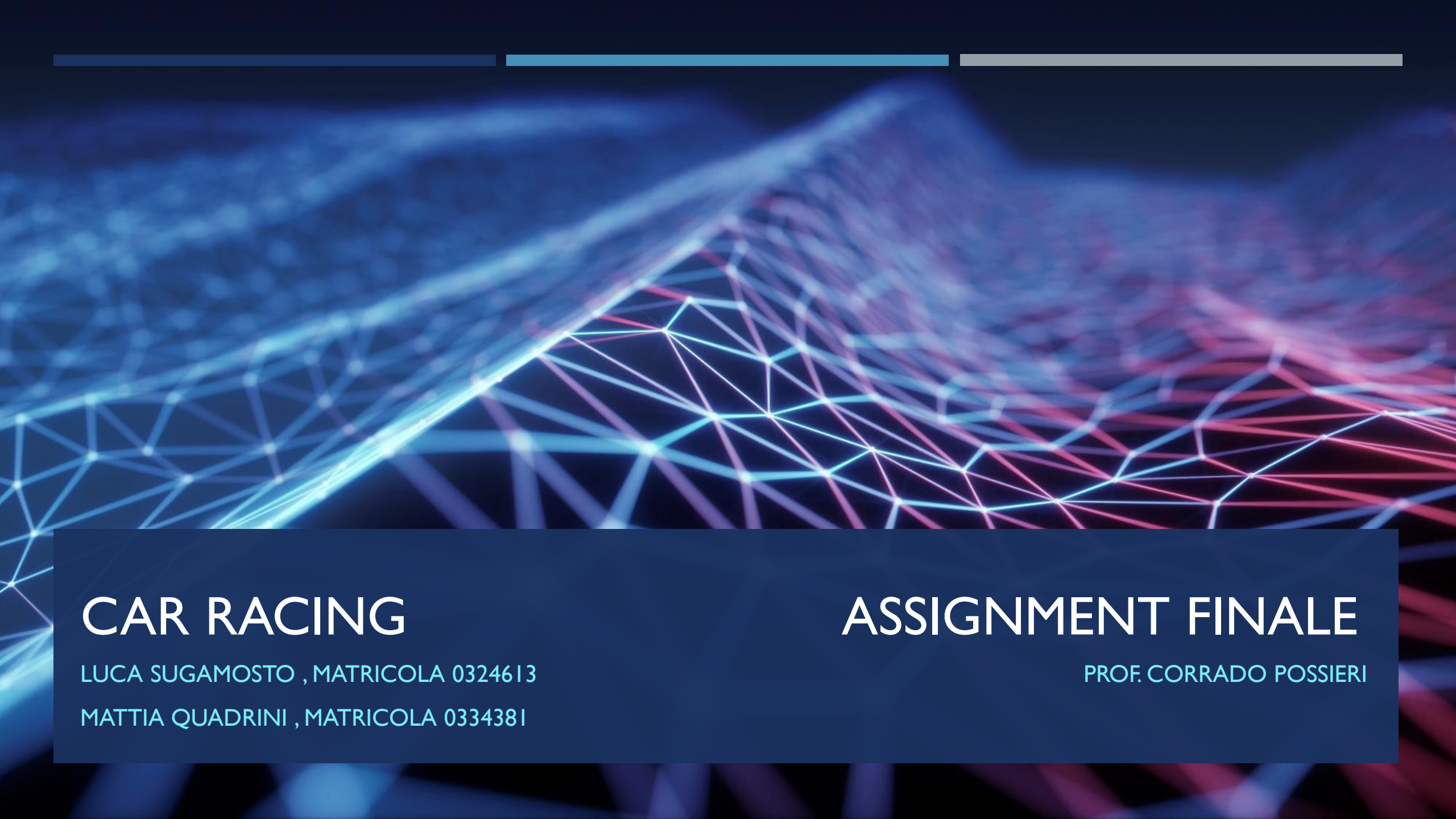


CAR RACING

ASSIGNMENT FINALE

LUCA SUGAMOSTO , MATRICOLA 0324613

MATTIA QUADRINI , MATRICOLA 0334381

PROF. CORRADO POSSIERI

OBIETTIVO DEL PROGETTO

Il $car - racing$ è un ambiente di gymnasium che consiste nell'addestrare un'auto all'interno di un circuito generato casualmente.

Le **azioni** possibili sono cinque in totale e tutte discrete :

- Non fare nulla;
- Girare tutto lo sterzo a sinistra;
- Girare tutto lo sterzo a destra;
- Accelerare;
- Frenare.

Lo **spazio osservabile** è una matrice $96\ x\ 96\ x\ 3$ (usando codifica RGB). Il punto di partenza è al centro della strada e deciso casualmente dall'ambiente durante l'inizializzazione. L'episodio termina quando l'auto esce dalla mappa oppure allo scadere di un numero fisso di iterazioni dettate dall'ambiente (1000).

Il **reward** è -0.1 per ogni frame e $1000/N$ per ogni porzione (tile) di pista visitata, dove N è il numero totale di porzioni di pista visitate.

ALGORITMO UTILIZZATO

$Sarsa(\lambda)$

TRACCE DI ELEGGIBILITÀ E LEGGI DI AGGIORNAMENTO

Una **traccia di eleggibilità** è una memorizzazione delle occorrenze associate ad un evento. Essa permette di colmare il divario tra eventi ed informazioni di addestramento.
Nella pratica consiste nell'aggiungere una variabile di memoria per ogni stato $s \in \mathcal{S}$, definita come $E(s)$.
La legge di aggiornamento delle tracce di eleggibilità dipende dallo stato considerato :

- Se questo non è stato visitato allora si ha un decadimento del valore di E, secondo la legge : $E_{t+1}(s) = \gamma\lambda E_t(s)$.
- Se questo è stato visitato si aggiorna secondo la seguente legge : $E_t(s_t) = (1 - \alpha)\gamma\lambda E_{t-1}(s_t) + 1$ (**Dutch trace**).

Per il calcolo della stima della funzione qualità, questa si compone di due fasi di aggiornamento :

- Nella prima fase si aggiorna l'errore TD per la predizione stato - valore :

$$\delta_t = R_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t);$$

- Successivamente si esegue l'aggiornamento della stima della funzione qualità Q :

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha\delta_t E_t(s,a) \qquad \forall s, a.$$

Funzione ‘carRacing_game2’

Funzione principale in cui viene inizializzato l’ambiente di lavoro (ed infine anche la sua chiusura).

Inoltre vengono settati i parametri utili per le leggi di aggiornamento ed il corretto funzionamento dell’algoritmo.

Le ultime linee di codice permettono di iniziare una nuova simulazione oppure continuare una simulazione già iniziata precedentemente (tecnica spiegata in dettaglio successivamente).

```python
import gymnasium as gym
import carRacingClasses2 as CRC2

#Il parametro 'render_mode = "human"' permette di rappresentare graficamente
#lo spazio di lavoro; se questo non si vuole visualizzare si esegue la seguente
#istruzione 'render_mode = None'
# env = gym.make("CarRacing-v2", render_mode = "human")
env = gym.make("CarRacing-v2", render_mode = None)

#Inizializzazione della classe in cui sono definite le variabili e le funzioni
#Primo input = numero di episodi
#Secondo input = alpha (Deve avere valore tale da rispettare le condizioni di convergenza => Alpha -> 0)
#Terzo input = epsilon (Epsilon -> 0 => si sfrutta la conoscenza per la scelta dell'azione | Epsilon -> 1 => si esplorano le azioni sconosciute)
#Quarto input = lambda (Lambda -> 0 => si ottiene un MC algorithm | Lambda -> 1 => si ottiene un one-step TD algorithm)
#Quinto input = gamma (Gamma -> 0 => importanza data ai reward immediati | Gamma -> 1 => importanza data ai reward futuri)
#Sesto input = k

numEpisodes = 10000
Alpha = 0.005
initialEpsilon = 0.8
Lambda = 0.8
Gamma = 0.5
k = 2
Class = CRC2.carRacingClass2(numEpisodes, Alpha, initialEpsilon, Lambda, Gamma, k)

#Inizializzazione delle variabili utili alla fase di aggiornamento.
#Se 1° parametro in input vale 0 allora si inizial una nuova simulazione,
#se invece vale 1 allora si continua una vecchia simulazione
Class.initStage(1)

#Esecuzione dell'algoritmo di apprendimento.
#Se 2° parametro in input vale 0 allora si inizial una nuova simulazione,
#se invece vale 1 allora si continua una vecchia simulazione
Class.SARSALambda(env, 1)

env.close()
```

Funzione ‘__init__’

La seguente funzione è utilizzata per inizializzare i parametri utili al fine del corretto funzionamento dell’algoritmo.
In particolare sono definiti ed inizializzati :

- La matrice delle azioni ‘actionMatrix’ contenente tutte le azioni discrete che può eseguire l’auto;
- I parametri necessari per le varie leggi di aggiornamento, per la gestione del numero di iterazioni e per l’algoritmo di scelta delle azioni;
- Alcuni parametri speciali per delle specifiche sotto - funzioni.

```python
class carRacingClass2:

    def __init__(self, numEp, alp, eps, lam, gam, K):
        #Inizializzazione delle variabili utili al fine del gioco
        self.sizeSpace = 96                     #Dimensione dello spazio osservato lungo una direzione
        self.forwardDistance = 68               #Dimensione dello spazio osservato in avanti
        self.A = 5                              #Numero di azioni possibili
        self.actionVariable = 3                 #Dimensione del vettore indicante l'azione
        self.steering = np.array([-1, 1])       #-1 = sterzata a SX, 1 = sterzata a DX
        self.gas = np.array([0, 1])             #0 = non accelerare, 1 = accelerare
        self.breaking = np.array([0, 1])        #0 = non frenare, 1 = frenare

        #Se l'azione "gas", l'azione "breaking" e l'azione "steering" valgono
        #0 allora si sta considerando l'azione "do nothing"

        #Inizializzazione della matrice delle azioni
        self.actionMatrix = np.zeros([self.A, self.actionVariable])

        for i in range(self.A):
            #Prima azione della matrice è "do nothing" quindi [0, 0, 0]

            #Seconda azione della matrice è "steer left" quindi [-1, 0, 0]
            if (i == 1):
                self.actionMatrix[i, 0] = self.steering[0]

            #Terza azione della matrice è "steer right" quindi [1, 0, 0]
            elif (i == 2):
                self.actionMatrix[i, 0] = self.steering[1]

            #Quarta azione della matrice è "gas" quindi [0, 1, 0]
            elif (i == 3):
                self.actionMatrix[i, 1] = self.gas[1]

            #Quinta azione della matrice è "breaking" quindi [0, 0, 1]
            elif (i == 4):
                self.actionMatrix[i, 2] = self.breaking[1]

        #Inizializzazione dei parametri decisi e modificati dall'utente
        self.numEpisodes = numEp                #Numero di episodi totali
        self.alpha = alp                        #Parametro applicato all'aggiornamento
        self.epsilon = eps                      #Parametro applicato all'aggiornamento
        self.Lambda = lam                       #Parametro applicato all'aggiornamento
        self.gamma = gam                        #Parametro applicato all'aggiornamento
        self.k = K

        self.epsUpdate = 0.000075               #Indica di quanto diminuire "epsilon" ad ogni episodio
        self.saveVariable = 25                  #Indica dopo ogni quanti episodi salvare le variabili utili
```

Funzione ‘initStage’

Funzione che riceve in ingresso una variabile $case$ che può assumere solo valori $\in \{0,1\}$.

- $case = 0 \rightarrow$ ciò significa che si deve iniziare una nuova simulazione e quindi sono inizializzate la matrice della stima della funzione qualità Q, il vettore delle ricompense G ed il parametro ε.
- $case = 1 \rightarrow$ ciò significa che si deve continuare una simulazione già avviata, quindi si caricano dalla cartella selezionata i vettori / matrici salvati in precedenza ed il parametro ε.

```
def initStage(self, case):
    #Inizializzazione del vettore della stima della funzione qualità e del
    #vettore dei rewards
    self.S = (self.sizeSpace ** 2) * self.forwardDistance   #Dimensione delle variabili dello spazio di stato
    if (case == 0):
        #Nuova simulazione che richiede l'inizializzazione (per la 1° volta)
        #del vettore delle stime della funzione qualità e nel vettore dei
        #rewards totali
        self.Q = np.random.randn(self.S, self.A)               #Inizializzazione casuale del vettore della stima della funzione qualità
        self.G = np.zeros([self.numEpisodes, 1])               #Inizializzazione nulla del vettore dei rewards
    elif (case == 1):
        #Si continua lo studio con un vettore Q già addestrato in precedenza
        #ed un vettore G già popolato di reward precedenti
        file_path_Q = r'C:\Users\quadr\Desktop\Assignment FINALE 2\Q.npy'
        self.Q = np.load(file_path_Q)
        file_path_G = r'C:\Users\quadr\Desktop\Assignment FINALE 2\G.npy'
        self.G = np.load(file_path_G)
        #Si carica anche il valore di Epsilon a cui si era arrivati in
        #precedenza
        file_path_eps = r'C:\Users\quadr\Desktop\Assignment FINALE 2\eps.pkl'
        with open(file_path_eps, 'rb') as file:
            self.epsilon = pc.load(file)
```

Funzione ‘convert_RGB_GrayScale’

La seguente funzione riceve in ingresso l’immagine a colori dell’ambiente (di dimensione 96×96) e calcola il valore in scala di grigi per ogni pixel appartenente alle sole righe e colonne interessate.

Il motivo per cui avviene ciò (e non si calcola tutta l’immagine in scala di grigi) è per ridurre il costo computazionale ed il tempo di esecuzione della funzione ad iterazione.

```
def convert_RGB_GrayScale(self, image):
    #Conversione dello spazio osservato dalla rappresentazione a colori RGB
    #in rappresentazione a scala di grigi

    #Si consideta la riga associata al baricentro dell'auto (Riga 67)
    if (self.k == 2):
        #Per risparmiare costo computazionale e tempo si considera solo la
        #riga utile al calcolo, quindi non si converte tutta l'immagine
        self.grayMatrix = np.zeros([2, self.sizeSpace])   #Utilizzato per il calcolo della distanza orizzontale (1° riga)
                                                          #e della distanza verticale (2° riga)
        for j in range(self.sizeSpace):
            red = image[67, j, 0]
            green = image[67, j, 1]
            blue = image[67, j, 2]

            #Riempimento del vettore in scala di grigi
            grayScale = (0.2989 * red) + (0.5870 * green) + (0.1140 * blue) #Calcolo del valore di luminanza
            self.grayMatrix[0, j] = grayScale                               #Componente del vettore in scala di grigi

        #Per calcolare la distanza in avanti dell'auto, si converte in
        #scala di grigi la colonna 48, dalla riga 0 alla riga 67 (dove si
        #trova la punta dell'auto)
        for j in range(self.forwardDistance):
            red = image[j, 48, 0]
            green = image[j, 48, 1]
            blue = image[j, 48, 2]

            grayScale = (0.2989 * red) + (0.5870 * green) + (0.1140 * blue) #Calcolo del valore di luminanza
            self.grayMatrix[1, j] = grayScale                               #Componente del vettore in scala di grigi
```

Funzione ‘epsGreedy’

Tale funzione sfrutta il concetto di *Esplorazione* & *Sfruttamento* per selezionare l’azione da eseguire.

Per fare questo viene utilizzato il parametro $epsValue$ che corrisponde ad un valore di probabilità compreso nell’intervallo $[0,1]$.

- Con probabilità pari a $epsValue$ si seleziona un’azione casuale tra quelle proposte,
- Con probabilità $1 - epsValue$ si seleziona l’azione greedy (cioè l’azione che massimizza la funzione qualità Q).

```
def epsGreedy(self, state):
    #Algoritmo per la scelta dell'azione da prendere in ogni singola
    #iterazione
    randomProb = np.random.rand()                          #Generazione di un numero casuale di probabilità tra 0 ed 1

    if (randomProb < self.epsilon):
        #Caso in cui si prende un'azione casualmente
        indAstar = rd.randint(0, self.A-1)                 #Scelta randomica dell'azione, prendendo un indice a caso
        Astar = self.actionMatrix[indAstar, :]             #Azione presa definitivamente
    else:
        #Caso in cui si prende l'azione con valore di stima della funzione
        #qualità più alta
        indAstar = np.where(self.Q == np.max(self.Q[state, :]))[1][0]      #Si cerca l'indice in Q con valore massimo
        Astar = self.actionMatrix[indAstar, :]                             #Azione presa definitivamente
    return([Astar,indAstar])
```

Funzione ‘distanceCalculation’

Questa funzione utilizza la variabile globale $grayMatrix$:

- La prima riga della matrice $grayMatrix$ è un vettore contenente il valore di grigio di ogni pixel di una determinata riga dell’immagine iniziale $observation$; questa viene utilizzata per calcolare la distanza dell’auto dal bordo sinistro e del bordo destro.
- La seconda riga della matrice $grayMatrix$ è un vettore contenente il valore di grigio di ogni pixel di una determinata colonna dell’immagine iniziale $observation$ e viene utilizzata per calcolare la distanza dell’auto dal bordo anteriore.

I tre valori calcolati permettono di identificare uno stato specifico nella matrice degli stati.

```
def distanceCalculation(self):
    #Riceve la matrice in scala di grigi (Vettore se si considera una sola
    #riga dell'immagine di partenza) e calcola la distanza dell'auto dai
    #bordi laterali
    if (self.k == 2):
        for j in range(self.sizeSpace):
            if (self.grayMatrix[0, j] < 110):
                x_SX = j            #Posizione del bordo pista sinistro
                break               #Uscita anticipata dal ciclo
        for j in range(self.sizeSpace - 1, 0, -1):
            if (self.grayMatrix[0, j] < 110):
                x_DX = j            #Posizione del bordo pista destro
                break               #Uscita anticipata dal ciclo
        for j in range(self.forwardDistance):
            if (self.grayMatrix[1,j] < 110):
                x_FD = j            #Posizione del bordo pista avanti
                break               #Uscita anticipata dal ciclo

        #Calcolo delle distanze tra bordo pista e bordo auto
        distance_SX = 46 - x_SX     #46 = bordo SX dell'auto
        distance_DX = x_DX - 49     #49 = bordo DX dell'auto
        distance_FD = 67 - x_FD     #67 = bordo anteriore dell'auto

        #Normalizzazione delle distanze per avere un indice compreso tra 0
        #e (sizeMatrix - 1)
        normDistance_SX = distance_SX + 49
        normDistance_DX = distance_DX + 49
        normDistance_FD = distance_FD

    return([normDistance_SX, normDistance_DX, normDistance_FD])
```

Funzione ‘convertCoordinate’

```
def convertCoordinate(self, distanceVector):
    #Funzione che prende in ingresso il vettore delle distanze tra l'auto e
    #i bordi della strada e restituisce lo stato corrispondente sulla
    #matrice delle stime della funzione qualità Q
    state = np.ravel_multi_index((distanceVector[0], distanceVector[1], distanceVector[2]), (self.sizeSpace, self.sizeSpace, self.forwardDistance))
    return(state)
```

La funzione qui rappresentata prende in ingresso il vettore delle distanze, calcolato dalla funzione precedentemente descritta, e restituisce in uscita lo stato associato.

Questo viene utilizzato per aggiornare la traccia di eleggibilità per la specifica coppia stato - azione e successivamente la matrice di stima della funzione qualità Q.

Implementazione del $Sarsa(\lambda)$ con aggiunta di istruzioni utili per la fase di simulazione

```python
def SARSALambda(self, env, case):
    #Algoritmo per l'addestramento del gioco
    self.stableIteration = 50                   #n° di iterazioni utili alla stabilizzazione dello zoom e dell'ambiente

    if (case == 0):
        #Nuova simulazione che richiede di partire dall'episodio 0
        initialEpisode = 0
    elif (case == 1):
        #Continuare lo studio dall'episodio in cui ci si è fermati prima,
        #quindi si carica l'episodio salvato
        file_path_e = r'C:\Users\quadr\Desktop\Assignment FINALE 2\e.pkl'
        with open(file_path_e, 'rb') as file:
            initialEpisode = pc.load(file) + 1

    for e in range(initialEpisode, self.numEpisodes):
        start = tm.time()                       #Inizio calcolo del tempo necessario ad analizzare un singolo episodio

        if (e >= self.numEpisodes - 25):        #Numero di episodi dopo il quale rappresentare graficamente la mappa di gioco
            env = gym.make("CarRacing-v2", render_mode = "human")

        observation, info = env.reset(seed = 1)   #Inizializzazione dell'ambiente 2D ad ogni episodio

        for i in range(self.stableIteration + 2):
            if (i <= self.stableIteration):
                #Inizialmente l'ambiente deve stabilizzarsi e zoomare lo
                #spazio osservabile, quindi la macchina per questi istanti
                #di tempo esegue l'azione "do nothing"
                env.action_space = self.actionMatrix[0, :]
                action = env.action_space
                observation, reward, terminated, truncated, info = env.step(action)

                if (i == self.stableIteration):
                    #L'ambiente si è stabilizzato quindi si calcola lo
                    #stato di partenza
                    stableObservation = observation
                    carRacingClass2.convert_RGB_GrayScale(self, stableObservation)
                    distanceVector = carRacingClass2.distanceCalculation(self)

            else:
                #Una volta stabilizzato lo spazio osservato, ci troviamo
                #nell'iterazione successiva e può iniziare l'episodio

                #Diminuzione del valore di Epsilon ad ogni episodio, per
                #ridurre l'esplorazione ed aumentare lo sfruttamento dei
                #valori noti
                if (self.epsilon > 0.05):
                    self.epsilon = self.epsilon - self.epsUpdate

                #Passaggio dal vettore delle distanze al corrispettivo
                #indice della matrice della stima della funzione qualità
                state = carRacingClass2.convertCoordinate(self, distanceVector)

                self.E = np.zeros([self.S, self.A])     #Inizializzazione della variabile utile all'apprendimento
                #Calcolo dell'azione da prendere utilizzando l'algoritmo di
                #Epsilon Greedy
                [a, ind_a] = carRacingClass2.epsGreedy(self, state)

                #Contatore delle iterazioni all'interno del ciclo while
                count = 0

                while (terminated == False and truncated == False):
                    #Finchè non si sarà raggiunto uno stato terminale o non
                    #sarà finito il numero massimo di iterazioni possibili
                    #si considerano le seguenti istruzioni
                    env.action_space = a
                    action = env.action_space
                    #La funzione env.step() è utilizzata per far avanzare
                    #l'ambiente simulato in base all'azione scelta
                    observation, reward, terminated, truncated, info = env.step(action)
                    #Immagine RGB catturata all'iterazione t
                    image = observation
                    #Calcoliamo la matrice in scala di grigi associata alla
                    #immagine appena ottenuta
                    carRacingClass2.convert_RGB_GrayScale(self, image)
                    #Calcolo della distanza tra l'auto ed i bordi laterali
                    #per la definizione dello stato successivo
                    distanceVector = carRacingClass2.distanceCalculation(self)
                    nextState = carRacingClass2.convertCoordinate(self, distanceVector)

                    [next_a, next_ind_a] = carRacingClass2.epsGreedy(self, nextState)

                    #Aggiornamento del parametro utile succcessivamente per
                    #aggiornare la stima della funzione qualità
                    self.delta = reward + (self.gamma * self.Q[nextState, next_ind_a]) - self.Q[state, ind_a]

                    #Aggiornamento del vettore dei rewards in cui ogni riga
                    #indica un episodio ed il valore in essa indica la
                    #ricompensa totale ottenuto nello specifico episodio
                    self.G[e, 0] = self.G[e, 0] + reward

                    #Aggiornamento della "dutch trace" associata alla sola
                    #coppia stato - azione interessata
                    self.E[state, ind_a] = ((1 - self.alpha) * self.E[state, ind_a]) + 1

                    #Aggiornamento della matrice di stima della funzione
                    #qualità
                    self.Q = self.Q + (self.alpha * self.delta * self.E)

                    #Aggiornamento della "dutch trace" totale
                    self.E = self.gamma * self.Lambda * self.E

                    #Le nuove variabili di stato e azione vengono salvate
                    #come quelle precedenti per eseguire tutti i passaggi
                    #nell'iterazione successiva
                    state = nextState
                    a = next_a
                    ind_a = next_ind_a

                    #Aggiornamento del contatore delle iterazioni
                    count = count + 1
                    #print("iterazione n°:", count)

                if (e % self.saveVariable == 0):        #Episodio nel quale si salvano i dati raccolti
                    print("Salvataggio dei dati fino all'episodio:", e)
                    #Salvataggio del vettore delle stime Q
                    file_path_Q = r'C:\Users\quadr\Desktop\Assignment FINALE 2\Q.npy'
                    np.save(file_path_Q, self.Q)
                    #Salvataggio del vettore dei reward G
                    file_path_G = r'C:\Users\quadr\Desktop\Assignment FINALE 2\G.npy'
                    np.save(file_path_G, self.G)
                    #Salvataggio del parametro Epsilon (probabilità di scelta dell'azione)
                    file_path_eps = r'C:\Users\quadr\Desktop\Assignment FINALE 2\eps.pkl'
                    with open(file_path_eps, 'wb') as file:
                        pc.dump(self.epsilon, file)
                    #Salvataggio del parametro e (indice dell'episodio corrente)
                    file_path_e = r'C:\Users\quadr\Desktop\Assignment FINALE 2\e.pkl'
                    with open(file_path_e, 'wb') as file:
                        pc.dump(e, file)

        finish = tm.time()
        totalTime = finish - start
        print("Tempo per l'episodio ", e, ":", totalTime, "\n")
```

ANALISI DEI RISULTATI

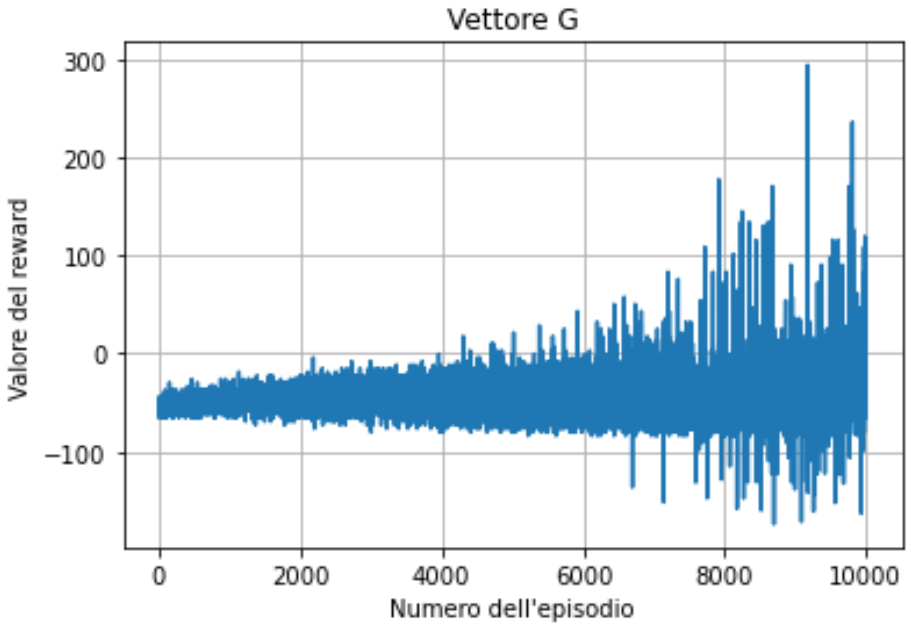
DI SEGUITO SONO RIPORTATI ALCUNI VIDEO RIGUARDANTI LE VARIE FASI DI STUDIO DEL PROBLEMA

1° *caso di studio* – TILES CODING

In questo primo caso di studio si utilizza l'algoritmo *Tiles Coding* per ridurre il numero di stati e generare una rappresentazione più semplice dell'ambiente. Inoltre, si considerano i bordi della strada come punti fondamentali per l'algoritmo.

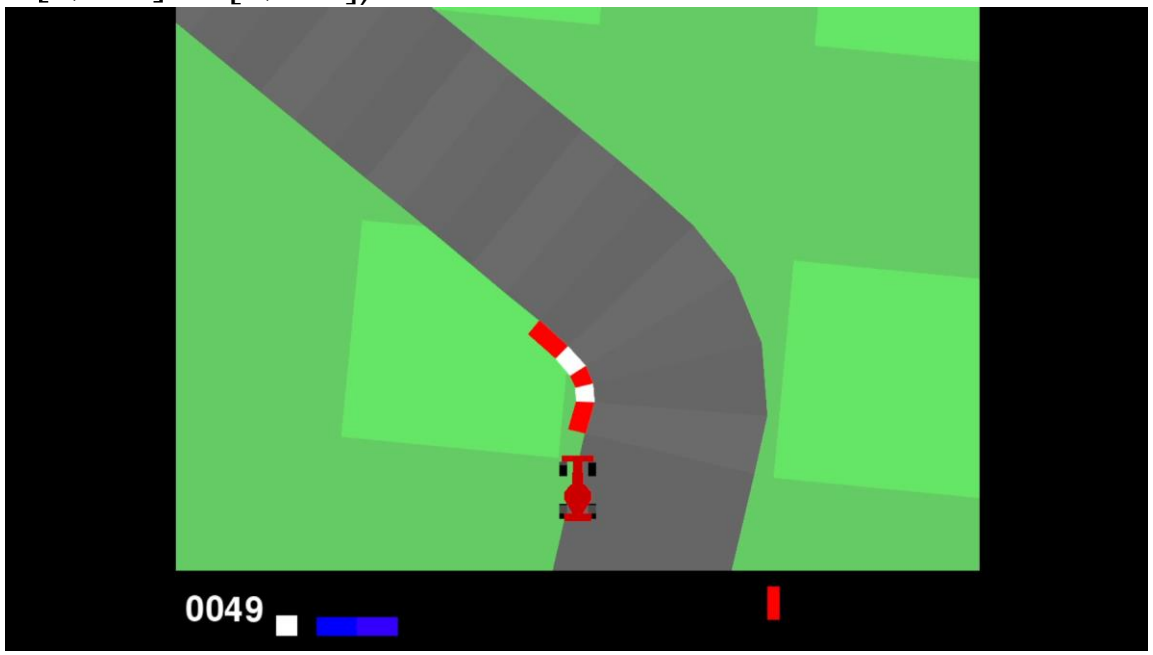
Osservando il comportamento dell'auto nel video e l'andamento delle ricompense si può notare come questo non sia l'ideale.

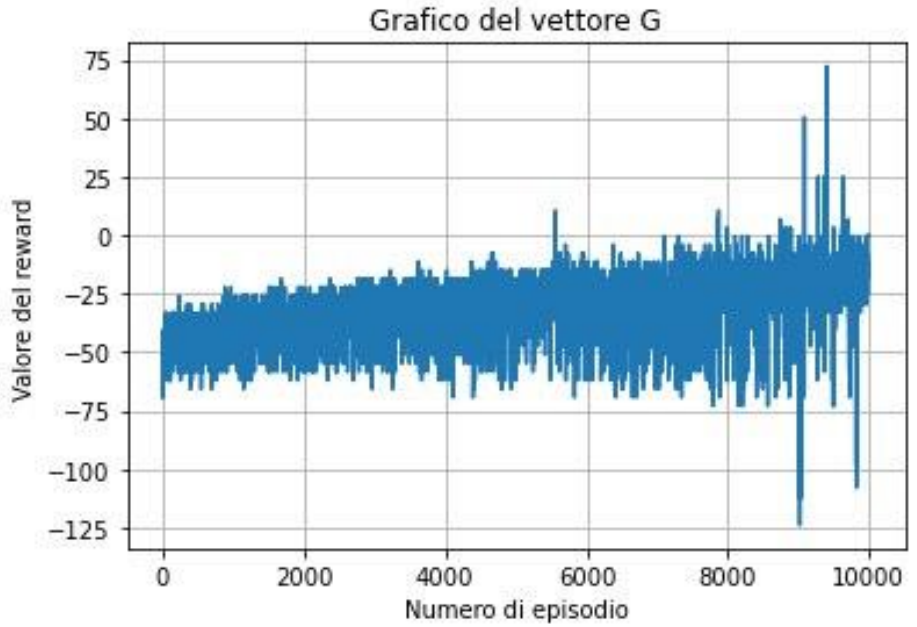
Perciò tale metodo è stato scartato e ci si è concentrati su un'altra tipologia di algoritmo.



$2° \, caso \, di \, studio \; - \mathrm{SARSA}(\lambda)$

Il nuovo algoritmo utilizzato è il $Sarsa(\lambda)$ dove ogni stato s è definito dalla distanza del bordo sinistro, destro ed anteriore dell'auto rispettivamente a quelli della strada. L'utilizzo del $Tiles \; Coding$ è stato sostituito dalla tecnica di conversione dell'immagine RGB in scala di grigi, infatti questa tecnica permette di ridurre le variabili dello spazio di stato ($[0,255] \times [0,255] \times [0,255] \rightarrow [0,255]$).



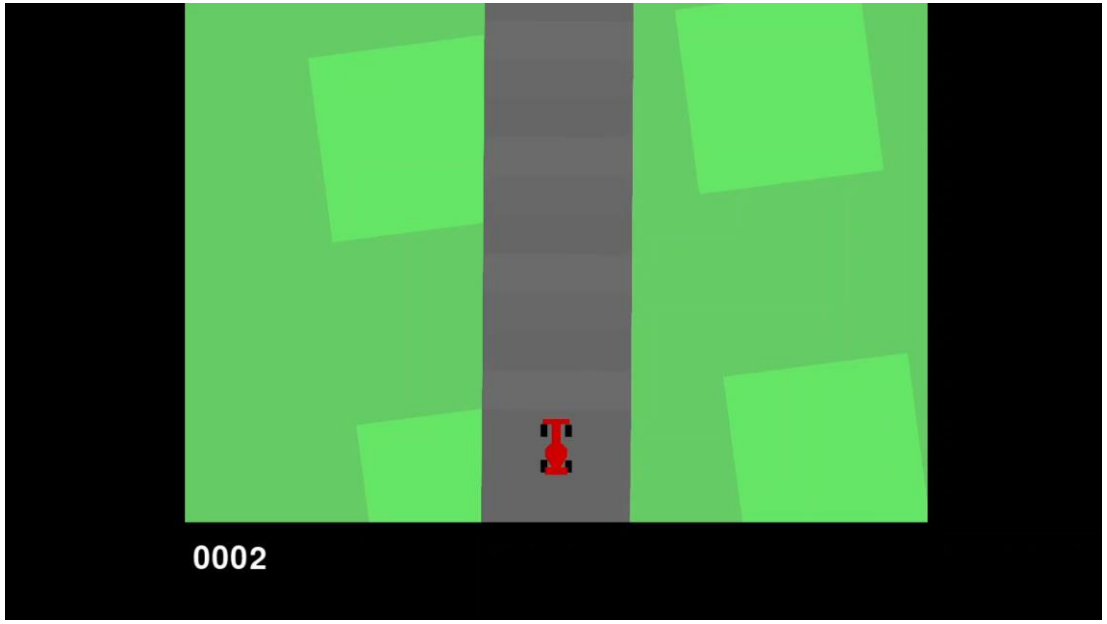


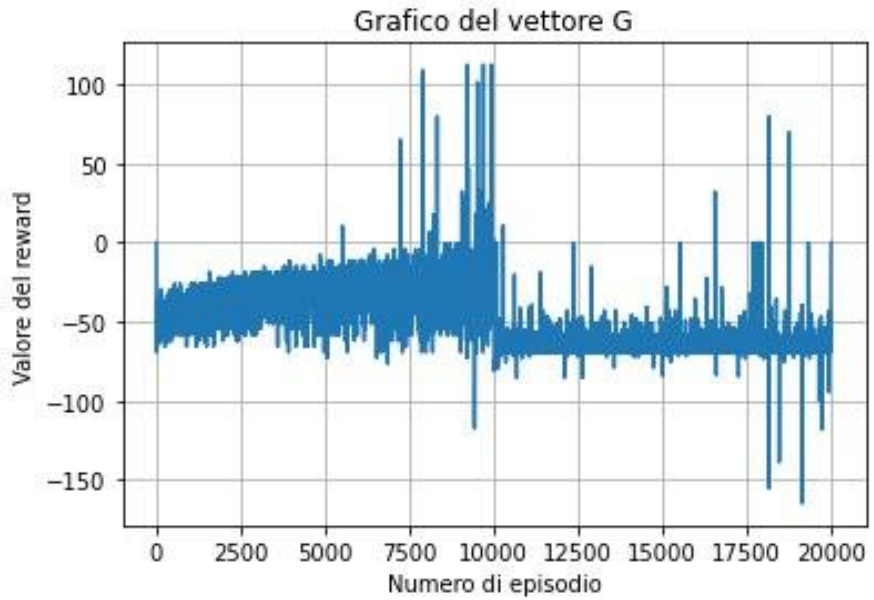
$3° \ caso \ di \ studio \ -$ SARSA(λ) $con \ doppio \ addestramento$

Partendo dalla base del caso precedente, si è implementato un doppio addestramento per far sì che l'agente imparasse la policy ottima in più punti del tracciato, questo perché essendo l'ambiente molto grande l'esplorazione data da $\varepsilon - greedy$ non basta.

Quindi vengono inizializzate ed utilizzate due variabili di probabilità $\varepsilon_1, \varepsilon_2$ in successione, cioè si eseguono le prime 10000 iterazioni decrementando $\varepsilon_1$ed le ultime 10000 iterazioni decrementando ε_2 dopo essersi spostati usando ε_1.

Nonostante l'idea sembrasse teoricamente valida, le simulazioni hanno contraddetto le aspettative.





GRAZIE

LUCA SUGAMOSTO (0324613)

MATTIA QUADRINI (0334381)

luca.sugamosto@students.uniroma2.eu

mattia.quadrini.1509@students.uniroma2.eu