

# Collaborative Learning

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

Daniele La Prova, Matteo Conti, Luca Falasca

Universita' degli Studi di Roma Tor Vergata

## Roadmap

## 1 Introduzione

- Contesto
  - Obiettivi

2 Metodología

- Modello
    - Stato
    - Azioni
    - Reward
  - Agente
    - Replay Buffer
    - AgentFacade
    - Decisions
  - Simulazione
    - Nodo
    - SrcNode

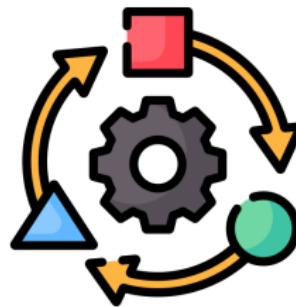
## 3 Risultati

- Coda singola, reward bilanciata
  - Coda singola, full power saving
  - Coda singola, no power saving
  - 3 Code, reward bilanciata

## 4 Conclusioni

## Contesto (1)

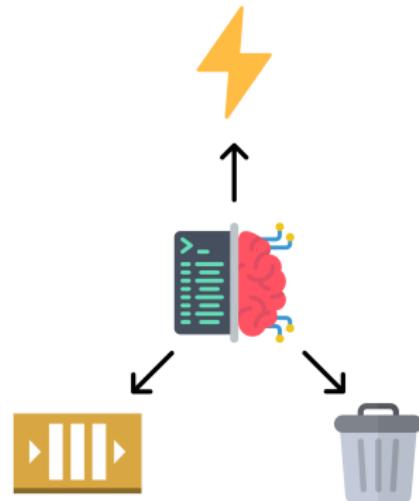
In un sistema moderno i QoS possono essere di vario genere e non sempre sono legati alle sole prestazioni, in particolare, hanno acquisito sempre più rilevanza aspetti come la sostenibilità ed il valore portato dal sistema. E' quindi necessario trovare delle soluzioni che siano flessibili rispetto a tali aspetti in modo da poter essere adattate a diversi contesti.



## Contesto (2)

Il progetto verte sulla realizzazione di un nodo di rete dotato di code di priorità e diverse fonti energetiche il cui comportamento è determinato da un agente di reinforcement learning che cerca di minimizzare una reward configurabile basata sui seguenti QoS:

- Costo energetico
  - Occupazione delle code
  - Perdita di pacchetti dalle code



## Obiettivi (1)

Le performance del nodo vengono valutate confrontando le scelte fatte dell'agente con quelle di un agente che fa scelte casuali, al variare di:

- Configurazione del modello di reward dell'agente
  - Numero di code del nodo



## Obiettivi (2)

La bontà delle scelte dell'agente viene misurata utilizzando diverse metriche, in particolare:

- Reward cumulativa per unità di tempo
- Reward per unità di tempo
- Utilizzazione del nodo 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

## Obiettivi (3)

- Costo energetico per unità energetica, definito come:

### Definition

$$\frac{\text{energy\_cost}}{\text{energy\_consumed}} \quad (1)$$

- Costo energetico per unità di tempo, definito come:

### Definition

$$\frac{\text{energy\_cost}}{\text{time\_step}} \quad (2)$$

# Obiettivi (4)

- Risparmio energetico per unità di tempo, definito come:

## Definition

$$100 \cdot \left( 1 - \frac{\text{energy\_cost}}{\text{energy\_cost\_only\_power\_chord}} \right) \quad (3)$$

# Roadmap

## 1 Introduzione

- Contesto
- Obiettivi

## 2 Metodologia

### ■ Modello

- Stato
- Azioni
- Reward

### ■ Agente

- Replay Buffer
- AgentFacade
- Decisions

### ■ Simulazione

- Nodo
- SrcNode

## 3 Risultati

- Coda singola, reward bilanciata
- Coda singola, full power saving
- Coda singola, no power saving
- 3 Code, reward bilanciata

## 4 Conclusioni

# Modello: 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.

## Modello: 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

## Modello: Reward (1)

## Definition (Reward per timestep)

$$\begin{aligned}
r(t) &= \sum_{i \in T} w_i \cdot r_i(t), \quad -1 \leq r_i \leq 0 \\
&\quad 0 \leq w_i \leq 1, \\
&\quad \sum_{i \in T} w_i = 1, \\
T &= \{e, o, d\}, \\
t &\geq 0 \in \mathbb{N}
\end{aligned} \tag{4}$$

- La reward  $r$  da inviare all'agente per un qualunque timestep  $t$  è calcolata come la somma pesata dei contributi normalizzati del termine energetico, di occupazione delle code e di drop dei pacchetti in quel timestep

# Modello: Reward (1)

## Definition (Reward per timestep)

$$\begin{aligned} r(t) &= \sum_{i \in T} w_i \cdot r_i(t), \quad -1 \leq r_i \leq 0 \\ &\qquad\qquad\qquad 0 \leq w_i \leq 1, \\ &\qquad\qquad\qquad \sum_{i \in T} w_i = 1, \\ T &= \{e, o, d\}, \\ t &\geq 0 \in \mathbb{N} \end{aligned} \tag{4}$$

- Tutti i termini della reward sono confrontabili tra loro;
- configurando opportunamente i pesi è possibile costringere l'agente a minimizzare dei termini anche a discapito di altri

## Modello: Reward (2)

### Definition (Reward Power Term)

$$r_e(t) = - \sum_{p \in P} \frac{\text{cons}_p(t) \cdot \text{cost}_p}{\max_{\tau < t} \{\text{cons}_p(\tau)\} \cdot \sum_{p \in P} \text{cost}_p}, \quad (5)$$

$$P = \{\text{battery, power chord}\}$$

- Il termine energetico per una power source è calcolato come 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.
- i costi e i consumi energetici sono tutti maggiori di zero

# Modello: Reward (3)

## Definition (Reward Queue Occupancy Term)

$$r_o(t) = - \sum_{q \in Q} \frac{\text{occ}_q(t) \cdot p_q}{\max_{\tau < t} \{\text{occ}_q(\tau)\} \cdot \sum_{q \in Q} p_q}, \quad (6)$$

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

- Il termine dell'occupazione di una coda è calcolato come 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.
- La priorità di una coda  $p_q$  è data dal suo indice di coda  $q + 1$

# Modello: Reward (4)

## Definition (Reward Packet Drop Term)

$$r_d(t) = - \sum_{q \in Q} \frac{\text{drop}_q(t) \cdot p_q}{\text{inbound}_q(t) \cdot \sum_{q \in Q} p_q}, \quad (7)$$

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

- Il termine del drop dei pacchetti di una coda è calcolato come 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.

# Agente (1)

- DQN (Deep Q-Network): Un algoritmo di reinforcement learning che utilizza reti neurali deep per apprendere una Q function.
- Q Function: Utilizzata per determinare una policy che guida le azioni dell'agente in base allo stato attuale.
- Addestramento: Basato sull'esperienza accumulata durante l'interazione con l'ambiente.

## Agente (2)

- Implementazione: Configurabile tramite la classe AgentFactory
- Replay Buffer:
  - ▶ TFUniformReplayBuffer: Buffer circolare fornito dalla libreria TF-Agents che pesca le esperienze con distribuzione uniforme.
  - ▶ Decoratore per integrazione dinamica con l'agente
- Ottimizzazione: Utilizzo dell'algoritmo Stochastic Gradient Descent (SGD).
- Esplorazione: Politica  $\epsilon$ -greedy per bilanciare exploration e exploitation.

# Agente: Replay Buffer

- Problema Identificato: Esperienze simili riempiono il buffer, limitando l'apprendimento<sup>1</sup>.
- Soluzione: Divisione del buffer in due parti: una per esperienze di esplorazione e una per esperienze di exploitation.
- Vantaggi: Addestramento con esperienze diversificate per migliorare la stabilità e la performance.

---

<sup>1</sup> Bruin, T. D., Jens Kober, and K. Tuyls. "The importance of experience replay database composition in deep reinforcement learning.", 2015

# Agente: Rete Neurale

- Architettura: La rete è composta da soli layer densi, tutti con funzione di attivazione sigmoidale.
  - ▶ Layer: Ultimo layer con neuroni pari al numero di azioni possibili.
- Preprocessamento: Un layer flatter appiattisce lo stato in un vettore unidimensionale prima di essere passato alla rete neurale.

# Agente: Azioni Illegali

- Azioni Illegali: Esempio: inviare un messaggio da una coda vuota.
- Metodi di Gestione:
  - ▶ Penalizzazione con reward negativa per azioni illegali.
  - ▶ Esclusione delle azioni illegali dal set di azioni possibili (non implementabile facilmente con la libreria usata).

# Agente: AgentFacade (1)

- **AgentFacade:** Un'interfaccia che espone i metodi necessari per l'interazione con l'agente.
  - ▶ Vantaggi: Nasconde le complessità e fornisce un'interfaccia unificata per la comunicazione con l'agente.
- **Configurazione:** Tramite file che specifica varie caratteristiche dell'agente per adattarlo a diversi scenari.
  - ▶ Parametri Aggiuntivi: Numero di code nel nodo per generare azioni valide.

# Agente: AgentFacade (1)

- **AgentFacade:** Un'interfaccia che espone i metodi necessari per l'interazione con l'agente.
  - ▶ Vantaggi: Nasconde le complessità e fornisce un'interfaccia unificata per la comunicazione con l'agente.
- **Configurazione:** Tramite file che specifica varie caratteristiche dell'agente per adattarlo a diversi scenari.
  - ▶ Parametri Aggiuntivi: Numero di code nel nodo per generare azioni valide.
- **Metodo get\_action:** Restituisce l'azione da intraprendere in base allo stato attuale.
- **Addestramento:** L'agente viene addestrato con una periodicità tarata affinchè velocizzi il training ma non rallenti l'apprendimento.

# Agente: Decisions (1)

- Lo spazio delle decisioni si estende attraverso diverse dimensioni, ciascuna rappresenta un parametro.
- Un'azione dell'agente è rappresentabile da un vettore a tre dimensioni:
  - ▶ Azione[0] = invia o non fare nulla
  - ▶ Azione[1] = preleva pacchetto da coda  $i$ , con  $i \in [0, \text{num\_queues} - 1]$
  - ▶ Azione[2] = seleziona power source  $j, j \in [0, \text{num\_power\_sources} - 1]$
- Non tutte le combinazioni di valori sono legali.

## Agente: Decisions (2)

Soluzione semplice: enumerare solo le combinazioni legali.

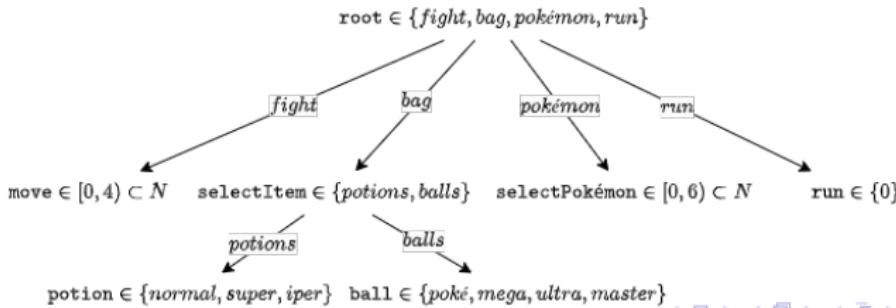
- Problemi dell'enumerazione:
  - ▶ Crescita rapida dello spazio delle azioni con l'aumentare delle opzioni.
  - ▶ Rallentamento dell'apprendimento dell'agente.

Decisions Idea:

- Delegare la scelta del valore di ogni componente di un'azione a un agente diverso.
- Gli agenti sono organizzati in un albero delle decisioni.

# Agente: Decisions (3)

- Nodo root: agente per la decisione iniziale.
- Due casi:
  - ▶ Nodo root è una foglia: un solo nodo root (equivalente all'enumerazione).
  - ▶ Nodo root ha figli: ogni scelta corrisponde a un nodo figlio.
- Decision path
  - ▶ Cammino dalla root a una foglia.



# Agente: Decisions (4)

- Responsabilità di un Nodo:

- ▶ Riferimento all'agente che prende la decisione.
- ▶ Lista di nodi figli.
- ▶ Metodo `get_decisions()` per aggiungere la propria decisione al decision path.
- ▶ Metodo `train()` per addestrare l'agente.

- Raffinazione stato-esperienza:

- ▶ `deduce_consultant_state()`
- ▶ `deduce_consultant_experience()`

# Agente: Decisions (5)

## Implicazioni:

- Gli agenti condividono lo stesso stato osservato dalla root o una sua deduzione.
- Gli agenti condividono la stessa esperienza della root al momento del training o una deduzione.
- Solo i nodi nel decision path vengono interrogati.
- Addestramento dei nodi può essere fatto in parallelo.
- Nodi in profondità ricevono meno esperienze.
  - ▶ Spazio delle azioni ridotto: meno esperienza necessaria per l'apprendimento.
- Evitare gli svantaggi dell'enumerazione dello spazio delle azioni.

# Simulazione

Per la simulazione è stato utilizzato il framework OMNeT++, il quale permette di definire e simulare una rete di nodi.

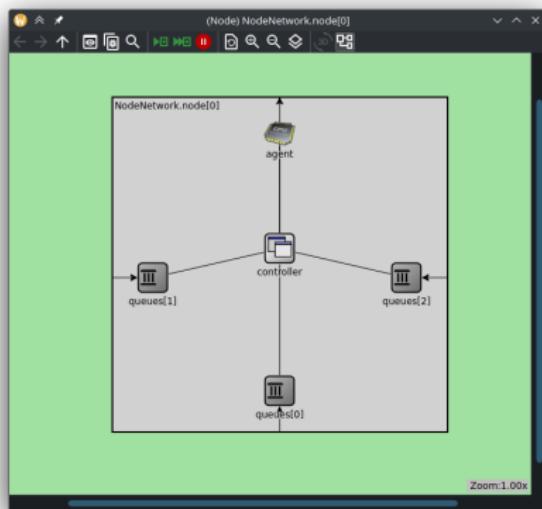
Dato che OMNeT++ non supporta nativamente Python per permettere la comunicazione tra l'agente ed il nodo, è stata utilizzata la libreria PyBind la quale permette di esporre simboli in Python e C++ e farne il binding.



# Nodo

E' l'elemento della rete che è in grado di ricevere e inviare messaggi sulla base della scelta dell'agente, è composto da:

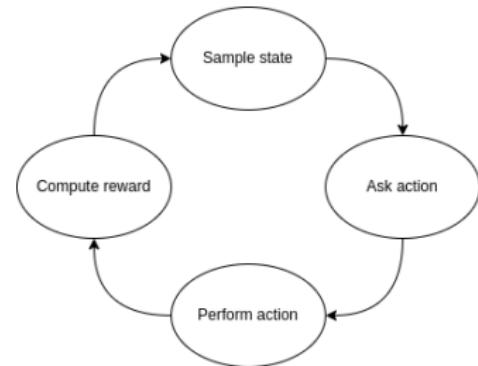
- Controller, che implementa la logica del nodo
- AgentClient, che permette la comunicazione con l'agente
- Queues, una o più code che accodano messaggi



# Nodo: Controller (1)

E' il componente che implementa la logica del nodo, in particolare:

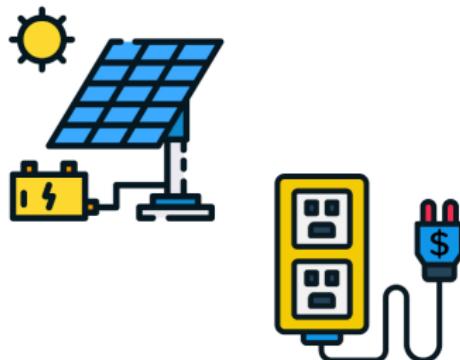
- Campiona lo stato visibile al nodo
- Interroga l'agente per ottenere l'azione consegnandogli la reward dell'azione precedente
- Attua l'azione ricevuta dall'agente
- Calcola la reward per le azioni ricevute dall'agente



## Nodo: Controller (2)

Il controller si occupa anche di gestire le sorgenti energetiche del nodo, in particolare:

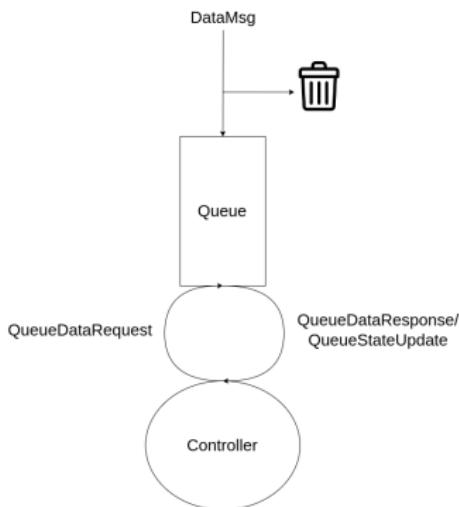
- una batteria, che ha un costo di utilizzo pari a zero ma una capacità limitata e una ricarica influenzata da fattori ambientali;
- Una presa di corrente, che ha un costo di utilizzo diverso da zero ma una capacità pressoché illimitata.



## Nodo: Queue

E' il componente che implementa la logica della coda, in particolare:

- Accoglie i pacchetti in arrivo, scartandoli se la coda è piena
- Riceve le QueueDataRequest di pacchetti da parte del controller
- Consegna i pacchetti al controller in delle QueueDataResponse
- Comunica al controller gli aggiornamenti sullo stato della coda, tramite una QueueStateUpdate, ogni volta che arriva un pacchetto



# Nodo: Agent Client

E' il componente che fa da intermediario tra il controller e l'Agente

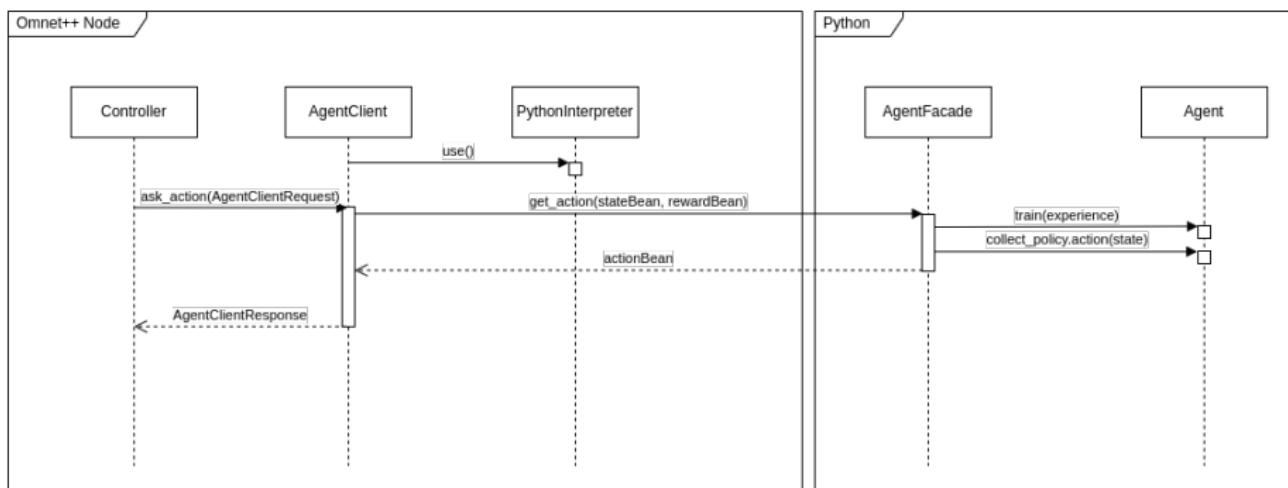
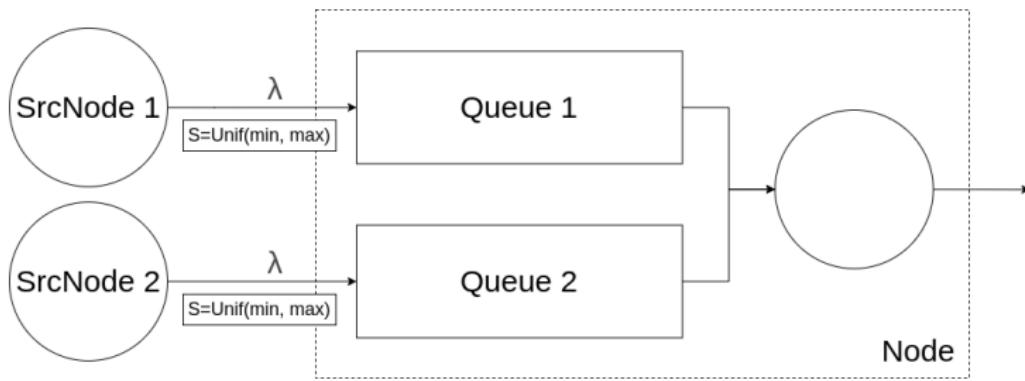


Figure 1: Sequence diagram per la servituta di una richiesta di azione del Controller nei confronti dell'agente.

# SrcNode

E' l'elemento della rete che si occupa di generare il traffico, ogni Queue ha associato un SrcNode. Il tempo di interarrivo dei pacchetti è regolato da una distribuzione esponenziale con tasso  $\lambda$  configurabile, mentre la dimensione dei pacchetti è regolata da una distribuzione uniforme su un intervallo configurabile.



# Roadmap

## 1 Introduzione

- Contesto
- Obiettivi

## 2 Metodologia

- Modello
  - Stato
  - Azioni
  - Reward
- Agente
  - Replay Buffer
  - AgentFacade
  - Decisions
- Simulazione
  - Nodo
  - SrcNode

## 3 Risultati

- Coda singola, reward bilanciata
- Coda singola, full power saving
- Coda singola, no power saving
- 3 Code, reward bilanciata

## 4 Conclusioni

# Risultati

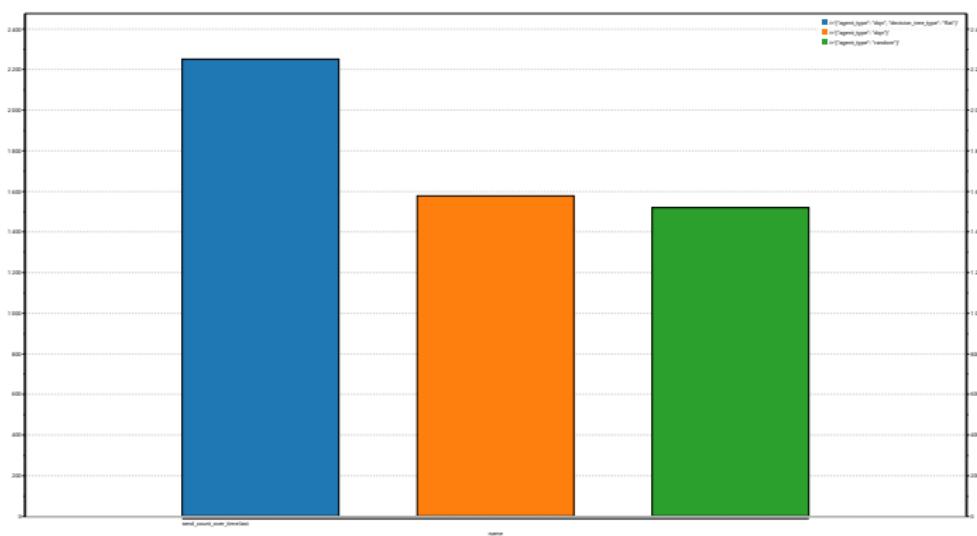
- Sono stati condotti diversi esperimenti per valutare le prestazioni del nodo in base alle azioni suggerite dal suo agente
- Per effettuare gli esperimenti si è partiti da una configurazione dei parametri di base<sup>2</sup>, a cui poi sono state apportate modifiche a seconda dello scenario considerato 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.

---

<sup>2</sup><https://github.com/retarded-reward/collaborative-learning/blob/main/simulations/res/omnetpp.ini>

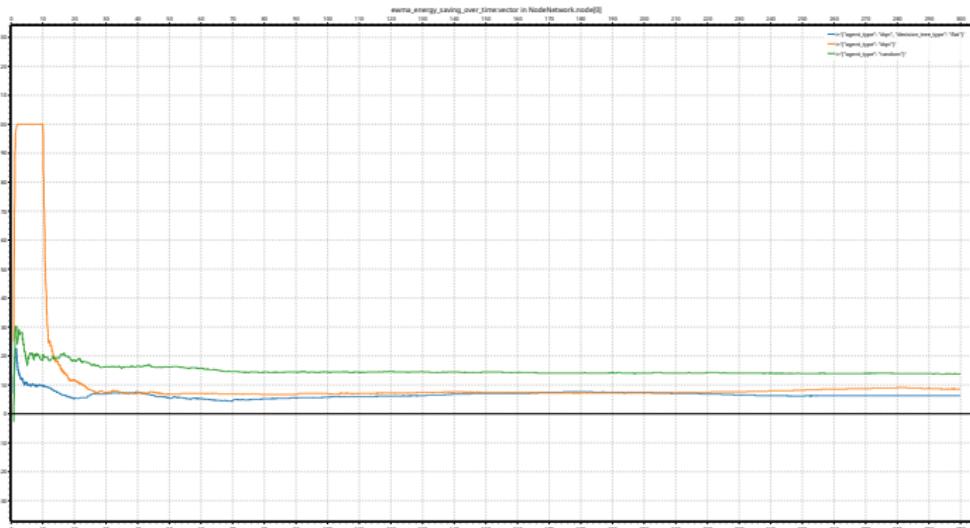


# Risultati: Coda singola, reward bilanciata



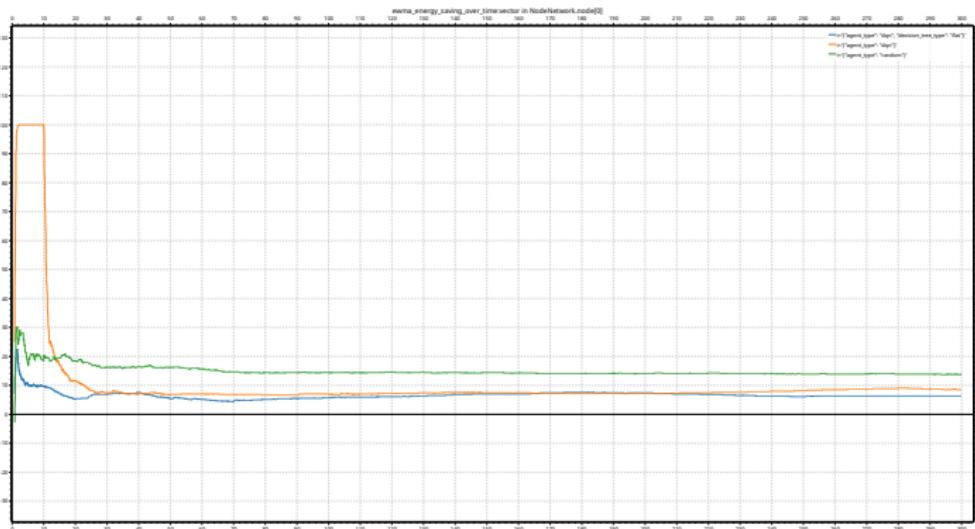
- l'agente root sembra essere scoraggiato dall'inviare pacchetti a causa delle scelte di agenti più in profondità nell'albero delle decisions, ad esempio se l'agente responsabile della scelta della power source sceglie spesso di ricorrere al power chord.

# Risultati: Coda singola, reward bilanciata



- gli agenti DQN e DQN flat presentano un risparmio di spesa energetica più basso (6%, 8%) rispetto al Random (14%)
- gli agenti DQN sembrano inviare più spesso pacchetti dalla power chord piuttosto che dalla batteria

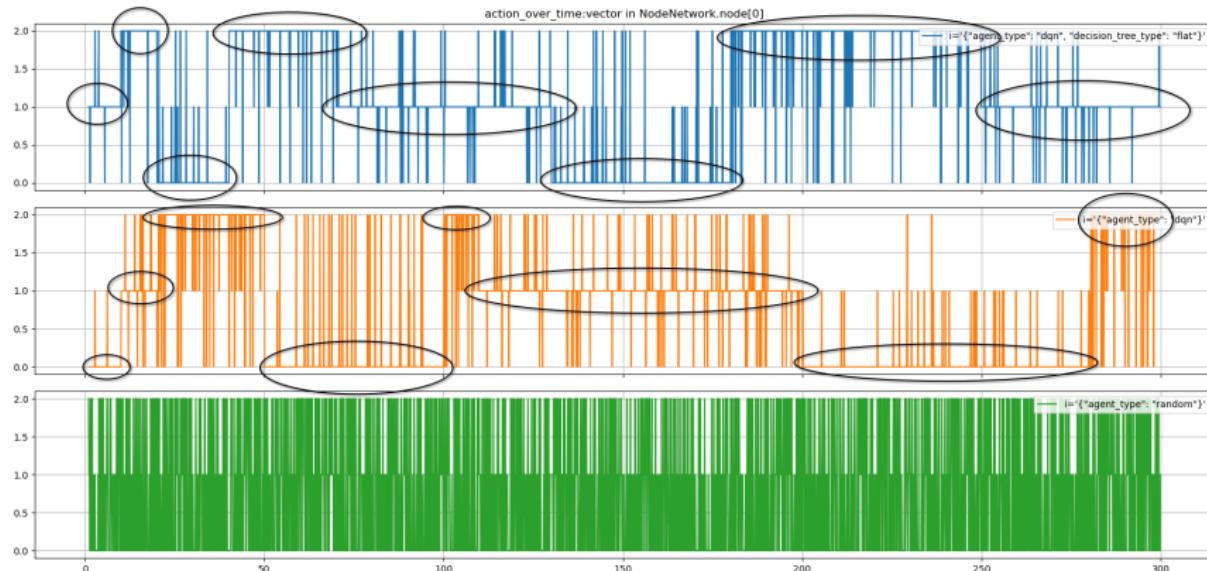
# Risultati: Coda singola, reward bilanciata



- Ciò potrebbe essere dovuto al fallback dato che nello stato dell'agente non è presente alcuna informazione relativa al consumo energetico

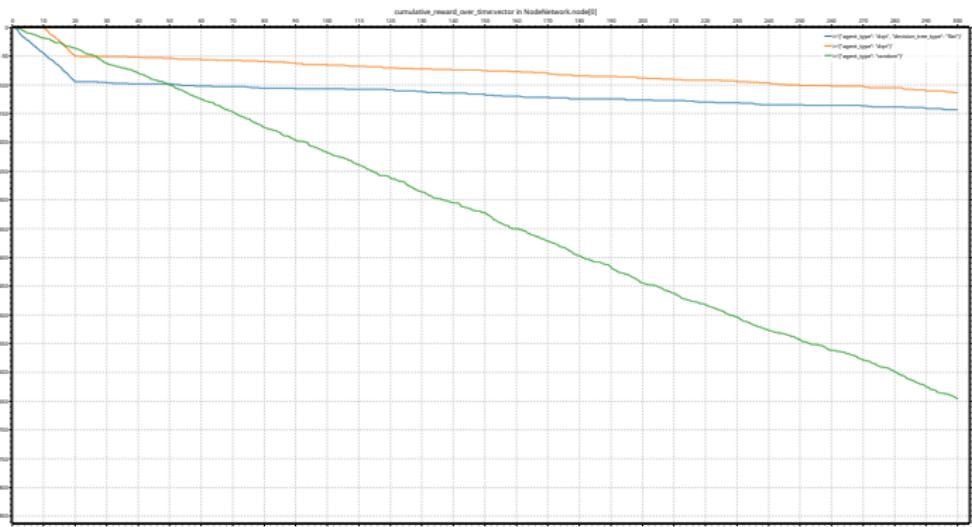


# Risultati: Coda singola, reward bilanciata



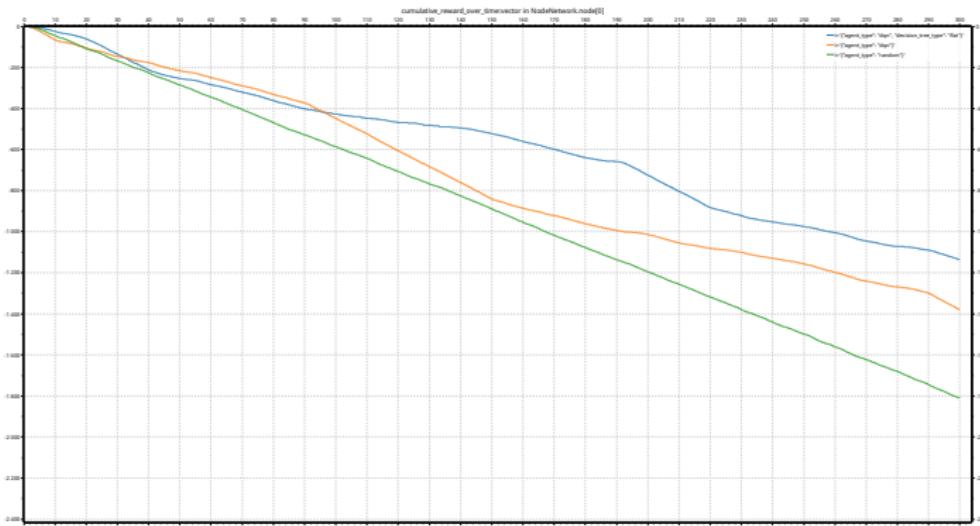
- il DQN con decision oscilla di meno rispetto alla versione flat, anche se accumula più reward negativa.

# Risultati: Coda singola, full power saving



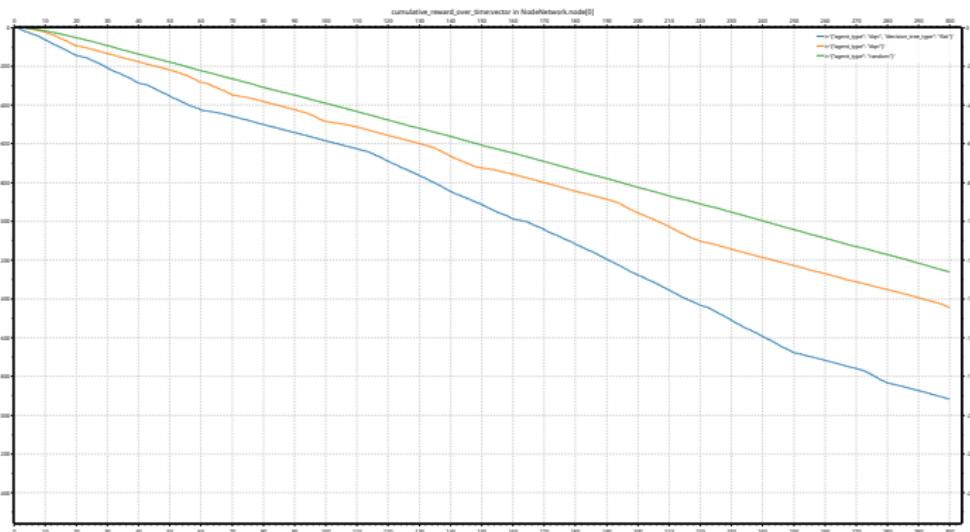
- il DQN con decisions sembra andare meglio rispetto alla controparte flat
- la decision root deve scegliere solo tra due azioni e quindi impara più in fretta che non conviene inviare mai rispetto al DQN flat

## Risultati: Coda singola, no power saving



- le varianti di DQN accumulano un quantitivo di reward simile rispetto allo scenario con reward bilanciata, mentre il random va peggio.
- a pesare maggiormente sulla reward sono i termini riguardanti le code, e la strategia imparata dal DQN punta sempre a minimizzarli.

# Risultati: 3 Code, reward bilanciata



- gli agenti DQN hanno un comportamento peggiore del random, segno che non sono riusciti a imparare una strategia appropriata
- il DQN con decisions sembra soffrire di meno rispetto alla variante flat.

# Roadmap

## 1 Introduzione

- Contesto
- Obiettivi

## 2 Metodologia

- Modello
  - Stato
  - Azioni
  - Reward
- Agente
  - Replay Buffer
  - AgentFacade
  - Decisions
- Simulazione
  - Nodo
  - SrcNode

## 3 Risultati

- Coda singola, reward bilanciata
- Coda singola, full power saving
- Coda singola, no power saving
- 3 Code, reward bilanciata

## 4 Conclusioni

# Conclusioni

- All'aumentare del numero delle code la difficoltà di apprendimento dell'agente aumenta considerevolmente, per cui forse è opportuno considerare un modello diverso;

# Conclusioni

- 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;

# 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, poiché un minimo errore ad esempio nel calcolo della reward può indurre comportamenti nell'agente molto differenti e difficilmente spiegabili;

# 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, poiché un minimo errore ad esempio nel calcolo della reward può indurre comportamenti nell'agente molto differenti e difficilmente spiegabili;
- Apportare delle soluzioni che permettono di non dimenticare facilmente esperienze di exploration può aiutare l'agente a imparare una strategia ottima;

# Grazie per l'attenzione!

- Tutto il codice che implementa ambiente di simulazione e agente è disponibile al seguente repository:  
<https://github.com/retarded-reward/collaborative-learning>
- Consultare README per dettagli build, e la relazione per ulteriori approfondimenti;
- contattaci a:
  - ▶ daniele.laprova@students.uniroma2.eu
  - ▶ matteo.conti@students.uniroma2.eu
  - ▶ luca.falasca@students.uniroma2.eu
- Domande ???
- Fatto con  