SnakeRL

Questo progetto usa come pretesto il gioco Snake per imparare e approfondire algoritmi, tecniche di reinforcement learning.

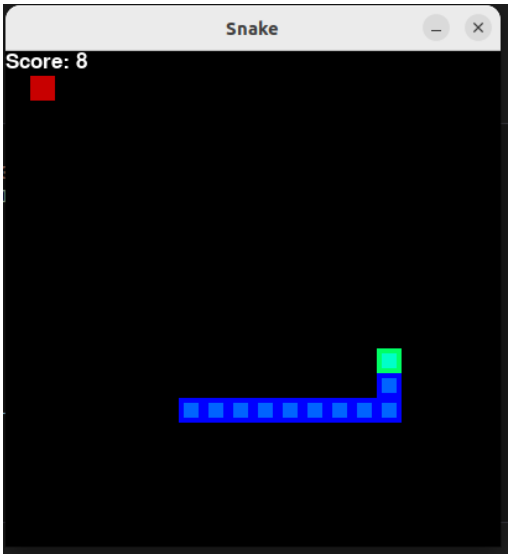
Il codice utilizzato per simulare il gioco di Snake è presente nel file **enviroment.py** e permette di eseguire singoli step e resettare la partita garantendo un training agevole con o senza visuale del gioco. Ogni volta che Snake esegue uno step è restituita un'esperienza composta da quattro elementi: il nuovo stato, la ricompensa, un valore booleano che indica se la partita è terminata, e lo score.

Lo stato è la rappresentazione dell'ambiente corrente in cui si trova Snake; la sua rappresentazione sarà soggetta a cambiamenti nell'arco del progetto. Di seguito sarà mostrata la prima rappresentazione di stato, la più semplice.

Prima rappresentazione di stato

Lo **stato** è espresso come un array di 11 valori booleani:

- **Pericolo**: I primi 3 valori indicano la presenza di un pericolo direttamente davanti, a destra o a sinistra. Il pericolo è definito dalla presenza di un ostacolo come il bordo o la coda di Snake a distanza di 1 blocco dalla testa.
- **Direzione corrente**: 4 valori indicano la direzione corrente di Snake: destra, sinistra, su e giù.
- **Posizione del frutto**: 4 valori rappresentano la posizione relativa del frutto rispetto alla testa di Snake: se si trova a destra, sinistra, sopra o sotto.



Nell'esempio illustrato a sinistra lo stato è rappresentato dall'array `[0,0,0, 0,0,1,0, 0,1,1,0]`.

Ricompense e azioni

Le **ricompense** sono assegnate in base agli eventi del gioco:

- **+10** se Snake mangia un frutto.
- **- 10** in caso di game over (quando Snake collide con un ostacolo).
- **0** in tutti gli altri casi.

Le **azioni** possibili sono rappresentate da un array di 3 elementi:

- `[1, 0, 0]`: Snake continua dritto.

- `[0, 0, 1]`: Snake gira a sinistra.
- `[0, 1, 0]`: Snake gira a destra.

Reinforcement Learning

In questo progetto il gioco Snake è affrontato tramite diverse tecniche di Reinforcement Learning. L'obiettivo di algoritmi RL è quello di apprendere una politica ottimale che consenta a un agente, in questo caso il serpente, di prendere decisioni ottimali in ogni stato dell'ambiente per massimizzare una ricompensa cumulativa.

Q-learning

Tra le diverse tecniche di Reinforcement Learning, una delle più comuni è il Q-learning. Questo algoritmo permette all'agente di apprendere, attraverso l'esplorazione e l'interazione con l'ambiente, la migliore sequenza di azioni per ottenere il massimo della ricompensa nel tempo .

Alla base di questo processo c'è la **funzione di valore Q** *Q(s,a),* che stima la qualità di un'azione *a* eseguita nello stato *s*. In sostanza, la funzione predice quanto sia vantaggioso per l'agente scegliere una determinata azione in un dato stato, tenendo conto non solo delle ricompense immediate, ma anche di quelle che potrebbero essere accumulate in futuro.

L'aggiornamento di *Q(s,a)* avviene utilizzando una versione iterativa dell'equazione di Bellman, nota come **Q-Value Iteration**, che descrive la relazione ricorsiva tra il valore atteso corrente e il massimo valore atteso futuro. L'equazione usata per aggiornare i valori *Q(s,a)* è la seguente:

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a)\right)$$

Dove:

- α è il tasso di apprendimento.
- γ è il discount factor, che determina l'importanza delle ricompense future rispetto a quelle immediate.
- r_t è la ricompensa che l'agente ottiene compiendo l'azione a nello stato s arrivando a s_{t+1}
- $max_a Q(s_{t+1}, a)$ è il Q-Value maggiore nel nuovo stato s_{t+1}, e stima il miglior guadagno futuro possibile.

Effettuato il training dei Q-Value, la policy ottimale corrisponde a scegliere l'azione con maggiore Q-Value.

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

Il file **TabQLearning.ipynb** implementa l'algoritmo Q-Value Iteration utilizzando una tabella Q (Q-Table), che viene creata per memorizzare i valori *Q(s,a)* per ogni possibile combinazione di stato *s* e azione *a*. In questo caso, ogni stato è associato a tre possibili azioni: proseguire dritto, girare a destra, oppure girare a sinistra.

Struttura della Q-Table

La Q-Table è una matrice di dimensioni 2048×3, dove:

- **2048** deriva da 2^{11}, con 11 valori booleani che descrivono lo stato.
- **3** rappresenta le tre possibili azioni.

All'inizio, tutti i valori della tabella sono inizializzati a zero, dato che l'agente non ha ancora informazioni sull'ambiente. Le dimensioni della tabella giustificano la scelta di stato di soli 11 elementi: a stati complessi corrispondono Q-Table enormi, il che avrebbe reso il training computazionalmente costoso o addirittura impossibile. Uno stato composto da 11 fornisce un compromesso tra la capacità di rappresentare l'ambiente e la semplicità computazionale. Inoltre anche se la Q-Table ha 2048 possibili combinazioni, non tutti questi stati sono raggiungibili. Per esempio, alcune combinazioni sono logicamente impossibili, come avere due direzioni correnti. In realtà, solo 256 stati sono possibili, risultanti da:

- **8 possibili pericoli** intorno al serpente (come muri o il corpo del serpente stesso),
- **4 direzioni** in cui il serpente può muoversi (su, giù, destra, sinistra),
- **8 possibili posizioni relative del frutto** rispetto alla testa del serpente.

L'implementazione della Q-Table segue quindi il codice:

```
def state_to_index(state_vector):
    # 3 bit con 8 posisibili pericoli
    pericoli = state_vector[0] * 4 + state_vector[1] * 2 + state_vector[2]
    # 4 bit ma solo 4 direzioni corrette
    direzione_corrente = state_vector[3] * 3 + state_vector[4] * 2 + state_vector[5] * 1
    # 4 bit ma solo 8 possibili posizioni frutto
    posizione_frutto = np.mod(state_vector[7] * 6 + state_vector[8] * 3 + state_vector[9] * 2 + state_vector[10], 8)
    return pericoli * 32 + direzione_corrente * 8 + posizione_frutto

def initialize_QTable():
    return np.zeros((256, 3))
```

Viene quindi inizializzata la Q-Table con dimensioni ottimizzate a solo 256 righe corrispondenti agli stati possibili. Per accedere alla tabella si inserisce a `state_to_index(state_vector)` un input corrispondente un vettore di stato di 11 elementi e restituisce l'indice della riga della Q-Table unico per stato possibile.

Exploration policies

Per quanto riguarda l'esplorazione dell'ambiente durante il training, sono state implementate tre possibili politiche di esplorazione che l'agente può seguire. La prima, random, in cui l'agente sceglie azioni in modo completamente casuale. La seconda è una politica epsilon greedy, dove l'agente esegue un'azione casuale con probabilità ε, o l'azione migliore con probabilità 1-ε. Infine, la terza politica di esplorazione è definita tramite la seguente equazione:

$$$$

con:

- $N(s', a')$ conta il numero di volte che un azione *a'* è stata eseguita nello stato *s'*.
- $f(Q, N)$ è la funzione di esplorazione, tale che $f(Q, N) = Q + k/(1 + N)$, dove k è un parametro di curiosità che misura quanto l'agente è attratto nel compiere un'azione intrapresa poco nello stato *s'*.

```
def random_policy():
    return np.random.randint(3)

def epsilon_greedy_policy(Q_table, state, epsilon):
    if random.uniform(0,1) < epsilon:
        return np.random.randint(3)
    else:
        row_Q_value = state_to_index(state)
        return np.argmax(Q_table[row_Q_value])

def exploration_function_policy(Q_table, N_table, state, k):
    row_Q_value = state_to_index(state)
    exploration_values = Q_table[row_Q_value] + k / (1 + N_table[row_Q_value])
    return np.argmax(exploration_values)
def update_N_table(N_table, state, action):
    row_N_value = state_to_index(state)
    N_table[row_N_value][action] += 1
```

Nota: N, la funzione che ricorda il numero di volte in cui un'azione è eseguita in un determinato stato, è inizializzata come la Q-Table e aggiornata secondo la funzione `update_N_table`. L'aggiornamento consiste nel aumentare di +1 il valore nella corrispettiva colonna dell'azione intrapresa nello riga corrispondente allo stato in cui è stata intrapresa. In questo modo è facile tenere conto del numero di volte che un azione *a'* è stata eseguita nello stato *s'*.

Training

A questo punto non resta altro che eseguire il training. La funzione adibita al training riceve in input 4 valori: il nome della policy di esplorazione da eseguire, il tasso di apprendimento, il fattore di sconto e il numero totale di passi di addestramento che l'agente eseguirà.

```
def trainAgent(POLICY_NAME, lr, gamma, N_STEPS):
    env = enviroment.SnakeGameAI()
    Q_table = initialize_QTable()
    if POLICY_NAME == "ExplorationFunction":
        N_table = initialize_QTable()
    epsilon = 1

    for iteration in range(N_STEPS+1):
        state = env.get_state()
        row_state = state_to_index(state)
        # exploration policies
        if POLICY_NAME == "RandomPolicy":
            action = random_policy()
        elif POLICY_NAME == "ExplorationFunction":
            action = exploration_function_policy(Q_table, N_table, state, k=100)
            update_N_table(N_table, state, action)
        else:
            action = epsilon_greedy_policy(Q_table, state, epsilon)
            epsilon = max((N_STEPS - iteration)/(N_STEPS), 0.1)

        next_state, reward, game_over = step(env, action)
        row_next_state = state_to_index(next_state)
        next_value = Q_table[row_next_state].max()
        # Equazione di Bellman
        Q_table[row_state, action] *= 1 - lr
        Q_table[row_state, action] += lr * (reward + gamma * next_value)
        if game_over:
            env.reset()
        if (iteration % 10_000 == 0) and (iteration != 0):
            np.save(f'Q_table/{POLICY_NAME}/{iteration}step.npy', Q_table)
            print(f"\rNumero step eseguiti: {iteration}\\{N_STEPS}", end="")
```

La funzione permette di scegliere una delle 3 policy d'esplorazione; in particolare per l'epsilon greedy policy ε ha una discesa lineare fino a un limite di 0.1. Per quanto riguarda la funzione di esplorazione si è deciso di partire con il parametro di curiosità **k**=100. Il valore scelto per **k** sarà motivato in una sezione dedicata.

Il cuore della funzione, in cui i i Q-value sono aggiornati secondo l'equazione di Bellman, è racchiuso nelle seguenti righe:

```
        Q_table[row_state, action] *= 1 - lr
        Q_table[row_state, action] += lr * (reward + gamma * next_value)
```

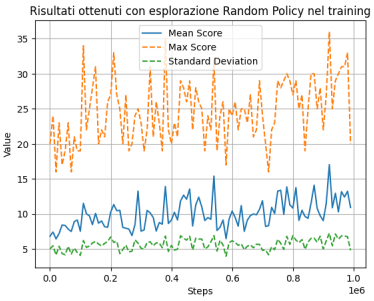
Il Q-Learning è un algoritmo *off-policy*, ovvero la policy eseguita durante il training non corrisponde alla policy che impara. In termini semplici, riesce a imparare una buona strategia anche da una random policy eseguita nel training. Di conseguenza è difficile stabilire durante il training quanto l'agente ha finora appreso. Per avere un'idea chiara del variare delle performance del modello per step di training eseguiti, la Q-Table è salvata ogni 10.000 step in una directory dedicata. Questo approccio consente di testare in seguito ciascuna Q-table salvata su più partite e analizzare le performance delle tabelle per numero di step di training.

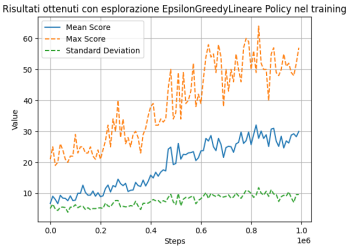
Si è quindi proceduto ad addestrare una Q-Table per ciascuna delle 3 strategie di esplorazione per 1.000.000 di step con learning rate pari a 0.05 e un fattore di sconto pari a 0.9.

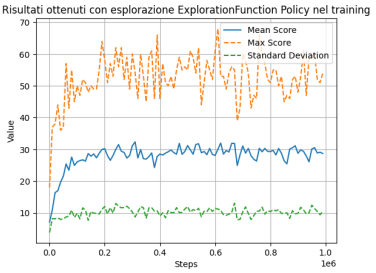
```
print("Training con RandomPolicy")
trainAgent("RandomPolicy", lr = 0.05, gamma = 0.9, N_STEPS = 1_000_000)
print("\nTraining con policy EpsilonGreedyLineare")
trainAgent("EpsilonGreedyLineare", lr = 0.05, gamma = 0.9, N_STEPS = 1_000_000)
print("\nTraining con policy ExplorationFunction")
trainAgent("ExplorationFunction", lr = 0.05, gamma = 0.9, N_STEPS = 1_000_000)
```

Quindi per valutare l'andamento nel training si è proceduto a far giocare ciascuna Q-Table salvata per 50 partite, salvando i risultati.

Di seguito sono presentati i risultati delle performance ottenute da ciascuna Q-table salvata divise per politica di esplorazione e in ordine di step di training eseguiti:



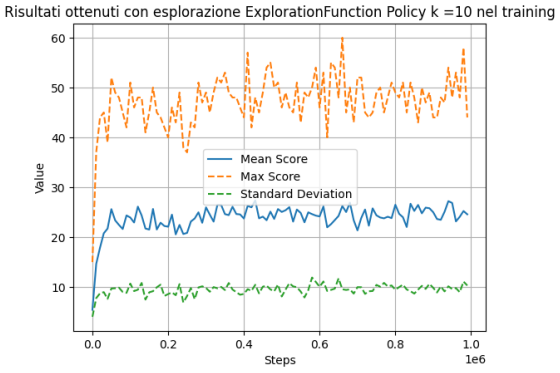


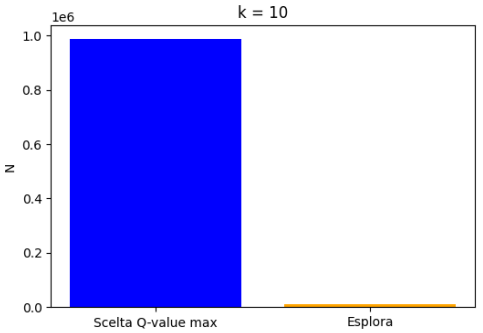


Si nota una notevole differenza degli andamenti per politica di esplorazione. Per quanto riguarda il training con esplorazione random si nota un aumento delle performance estremamente lento e instabile rispetto agli altri grafici; dopo un milione di step la Q-Table non riesce ad ottenere una media dello score maggiore di 15. Per quanto riguarda la politica epsilon greedy si osserva un aumento lineare per lo score medio raggiungendo una media di 28 punti dopo circa 800.000 step. I risultati più soddisfacenti sono ottenuti tramite funzione di esplorazione: dopo solo 100.000 passi l'andamento ha raggiunto una media di circa 30 punti risultando estremamente veloce rispetto all'epsilon greedy policy nell'ottenere una performance buona.

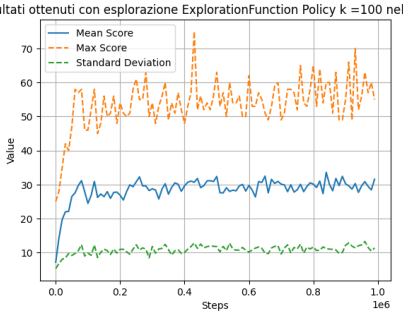
Analisi sul parametro di curiosità

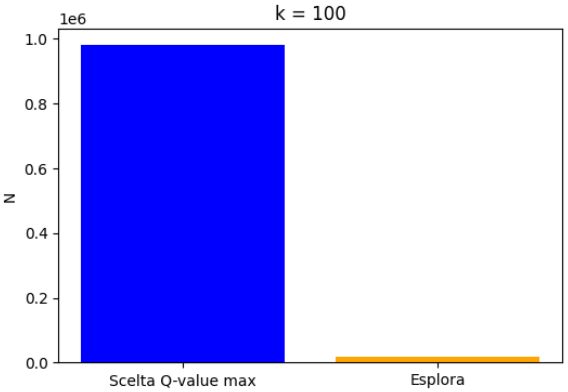
Per quanto riguarda il tasso di apprendimento e il fattore di sconto, i valori scelti sin dall'inizio hanno fornito buone performance. Non si è quindi proceduto a effettuare una scelta di iperparametri, ad esempio tramite grid-search, in quanto il risultato ottenuto è quello sperato. Tuttavia, il parametro di curiosità **k** ha richiesto un'analisi più approfondita, poiché le prestazioni del modello variano in base al suo valore. Sono stati quindi effettuati diversi test per analizzare le performance al suo variare. Accanto agli andamenti sono mostrati gli istogrammi indicanti il numero di volte in cui `exploration_function_policy` ha scelto l'azione ottimale, Q-Value maggiore, o eseguito un azione "d'esplorazione".



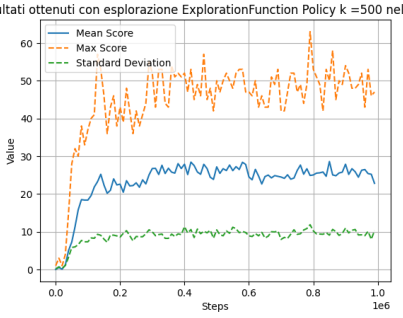


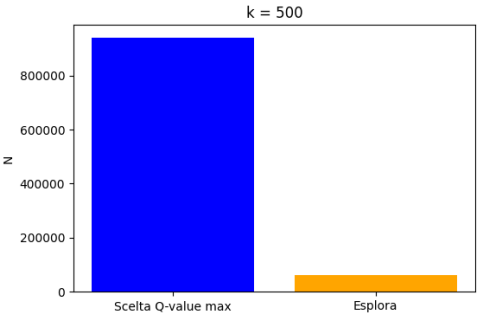
Risultati ottenuti con esplorazione ExplorationFunction Policy k =100 nel training



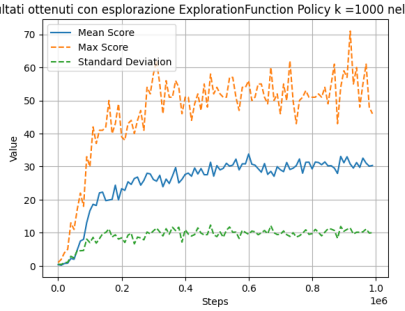


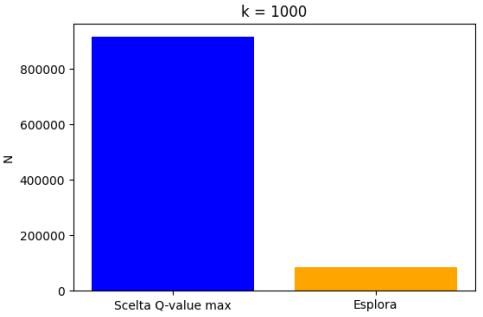
Risultati ottenuti con esplorazione ExplorationFunction Policy k =500 nel training



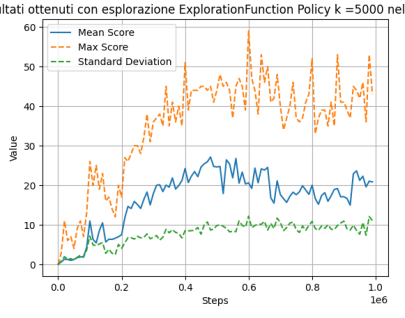


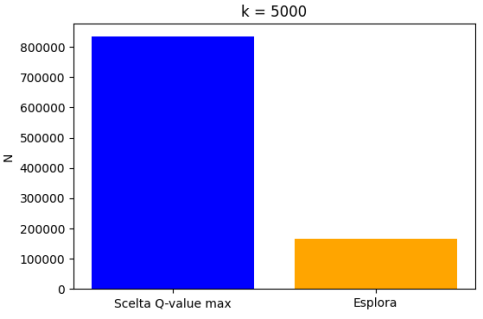
Risultati ottenuti con esplorazione ExplorationFunction Policy k =1000 nel training

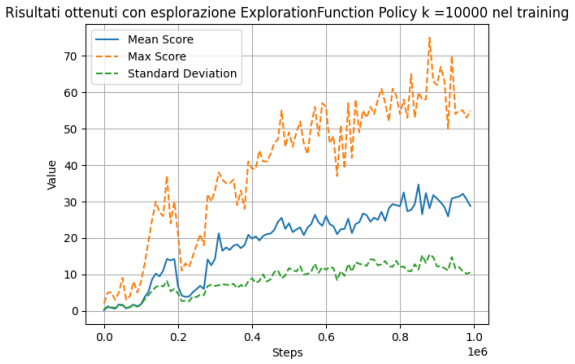


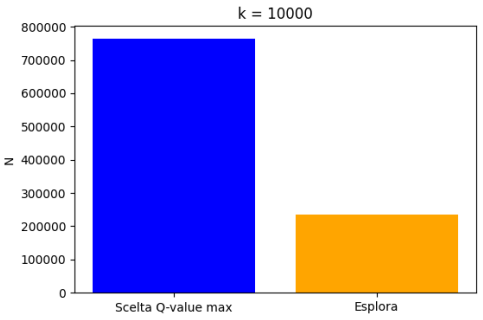


Risultati ottenuti con esplorazione ExplorationFunction Policy k =5000 nel training





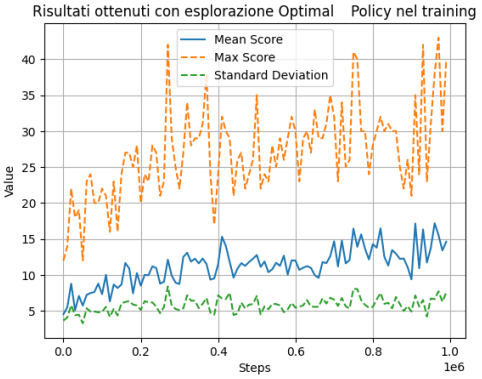




L'analisi ha mostrato che un valore elevato di **k**, quindi a incitare maggiormente l'esplorazione, non solo rallenta il processo di apprendimento, ma porta anche a prestazioni peggiori. A valori k inferiori a 100 invece si stabilizza a un punteggio medio di 25. Un valore di k pari a 100 porta invece una media di 30 ma con un numero di esplorazioni molto basso.

Di conseguenza, ci si è interrogati sulla validità di compiere un'azione intrapresa poco. È stato quindi eseguito un test optando per una strategia basata esclusivamente sul Q-value attuale più alto.

I risultati ottenuti, riportati nella figura a destra, dimostrano che questo approccio è inefficace: l'assenza di esplorazione riduce significativamente le performance. Al contrario, un valore di **k** pari a 100, anche se comporta un numero limitato di azioni esplorative, si è rivelato essere sufficiente e ha avuto un impatto positivo sulle prestazioni complessive del modello. Pertanto, si è deciso di mantenere il parametro di curiosità a 100.

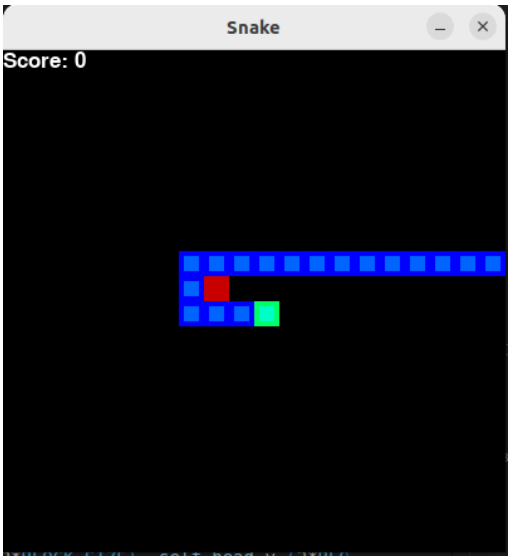


Considerazioni

La valutazione precedente, che stabilisce una media di 30 punti come un buon risultato, non è soggettiva. Infatti, le aspettative per questo modello sono limitate a tale soglia. Questa restrizione è dovuta al fatto che lo stato in input è rappresentato da soli 11 elementi: se da una parte facilita una rappresentazione semplificata dell'ambiente corrente, dall'altra parte non permette all'agente di acquisire una conoscenza profonda dell'ambiente che gli permetta di sfuggire ai pericoli rappresentati dalla sua coda. Dopo una certa lunghezza è solo questione di tempo, tipicamente per posizionamento sfavorevole del nuovo frutto, che l'agente si circondi con la sua coda precludendosi qualsiasi via di fuga.

Dentro al notebook **test.ipynb** c'è una sezione dedicata a un test in cui la scelta di dirigersi al frutto porta a game over certo, come mostra la figura a destra. La scelta giusta è quella di proseguire a destra ma qualsiasi modello allenato prosegue verso il frutto non superando il test, a dimostrazione che la rappresentazione di stato usata per il momento è troppo miope.

Una possibile soluzione al problema potrebbe è quella di utilizzare uno stato che rappresenta l'intera matrice del gioco. Tuttavia questo approccio è impraticabile con l'algoritmo appena adottato, poiché comporterebbe un aumento enorme del numero di stati possibili, rendendo il processo di apprendimento inefficace e computazionalmente proibitivo.



Deep Q-Learning

Il numero di possibili stati considerando la matrice del gioco diventa infatti troppo complesso da gestire. Bisogna infatti considerare una mappa 20×20 in cui ciascun blocco può essere vuoto o con un ostacolo: sono necessari 2^{400} possibili stati, senza considerare la posizione del frutto. Se anche avessimo una memoria che riuscisse a gestire così tanti stati (impossibile), l'allenamento richiederebbe anni.

La soluzione consiste nel calcolare una funzione $Q_\theta(s,a)$ che approssima i Q-Value di ciascun stato-azione utilizzando un numero di parametri gestibile. Questa tecnica, nota come Approximate Q-Learning, viene implementata attraverso l'uso di una rete neurale. Il principio alla base è che, invece di memorizzare esattamente i Q-values per tutte le combinazioni di stati e azioni (cosa impraticabile in ambienti di grandi dimensioni), la rete approssima tali valori. In questo modo, può generalizzare a nuove combinazioni di stato-azione mai incontrate, ma simili a quelle già sperimentate.

La rete neurale è sempre allenata tramite equazione di Bellman: vogliamo che l'approssimazione del Q-Value calcolato per uno stato-azione (s,a) sia più vicino possibile alla ricompensa *r* ottenuta compiendo l'azione *a* nello stato *s*, sommata al valore scontato delle possibili ricompense nel giocare in maniera ottimale. Per calcolare quest ultimo valore è necessario predire il Q-Value nello stato successivo *s'* per tute le sue azioni *a', s*elezionare il Q-value maggiore e moltiplicarlo per il fattore di sconto. La formula:

$$Q_{target}(s,a) = r + \gamma * max_{a'} Q_\theta(s',a')$$

Con questa formula è possibile eseguire la discesa del gradiente, minimizzando l'errore quadrato tra il Q-Value predetto e il Q-Value target.

Implementazione DQN

Di seguito sono descritte le tre classi cuore dell'algoritmo DQN.

La classe QNetwork, incaricata di gestire i parametri e le funzioni della rete neurale e il suo aggiornamento pesi secondo l'espressione precedentemente descritta.

```
class QNetwork:
    def __init__(self, lr, gamma):
        self.model = Linear_QNet()
        self.gamma = gamma
        self.optimizer = keras.optimizers.Adam(learning_rate=lr)
        self.loss_fn = keras.losses.mean_squared_error
```

```
    @tf.function
    def train_step(self, states, actions, rewards, next_states, dones):
        next_Q_values = self.model(next_states)
        max_next_Q_values = tf.reduce_max(next_Q_values, axis=1)
        # Equazione di Bellman: Q value = reward + discount factor * expected future reward
        target_Q_values = rewards + (1 - dones) * self.gamma * max_next_Q_values
        with tf.GradientTape() as tape:
        all_Q_values = self.model(states)
        Q_values = tf.reduce_sum(all_Q_values * actions, axis=1, keepdims=True)
        loss = tf.reduce_mean(self.loss_fn(target_Q_values, Q_values))
        # Backpropagation
        grads = tape.gradient(loss, self.model.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))
```

La classe ReplayBuffer gestisce un buffer circolare per memorizzare le esperienze passate durante il training di un agente di reinforcement learning. Il training non è più effettuata ad ogni step, come nel Tabular QLearning, ma su batch di esperienze memorizzate, appunto da ReplayBuffer. Questo approccio tramite batch, rispetto a un aggiornamento dei pesi effettuato ad ogni step, migliora la capacità del modello di generalizzare, evitando che si basi solo sull'ultima esperienza. Il buffer ha una dimensione massima definita da max_size e utilizza un array per memorizzare ogni esperienza, che consiste in tuple di (stato, azione, ricompensa, stato successivo, fine partita). Il sampling dell'esperienze è casuale. Il vantaggio nell'uso di un buffer circolare sta nell'aggiungere e eliminare elementi molto velocemente e riduce il tempo di accesso rispetto a una lista a $O(1)$ rispetto a $O(N)$, in quanto il buffer è implementato come un array di dimensione fissa, in cui ogni elemento può essere indicizzato direttamente.

```
class ReplayBuffer:
    def __init__(self, max_size):
        self.buffer = np.empty(max_size, dtype=object)
        self.max_size = max_size
        self.index = 0
        self.size = 0
    def append(self, obj):
        self.buffer[self.index] = obj
        self.size = min(self.size + 1, self.max_size)
        self.index = (self.index + 1) % self.max_size
    def sample(self, batch_size):
        indices = np.random.randint(self.size, size=batch_size)
        return self.buffer[indices]
    def sample_experiences(self, batch_size):
        batch = self.sample(batch_size)
        states, actions, rewards, next_states, game_over = map(np.array, zip(*batch))
        return states, actions, rewards, next_states, game_over
```

La classe Agent ha il compito di gestire il processo di l'allenamento. Possiede un'istanza di QTrainer, ReplayBuffer e del simulatore di gioco e ne coordina l'utilizzo per trovare la politica ottimale di gioco. Oltre alla epsilon greedy policy nel training è aggiunta la *softmax exploration policy*. Questa assegna probabilità diverse a ciascuna azione in base ai loro valori Q stimati, permettendo così una scelta probabilistica delle azioni. In questo modo, azioni con valori Q più elevati hanno una maggiore probabilità di essere selezionate, ma anche azioni con valori Q inferiori hanno la possibilità di essere esplorate. Questo approccio riduce il rischio di rimanere bloccati in un comportamento subottimale dovuto a scelte esplorative casuali. In particolare nel problema di Snake le probabilità che la scelta della prossima azione comporti la morte è ridotta rispetta a una random, in quanto ci sia aspetta che abbia probabilità minore rispetto alle altre, permettendo quindi un esplorazione più duratura anche nelle prime partite. La funzione è espressa tramite la formula:

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}$$

dove:

- $Q(s, a)$ è il valore Q stimato per l'azione a nello stato s.
- τ è il parametro di temperatura che controlla il grado di esplorazione. Un valore alto di τ porta a una maggiore esplorazione, mentre un valore basso di τ favorisce lo sfruttamento delle azioni migliori.

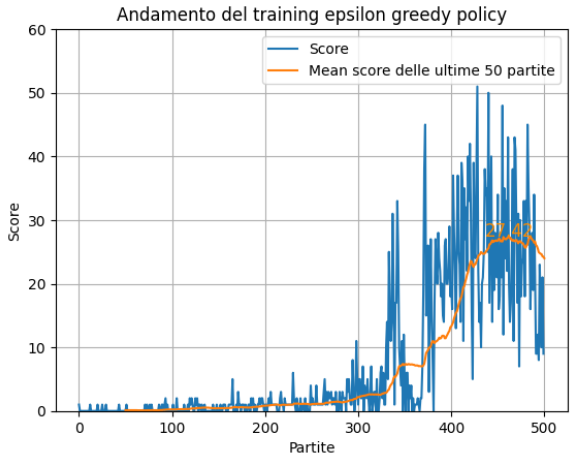
```
class Agent:
    def __init__(self, lr, gamma, max_memory, batch_size):
        self.n_games = 0
        self.epsilon = 1
        self.batch_size = batch_size
        self.memory = ReplayBuffer(max_size=max_memory)
        self.qnetwork = QNetwork(lr=lr, gamma=gamma)
        self.env = enviroment_visual.SnakeGameAI()

    def remember(self, state, action, reward, next_state, done):..
    def train_memory(self):...
    def epsilon_greedy_policy(self, state):...
    def softmax_policy(self, state):...
    def get_action(self, state):...

    def train_agent(self, N_GAME):
        score_list = []
        record = 0
        step=0
        n_eps_zero = int(N_GAME*0.7)
        while self.n_games < N_GAME:
            state_old = self.env.get_state()
            final_move = self.get_action(state_old)
            state_new, reward, done, score = self.env.play_step(final_move)
            self.remember(state_old, final_move, reward, state_new, done)
            if done:
                self.env.reset()
                self.n_games += 1
                self.train_memory()
                print(f"\rGame: {self.n_games}, Epsilon: {self.epsilon:3f}, Score: {score}, Record: {record}, Step eseguiti: {step}. ", end="")
                self.epsilon = max(((n_eps_zero - self.n_games) / n_eps_zero), 0)
                if score > record:
                    record = score
                    self.qnetwork.save_model()
                    score_list.append(score)
      step+=1
    return score_list
```

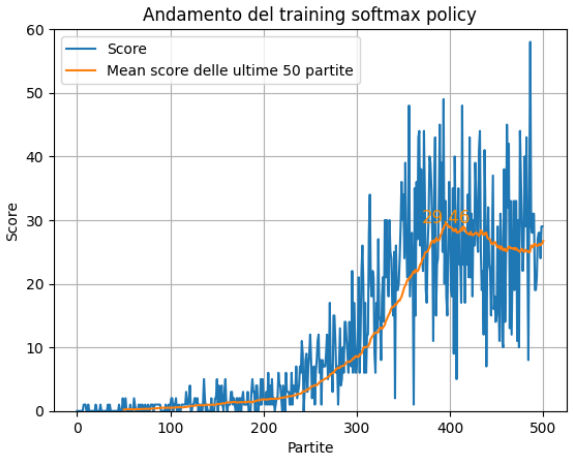
In questo modo è possibile creare, addestrare e mostrare i risultati dell'agente in poche righe di codice:

```
agent = Agent(lr=0.001, gamma=0.9, max_memory=50_000, batch_size=1024)
score_list = agent.train_agent(N_GAME=500)
```

Per eseguire il training è stato introdotto un accorgimento necessario per risolvere un comportamento indesiderato dell'agente Snake: per evitare di colpire un ostacolo e subire penalità, il serpente imparava a girare su se stesso, evitando la morte ma senza mai raggiungere il frutto, prolungando indefinitamente la partita e, di conseguenza, il training. Per correggere questo comportamento il gioco si interrompe ogni 50* (lunghezza di Snake) step.

Per scegliere i valori di learning rate, discount factor e il numero di unità del modello è stata operata una piccola grid search che ha riportato i risultati migliori con lr pari a 0.001 e 250 unità. Al variare di gamma non sono stati notati differenze nel training se non una maggiore instabilità al tendere di 1, quindi è rimasto a 0.9. Un altro aspetto rivisto è stata la dimensione del replay buffer, inizialmente sovrastimata a 100.000 esperienze. Tuttavia, dopo 500 partite di training, si raccoglievano circa 100.000 esperienze, il che impediva il ricambio delle vecchie esperienze con le nuove. Questo portava l'agente a continuare a imparare da esperienze obsolete, meno rilevanti per lo stato attuale. Per migliorare l'efficienza del replay buffer, la sua dimensione è stata dimezzata a 50.000 esperienze. Di seguito sono riportati i risultati del training:





Entrambi i training hanno raggiunto l'obiettivo desiderato, con uno score di circa 30 punti. La principale differenza tra i due risiede nella velocità d'apprendimento di una buona politica e di stabilizzazione, dove l'agente che utilizza la funzione di esplorazione softmax si dimostra superiore. Questo accade perché, con epsilon-greedy, anche una singola mossa casuale può condurre immediatamente al game over. Al contrario, la softmax ha una probabilità maggiore di selezionare una mossa vantaggiosa, o meglio, una probabilità inferiore di scegliere la mossa che causerebbe il game over, facilitando così una migliore esplorazione del gioco.

Il lavoro successivo prevede l'implementazione di alcune varianti del Deep Q-Learning, proposte da OpenAI, con l'obiettivo di stabilizzare e velocizzare il processo di training.

Prioritezed Experience Replay

Nel DQN classico, le esperienze vengono campionate casualmente dal buffer di replay per addestrare la rete. L'idea alla base del Prioritized Experience Replay è invece quella di campionare le esperienze in base alla loro importanza. Un'esperienza è determinata importante se questa porta progressi veloci nella fase di learning. Un modo per stabilire una stima al progresso determinato da un'esperienza, è quello di misurare la magnitudine dell'errore $TD\ \delta = r + \gamma * V(s') - V(s)$. Un TD error alto indica una transizione (s, r, s') sorprendente e che quindi vale la pena esplorare. Ogni qual volta che un esperienza è memorizzata, a questa è assegnato un valore alto in modo che questa sia scelta almeno una volta nel training batch. Ogni volta che un'esperienza ha subito questo sampling, l'errore $TD\ \delta$ è calcolato, e questa esperienza viene assegnato un nuovo valore di priorità paria $p = |\delta| + \epsilon$, con ϵ un valore piccolo per garantire che ogni esperienza abbia priorità maggiore di zero. La probabilità P di selezionare un esperienza con priorità p è proporzionale a p^{ζ},con ζ un iperparametro che se uguale a 0 si ottiene un sampling uniforme tra esperienze, se uguale a 1 si da massima importanaza alla priorità. Nel progetto è utilizzato $\zeta = 0.6$ in quanto valore comunemente usato.

Dato che le esperienze importanti in questo modo saranno scelte maggiormente bisogna abbassare il loro peso durante il training per evitare overfitting. Per far ciò si definisce per ciascuna esperienza un peso $w = (nP)^{-\beta}$, con n il numero di esperienze nel replay buffer, β un iperparametro che controlla quanto si vuole compensare l'importanza del sampling bias. Una scelta comune è iniziare con 0.4 e aumentare linearmente a 1.

Di seguito l'implementazione del PER:

```
class PrioritizedReplayBuffer:
    def __init__(self, max_size, zeta=0.6):
        self.max_size = max_size
        self.buffer = []
        self.zeta = zeta
        self.priorities =  np.zeros((max_size,), dtype=np.float32)
        self.index = 0
```

```
    def append(self, experience):
        max_priority = self.priorities.max() if self.buffer else 1.0
        if len(self.buffer) < self.max_size:
            self.buffer.append(experience)
        else:
            self.buffer[self.index] = experience
        self.priorities[self.index] = max_priority
        self.index = (self.index + 1) % self.max_size

    def sample(self, batch_size, beta=0.4):
        if len(self.buffer) == self.max_size:
            priorities = self.priorities
        else:
            priorities = self.priorities[:self.index]

        probabilities = priorities ** self.zeta
        probabilities /= probabilities.sum()
        indices = np.random.choice(len(self.buffer), batch_size, p=probabilities)
        samples = [self.buffer[idx] for idx in indices]

        total = len(self.buffer)
        sampling_probabilities = probabilities[indices]
        weights = (total * sampling_probabilities) ** (-beta)
        weights /= weights.max()

        states, actions, rewards, next_states, dones = map(np.array, zip(*samples))
        return states, actions, rewards, next_states, dones, indices, weights

    def update_priorities(self, batch_indices, batch_errors):
        for i, error in zip(batch_indices, batch_errors):
            self.priorities[i] = np.abs(error) + 1e-5
```

```
class DQNetwork:
    ...
    @tf.function
    def train_step(self, states, actions, rewards, next_states, dones, weights):
        next_Q_values = self.model(next_states)
        max_next_Q_values = tf.reduce_max(next_Q_values, axis=1)
        target_Q_values = rewards + (1 - dones) * self.gamma * max_next_Q_values
        with tf.GradientTape() as tape:
            all_Q_values = self.model(states)
            Q_values = tf.reduce_sum(all_Q_values * actions, axis=1, keepdims=False)
            loss = tf.reduce_mean(weights * self.loss_fn(target_Q_values, Q_values))  #RIGHE CAMBIATE
        grads = tape.gradient(loss, self.model.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))
        td_errors = tf.abs(tf.subtract(target_Q_values, Q_values))  #RIGHE CAMBIATE
        return td_errors  #RIGHE CAMBIATE
```

FIXED Q_VALUES TARGETS

Nelle architetture DQN standard il modello è usato sia per calcolare il Q-Value che il Q-Value Target. Questo loop può portare a oscillazioni e instabilità nel training. Per risolvere questo problema si utilizzano due modelli separati: l'online model che è allenato ad ogni step e muove l'agente nell'ambiente, e il target model utilizzato solo per definire i Q-Value targets. Il target model è un clone di quello online:

```
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())
```

Nella funzione `train_step` è cambiata giusto una riga di codice in modo da usare il target model al posto di quello online quando sono calcolati i Q-Values dei prossimi stati.

```
next_Q_values = target.predict(next_states)
```

Nel training loop sono copiati i pesi dell'online model e impostati nel target model a intervalli regolari (nel nostro caso 5):

```
if self.n_games % 5 ==0:
  self.target_model.set_weights(self.online_model.get_weights())
```

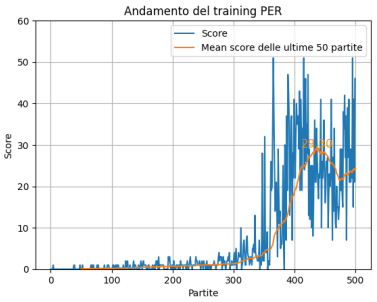
Dato che il target model è aggiornato meno frequentemente i Q-Value target sono più stabili.

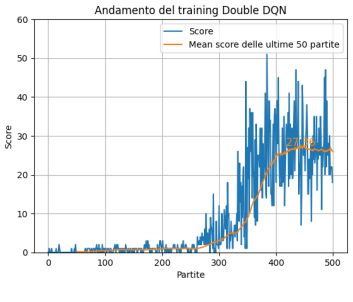
Double DQN

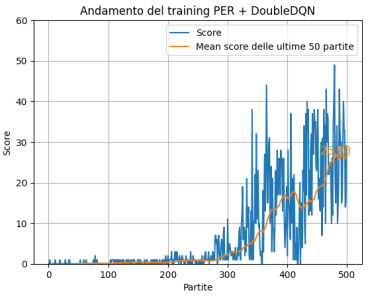
Il Double DQN riprende la tecnica precedente considerando un secondo problema. OpenaAI ha osservato che il modello target tende a sovrastimare i Q-Values. Per risolvere questo problema hanno utilizzato il modello online al posto di quello target nello scegliere la migliore azione per i prossimi stati, e usare il target model solo per stimare i Q-Values per le loro migliori azioni.

```
  def train_step(self, states, actions, rewards, next_states, dones):
      next_Q_values = self.online_model(next_states)
      # Double DQN: l'online model sceglie l'azione dei prossimi stati ma i Q-Value sono stimati da target_model
      max_next_Q_values_by_model = tf.argmax(next_Q_values, axis=1)
      mask_for_target = tf.one_hot(max_next_Q_values_by_model, 3)
      max_next_Q_values_by_target = tf.reduce_sum(self.target_model(next_states) * mask_for_target, axis=1)
      ...
```

Di seguito riportiamo i risultati delle tecniche utilizzate singolarmente e combinate.







Come si può notare le tecniche non hanno prodotto risultati migliori. Questo però deve essere attribuito ancora una volta alla rappresentazione miope utilizzata che pone un pesante limite al punteggio raggiungibile.

Refactoring del codice DQN

A causa dell'introduzione di nuove tecniche come la Prioritized Experience Replay e il Double DQN nel progetto di reinforcement learning per il gioco Snake, si è reso necessario un refactoring del codice. Questo ha portato alla creazione di una mini libreria modulare che implementa queste varianti in modo flessibile. L'obiettivo è stato quello di rendere il codice più scalabile, permettendo di aggiungere facilmente nuove tecniche e algoritmi senza modificare pesantemente l'architettura esistente. Il codice è contenuto nella cartella DQN_components, che consente di costruire rapidamente un agente, permettendo al programmatore di definire le componenti e avviare il training con poche righe di codice. Di seguito un esempio che mostra come inizializzare un agente e allenarlo:

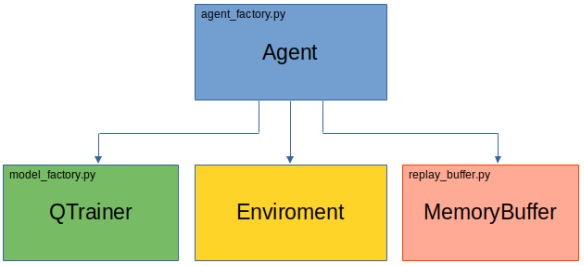
```
agent = agent_factory.Agent_DoubleDQN(env, lr=0.001, gamma=0.95, max_memory=50_000, batch_size=1024,
                                      model_units=[200, 400], input_shape=[53], n_actions=4)
train_result = agent.train_agent(N_GAME=1000, episode_decay=350, eps_greedy=True,
                                 update_target_model=5,
                                 directory_path="./DQN_saved_model/matrix_state/",
                                 file_name_model="model_DDQN.keras")
```

Questo esempio mostrala configurazione di un agente con una rete neurale personalizzata, specificando nei parametri il learning rate, il fattore di sconto, la capacità della memoria, e la dimensione del batch. Inoltre, si definisce l'architettura della rete tramite il parametro model_units, che rappresenta le unità per hidden layers

passate come array, e si specificano l'input (input_shape) e il numero di azioni possibili (n_actions). Da sottolineare che agent_factory permette la scelta di creazione delle differenti tipologie di agenti.

Dopo aver creato l'agente, viene lanciata la procedura di allenamento con train_agent, che permette di configurare il numero di episodi di gioco (N_GAME), il decadimento dell'esplorazione (episode_decay), e la scelta della politica di esplorazione (eps_greedy). Infine, si specifica dove salvare il modello addestrato attraverso il percorso e il nome del file (directory_path e file_name_model).

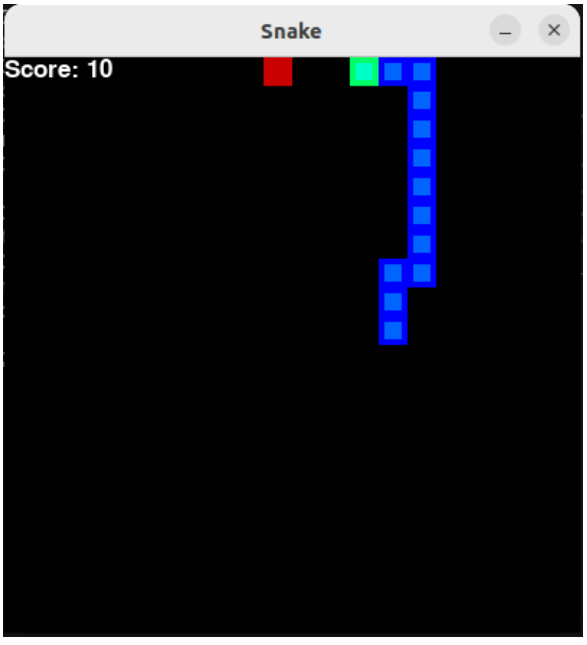
I file che gestiscono questo piccolo modulo RL sono agent_factory.py, model_factory.py e replay_buffer, la cui gerarchia è riportata a destra. Agent è colui che si occupa di gestisce le altre classi al fine di eseguire il training. Inoltre è facile estendere l'applicazione dell'agente a qualsiasi altro ambiente; le uniche richieste per il corretto funzionamento è che l'istanza di *eviroment* sia passata come parametro alla creazione di Agent e che offra le funzioni `get_state()` e `play_step()`, al resto ci pensa l'agente.



In questo modo, il package offre una soluzione modulare e flessibile per gestire la creazione e l'allenamento degli agenti.

Seconda rappresentazione di stato

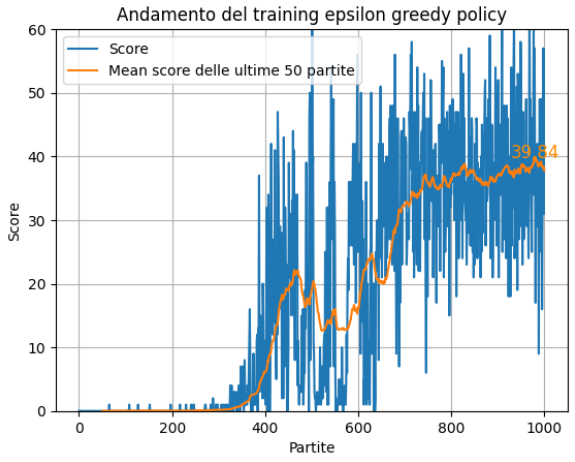
A questo punto si vuole cercare di migliorare i risultati ottenuti, rendendo la rappresentazione dello stato più complessa per permettere all'agente di comprendere meglio i pericoli circostanti. Una soluzione ovvia sarebbe quella di fornire come input i 20×20 blocchi dell'intera griglia di gioco. Tuttavia, questo approccio comporta un elevato costo computazionale durante l'allenamento, richiedendo un modello molto più grande rispetto a quello utilizzato finora. Si è quindi adottato un compromesso, utilizzando una rappresentazione dell'intorno della testa di Snake che le consenta di rilevare i pericoli nelle immediate vicinanze. Il nuovo stato è una matrice 7×7, con la testa di Snake al centro, rappresentata da un 2, i pericoli (coda e muro) con 1, il frutto -1 e il resto con 0. Lo stato sarà quindi formato da questi 49 valori a cui sono sommati altri 4 che indicano la posizione del frutto rispetto alla testa per dare indicazione su dove proseguire.

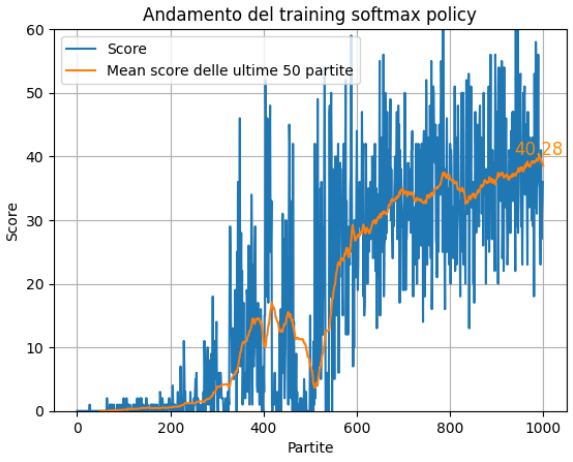


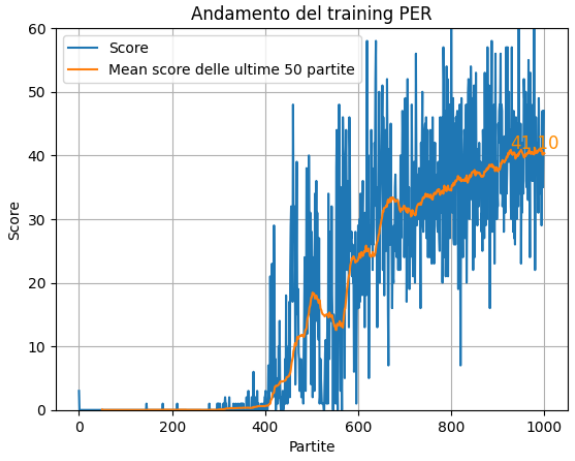
$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & 0 & 0 & 2 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

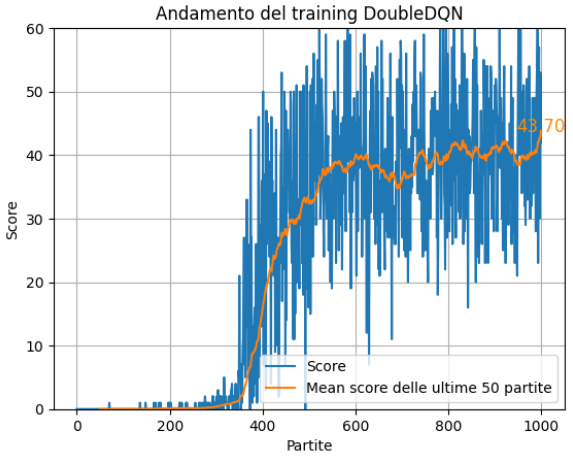
Se si continuassero a utilizzare le stesse azioni finora adottate, nel caso in cui la testa dello Snake entri in contatto con un pezzo o più della sua coda, il modello non sarebbe in grado di determinare la posizione del "collo" e quindi quale azione compiere per salvarsi. Questo accade perché le azioni dipendono dalla direzione corrente, un'informazione che in questa situazione non è più disponibile. Per risolvere il problema, si è deciso di estendere il set di azioni da 3 a 4: vai su, giù, destra o sinistra.

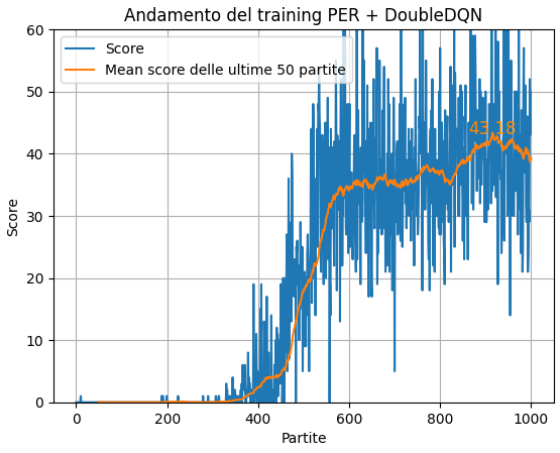
Di seguito sono riportati i risultati con l'architettura finale scelta con due hidden layers rispettivamente di 200 e 400 unità.





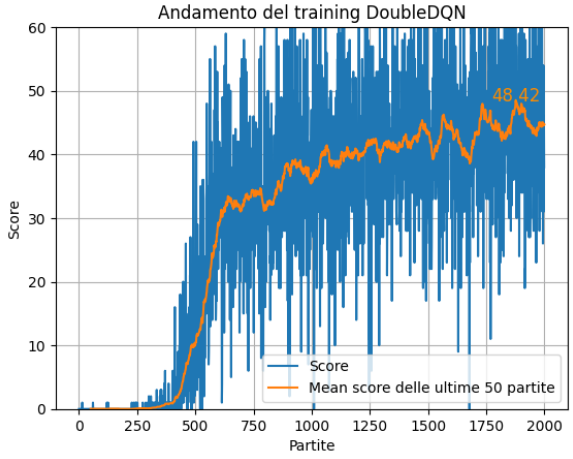


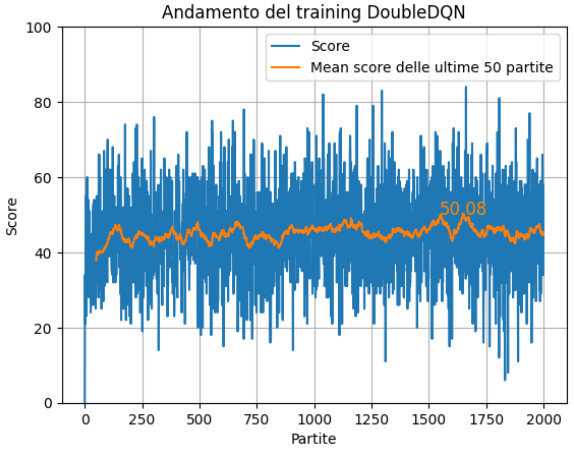




Si può osservare una differenza nell'andamento dei grafici: ancora una volta, la softmax ha garantito una velocità di apprendimento maggiore rispetto alla epsilon-greedy policy. Un'altra considerazione importante riguarda la stabilità e i risultati migliori ottenuti applicando le varie tecniche proposte da OpenAI. L'andamento risulta più rapido e stabile utilizzando una Double DQN, raggiungendo una stabilità intorno a uno score di 43,7 punti maggiore agli altri approcci. Inoltre, durante il tuning manuale dei parametri, la DDQN ha costantemente fornito i migliori risultati. Il punteggio massimo raggiunto è stato 85, vetta mai toccata dagli approcci precedenti. Per quanto riguarda l'uso di buffer PER questo non sembra portare un particolare contributo, non distinguendosi dagli altri andamenti.

Una visuale limitata a un range di 3 blocchi di distanza dalla testa ha consentito di evitare mosse che avrebbero inevitabilmente portato al game over, specialmente quando si cercava di raggiungere il frutto, superando così il test che falliva con l'approccio precedente. Tuttavia, questa visuale ristretta non ha ancora permesso di ottenere punteggi superiori alla soglia di 45 punti. Un possibile miglioramento potrebbe consistere nell'ampliare notevolmente la visuale o addirittura passare in input la totalità dei blocchi di gioco, ma ciò richiederebbe una rete più complessa e un tempo di addestramento più lungo, aumentando lo sforzo computazionale insostenibile per il computer a disposizione. Si è quindi optato per aumentare la visuale a una matrice 9×9, per valutare i risultati ottenuti con questa configurazione.

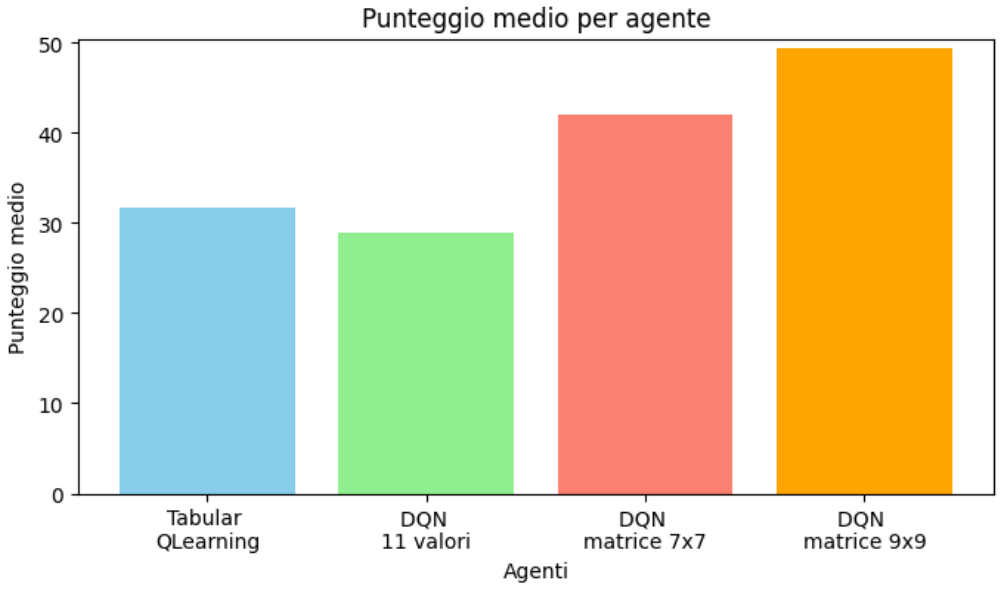




Confronto tra approcci

Di seguito è mostrato il punteggio medio di ciascun agente allenato con differente approccio.

Si nota che, nonostante la rappresentazione dello stato con 11 valori, il Q-learning tabellare fornisce risultati leggermente migliori. Questo avviene perché il Q-learning memorizza esattamente ogni Q-value per coppia stato-azione, un approccio più preciso rispetto alla DQN, che invece tenta di approssimare la funzione Q con una rete neurale. Gli altri due approcci superano il limite precedentemente individuato sul punteggio di 30, ma anche loro con visuale ridotta non lasciano sperare di risolvere il gioco nel completo.



Conclusione

In conclusione, il progetto ha rappresentato un'opportunità significativa per acquisire conoscenze approfondite nel campo del reinforcement learning, grazie all'implementazione di diversi algoritmi quasi da zero. Questa esperienza ha permesso di comprendere non solo le fondamenta teoriche, ma anche le sfide pratiche legate all'addestramento di agenti intelligenti. Un possibile proseguimento del lavoro potrebbe essere l'integrazione di tecniche più avanzate, come l'algoritmo Proximal Policy Optimization (PPO) algoritmo standard nel campo. Per quanto riguarda il gioco di Snake i mezzi a disposizione non ci permettono di proseguire con modelli e rappresentazioni più complesse. Nonostante questi algoritmi abbiano dimostrato una buona capacità di apprendere strategie per ottenere punteggi alti, se l'obiettivo fosse risolvere il gioco in modo ottimale, gli algoritmi di reinforcement learning potrebbero non essere la scelta più indicata. Un approccio più efficace potrebbe essere l'uso di cicli hamiltoniani (https://github.com/BrianHaidet/AlphaPhoenix/tree/master/Snake_AI_(2020a)_DHCR_with_strategy) che garantirebbe un percorso sempre valido per la crescita infinita dello Snake, risolvendo così il gioco in maniera deterministica ed efficiente.