteriore aiuto al gatto, aumentando la difficoltà per il topo nel raggiungere il suo obiettivo, si nota infatti come quasi il 99% delle epoche (in fase di test) terminano in stallo, e nessuno dei due agenti riesce a raggiungere il proprio obiettivo.
Il verificarsi di questo problema è comunque segno del fatto che entrambi gli agenti abbiano appreso una policy, nonostante nessuno dei due riesca a sovrastare l'altro (nell'istanza più difficile).
Per quanto riguarda il train, quest'ultimo è piuttosto costoso in termini di tempo. Per risolvere il problema si potrebbe pensare di utilizzare il **deep Q-learning** costruendo

un modello, invece di creare tutte le possibili combinazioni dello stato per la Q-table da cui ricavare la policy (passaggio necessario utilizzando il semplice Q-learning).
In conclusione, i risultati ottenuti sono molto positivi, nonostante il Q-learning presenti molti limiti e sia solo un punto di partenza per studiare il Reinforcement Learning.