

Collaborative Learning

Una soluzione di reinforcement learning
applicata a un nodo router di messaggi

Progetto Machine Learning

Matteo Conti
0323728

Daniele La Prova
0320429

Luca Falasca
0334722

Abstract—In questo lavoro, viene proposto il design di un nodo di rete dotato di code di priorità, una batteria e una presa elettrica, che implementa un agente di reinforcement learning per prendere decisioni basate su un modello di reward. Questo modello tiene conto di vari parametri di qualità del servizio (QoS), come il consumo energetico, l'occupazione delle code e il numero di pacchetti persi. L'obiettivo dell'agente è ottimizzare il comportamento del nodo in modo da massimizzare la reward accumulata. Il modello di reward è configurabile per dare più o meno peso a ciascuna metrica QoS, permettendo di adattare il comportamento dell'agente a diversi scenari. Nel paper viene inoltre esplorata una implementazione personalizzata di un multi-agente collaborativo e confrontata con l'approccio più classico. I risultati mostrano che l'agente di reinforcement learning è in grado di migliorare l'efficienza operativa del nodo rispetto a un agente che prende decisioni casuali, nel caso di coda singola. Tuttavia, l'agente non è in grado di migliorare le prestazioni del nodo nel caso di più code, a causa della complessità aggiuntiva introdotta da questo scenario.

I. INTRODUZIONE

A. Contesto

In un sistema moderno i QoS possono essere di vario genere e non sempre sono legati alle sole prestazioni, in particolare, hanno acquistato sempre più rilevanza aspetti come il valore portato dal sistema e la sostenibilità dello stesso. In un sistema di comunicazione, ad esempio, il valore portato da un messaggio può variare a seconda del tempo che impiega ad essere processato, allo stesso tempo un messaggio inviato utilizzando energia green può avere un valore maggiore rispetto ad un messaggio inviato tramite energia tradizionale.

Il progetto in questione riguarda il design di un nodo di rete dotato di code di priorità, di una batteria e di una presa elettrica, il quale monta un agente di reinforcement learning che impara a prendere decisioni sulla base di un modello di reward che tiene conto di diversi QoS come il consumo energetico, l'occupazione delle code ed il numero di pacchetti persi. L'agente è in grado di interagire con l'ambiente e di apprendere da queste interazioni, in modo da ottimizzare il suo comportamento e massimizzare la reward accumulata. Il modello di reward è configurabile in modo da dare più o meno peso a ciascuna delle metriche di QoS, in modo da poter adattare il comportamento dell'agente a diversi scenari.

B. Obiettivi

Le performance del nodo vengono valutate sulla base delle scelte prese dall'agente in diversi scenari, in particolare al variare di:

- Configurazione del modello di reward dell'agente
- Grado di esplorazione dell'agente
- Frequenza di interrogazione dell'agente
- Numero di code del nodo
- Capacità delle code del nodo
- Costo dell'energia
- Tasso di arrivo dei pacchetti

Per valutare le performance dell'agente esso verrà confrontato con un agente che fa scelte casuali, sulla base di diverse metriche, alcune derivanti dai QoS del nodo ed altre derivanti dall'agente stesso, in particolare:

- Reward per unità di tempo
- Reward cumulativa per unità di tempo
- Tempo medio in coda per unità di tempo, per ogni coda
- Percentuale di occupazione della coda per unità di tempo, per ogni coda
- Numero di pacchetti persi dalla coda per unità di tempo, per ogni coda
- Percentuale di carica della batteria per unità di tempo
- Utilizzazione del nodo per unità di tempo, definito come:

$$\text{arrivalRate} \cdot \text{timeStep} \quad (1)$$

- Costo energetico per unità energetica, definito come:

$$\frac{\text{energyCost}}{\text{energyConsumed}} \quad (2)$$

- Costo energetico per unità di tempo, definito come:

$$\frac{\text{energyCost}}{\text{timeStep}} \quad (3)$$

- Risparmio energetico per unità di tempo, in termini di costo, definito come:

$$100 \cdot \left(1 - \frac{\text{energyCost}}{\text{energyCostOnlyPowerChord}}\right) \quad (4)$$

È opportuno specificare che nella sezione ?? sono state considerate solo un sottinsieme ritenuto maggiormente rilevante delle suddette metriche.

II. METODOLOGIA

Di seguito sono descritti i dettagli relativi alla specifica del modello, dell'agente e della simulazione.

A. Modello

A partire dalla specifica del problema è stato definito un modello che delinea come sono stati rappresentati lo Stato, le Azioni e le Reward della simulazione.

1) *Stato*: Per catturare lo stato della simulazione è stato deciso di campionare a ogni timestep le seguenti variabili:

- Percentuale di carica della batteria;
- Quanta percentuale di batteria è stata ricaricata nell'ultimo timestep;
- La percentuale di occupazione delle code, un valore per ogni coda.

Tali valori sono stati discretizzati ove possibile per permetterne la comprensione da parte di alcune implementazioni di agenti, ad esempio il DQN.

2) *Azioni*: Un nodo può essenzialmente svolgere due operazioni in un qualunque timestep:

- Non fare nulla
- Inviare un pacchetto

Qualora l'azione scelta sia quella di inviare un pacchetto, allora è necessario anche scegliere:

- Da quale coda prelevare il pacchetto
- Da quale power source attingere l'energia necessaria per l'invio

3) *Reward*: La reward r da inviare all'agente per un qualunque timestep t è calcolata secondo la seguente formula:

$$\begin{aligned} r(t) = \sum_{i \in T} w_i \cdot r_i(t), \quad -1 \leq r_i \leq 0 \\ 0 \leq w_i \leq 1, \\ \sum_{i \in T} w_i = 1, \\ T = \{e, o, d\}, \\ t \geq 0 \in \mathbb{N} \end{aligned} \quad (5)$$

In sostanza, la reward è calcolata come la somma pesata dei contributi normalizzati di diversi termini. I pedici $\{e, o, d\}$ si riferiscono rispettivamente ai contributi derivati dal termine energetico, di occupazione delle code e di drop dei pacchetti in quel timestep.

Il termine energetico è calcolato come:

$$\begin{aligned} r_e(t) = - \sum_{p \in P} \frac{\text{cons}_p(t) \cdot \text{cost}_p}{\max_{\tau < t} \{\text{cons}_p(\tau)\} \cdot \sum_{p \in P} \text{cost}_p}, \\ P = \{\text{battery, power chord}\} \end{aligned} \quad (6)$$

Al numeratore abbiamo il prodotto tra il consumo energetico della power source p e il suo costo di utilizzo, normalizzato sul massimo consumo energetico registrato fino a quel time step per quella power source moltiplicato per la somma dei costi di utilizzo. La batteria ha un costo di utilizzo pari a zero ma una capacità limitata e una ricarica influenzata da fattori

casuali, mentre il Power Chord ha un costo di utilizzo diverso da zero ma una capacità pressoché illimitata.

Il termine dell'occupazione delle code è calcolato come:

$$r_o(t) = - \sum_{q \in Q} \frac{\text{occ}_q(t) \cdot p_q}{\max_{\tau < t} \{\text{occ}_q(\tau)\} \cdot \sum_{q \in Q} p_q}, \quad (7)$$

$$Q = \{0, 1, \dots, \#Q - 1\}$$

Al numeratore abbiamo il prodotto tra la percentuale di occupazione della coda q nel timestep t e la sua priorità p_q , normalizzato sul massimo valore di occupazione registrato fino a quel time step per quella coda moltiplicato per la somma delle priorità. Essendo il termine dell'occupazione delle code una percentuale e la priorità definita come $p_q = q + 1$ possiamo riscrivere [Equation 7](#) come:

$$r_o(t) = - \sum_{q \in Q} \frac{\text{occ}_q(t) \cdot (q + 1)}{100 \cdot \frac{1}{2} \#Q (\#Q + 1)}, \quad (8)$$

$$Q = \{0, 1, \dots, \#Q - 1\}$$

Il termine del drop dei pacchetti delle code è calcolato come:

$$r_d(t) = - \sum_{q \in Q} \frac{\text{drop}_q(t) \cdot p_q}{\text{inbound}_q(t) \cdot \sum_{q \in Q} p_q}, \quad (9)$$

$$Q = \{0, 1, \dots, \#Q - 1\}$$

Al numeratore abbiamo il prodotto tra il numero di pacchetti persi dalla coda q nel timestep t e la sua priorità p_q , normalizzato sul numero di pacchetti arrivati alla coda in quel time step moltiplicato per la somma delle priorità.

In un modo analogo per scrivere [Equation 8](#) possiamo riscrivere [Equation 9](#) come:

$$r_d(t) = - \sum_{q \in Q} \frac{\text{drop}_q(t) \cdot (q + 1)}{\text{inbound}_q(t) \cdot \frac{1}{2} \#Q (\#Q + 1)}, \quad (10)$$

$$Q = \{0, 1, \dots, \#Q - 1\}$$

Con i termini della reward così definiti, è possibile dimostrare che $-1 \leq r_i \leq 0 \forall i \in T$, e dunque tutti i termini della reward normalizzati sono confrontabili.

B. Agente

L'agente preso utilizzato per il progetto è un DQN (Deep Q-Network). Il DQN è un algoritmo di reinforcement learning che apprende una Q function utilizzando una rete neurale profonda. L'agente utilizza la Q function per determinare una policy tramite la quale può scegliere l'azione da intraprendere in base allo stato attuale. L'addestramento dell'agente avviene tramite l'esperienza accumulata durante l'interazione con l'ambiente. L'agente memorizza l'esperienza in un replay buffer e addestra la rete neurale utilizzando un algoritmo di ottimizzazione come Stochastic Gradient Descent (SGD). L'agente utilizza una politica di esplorazione ϵ -greedy per esplorare lo spazio delle azioni. L'agente non è stato implementato da zero, ma è stato utilizzato un'implementazione già esistente fornita dalla libreria TF-Agents¹. Tuttavia sono state adattate diverse caratteristiche peculiari.

¹<https://www.tensorflow.org/agents>

1) *Rete neurale*: La rete neurale utilizzata fa uso di soli layer densi, tutti con la stessa funzione di attivazione sigmoidale. In particolare la rete è composta da cinque layer densi, con rispettivamente 500, 250, 100, 50, 20 neuroni, più un ultimo layer che ha un numero di neuroni pari al numero di azioni possibili. In questo modo la rete neurale approssima la funzione Q per ogni azione. Inoltre prima di essere passato alla rete neurale, lo stato viene preprocessato tramite un layer flatter che permette di appiattire il tensore dello stato in un vettore unidimensionale.

2) *Replay buffer*: Come replay buffer è stato usato quello messo a disposizione dalla libreria TF-Agents: `TFUniformReplayBuffer`. Questo buffer è un semplice buffer circolare che memorizza l'esperienza in un vettore di tuple. Una volta che il buffer è pieno, l'esperienza più vecchia viene sovrascritta. L'agente addestra la rete neurale utilizzando un minibatch di esperienze prelevate casualmente in modo uniforme dal replay buffer. Questo metodo di addestramento permette di decorrelare le esperienze e di evitare che l'agente si adatti troppo velocemente a uno scenario specifico. Nel buffer è possibile specificare vari parametri di configurazione:

- Max length: la lunghezza massima del buffer. Nel nostro caso è stato scelto 1000;
- Train frequency: la frequenza di step con cui l'agente addestra la rete neurale. Nel nostro caso è stato scelto 100 step, perché sperimentando abbiamo notato che il comportamento dell'agente non cambiava significativamente addestrandolo più frequentemente, mentre invece migliorava molto la velocità di esecuzione.
- Sample batch size: la dimensione del minibatch da prelevare dal replay buffer. Nel nostro caso è stato scelto 100 come la train frequency in modo da prendere in media lo stesso numero di esperienze inserite nel buffer ed evitare così di perderci troppe esperienze.

Tuttavia in seguito alla lettura del paper "*The importance of experience replay database composition in deep reinforcement learning*" [1] abbiamo deciso di fare dei cambiamenti all'implementazione del buffer. Il paper in questione mostra che per avere una buon stabilità nel processo di learning è necessario avere un replay buffer composto da esperienze abbastanza diversificate. Nella nostra situazione accadeva spesso che l'agente non imparasse abbastanza in fretta e che quindi lo stato tendesse a convergere ad una situazione molto specifica (e.g esempio le code piene). Questo faceva sì che replay buffer si andasse a riempire di esperienze molto simili tra loro, bloccando l'apprendimento dell'agente; l'esplorazione non è risultata sufficiente ad uscire da questa situazione in quanto essa componeva solo una minima parte del buffer. Per risolvere questo problema ed usare i criteri di diversificazione proposti dal paper, abbiamo deciso di dividere il replay buffer in due parti, una contenente solo esperienze di esplorazione ed una contenente solo esperienze di exploitation. In questo modo è possibile addestrare l'agente con esperienze diverse senza compromettere la quantità di esplorazione e facendo in modo che esso riesca sempre a vedere esperienze abbastanza

variegate anche nel caso il buffer di exploitation contenga solo esperienze molto simili tra loro. Inoltre questo ha anche l'effetto di aumentare la memoria dell'agente rispetto alle esplorazioni fatte in passato, in modo da non dimenticare le esperienze di esplorazioni passate e rimanere bloccato su un comportamento specifico.

3) *Politica di esplorazione*: Come politica di esplorazione abbiamo usato una politica ϵ -greedy. Questa politica permette all'agente di esplorare lo spazio delle azioni con una probabilità ϵ e di sfruttare la conoscenza acquisita con una probabilità $1 - \epsilon$.

4) *Optimizer*: Per l'ottimizzazione della rete neurale è stato utilizzato Stochastic Gradient Descent (SGD) con un learning rate di 0.0001. Questo ottimizzatore è stato scelto perché è uno degli ottimizzatori più semplici e più utilizzati per l'addestramento di reti neurali. Inoltre, è stato scelto un learning rate molto basso per evitare che l'addestramento della rete neurale avvenga troppo velocemente e che l'agente non sia in grado di generalizzare. La funzione da minimizzare è la squared loss, che è la funzione di loss più utilizzata per l'addestramento di reti neurali.

5) *Gamma*: Il parametro gamma è stato impostato a 0.99. Questo parametro è importante perché definisce il peso che viene dato alle reward future rispetto alle reward attuali. Un valore di gamma vicino a 1 indica che l'agente darà molto peso alle reward future, mentre un valore di gamma vicino a 0 indica che l'agente darà molto peso alle reward attuali.

6) *Gestione delle azioni illegali*: Nel modello di azioni definito, esiste solo un tipo di azione illegale, cioè inviare un messaggio da una coda vuota.

Esistono due modi per gestire le azioni illegali:

- Specificare nel caso che lo stato della coda sia vuota un sottoinsieme delle azioni tra cui scelgire contenente solo quelle legali.
- Penalizzare con una reward negativa più bassa possibile l'azione illegale.

Nel secondo caso l'agente deve imparare oltre a al problema principale anche quali azioni sono illegali. Invece nel primo caso questo non succede, perché ciò è già noto all'agente. Sembra chiaro che la prima soluzione sia migliore, ma la libreria utilizzata non permette di implementare questo comportamento in maniera semplice. Perciò abbiamo scelto di utilizzare la seconda soluzione.

7) *AgentFacade*: L'Agent Façade rappresenta un'interfaccia che espone i metodi necessari per l'interazione con l'agente. Questa soluzione consente di nascondere tutte le interazioni tra i sottosistemi e l'agente, offrendo un'interfaccia unificata per la comunicazione con l'agente stesso. Nella nostra implementazione, non è l'agente a restituire direttamente l'azione da intraprendere; piuttosto, è il simulatore (o, in uno scenario reale, il nodo) che richiede, dato un determinato stato, quale azione intraprendere in quel momento. Ciò implica che il simulatore scandisce il tempo. In questo modo, l'agente non è consapevole del tempo trascorso e può facilmente adattarsi a uno scenario reale.

8) *AgentFacade - Configurazione:* Per utilizzare l'AgentFacade è necessario configurare l'agente. La configurazione può essere fatta tramite un file in cui è possibile specificare varie caratteristiche dell'agente in modo da poterlo adattare a diversi scenari. La documentazione per la configurazione è disponibile qui². Un altro parametro di configurazione da specificare all'AgentFacade è il numero di code presenti nel nodo. Questo parametro è necessario per l'agente in modo da poter generare azioni valide.

9) *AgentFacade - Rappresentazione:* Per rappresentare lo stato è stato fatto uso di un tensore unidimensionale, in cui ogni componente rappresenta una variabile dello stato. Questo vuol dire che variabili di stato multidimensionali, come ad esempio le occupancies delle code, sono state appiattite in un vettore unidimensionale, e poi concatenate alle altre variabili in modo da produrre un unico tensore.

L'azione che il nodo può intraprendere è composta da valori di diverse componenti, ognuna delle quali è rappresentata da un valore scalare intero. Come è spiegato meglio nella sezione delle Decisions (section II-B12), ogni componente dell'azione è chiamata Decision, e la combinazione di Decisions di diversi agenti produce l'azione da intraprendere.

10) *AgentFacade - Interazione:* Il metodo principale per interagire con l'Agent Façade è `get_action`, che restituisce l'azione da intraprendere in base allo stato attuale. Questo metodo prende in input lo stato attuale e la reward associata all'azione precedente, e restituisce l'azione da intraprendere. All'interno di questo metodo, l'Agent Façade addestra l'agente con l'esperienza passata e interroga l'agente per ottenere l'azione da intraprendere utilizzando l'albero delle Decisions (vedi section II-B12). L'addestramento non avviene ad ogni chiamata, ma secondo un intervallo di steps configurabile. Questo intervallo è necessario per evitare che l'agente si adatti troppo velocemente a uno scenario specifico e non sia in grado di generalizzare, oltre a migliorare le prestazioni computazionali.

11) *AgentFactory:* L'Agent Factory è una classe che configura e crea diversi agenti disponibili nella libreria. Nel nostro caso, abbiamo utilizzato il DQN (Deep Q-Network) ed un RandomAgent come benchmark. Inoltre, è stato implementato un decoratore per gli agenti, in modo da poterli integrare dinamicamente con i replay buffer forniti dalla libreria. Utilizzando il metodo `create_agent`, è possibile creare un'istanza di un agente specificando il tipo di agente e i parametri necessari per la configurazione. Questa operazione viene eseguita all'interno dell'AgentFacade.

12) *Decisions:* Lo spazio di decisioni in cui un agente può trovarsi ad operare può estendersi attraverso diverse dimensioni, ognuna delle quali rappresenta un parametro che costituisce tale azione. Ad esempio, nel caso del problema in esame un'azione dell'agente è rappresentabile da un vettore a tre dimensioni così codificato:

- Azione[0] = invia o non fare nulla;

²<https://github.com/retarded-reward/collaborative-learning/wiki/Agent-Configuration>

- Azione[1] = preleva pacchetto da coda i, con $i \in [0, num_queues - 1]$;
- Azione[2] = seleziona power source j, $j \in [0, num_power_sources - 1]$.

Si noti che non tutte le combinazioni di valori di questi parametri rappresentano dei punti legali all'interno dello spazio delle azioni. Ad esempio, se la prima componente è pari a 0 (non fare nulla), allora i valori delle altre componenti devono essere don't care.

Una soluzione può essere ricorrere a un'enumerazione, ovvero si "schiaffia" lo spazio delle azioni lungo un'unica dimensione i cui valori sono rappresentati dalle sole combinazioni che rappresentano azioni legali. È una soluzione semplice e immediata, tuttavia comporta alcuni problemi:

- La dimensione dello spazio delle azioni aumenta molto velocemente rispetto all'aumentare delle opzioni disponibili. Nel caso del problema, avere in generale 10 code e 2 power sources comporta 20 possibili azioni diverse. Aggiungere una sola power source aggiunge altre 10 azioni disponibili all'agente.
- Come conseguenza del primo problema, l'apprendimento dell'agente può rallentare.

Le decisions sono una possibile soluzione che combina i vantaggi dell'enumerazione cercando di limitarne gli svantaggi. Il concetto chiave consiste nel delegare la scelta del valore di ogni componente di un'azione a un agente diverso. In questa maniera, ogni agente è responsabile di prendere una decisione, e l'insieme di decisioni compone l'azione da intraprendere.

Talvolta può accadere che alcune decisioni vadano intraprese solo se le decisioni prese precedentemente lo consentono. Ad esempio, se una decisione ha dato come esito "non fare nulla", non ha senso che si interroghino gli agenti responsabili della decisione su quale coda e quale power source usare. Per coprire questo aspetto, gli agenti che prendono le decisioni sono organizzati in un albero, detto appunto albero delle decisioni. Il nodo root contiene un riferimento all'agente che prende la decisione iniziale, che dovrebbe essere di alto livello. A questo punto possono verificarsi due casi:

- Il nodo root è una foglia: Nell'albero esiste solo il nodo root. questo è il caso banale in cui l'albero di decisione regredisce a un singolo agente. La decisione corrisponde al valore ritornato dall'agente, che dunque è anche l'azione da intraprendere.
- Il nodo root ha figli: La root dispone di più scelte per la decisione. Ad ogni scelta corrisponde un nodo figlio, a cui verrà delegata la successiva decisione. Il nodo root interroga l'agente associato alla scelta selezionata dal suo agente, e così via fino a raggiungere una foglia.

La sequenza di decisioni intraprese dagli agenti rappresenta un cammino dalla root a una foglia dell'albero delle decisioni, ed è chiamato decision path. Ogni nodo interrogato registra la sua decisione all'interno del decision path, associando alla propria posizione nel path il nome della sua decisione e il suo valore. Un esempio è visibile in Figure 1.

Ogni nodo dell'albero delle decisioni è un Consultant. Un Consultant si occupa delle seguenti responsabilità:

- Mantiene un riferimento all'agente che prende la decisione;
- Mantiene una lista di consultant figli, ognuno dei quali corrisponde a una possibile scelta che il suo agente può prendere;
- Espone un metodo `get_decisions()` (Figure 3), che prende in input lo stato e un decision path (possibilmente vuoto) e aggiunge il suo contributo al decision path interrogando il suo agente;
- Espone un metodo `train()` (Figure 4), che prende in input un'Esperienza, ovvero una tripla decision path, stato, reward e addestra il suo agente e quello dei nodi figli che fanno parte del decision path.

Lo stato e l'esperienza presi in input da un Consultant sono a loro volta propagati ai consultant figli che fanno parte del decision path. Inoltre, ogni Consultant può raffinare lo stato e l'esperienza presi in input dal padre se vengono specificate delle implementazioni per i metodi `deduce_consultant_state()` e `deduce_consultant_experience()`. Tuttavia, tali deduzioni non sono propagate ai figli per impedire perdite di informazioni.

Le caratteristiche delle decisions descritte finora comportano le seguenti implicazioni:

- Tutti gli agenti dei Consultant di uno stesso decision path condividono lo stesso stato osservato dalla root, oppure ne osservano una deduzione. Per esempio, un consultant potrebbe scartare alcuni parametri dello stato poiché non rilevanti per la decisione che deve prendere;
- Tutti gli agenti dei Consultant di uno stesso decision path condividono la stessa esperienza della root al momento del training, oppure ne osservano una deduzione. Ad esempio, un consultant potrebbe usare come reward un valore derivato da quello osservato dalla root;
- Agenti che non fanno parte di un decision path non sono interrogati al momento della richiesta delle decisioni per quel path e non partecipano al training corrispondente. Questo vuol dire che tali agenti osserveranno solo gli stati che implicano una loro partecipazione nel decision path e rewards che sono frutto di essa;
- L'addestramento dei consultant di un decision path può essere fatto in parallelo;
- Consultant in profondità nell'albero delle decisioni potrebbero ricevere molte meno esperienze, a seconda che i consultant dei livelli superiori abbiano preso decisioni che portino a un loro coinvolgimento oppure no. Sebbene ciò possa sembrare che ne rallenti l'apprendimento, bisogna considerare che tali consultant dispongono di uno spazio delle azioni molto più ridotto rispetto a quello di un singolo, monolitico agente che deve orientarsi in uno spazio delle azioni che enumera tutte le possibili combinazioni legali di valori delle componenti delle azioni. Inoltre, se non ricevono esperienze vuol dire che non

vengono coinvolti così spesso nella costruzione di un decision path. In sostanza, un consultant in profondità nell'albero delle decisioni riceve meno esperienze, ma ne necessita di meno affinché l'apprendimento converga;

- Agenti di consultants in cima all'albero delle decisions potrebbero essere portati a ritenere alcuni dei propri rami come sconvenienti a causa di scelte sfortunate dei consultant più in profondità, e potrebbero avere difficoltà a scoprire che effettuare la stessa scelta potrebbe portare a un esito migliore in futuro se i consultant in profondità scelgono differentemente.

In sostanza, l'obbiettivo del framework delle decisions è quello di poter costruire un albero delle decisioni dalla cui collaborazione dei consultant emerge un comportamento equivalente a quello di un agente monolitico basato su un'enumerazione dello spazio delle azioni, senza dover esserne vincolati dagli svantaggi. Da questa idea viene il nome di collaborative learning.

C. Simulazione

Per la simulazione è stato utilizzato il framework OMNeT++³, il quale offre degli strumenti per definire e simulare una rete di nodi. In particolare, il framework è stato utilizzato per simulare un nodo router che riceve traffico da più nodi sorgente e prende decisioni sulla base dell'agente di reinforcement learning montato sul nodo. Dato che OMNeT++ non supporta nativamente il linguaggio Python tramite il quale è stato definito l'agente, per integrare simulatore ed agente è stata utilizzata la libreria PyBind⁴ che permette di esporre e fare binding tra simboli definiti in Python e simboli definiti C++.

1) *Nodo*: Un nodo corrisponde a un elemento della rete capace di ricevere pacchetti e di inoltrarli al resto dei nodi nella rete a seconda delle decisioni intraprese dal suo agente. È suddiviso nei seguenti componenti (esempio in Figure 8):

- Controller: È il componente che implementa la logica applicativa del nodo. Tra le sue responsabilità figurano l'interrogazione dell'agente e la conseguente attuazione dell'azione suggerita, la gestione delle code e delle power sources, il calcolo della reward;
- AgentClient: È il componente attraverso il quale il Controller e l'agente comunicano;
- Queues: Una o più code che accodano i pacchetti in arrivo e li consegnano al Controller dietro una sua richiesta. Ogni coda ha una capacità limitata e una priorità.

2) *Nodo - Controller*: Il Controller è il componente del nodo necessario affinché l'agente possa interagire con l'ambiente simulato. Esso si preoccupa di:

- Campionare lo stato visibile dal nodo;
- Calcolare la reward frutto dell'ultima azione effettuata;
- Interrogare l'agente per ottenere l'azione da intraprendere fornendogli l'ultimo campione dello stato e la reward calcolata dall'ultima azione;

³<https://omnetpp.org/>

⁴<https://pybind11.readthedocs.io/en/stable/>

- Interpretare l'azione restituita dall'agente e attuarla;

Queste mansioni sono svolte ciclicamente nell'ordine in cui sono presentate a intervalli scanditi da un Timeout dedicato denominato `ask_action_timeout`. La durata di tale intervallo definisce la durata di un timestep dal punto di vista dell'agente.

3) *Nodo - Controller - Action Execution:* Il controller utilizza l'Agent Client per interfacciarsi con l'agente, fornendogli un campione dello stato della simulazione e la reward dell'ultima azione intrapresa per poterlo interrogare sull'azione da intraprendere. Una volta ricevuta la risposta, il controller interpreta il da farsi (Figure 2):

- Se l'azione è non fare nulla, il controller procede direttamente al calcolo della reward;
- Se l'azione è inviare un pacchetto, il controller procede a prelevare dalla coda specificata dall'agente un pacchetto inviando una richiesta al componente Queue corrispondente. Una volta recuperato il pacchetto, il controller ne simula l'invio calcolando il consumo energetico in funzione del power model in uso, della dimensione del pacchetto e della banda disponibile. Il controller soddisfa tale consumo energetico attingendo alla power source specificata dall'agente, e ne registra l'uso per il calcolo della reward.

Da notare che se l'agente seleziona la batteria come power source e la carica di cui dispone non fosse sufficiente a soddisfare la domanda energetica, il controller prosciuga la batteria e soddisfa la rimanente domanda usando il Power Chord.

4) *Nodo - Controller - Power Model:* Il Controller si occupa anche di gestire le Power Sources da cui il Nodo può attingere l'energia necessaria per svolgere le sue operazioni, in particolare l'operazione di invio dei pacchetti. Ogni power source implementa un'interfaccia `PowerSource` e quelle disponibili per le operazioni del nodo sono registrate nel vettore `power_sources`. Le power sources registrate in tale vettore al momento sono:

- Una `Battery`: Una power source dotata di una capacità ricaricabile, il cui uso costa zero;
- Un `PowerChord`: Una power source che non si scarica mai, ma che ha un costo di utilizzo diverso da zero.

Tali PowerSources sono istanziate secondo i parametri definiti dai rispettivi modelli indicati nel file di configurazione `omnetpp.ini`, che specificano i valori per la capacità energetica e il costo in reward per unità di energia.

Per simulare la ricarica della batteria è stata definita una nuova `PowerSource` chiamata `RandomCharger`, la cui peculiarità è quella di erogare una percentuale aleatoria dell'energia richiesta secondo una distribuzione specificata alla costruzione. L'introduzione dell'aleatorietà dovrebbe simulare l'influenza di fattori ambientali nella ricarica della batteria, ad esempio se tale carcabatteria fosse alimentato da energia solare e quindi dipendesse dalle condizioni del meteo. Gli istanti di ricarica della batteria sono scanditi da un timer dedicato denominato `charge_battery_timeout`.

I costi energetici delle operazioni del nodo sono definiti in modelli denominati `power_models` e specificati nel file `omnetpp.ini`.

5) *Nodo - Controller - Reward Computation:* Il calcolo della reward avviene sempre subito dopo che il controller ha effettuato l'azione invocando il metodo `compute_reward()`. La reward è concepita come una somma di una serie di termini, ognuno modellato come un oggetto `RewardTerm`. Per ogni termine è possibile specificare i seguenti attributi:

- Un *segna*, ovvero un'espressione che combina i valori di una o più grandezze osservabili nella simulazione per produrre un valore di reward;
- Un *peso*, che moltiplicato per il valore del segnale, ne determina il contributo al calcolo della reward finale;

I suddetti attributi possono essere specificati nel file di configurazione `omnetpp.ini`.

Per ogni `RewardTerm` è possibile specificare un metodo di normalizzazione da applicare al valore del segnale prima di moltiplicarlo per il peso. Per farlo, è sufficiente passare alla costruzione del termine un oggetto `Normalizer`. In particolare, per implementare la reward del modello è stato usato un `MinMaxNormalizer` per ogni termine, in maniera tale che la somma pesata di tutti i termini sia compresa tra -1 e 0.

A volte può capitare che l'Agente opti per delle azioni che sono ovviamente inutili, come ad esempio decidere di inviare un messaggio da una coda vuota. Per scoraggiare l'agente da tali comportamenti, ogni qual volta un'azione del genere è suggerita dall'agente il controller invoca `illegal_action_penalty()` per assegnare la reward più negativa possibile a tale azione.

6) *Nodo - Agent Client:* Ogni nodo dispone di un agente di reinforcement learning che può essere interrogato dal controller per essere istruito su qual'è la migliore azione da intraprendere in un determinato momento. L'AgentClient è il componente che si occupa di fare da intermediario tra il controller del nodo e il suo agente. Il controller periodicamente contatta l'AgentClient con una `AgentClientRequest` per poter interrogare l'agente su quale azione intraprendere. L'AgentClient risponderà con una `AgentClientResponse` contenente una descrizione dell'azione da intraprendere, che il controller interpreterà ed eseguirà.

Nel caso di studio affrontato è stata implementata una versione dell'AgentClient chiamata `AgentClientPybind`, che sfrutta la libreria PyBind per poter invocare codice Python da C++. In particolare, le operazioni svolte da tale componente sono:

- Accogliere le richieste del controller, le quali contengono un campionamento dello stato e la reward dell'azione precedente;
- Convertire la richiesta in oggetti Bean compatibili con l'implementazione Python dell'agente;
- Addestrare l'agente con la reward ricevuta e chiedere l'azione in base allo stato, ottenendo così una `ActionBean`;

- Convertire la ActionBean in un messaggio inscrivibile in una AgentClientResponse;
- Inviare la risposta al controller.

Il flusso di esecuzione delle suddette operazioni è illustrato in [Figure 7](#).

Per poter eseguire codice python l'agent client sfrutta un'istanza di `PythonInterpreter`, ovvero una classe singleton con il compito di mantenere un riferimento a un interprete Python.

7) *Nodo - Queue*: Ogni nodo dispone di una o più code adibite a ospitare i pacchetti in arrivo. Il controller del nodo registra ogni coda in un array, e assegna ad ognuna di esse una priorità pari al suo indice in tale array. Inoltre, ogni coda ha una capacità limitata, che se raggiunta comporta la perdita di nuovi pacchetti in arrivo. Infine, ogni coda può gestire più serventi.

L'implementazione della coda è prodotta da una decorazione della classe `cQueue` del framework di OMNeT++. In particolare, sono stati applicati i decoratori:

- `FixedCapCQueue` che limita la capacità della coda a un valore desiderato. La coda dunque accetterà nuovi pacchetti solo se la sua dimensione è minore della sua capacità. Se tale condizione non sussiste, l'inserimento di nuovi pacchetti provoca il sollevamento di un'eccezione;
- `PriorityCQueue` che registra la priorità desiderata nella coda;

Il modulo della coda mantiene un riferimento a un oggetto `cQueue` chiamato `data_buffer` che viene usato come coda vera e propria per i pacchetti dati in arrivo.

Una coda può ricevere i seguenti messaggi:

- `DataMsg`: un pacchetto dati in arrivo nella coda. La coda tenterà di inserire tale pacchetto nel suo `data_buffer`. Se il pacchetto è accettato, viene accodato e annotato il tempo di arrivo, altrimenti è scartato.
- `QueueDataRequest`: Un servente vuole prelevare uno o più pacchetti dalla coda. La coda tenta di soddisfare la richiesta prelevando un numero di pacchetti dal suo buffer che è al più pari al numero di pacchetti richiesti. Tali pacchetti sono poi inviati al servente richiedente incartati in una `QueueDataResponse`.

La coda comunica informazioni relative al suo stato inviando ai suoi serventi un `QueueStateUpdate` contenente le seguenti informazioni:

- percentuale di buffer occupato della coda rispetto alla sua capacità;
- numero di pacchetti arrivato alla coda (accettati e scartati) dall'ultimo aggiornamento sullo stato della coda;
- numero di pacchetti scartati dalla coda dall'ultimo aggiornamento sullo stato della coda;

La coda emette un aggiornamento sul suo stato ogni volta riceve un pacchetto, sia che venga accettato o meno. Inoltre, la coda include un aggiornamento sul suo stato nella risposta a una richiesta di pacchetti dati da parte di un servente.

Una descrizione grafica dei comportamenti della coda discussi finora è presentata nei diagrammi in [Figure 6](#) e [Figure 5](#).

8) *SrcNode*: E' un componente che ha il compito di generare il traffico da inviare al nodo router, in particolare, ad ogni coda del nodo router è associato un `SrcNode` differente. Il traffico è composto di pacchetti di dimensione casuale che vengono inviati ad intervalli di tempo casuali. La dimensione dei pacchetti è ottenuta campionando da una distribuzione uniforme su un intervallo configurabile che rappresenta la taglia minima e la taglia massima in byte del pacchetto, mentre l'intervalllo di tempo tra l'invio di due pacchetti è ottenuto campionando da una distribuzione esponenziale di media configurabile.

9) *Network*: E' l'ambiente in cui vivono il nodo router e tutte i nodi sorgente, in particolare è qui che viene esplicitata la connessione tra i nodi sorgenti e le code del nodo router. ([Figure 9](#))

III. RISULTATI

sec:risultati) Sono stati condotti diversi esperimenti per valutare le prestazioni del nodo in base alle azioni suggerite dal suo agente. Le specifiche della macchina su cui sono stati eseguiti gli esperimenti sono elencate in [Listing 1](#). Per effettuare gli esperimenti si è partiti da una configurazione dei parametri di base⁵, a cui poi sono state apportate modifiche a seconda dello scenario considerato nell'esperimento.

Negli esperimenti andiamo a confrontare tre tipologie di agenti:

- `DQN`: Agente DQN implementato secondo le specifiche delle `Decisions`
- `DQN flat`: Agente DQN in cui le azioni sono l'enumerazione di tutte le possibili azioni (Albero delle decisions con 1 singolo nodo).
- `Random`: Agente che sceglie casualmente tra le azioni, utilizzato per avere un benchmark di confronto.

Nel primo scenario sono state valutate le prestazioni del nodo router con valori dei parametri della soluzione di default. In particolare, l'agente del nodo tiene conto di tutti i termini della reward in ugual misura, il nodo dispone di una sola coda e la frequenza di richiesta delle azioni egualgia il tasso di arrivo dei pacchetti. In [Figure 10](#) si può notare come dal punto di vista della reward cumulativa gli agenti DQN, DQN flat e Random si comportino in maniera simile fino a circa 50 secondi di simulazione, dopodiché l'agente DQN sembra imparare una politica più vantaggiosa. Il DQN con decisions sembra imparare più lentamente rispetto al DQN flat, poichè notiamo uno scostamento dalle prestazioni del Random intorno ai 110 secondi. Inoltre, andando avanti con la simulazione, le sue prestazioni sembrano peggiorare. Ciò potrebbe essere dovuto al fatto che l'agente root venga scoraggiato dall'inviare pacchetti a causa delle scelte di agenti più in profondità nell'albero delle decisions, ad esempio se l'agente responsabile della scelta della power source scelga spesso di ricorrere al power chord.

⁵<https://github.com/retarded-reward/collaborative-learning/blob/main/simulations/res/omnetpp.ini>

Il grafico in [Figure 11](#) sembra andare a supporto di questa ipotesi.

Se osserviamo il grafico in [Figure 12](#) notiamo che gli agenti DQN e DQN flat presentano un risparmio di spesa energetica più basso (6%, 8%) rispetto al Random (14%). Questo vuol dire che, per qualche ragione, gli agenti DQN sembrano inviare più spesso pacchetti dalla power chord piuttosto che dalla batteria, nonostante quest'ultima sia gratuita in termini di costo di utilizzo. Ciò potrebbe essere dovuto al fallback automatico alla power chord quando la batteria non riesce a soddisfare la domanda energetica. Si tenga presente che nello stato dell'agente non è presente alcuna informazione relativa al consumo energetico, ma solo allo stato di carica della batteria.

Il grafico in [Figure 13](#) mostra come gli agenti DQN e DQN flat riescano a gestire meglio la popolazione della coda rispetto al Random, riducendo i periodi di tempo in cui si hanno perdite di pacchetti. Inoltre, si nota anche come gli agenti DQN abbiano dei comportamenti oscillanti: in certi periodi si concentrano nell'invio di pacchetti continuo, mentre in altri non fanno niente. Ciò è più evidente in [Figure 14](#), dove sono evidenziati i periodi di tempo in cui l'agente DQN si concentra su una particolare azione. Da notare che il DQN con decision oscilla di meno rispetto alla versione flat, ricordando però che il primo accumula più reward negativa rispetto al secondo.

Nelle figure [Figure 15](#) e [Figure 16](#) possiamo vedere i comportamenti dei vari agenti a seguito del cambio dei pesi assegnati ai termini della reward. Lo scenario considerato è quello in cui il peso energetico è 0 e i pesi del packet drop e dell'occupazione della coda sono entrambi 1/2. In particolare, nella figura [Figure 15](#) si può notare come le varianti di DQN accumulino un quantitativo di reward simile rispetto a quello ottenuto ponendo i pesi di tutti i termini della reward a 1/3. Il random, invece, va molto peggio. Ciò sembra suggerire che a inficiare maggiormente sulla reward siano proprio i termini che riguardano gli aspetti delle code, e quindi che la strategia imparata dal DQN in entrambi i casi punti ad azzerare tali termini. In [Figure 16](#) si può notare come gli agenti DQN, dopo un transitorio iniziale, imparino che non conviene inviare pacchetti praticamente mai e non consumino più energia. Inoltre, il DQN con decisions sembra andare meglio rispetto alla controparte flat. Questo potrebbe essere dovuto al fatto che la root deve decidere solo tra due azioni, ovvero inviare e non fare nulla, e quindi impari più in fretta che non conviene inviare mai rispetto al DQN flat, che ha a disposizione più opzioni per la sua azione.

In figura [Figure 17](#), viene mostrato il comportamento dell'agente nel caso in cui ci siano 3 code con priorità. Possiamo vedere che gli agenti DQN hanno un comportamento peggiore del random, segno che non sono a imparare una strategia appropriata. Tuttavia, il DQN con decisions sembra soffrire di meno rispetto alla variante flat.

IV. CONCLUSIONI

- Nello sviluppo di soluzioni di reinforcement learning è di importanza fondamentale che il modello di stato, azioni e reward utilizzati siano rappresentativi del problema,

validi e verificati, poichè un minimo errore ad esempio nel calcolo della reward può indurre comportamenti nell'agente molto differenti;

- Apportare delle soluzioni che permettono di non dimenticare facilmente esperienze di exploration può aiutare l'agente a imparare una strategia ottima;
- All'aumentare del numero delle code la difficoltà di apprendimento dell'agente aumenta considerevolmente, per cui forse è opportuno considerare un modello diverso;
- L'utilità dell'uso di un albero delle decisioni sembra essere limitata rispetto a un DQN normale, anche se sembra migliorare le prestazioni del DQN con'aumentare del numero delle azioni possibili;

Tutto il codice che implementa ambiente di simulazione e agente è disponibile al seguente repository: <https://github.com/retarded-reward/collaborative-learning>

REFERENCES

- [1] Bruin, T. D., Jens Kober, and K. Tuyls. "The importance of experience replay database composition in deep reinforcement learning." Computer Science, 2015, pp. 5-8.

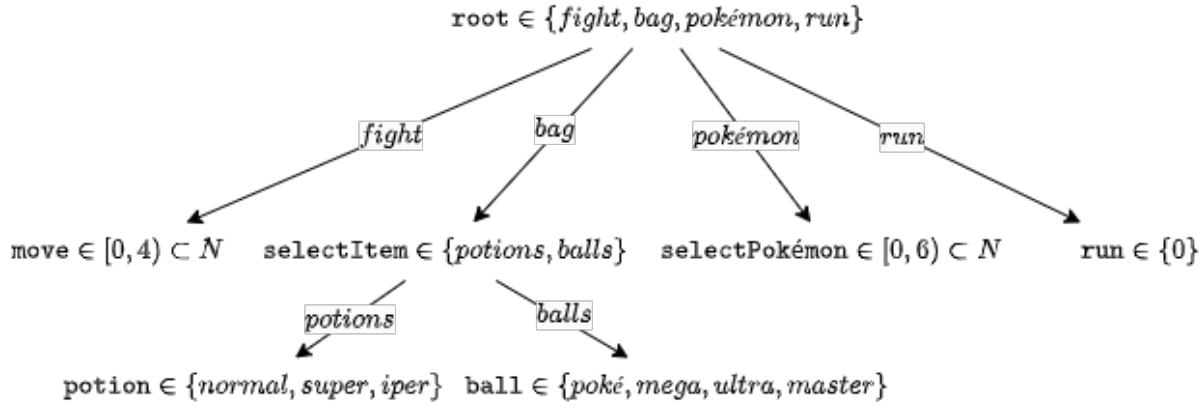


Fig. 1. Esempio di albero delle decisioni, usando come esempio il combattimento nel gioco dei Pokémon. Un esempio di decision path:
 $\{\text{root: bag, select_item: potions, potion: iper}\}$

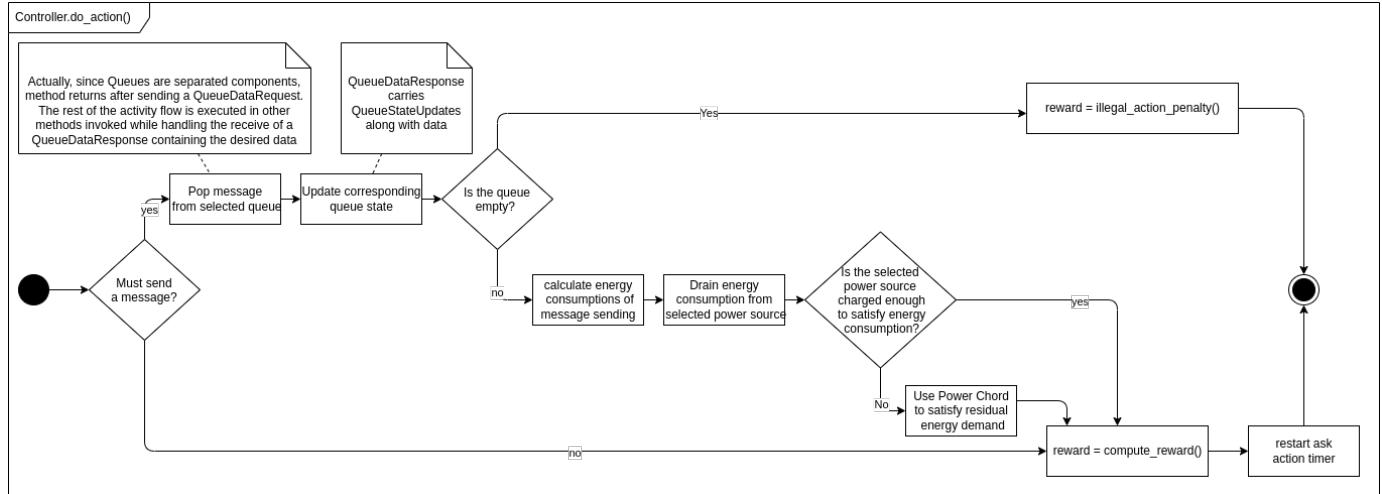


Fig. 2. `do_action()` Activity diagram. I consumi energetici sono annotati dal controller per poter essere utilizzati dal metodo `compute_reward()` in modo da poter essere combinati con i costi di utilizzo delle rispettive power sources ai fini del calcolo della reward.

```

Operating System: Fedora Linux 40
KDE Plasma Version: 6.0.5
KDE Frameworks Version: 6.2.0
Qt Version: 6.7.1
Kernel Version: 6.8.11-300.fc40.x86_64 (64-bit)
Graphics Platform: Wayland
Processors: 12 x AMD Ryzen 5 7530U with Radeon Graphics
Memory: 13.5 GiB of RAM
Graphics Processor: AMD Radeon Graphics
Manufacturer: LENOVO
Product Name: 21KK
System Version: ThinkBook 16 G6 ABP

```

Listing 1: Specifiche della macchina su cui sono stati eseguiti gli esperimenti

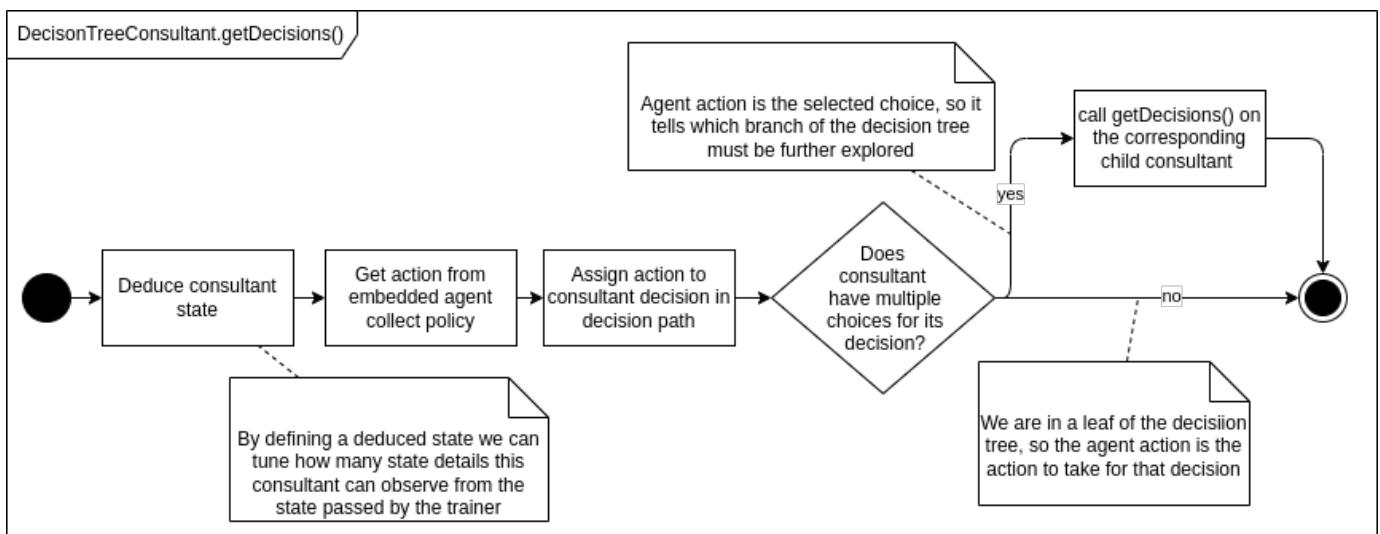


Fig. 3. activity diagram per il metodo `getDecisions()` di un Consultant

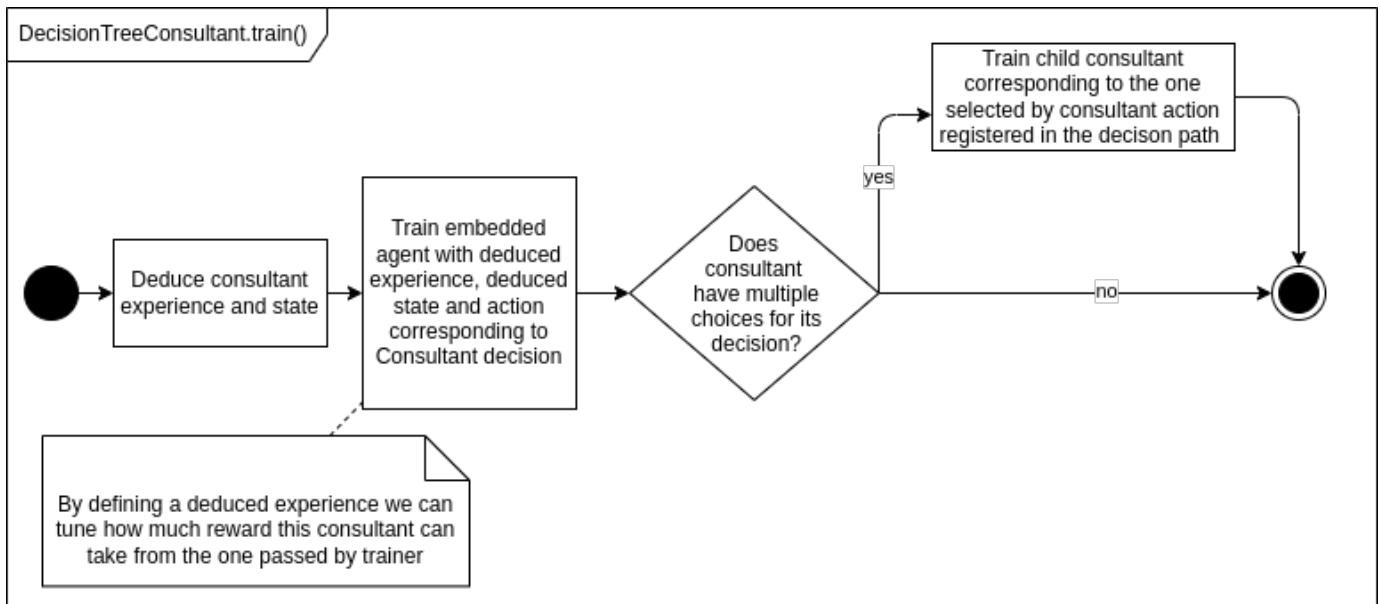


Fig. 4. activity diagram per il metodo `train()` di un Consultant

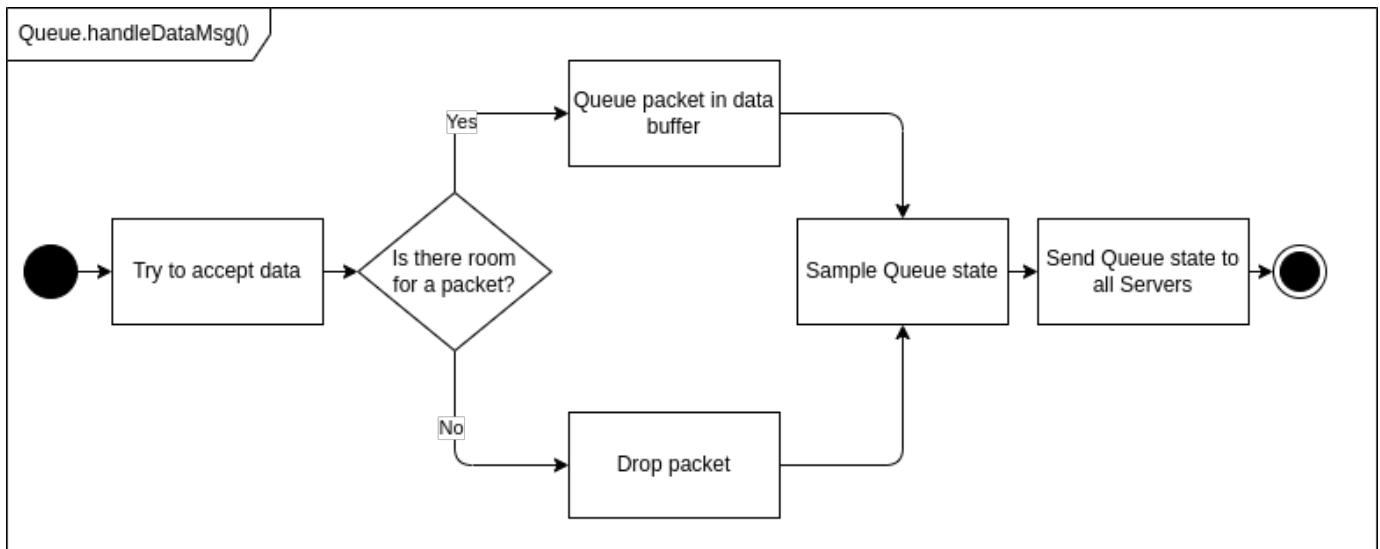


Fig. 5. activity diagram per la gestione dell'arrivo di un pacchetto nella coda

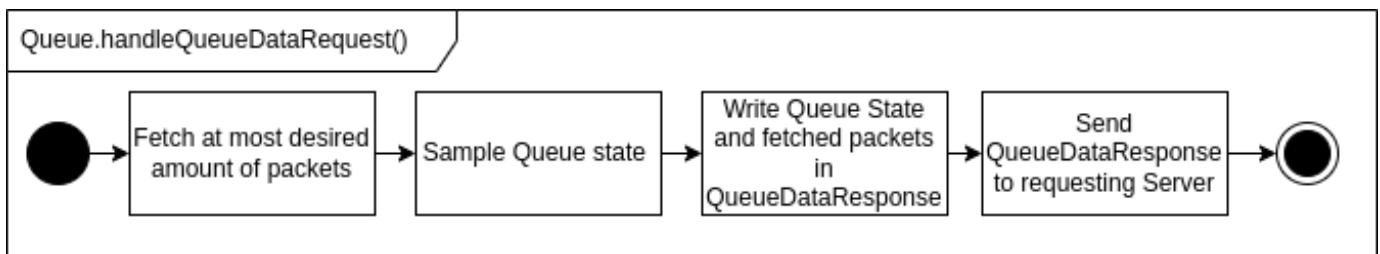


Fig. 6. activity diagram per la gestione di una richiesta di dati da parte di un servente di pacchetti nella coda

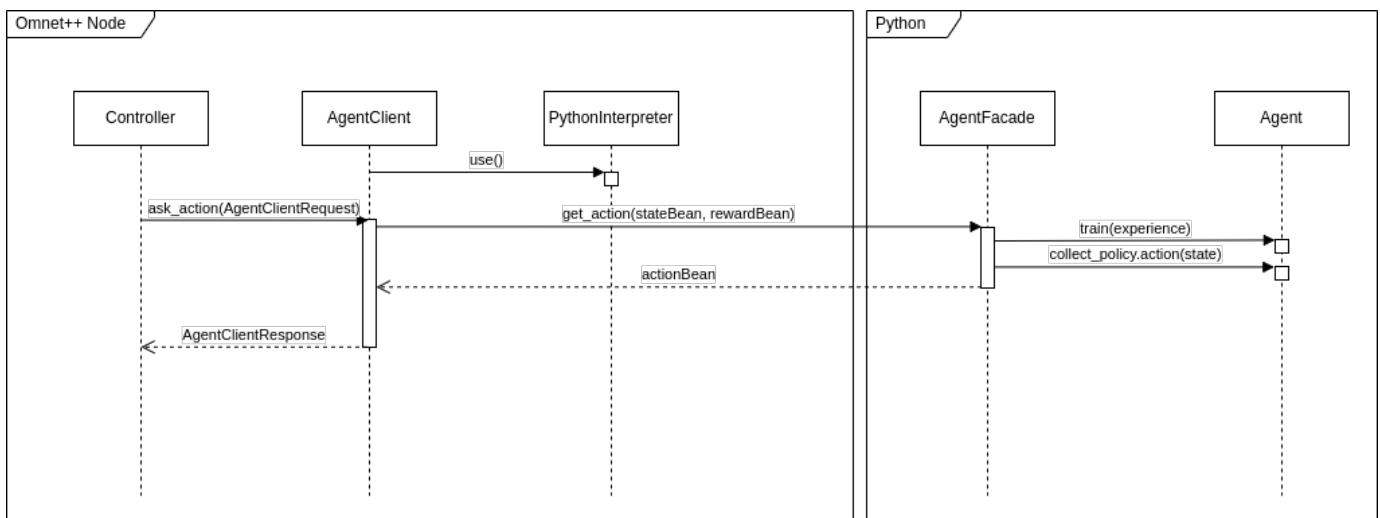


Fig. 7. sequence diagram per la servitù di una richiesta di azione del Controller nei confronti dell'agente.

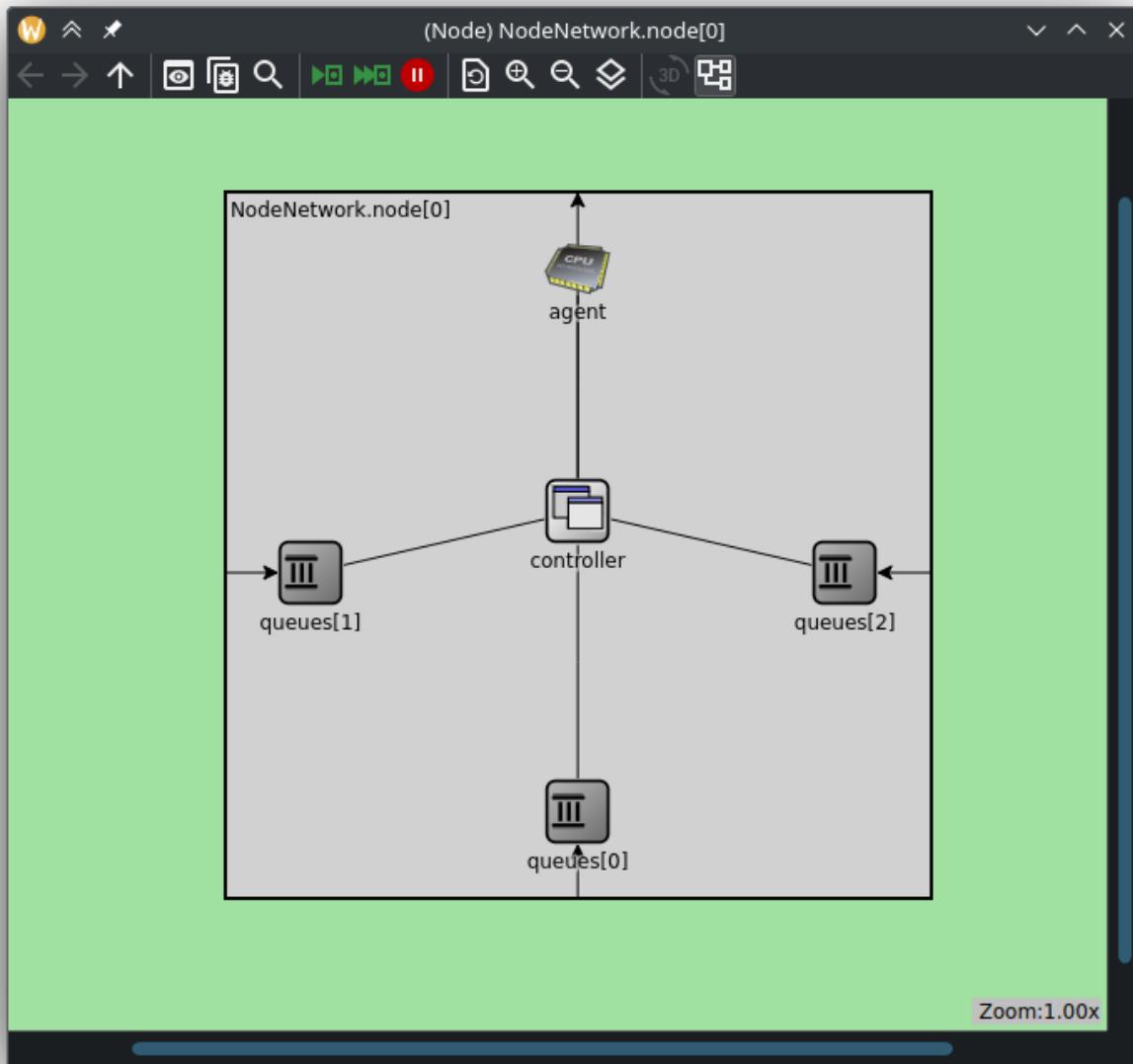


Fig. 8. Nodo con tre code

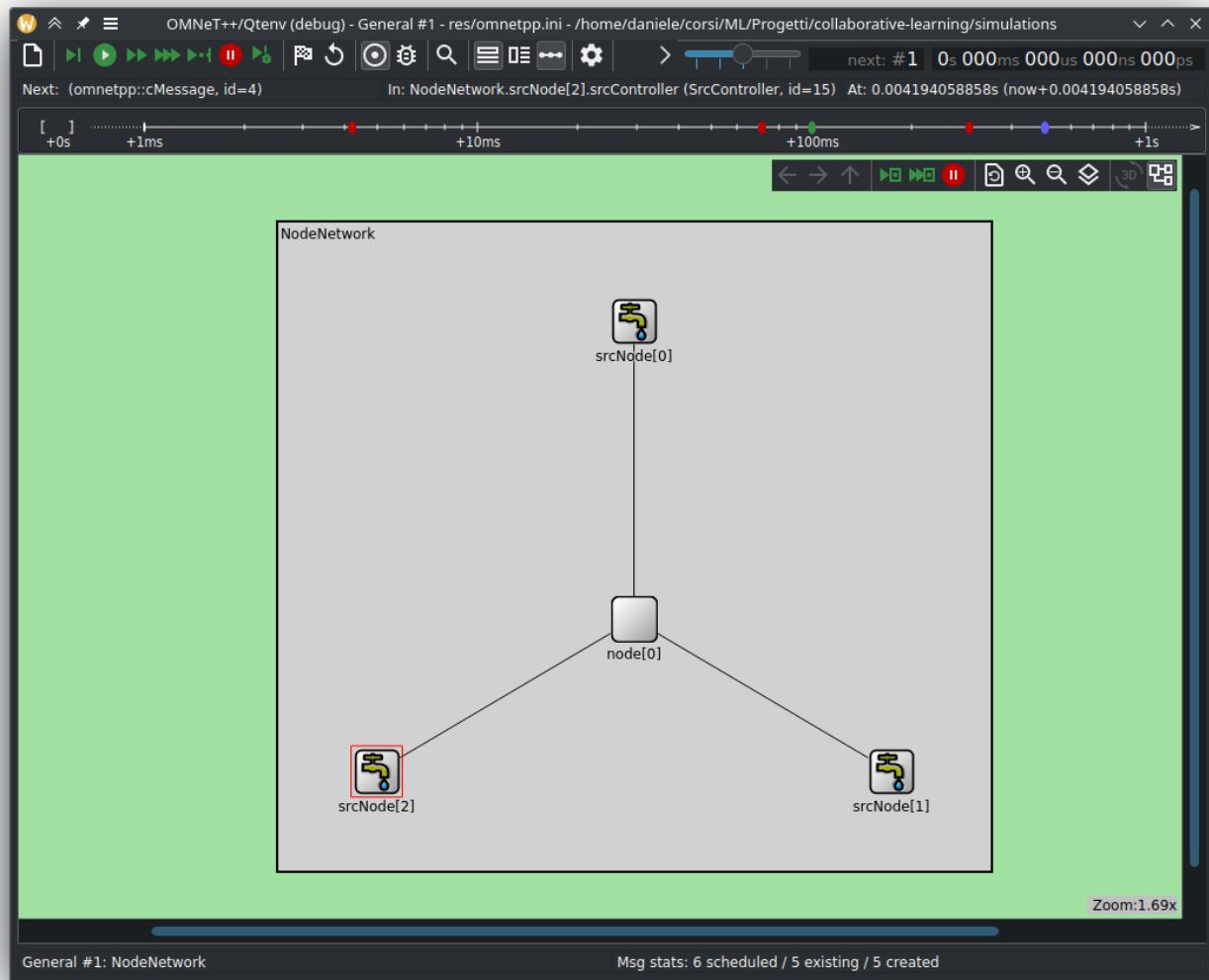


Fig. 9. Rete con tre nodi sorgente e un solo nodo con agente.

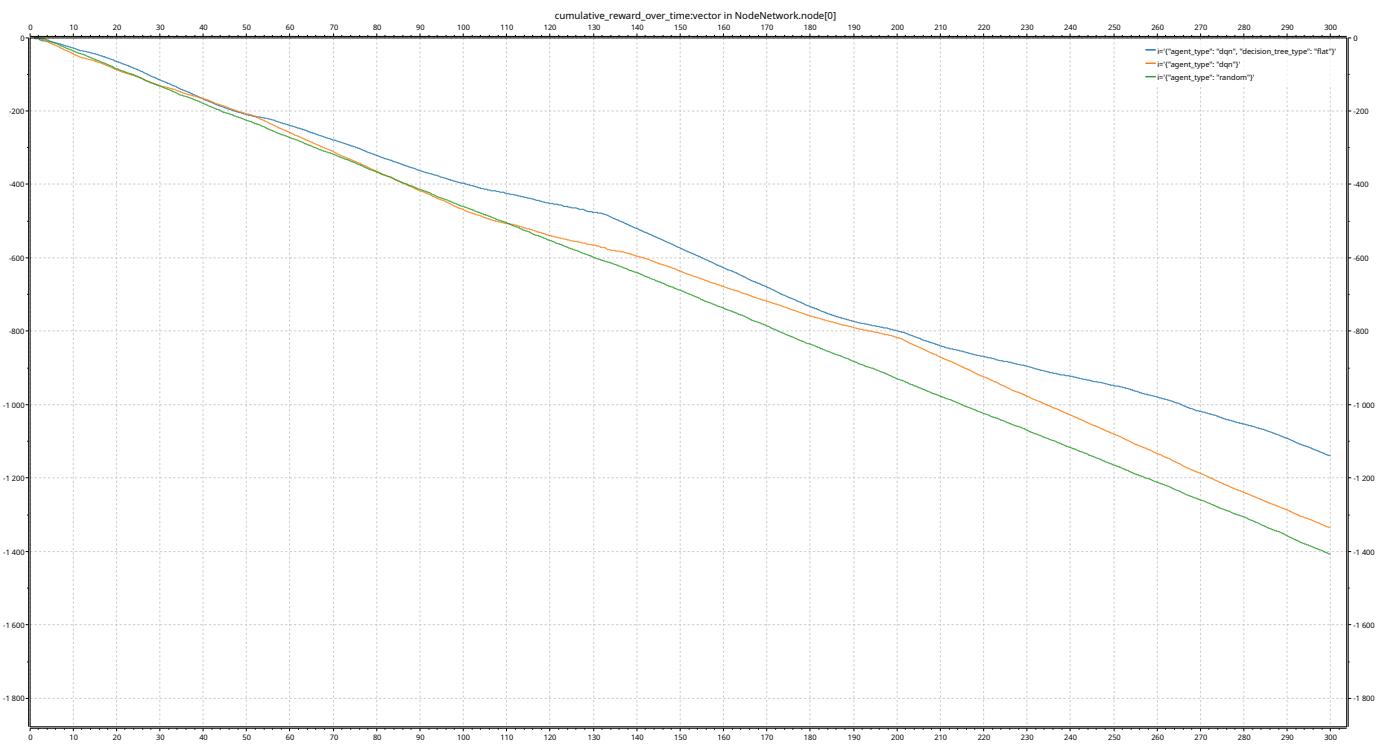


Fig. 10. Confronto cumulative reward tra agenti DQN con decision, DQN semplice ("flat") e Random.

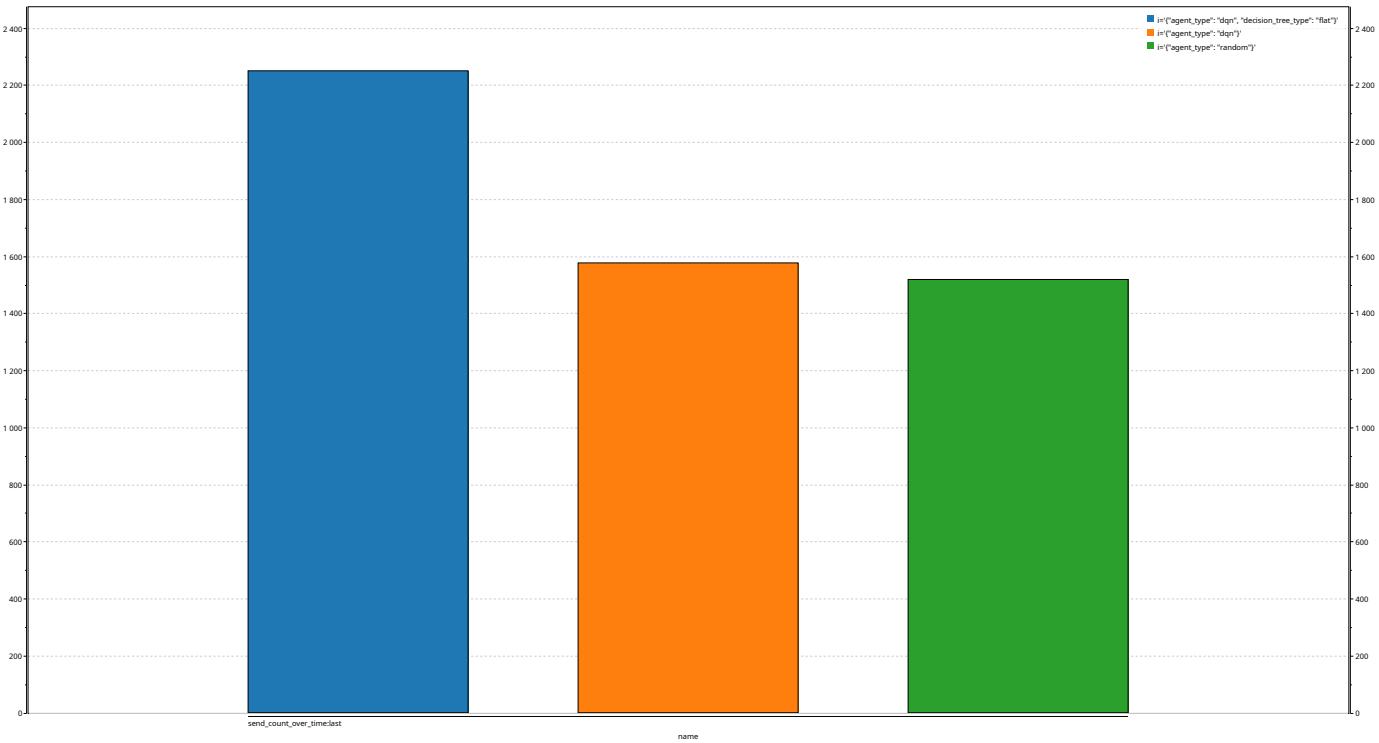


Fig. 11. Confronto numero di volte che è stata intrapresa l'azione di inviare un messaggio, a prescindere dalla power source utilizzata, tra agenti DQN con decision, DQN semplice ("flat") e Random.

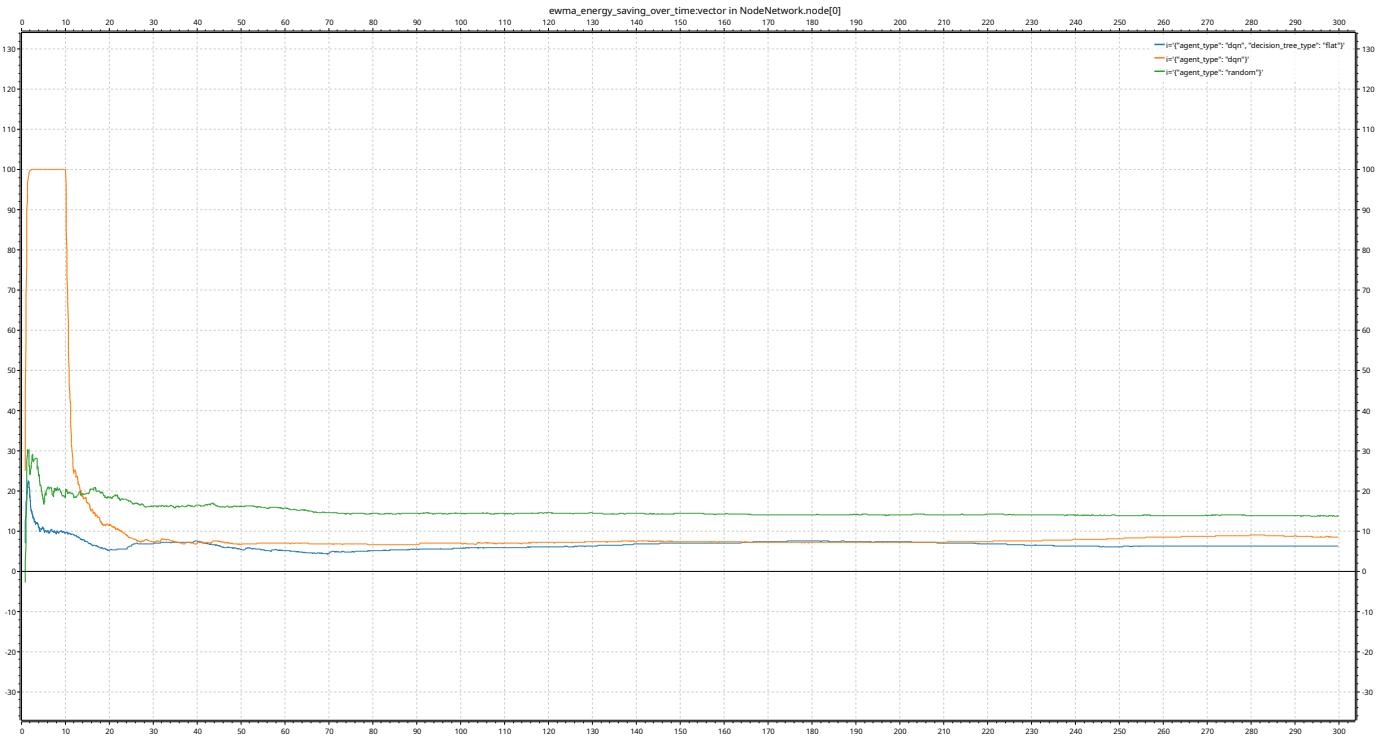


Fig. 12. Confronto media mobile pesata esponenzialmente del risparmio della spesa energetica in percentuale tra agenti DQN con decision, DQN semplice ("flat") e Random. Il risparmio è calcolato come $100 - \frac{\text{spesa energetica dell'agente}}{\text{potenziale spesa energetica}}$ se avesse usato la power source più costosa.

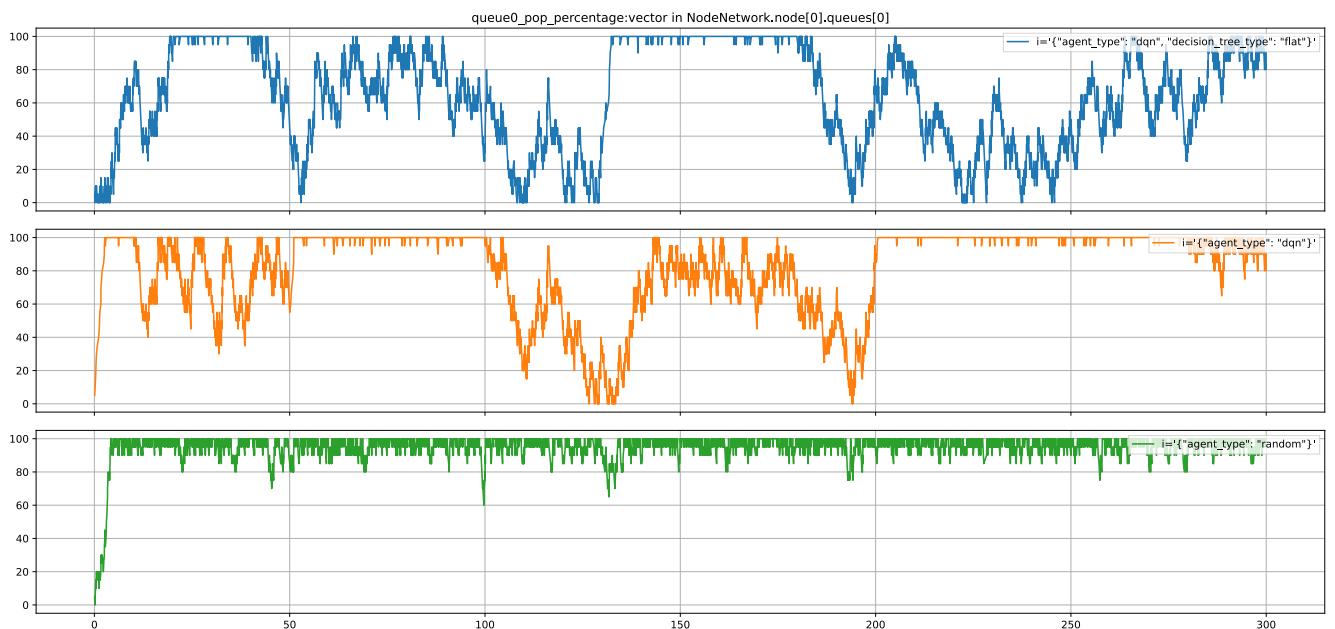


Fig. 13. Confronto delle percentuali di occupazione delle code tra agenti DQN con decision, DQN semplice ("flat") e Random.

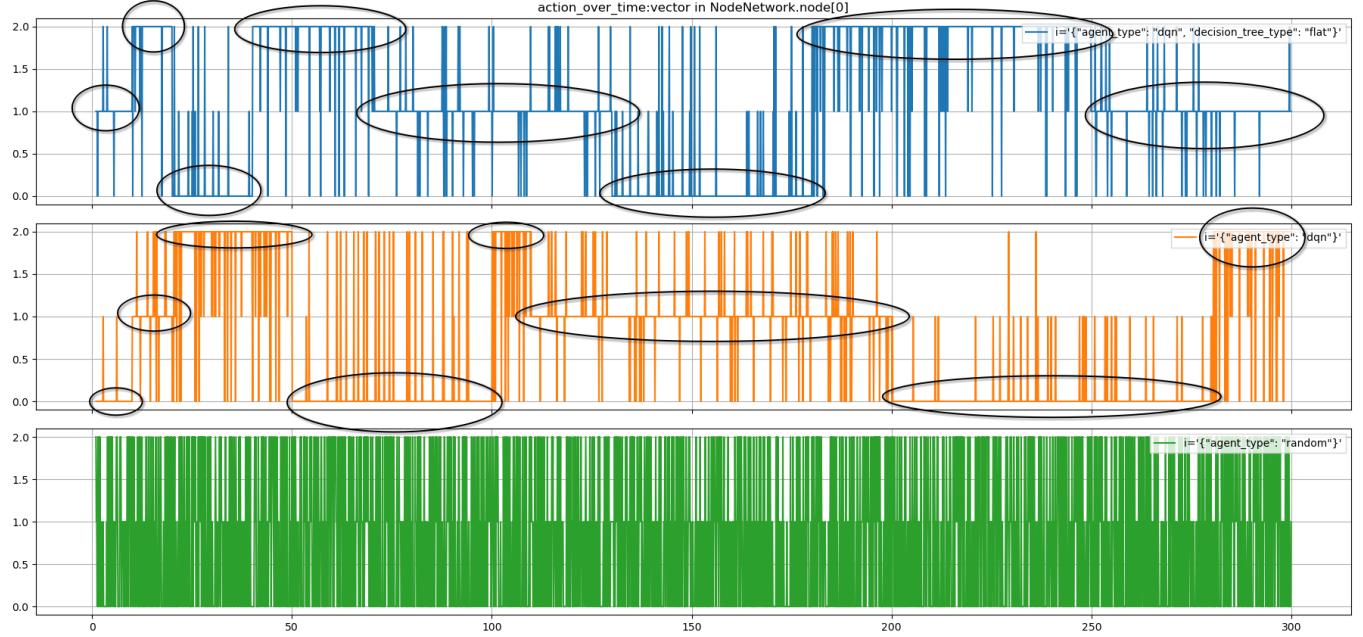


Fig. 14. Confronto delle azioni scelte per ogni istante di tempo tra agenti DQN con decision, DQN semplice ("flat") e Random. 0 corrisponde a do nothing, 1 a send message dalla batteria, 2 a send message dal power chord.

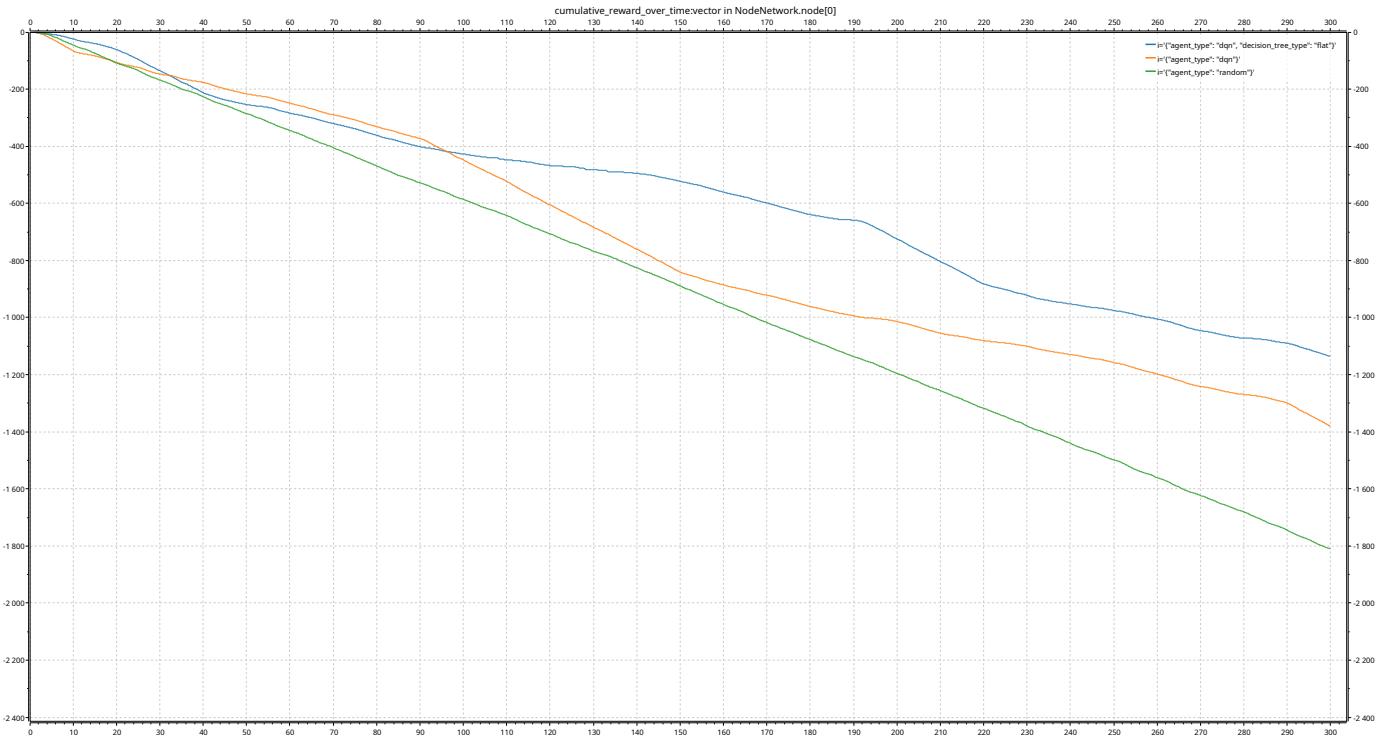


Fig. 15. Confronto cumulative reward tra agenti DQN con decision, DQN semplice ("flat") e Random nel caso in cui le priorità dell'agente sono distribuite unicamente tra il ridurre le perdite e le occupazioni delle code.

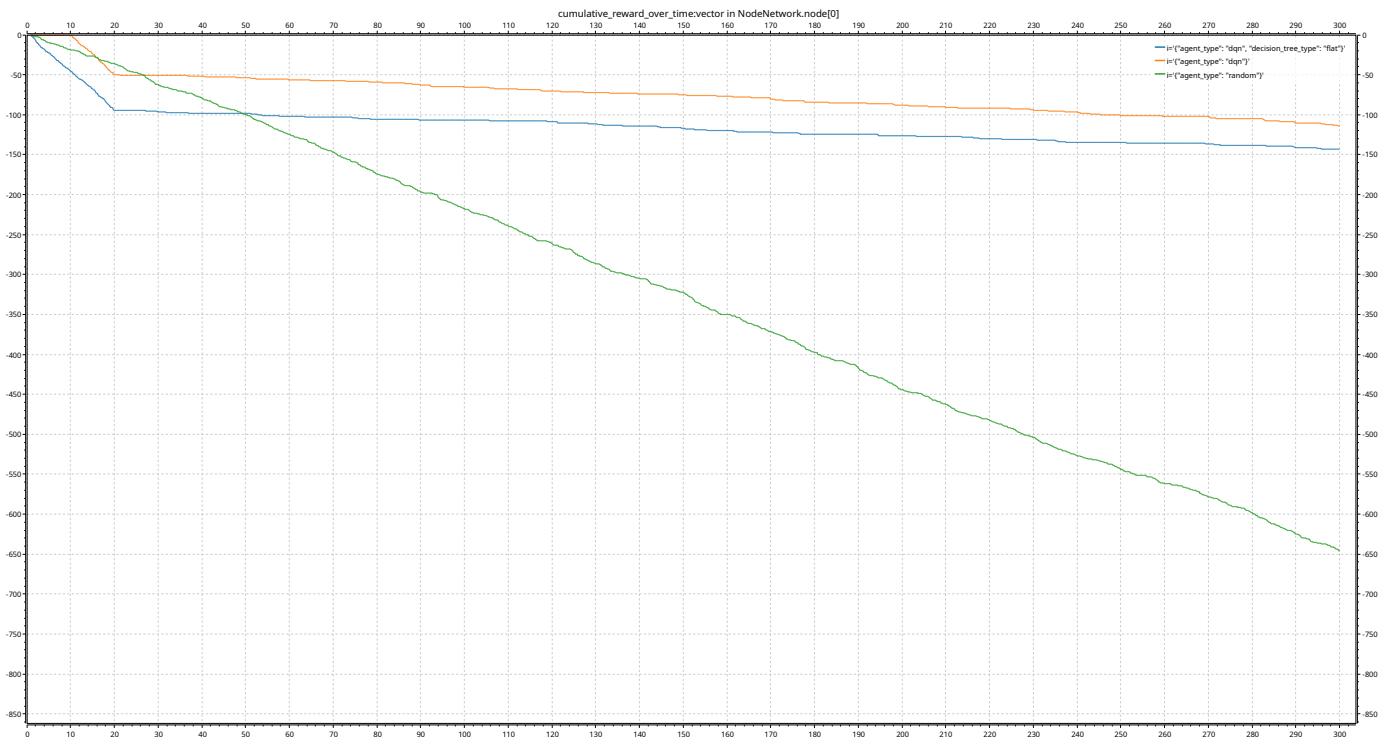


Fig. 16. Confronto cumulativo reward tra agenti DQN con decision, DQN semplice ("flat") e Random nel caso in cui l'agente si concentri solo sul risparmio energetico.

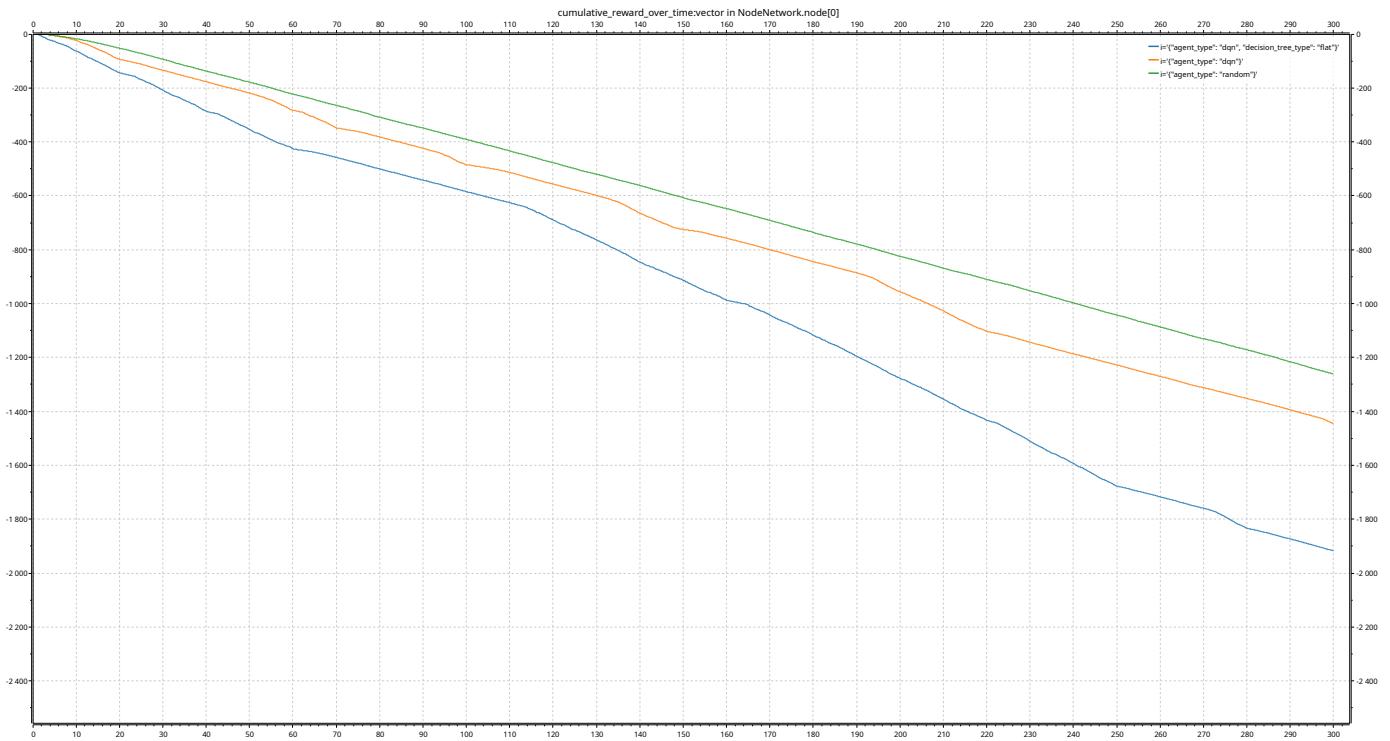


Fig. 17. Confronto cumulativo reward tra agenti DQN con decision, DQN semplice ("flat") e Random nel caso in cui l'agente abbia tre code da gestire.