# Super Mario Bros: Progetto IA

*Alessandro* Aquino, *Mattia* d'Argenio e *Alberto* Montefusco

**Abstract.** In questa relazione esamineremo due tipologie di algoritmi di Reinforcment Learning: il Q-Learning e Sarsa. In seguito analizzeremo i problemi riscontrati e la logica, quindi l'implementazione, degli agenti per poi terminare con un'analisi dei modelli ottenuti dopo la fase di addestramento. L'obiettivo è stato di effettuare benchmarking tra i vari algoritmi di RL: QL - Sarsa, Double QL - Double Sarsa, DQN - DN Sarsa, DDQN - DDN Sarsa.

## 1 Introduzione

### 1.1 Problema

Il seguente documento riguarda lo sviluppo di un agente intelligente per il famoso gioco prodotto dalla Nintendo: Super Mario Bros. Più nel dettaglio, l'obiettivo è stato progettare, implementare e addestrare un agente con l'algoritmo di apprendimento per rinforzo Q-learning. I risultati ottenuti dall'apprendimento sono stati, in seguito, confrontati con l'algoritmo SARSA. Tra le varie tipologie di algoritmi analizzati, troviamo:

- Q-Learning classico (QL);
- Double Q-Learning (DQL);
- Deep Q-Network (DQN);
- Double Deep Q-Network (DDQN).

Il motivo per cui abbiamo analizzato questi diversi tipi di algoritmi è per questioni di prestazioni che poi approfondiremo.

### 1.2 Workflow: Super Mario

L'agente dovrà essere in grado di attraversare il primo livello del primo mondo del gioco Super Mario Bros evitando ostacoli, nemici e raccogliendo power-up. Per ogni azione compiuta da Mario è prevista una ricompensa o penalità per guidare l'apprendimento dell'agente. Il sistema di gioco è stato implementato tramite la libreria gym-super-mario-bros: un ambiente OpenAI Gym per Super Mario Bros su Nintendo Entertainment System (NES) che utilizza l'emulatore nes-py. Per impostazione predefinita, gli ambienti gym-super-mario-bros utilizzano l'intero spazio d'azione del NES, composto da 256 azioni discrete. Per limitare questo aspetto, *gym-super-mario-bros.actions* fornisce tre elenchi di azioni (`RIGHT_ONLY`, `SIMPLE_MOVEMENT` e `COMPLEX_MOVEMENT`).
Di seguito tutte le azioni che può compiere Mario in ognuna delle tre configurazioni.

- `RIGHT_ONLY` = (['NOOP'], ['right'], ['right', 'A'], ['right', 'B'], ['right', 'A', 'B'])
- `SIMPLE_MOVEMENT` = (['NOOP'], ['right'],[ 'right', 'A'], ['right', 'B'], ['right', 'A', 'B'], ['A'], ['left'])
- `COMPLEX_MOVEMENT` = (['NOOP'], ['right'], ['right', 'A'], ['right', 'B'], ['right', 'A', 'B'], ['A'], ['left'], ['left', 'A'], ['left', 'B'], ['left', 'A', 'B'], ['down'], ['up'])

Inoltre, per guidare l'agente nel superamento del livello col fine di prendere il maggior numero di power-up e score, abbiamo previsto l'implementazione di ricompense personalizzate che possano aiutare lo sviluppo dell'agente. Le ricompense sono:

- "time": -0.1 per ogni secondo che passa;
- "death": -100 viene attribuita quando Mario muore;
- "extra-life": 100 quando Mario guadagna una vita extra;
- "mushroom": 20 quando Mario prende un fungo e diventa grande;
- "flower": 25 Mario mangia un fiore;
- "mushroom-hit": -10 Mario colpisce un fungo mentre è grande;
- "flower-hit": -15 Mario viene colpito quando è in modalità fuoco;
- "coin": 15 quando prende una moneta;
- "score": 15 quando Mario colpisce un nemico;
- "victory": 1000 viene attribuita quando Mario termina il livello.

## 2 Probemi riscontrati

La principale problematica riscontrata riguarda le prestazioni dell'algoritmo Q-learning. Infatti, Mario non vince in nessun episodio mandato in esecuzione, in modo analogo con l'algoritmo SARSA. Più nel dettaglio, tramite le sperimentazioni effettuate, si è notato che l'algoritmo del Q-learning ha impiegato circa 28 ore addestrato su 3181 episodi (durante tutto il periodo di addestramento Mario non è riuscito a superare con successo il livello,

registrando risultati pessimi). Lo stesso andamento è stato riscontrato con l'utilizzo del Double Q-learning e con Double Sarsa.

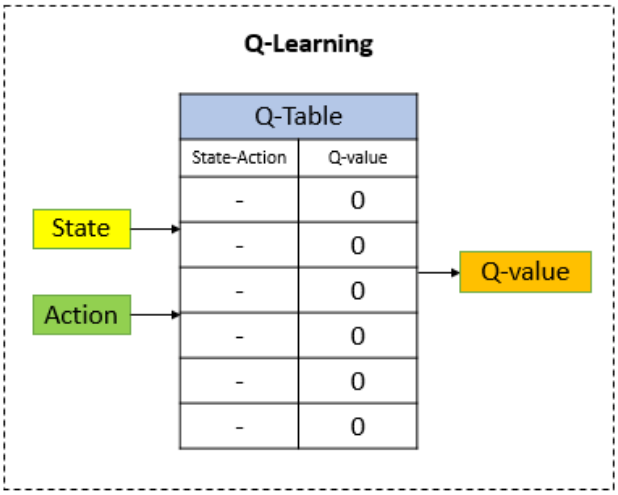


**Figure 1.** Struttura della tabella del Q-Learning

Durante l'analisi delle prestazioni dell'agente intelligente, abbiamo poi analizzato altri due algoritmi di Q-Learning che sfruttavano approcci più avanzati: Deep Q-Network (DQN) e Double Deep Q-Network (DDQN). Le differenze fondamentali tra questi algoritmi giocano un ruolo significativo nel determinare le prestazioni dell'agente, dato che il Q-learning e il Double Q-learning, in scenari complessi come Super Mario Bros, possono incontrare difficoltà nell'apprendere strategie efficienti a causa della vastità dello spazio delle azioni. Abbiamo, pertanto, pensato di utilizzare approcci differenti basati su reti neurali.

DQN rappresenta un notevole avanzamento rispetto all'approccio tradizionale di Q-learning, specialmente in contesti complessi come Super Mario Bros. Una delle sfide affrontate da DQN è la gestione degli spazi di azione estremamente ampi, tipici dei giochi più complessi. Per far fronte a questa complessità, DQN impiega una rete neurale profonda per approssimare la funzione di valore Q. L'architettura di DQN è progettata per ricevere in input la rappresentazione di stato e restituire un vettore di valori Q associati a ciascuna possibile azione. L'addestramento avviene attraverso l'ottimizzazione della differenza tra i valori predetti e quelli effettivi, contribuendo a migliorare la capacità dell'agente di prendere decisioni intelligenti nel gioco.

DDQN rappresenta un ulteriore passo avanti nell'affinare le prestazioni di DQN, concentrandosi sulla mitigazione del problema di sovrastimazione dei valori Q. Nel contesto di Super Mario Bros, l'overestimation può portare a decisioni sub-ottimali, influenzando negativamente l'apprendimento dell'agente. Per risolvere questo problema, DDQN introduce la distinzione tra la selezione dell'azione e la valutazione, utilizzando due reti neurali separate, rispettivamente, per stimare il valore target e selezionare l'azione. Questo approccio riduce l'errore di sovrastimazione, migliorando la stabilità e l'efficacia dell'apprendimento.

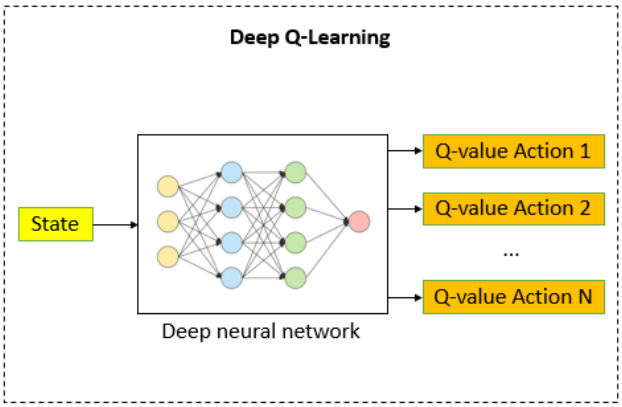


**Figure 2.** Deep Q-Network

Durante le nostre sperimentazioni, abbiamo osservato che l'utilizzo di DQN e DDQN hanno prodotto risultati superiori rispetto agli approcci più tradizionali come Q-learning e Double Q-learning. In particolare, l'algoritmo che ha prodotto i risultati migliori e ha impiegato il minor tempo di addestramento è stato DDQN per un totale di 10.000 episodi eseguiti in 13 ore circa di addestramento. Il restante dei risultati verrà analizzato nel paragrafo 4 chiamato "Analisi dei risultati".

## 3 Implementazione

In questa sezione andremo a trattare le parti implementative salienti per sviluppare un agente Q-learning, DQN e DDQN (e i corrispettivi algoritmi Sarsa) capace di apprendere strategie ottimali per navigare attraverso il livello e raggiungere l'obiettivo.

### 3.1 Enviroment

Questa sezione definisce una serie di wrapper per ambienti Gym, utilizzati per pre-processare le osservazioni in modo da renderle più adatte all'addestramento della QL, Double QL, DQN, DDQN e i rispettivi Sarsa.

- MaxAndSkipEnv: questo wrapper restituisce ogni i-esimo frame durante la chiamata del metodo step, consentendo un'approssimazione più rapida dell'azione in situazioni in cui i frame consecutivi possono essere simili. Usa il massimo valore tra i frame consecutivi.
- ProcessFrame84: Questo wrapper riduce la risoluzione delle osservazioni a 84x84 pixel e le converte in scala di grigi. Viene utilizzato per rendere più efficiente l'addestramento riducendo la dimensione dell'input.
- ImageToPyTorch: questo wrapper modifica le osservazioni per adattarle a PyTorch, riorganizzando gli assi in modo che siano compatibili con l'ordine comune utilizzato in PyTorch (canale, altezza, larghezza).

- ScaledFloatFrame: normalizza i valori dei pixel nell'intervallo da 0 a 1. Questo è utile per garantire che le osservazioni siano in una scala comune per l'addestramento di reti neurali.
- BufferWrapper: crea un buffer di osservazioni per tener traccia degli ultimi n_steps frame. Ogni osservazione è ripetuta per n_steps. Utile per fornire al modello informazioni temporali sulle dinamiche dell'ambiente.
- PixelNormalization: normalizza i valori dei pixel nell'intervallo da 0 a 1. Questo wrapper sembra duplicare la funzionalità di ScaledFloatFrame. Potrebbe essere un duplicato o potrebbe essere utilizzato in combinazione con altri wrapper a seconda delle esigenze specifiche.

### 3.2 Q-Learing & Sarsa: implementazione

La logica di base dell'agente utilizza una politica *epsilon-greedy* per esplorare e sfruttare l'ambiente nel processo di apprendimento. I valori Q sono inizializzati casualmente e vengono aggiornati utilizzando le equazioni di Q-learning. L'agente apprende valutando le azioni in base alle ricompense ricevute e cerca di massimizzare la somma delle ricompense future.

#### *3.2.1 Q-Learing: implementazione Agent*

Il primo metodo che andremo a trattare è **obs_to_state**. Converte un'osservazione (stato dell'ambiente) in uno stato numerico. Se l'osservazione è già presente nel vettore di osservazioni (obs_vec), restituisce l'indice corrispondente. Altrimenti, aggiunge l'osservazione a obs_vec e restituisce il nuovo indice.

```
def obs_to_state(self, observation):

    state = -1
    for i in range(len(self.obs_vec)):
        if np.array_equal(observation, self.
    obs_vec[i]):
            state = i
            break
    if state == -1:
        state = len(self.obs_vec)
        self.obs_vec.append(observation)
    return state
```

**take_action**: l'agente sceglie un'azione in base alla politica epsilon-greedy; se un valore casuale è maggiore di exploreP (la probabilità di esplorazione iniziale), l'agente esegue l'azione corrispondente al massimo valore Q; altrimenti, l'agente esegue un'azione casuale. La politica dell'agente è controllata dalla condizione *np.random.rand() > self.exploreP*. Se questa condizione è vera, l'agente esegue l'azione corrispondente al massimo valore Q (exploitation). Altrimenti, l'agente esegue un'azione casuale (exploration). La probabilità di esplorazione exploreP diminuisce nel tempo, poiché viene moltiplicata per 0.99 ad ogni chiamata di *take_action*. Quindi, in questo caso, la politica è una combinazione di esplorazione e sfruttamento, dove l'agente inizialmente esplora di più (azioni casuali) e gradualmente si orienta verso l'exploitation (sfruttamento dei valori Q appresi).

```
def take_action(self, state):

    q_a = self.get_qval(state)
    if np.random.rand() > self.exploreP:
        action = np.argmax(q_a)
    else:
        action = self.env.action_space.sample()
    self.exploreP *= 0.99
    return action
```

**update_qval**: aggiorna i valori Q in base all'equazione di aggiornamento di Q-learning. Calcola il target temporale TD_target e il calcolo dell'errore temporale td_error per aggiornare i valori Q associati all'azione e allo stato corrente.

```
def update_qval(self, action, state, reward,
    next_state, terminal):
    if terminal:
        td_target = reward
    else:
        td_target = reward + self.gamma * np.
    amax(self.get_qval(next_state))
    td_error = td_target - self.get_qval(state)[
    action]
    self.state_a_dict[state][action] += self.
    alpha * td_error
```

#### *3.2.2 SARSA: implementazione Agent*

SARSA aggiorna i valori Q in modo incrementale, prendendo in considerazione le azioni attuali e future. Questo approccio consente all'agente di apprendere una politica basata sull'esperienza, considerando come le sue azioni influenzano direttamente il suo futuro.

**update_qval_SARSA**: nello specifico se terminal è vero, cioè se siamo nel termine dell'episodio, allora il td_target (target temporale) è impostato alla sola ricompensa ottenuta nell'ultimo passo (reward). Altrimenti, se non siamo nel termine dell'episodio, calcoliamo il td_target utilizzando l'equazione SARSA: **td_target = reward + gamma * Q(next_state, next_action)**. Calcoliamo l'errore temporale (td_error), che rappresenta la differenza tra il td_target e il valore Q stimato per l'azione corrente nello stato attuale. Aggiorniamo il valore Q corrispondente utilizzando il tasso di apprendimento (alpha) e l'errore temporale (td_error):

```
def update_qval_sarsa(self, action, state,
    reward, next_state, next_action, terminal):
    if terminal:
        td_target = reward
    else:
        td_target = reward + self.gamma * self.
    get_qval(next_state)[next_action]

    td_error = td_target - self.get_qval(state)[
    action]
    self.state_a_dict[state][action] += self.
    alpha * td_error
```

### 3.3 Double QL & Double Sarsa: implementazione

In Double Q-Learning, vengono mantenuti due set di valori Q, mentre la scelta delle azioni e gli aggiornamenti dei valori Q sono effettuati utilizzando entrambi i set. Questo è

fatto per mitigare il problema della sovrastima dei valori Q che può verificarsi nel Q-Learning classico. La scelta delle azioni e gli aggiornamenti dei valori Q vengono effettuati utilizzando i due set in modo alternato e periodicamente i valori Q da uno dei set vengono copiati nell'altro.

#### 3.3.1 Double QL: implementazione Agent

In questa sezione verranno trattate solamente le funzioni che differiscono da quelle del Q-learning. La funzione copy non è presente nel Q-Learning ed il suo compito è di copiare i valori da state_a_dict a Q_target. Questa operazione avviene periodicamente, ad ogni copy_steps passi.

```
    def copy(self):
        self.Q_target = self.state_a_dict.copy()
```

Il metodo update_Qval aggiorna i valori Q utilizzando l'equazione di aggiornamento del Q-Learning, ma prende in considerazione il massimo valore Q dal set target (Q_target) durante l'aggiornamento.

```
def update_Qval(self, state, action, reward,
    next_state, terminal):
    if terminal:
        td_target = reward
    else:
        td_target = reward + self.gamma * np.
    amax(self.get_Qtarget(next_state))

    td_error = td_target - self.get_Qval(state)[
    action]
    self.state_a_dict[state][action] += self.
    alpha * td_error
```

#### 3.3.2 Double SARSA: implementazione Agent

Double Sarsa utilizza due dizionari: (state_a_dict1 e state_a_dict2). La logica dell'agente consiste nel fare la media tra le stime successive dei valori Q durante l'aggiornamento per ridurre la varianza. Entrambi gli approcci (Double QL e Double Sarsa) cercano di migliorare la stabilità dell'apprendimento nei casi in cui l'aggiornamento di un valore Q influenza la stima dell'altro.

**update_Qval:** verifica se lo stato successivo è terminale (terminal è vero), se è così il target di differenza temporale (td_target) è impostato sulla ricompensa ottenuta. Se lo stato successivo non è terminale, vengono effettuati i seguenti passaggi:

- viene effettuata una scelta casuale tra i due dizionari (state_a_dict1 e state_a_dict2) per selezionare quale stima dei valori Q utilizzare per calcolare il target, questo viene fatto tramite *np.random.rand() < 0.5*;
- viene scelta l'azione successiva (next_action) in base al dizionario scelto in precedenza;
- calcoliamo il target di differenza temporale (td_target) utilizzando la ricompensa corrente, il fattore di sconto gamma e il valore Q del dizionario non scelto;
- viene effettuata un'altra scelta casuale tra i due dizionari per decidere quale dei due valori Q (corrispondente all'azione corrente nello stato corrente) aggiornare. L'errore temporale (td_error) viene calcolato sottraendo il valore Q attuale dalla sua controparte nel dizionario scelto. L'aggiornamento effettivo avviene incrementando il valore Q corrente con un passo proporzionale all'errore temporale moltiplicato per il tasso di apprendimento.

```
def update_Qval(self, state, action, reward,
    next_state, terminal):
    if terminal:
        td_target = reward
    else:
        if np.random.rand() < 0.5:
            next_action = np.argmax(self.
    get_Qval2(next_state))
            td_target = reward + self.gamma *
    self.get_Qval1(next_state)[next_action]
        else:
            next_action = np.argmax(self.
    get_Qval1(next_state))
            td_target = reward + self.gamma *
    self.get_Qval2(next_state)[next_action]
    if np.random.rand() < 0.5:
        td_error = td_target - self.get_Qval1(
    state)[action]
        self.state_a_dict1[state][action] +=
    self.alpha * td_error
    else:
        td_error = td_target - self.get_Qval2(
    state)[action]
        self.state_a_dict2[state][action] +=
    self.alpha * td_error
```

### 3.4 Rete Neurale: implementazione DQN - DDQN & DN Sarsa - DDN Sarsa

La rete apprende una rappresentazione delle immagini degli stati dell'ambiente attraverso i layer convoluzionali e utilizza questa rappresentazione per stimare i valori Q associati alle diverse azioni. L'architettura è progettata per catturare pattern spaziali e gerarchie di features che possono emergere dagli stati dell'ambiente. L'output finale fornisce una stima dei valori Q per ciascuna azione, che viene utilizzata per prendere decisioni durante l'addestramento e il processo decisionale in fase di test. In questo paragrafo analizzeremo sia la Deep QN (DQN) che la Double Deep QN (DDQN) e i rispettivi Deep Network Sarsa (DN Sarsa) e Double Deep Network Sarsa (DDN Sarsa).

#### 3.4.1 DQN & DDQN: implementazione degli Agents

Il metodo **remember** è responsabile di memorizzare le esperienze (stato, azione, ricompensa, nuovo stato, flag di terminazione) nella memoria di riproduzione dell'agente. La memoria di riproduzione viene utilizzata successivamente per il replay dell'esperienza durante l'addestramento della rete neurale.

```
def remember(self, state, action, reward, state2
    , done):
    self.STATE_MEM[self.ending_position] = state
    .float()
    self.ACTION_MEM[self.ending_position] =
    action.float()
```

```
    self.REWARD_MEM[self.ending_position] =
    reward.float()
    self.STATE2_MEM[self.ending_position] =
    state2.float()
    self.DONE_MEM[self.ending_position] = done.
    float()
    self.ending_position = (self.ending_position
     + 1) % self.max_memory_size  # FIFO tensor
    self.num_in_queue = min(self.num_in_queue +
    1, self.max_memory_size)
```

**batch_experiences**: il metodo campiona casualmente un insieme di indici dalla memoria di riproduzione per estrarre un batch di esperienze. Questo approccio di campionamento casuale aiuta a introdurre casualità durante l'addestramento, contribuendo a ridurre la correlazione temporale tra le transizioni degli stati.

```
def batch_experiences(self):
    """Randomly sample 'batch size' experiences
    """
    idx = random.choices(range(self.num_in_queue
    ), k=self.memory_sample_size)
    state = self.STATE_MEM[idx]
    action = self.ACTION_MEM[idx]
    reward = self.REWARD_MEM[idx]
    state2 = self.STATE2_MEM[idx]
    done = self.DONE_MEM[idx]
    return state, action, reward, state2, done
```

**act**: è responsabile di selezionare un'azione da intraprendere in uno stato dato, seguendo una politica epsilon-greedy. Questa politica bilancia l'esplorazione (prova di azioni casuali per scoprire nuove strategie) e lo sfruttamento (scegliere l'azione migliore stimata dal modello). Nel caso di DDQN, la rete locale viene utilizzata per selezionare l'azione migliore.

```
def act(self, state):
    """Epsilon-greedy action"""
    if self.double_dqn:
        self.step += 1
    if random.random() < self.exploration_rate:
        return torch.tensor([[random.randrange(
    self.action_space)]])
    if self.double_dqn:
        # Local net is used for the policy
        return torch.argmax(self.local_net(state
    .to(self.device))).unsqueeze(0).unsqueeze(0)
    .cpu()
    else:
        return torch.argmax(self.dqn(state.to(
    self.device))).unsqueeze(0).unsqueeze(0).cpu
    ()
```

**experience_replay**: sviluppa la logica principale in diversi punti.

- Controllo Dimensione della Memoria: se la dimensione della memoria di riproduzione è inferiore alla dimensione del batch, l'agente esce dalla funzione senza fare nulla.
- Estrazione Batch di Esperienze: viene campionato casualmente un batch di esperienze dalla memoria di riproduzione utilizzando il metodo batch_experiences.
- Calcolo dei Target: viene calcolato il target di apprendimento basato sulla formula di aggiornamento Q-Network o Double Deep Q-Network, a seconda della configurazione. La rete target è utilizzata per stimare i valori Q nel caso di Double Q-Network.
- Calcolo dell'Errore: viene calcolata la perdita (errore) tra i valori stimati correnti e i valori target.
- Retropropagazione dell'errore, aggiornamento dei pesi e aggiornamento del tasso di esplorazione.
- Controllo Minimo del Tasso di Esplorazione: garantisce che il tasso di esplorazione non scenda al di sotto del valore minimo specificato (self.exploration_min).

Nel caso in cui venga utilizzata la DDQN e il numero di passi (*self.step*) è un multiplo di *self.copy*, viene copiato il modello.

```
def experience_replay(self):
        if self.double_dqn and self.step % self.
    copy == 0:
            self.copy_model()

        if self.memory_sample_size > self.
    num_in_queue:
            return

        state, action, reward, state2, done =
    self.batch_experiences()
        state = state.to(self.device)
        action = action.to(self.device)
        reward = reward.to(self.device)
        state2 = state2.to(self.device)
        done = done.to(self.device)

        self.optimizer.zero_grad()
        if self.double_dqn:
            target = reward + torch.mul((self.
    gamma * self.target_net(state2).max(1).
    values.unsqueeze(1)), 1 - done)

            current = self.local_net(state).
    gather(1, action.long())  # Local net
    approximation of Q-value
        else:
            target = reward + torch.mul((self.
    gamma * self.dqn(state2).max(1).values.
    unsqueeze(1)), 1 - done)

            current = self.dqn(state).gather(1,
    action.long())

        loss = self.l1(current, target)
        loss.backward()  # Compute gradients
        self.optimizer.step()  # Backpropagate
    error

        self.exploration_rate *= self.
    exploration_decay


        self.exploration_rate = max(self.
    exploration_rate, self.exploration_min)
```

### *3.4.2 DN Sarsa & DDN Sarsa: implementazione Agents*

La principale differenza tra experience_replay e experience_replay_sarsa riguarda il modo in cui viene calcolato il target Q e come viene gestita l'azione successiva nell'aggiornamento dei pesi. experience_replay segue l'approccio classico di Q-learning, mentre experience_replay_sarsa adatta questo approccio per supportare SARSA, tenendo conto dell'azione successiva nella stima dei valori Q.

```
def experience_replay_sarsa(self):
    """Use the double Sarsa-update or Sarsa-
    update equations to update the network
    weights"""
    if self.double_dqn and self.step % self.copy
     == 0:
        self.copy_model()

    if self.memory_sample_size > self.
    num_in_queue:
        return

    # Sample a batch of experiences
    state, action, reward, state2, done = self.
    batch_experiences()
    state = state.to(self.device)
    action = action.to(self.device)
    reward = reward.to(self.device)
    state2 = state2.to(self.device)
    done = done.to(self.device)

    self.optimizer.zero_grad()

    # double_sarsa
    if self.double_dqn:

        next_action = self.local_net(state2).max
    (1).indices.unsqueeze(1)
        target = reward + torch.mul(self.gamma *
     self.target_net(state2).gather(1,
    next_action), 1 - done)
        current = self.local_net(state).gather
    (1, action.long())
    else:

        next_action = self.dqn(state2).argmax(1)
    .unsqueeze(1)
        target = reward + torch.mul(self.gamma *
     self.dqn(state2).gather(1, next_action), 1
    - done)
        current = self.dqn(state).gather(1,
    action.long())

    loss = self.l1(current, target)
    loss.backward()
    self.optimizer.step()

    self.exploration_rate *= self.
    exploration_decay
    self.exploration_rate = max(self.
    exploration_rate, self.exploration_min)
```

## 4 Analisi dei risultati

In questa sezione andremo a mostrare e a discutere dei risultati ottenuti dai diversi modelli stabilendo quale caso ha prodotto metriche migliori. Per una visione completa andare al GitHub del progetto.

La figura 3 riguarda il confronto fatto tra l'algoritmo di Q-learning e Sarsa su 1000 episodi. Evidenziamo dal grafico, per quanto riguarda l'algoritmo Sarsa, una salita durante i primi 100 episodi per poi proseguire con una tendenza continua e discendente. L'addestramento è stato eseguito sulle nostre macchine e si può notare che l'algoritmo Sarsa ha ottenuto i risultati migliori.

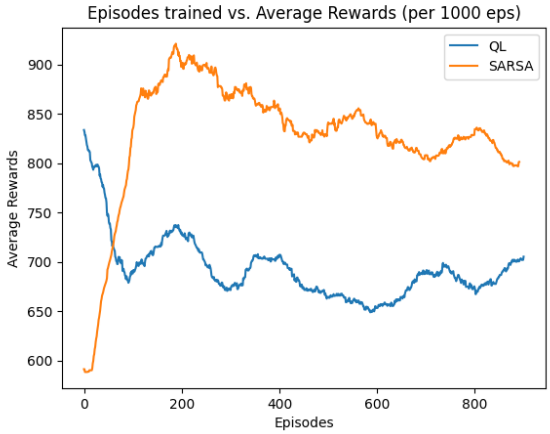


**Figure 3.** Confronto tra Q-learning e Sarsa

La figura 4 evidenzia il confronto tra Double Q-learning e Double Sarsa su 1000 episodi. L'esperimento è stato condotto sulle nostre macchine e possiamo notare un andamento molto simile tra i due algoritmi, con il secondo leggermente migliore rispetto al primo.

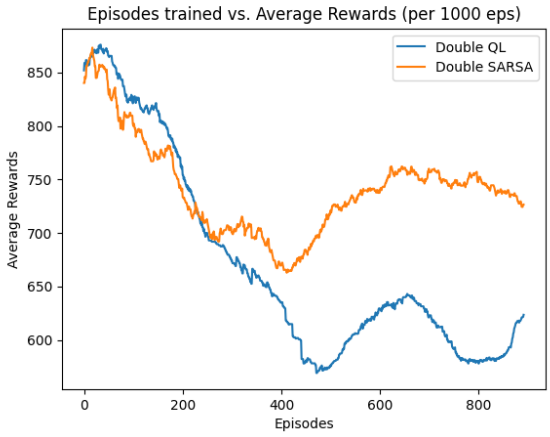


**Figure 4.** Confront tra Double Q-learning e Double Sarsa

La figura 5 rappresenta il confronto tra Deep Q-Network e Deep Q-Sarsa su 1000 episodi. L'addestramento è stato eseguito sulle nostre macchine ed evidenzia una tendenza simile con un incremento dell'algoritmo Sarsa nei primi episodi, diversamente dalla Deep Q-Network.

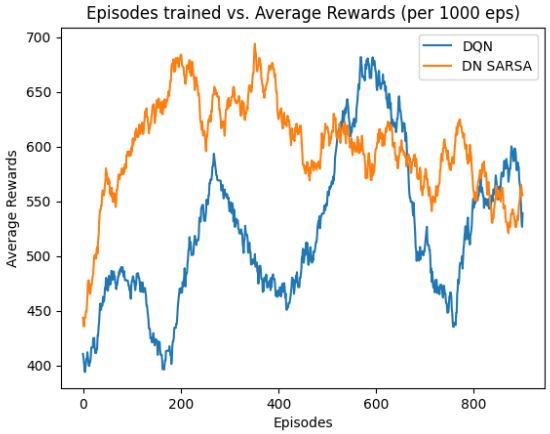


**Figure 5.** Confront tra Deep Q-Network e Deep Network Sarsa

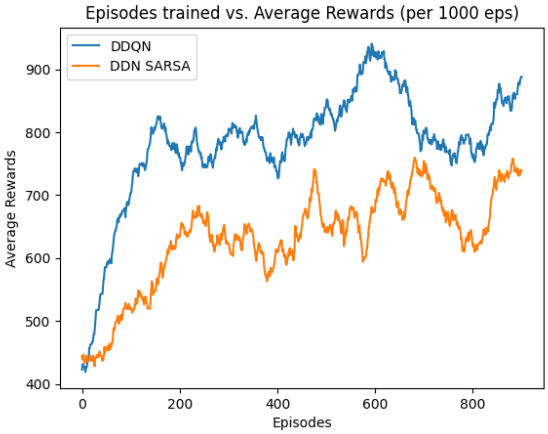


**Figure 6.** Confronto tra Double Deep Q-Network e Double Deep Network Sarsa su 1000 episodi

La figura 6 rappresenta il confronto tra Double Deep Q-Network e Double Deep Q-Sarsa su 1000 episodi. L'esperimento è stato eseguito sulle nostre macchine e possiamo notare anche in questo caso una tendenza pressoché identica con delle prestazioni migliori da parte dell'algoritmo Double Deep Q-Network.

## 5 Data availability

Codice Workflow: GitHub.