Progetto Intelligenza Artificiale
Deep Reinforcement Learning agents for Ambulance Location Problem

Napolitano Margherita Maria & Nappi Severino

a.a. 2024/2025

Abstract

L'obiettivo del progetto originale [1] è sviluppare un framework e un ambiente di simulazione compatibili con OpenAI Gym per testare agenti Deep RL per risolvere il problema della localizzazione di ambulanze.
https://github.com/MichaelAllen1966/qambo.git
Le modifiche da noi apportate mirano a: considerare che l'incidente può coinvolgere una o più parti del corpo del paziente; gli ospedali sono specializzati per la cura di una o più parti del corpo; l'obiettivo dell'ambulanza è portare il paziente all'ospedale più vicino che sia in grado di curare la maggior parte degli infortuni subiti.
https://github.com/Marghe19/qambo.git

Introduzione

Il **Deep Reinforcement Learning (Deep RL)** è un campo storicamente applicato ai giochi come gli scacchi o il go. Ad oggi è in grado di offrire valore nei problemi del mondo reale, tra cui l'ottimizzazione dei sistemi sanitari. Uno di questi problemi è la localizzazione delle ambulanze tra una chiamata e l'altra, al fine di ridurre al minimo il tempo che intercorre tra la chiamata di emergenza e l'arrivo dell'ambulanza.

A tal fine, sono stati costruiti **agenti** Deep RL utilizzando `PyTorch`.

È stato sviluppato un **ambiente di simulazione** personalizzato per l'invio di ambulanze utilizzando `OpenAI Gym` e `SimPy`. L'ambiente è una semplificazione del mondo reale, ma consente di controllare il numero di cluster di

incidenti, il numero di possibili luoghi di spedizione, il numero di ospedali e di creare incidenti che si verificano in luoghi diversi nel corso della giornata.

In questo ambiente personalizzato è stata testata una serie di agenti Deep RL basati su **reti Deep Q**. Tutti hanno ridotto i tempi di risposta alle chiamate di emergenza rispetto all'assegnazione casuale ai punti di smistamento. Le reti **Deep Q Bagging Noisy Duelling** hanno fornito le prestazioni più consistenti. Per ogni tipologia di agente Deep RL testato è incluso un `notebook Jupyter`.

Progetto di riferimento: Amboworld

Il codice del progetto originale è disponibile nella repository GitHub `qambo`.

Simulation Environment

L'ambiente di simulazione, implementato in '`amboworld/environment.py`', simula un sistema di incidenti di emergenza che richiede il trasporto di pazienti in ospedale con ambulanze. La simulazione comprende un mondo vincolato di dimensioni determinate. Quando tali dimensioni sono libere, le ambulanze attendono nei punti di smistamento sparsi per il mondo. L'ambiente di simulazione consente di controllare l'azione degli oggetti contenuti semplificando un problema del mondo reale, quello della localizzazione delle ambulanze.

In particolare è così caratterizzato:

- Gli incidenti si verificano in spazi con dimensioni fisse. Il modello geografico degli incidenti può cambiare giornalmente.
- I punti di spedizione fissi forniscono ambulanze quando si verifica un incidente; È utilizzata l'ambulanza gratuita più vicina.
- Il paziente viene preso dall'ambulanza e trasportato all'ospedale più vicino.
- Quindi, l'agente assegna l'ambulanza a qualsiasi punto di spedizione. L'ambulanza viaggia verso il punto di spedizione, dove diventa disponibile per rispondere agli incidenti.
- L'agente deve assegnare le ambulanze ai punti di smistamento per ridurre il tempo che intercorre tra l'incidente e l'arrivo dell'ambulanza al punto di smistamento.

In tale progetto l'ambiente di simulazione è stato impostato con le seguenti caratteristiche:

- La dimensione del mondo è di 50 km^2
- Un ospedale si trova al centro del mondo
- Le ambulanze, in media, rispondono ciascuna a otto incidenti al giorno (viene utilizzato un basso utilizzo in modo che il tempo di risposta alla chiamata dipenda principalmente dal posizionamento delle ambulanze, piuttosto che da eventuali code per le ambulanze nel sistema).
- Le ambulanze devono arrivare a un punto di smistamento prima di essere disponibili per gli incidenti.
- Le ambulanze viaggiano in linea retta a 60 km/h.
- Ci sono 25 punti di smistamento distribuiti uniformemente su un territorio di 50 km^2.
- Ci sono due schemi di localizzazione degli incidenti al giorno.
- Gli incidenti si verificano con un jitter casuale di $\pm$ 2 km in x e y intorno al centro della posizione dell'incidente.
- Sono stati testati tre scenari: un'area di incidente in qualsiasi momento della giornata e tre ambulanze; due aree di incidente in qualsiasi momento della giornata e sei ambulanze; tre aree di incidente in qualsiasi momento della giornata e nove ambulanze.

L'ambiente di simulazione viene avviato una sola volta per tutto l'addestramento e il test di un agente. I seguenti elementi vengono impostati all'avvio dell'oggetto `simulation environment`:

- *Sistema di coordinate mondiali* con le coordinate massime x e y.
- *Punti di spedizione* utilizzando il seme casuale passato.
- *Punti di localizzazione dell'ospedale* utilizzando il seme casuale passato.
- *Centri dei punti di incidente* utilizzando il seme casuale passato. Per ogni periodo della giornata viene impostato un insieme diverso di punti di incidente.

Esistono tre metodi che interfacciano l'ambiente di simulazione e l'agente:

- `reset`: ripristina l'ambiente di simulazione a uno stato iniziale e passa all'agente la prima serie di osservazioni.

- `step`: la simulazione passa tra i momenti in cui un'ambulanza è in attesa di essere assegnata a un punto di spedizione. L'ambiente di simulazione restituisce una tupla di osservazioni, ricompensa, terminale, info.

- `render`: visualizza lo stato attuale della simulazione (opzionale).

L'ambiente viene reimpostato per ogni sessione di allenamento e di test. Il metodo `reset(self)` permette di:

1. creare un nuovo ambiente SimPy

2. avviare i processi SimPy (incidenti)

3. scegliere un'ambulanza da assegnare al punto di smistamento

4. riportare le prime osservazioni

Nel dettaglio, le *ambulanze* sono tutte libere all'inizio della simulazione e vengono assegnate in modo casuale ai punti di smistamento (ogni sessione inizierà con un'assegnazione casuale diversa) e iniziano la simulazione da quel punto di smistamento.

Gli *incidenti* che richiedono l'intervento di un'ambulanza si verificano in modo casuale (o pseudo-casuale), con tempi di intervallo campionati da una distribuzione esponenziale negativa. Ogni corsa avrà diversi schemi casuali. Ogni incidente viene aggiunto a un elenco ordinato di incidenti senza ambulanza ancora assegnata. Ogni minuto, nel mondo simulato, questo elenco viene controllato e le ambulanze vengono assegnate, se possibile, in base a una semplice priorità "first-in-first-out".

La prima serie di *osservazioni* viene restituita dal metodo di reset, mentre le successive vengono restituite ogni volta che un'ambulanza libera richiede l'assegnazione a un punto di dispacciamento.

Successivamente, è implementato il metodo `step(self, action)`, ovvero il passo dell'ambiente di simulazione che inizia con un'ambulanza in attesa di essere assegnata a un punto di spedizione. Il metodo del passo contiene i seguenti componenti chiave:

- L'*azione* passa l'indice del punto di spedizione che verrà assegnato all'ambulanza in attesa di assegnazione. L'ambulanza si dirige quindi verso quel punto di smistamento alla velocità indicata nei parametri dell'ambulanza. Il viaggio, inoltre, può essere interrotto per raccogliere un nuovo paziente, se si tratta dell'ambulanza più vicina al momento in cui l'ambulanza è stata richiesta.

- La simulazione procede per passi temporali di 1 minuto, *simulation time step loop*. In questo lasso di tempo le ambulanze possono recarsi in ospedale o nei punti di smistamento assegnati, possono verificarsi nuovi incidenti e le ambulanze possono essere inviate agli incidenti. La simulazione esce da questo ciclo se c'è un'ambulanza in attesa di essere assegnata a un punto di smistamento (dopo aver trasportato un paziente in ospedale).

- Alla fine del ciclo dei passi temporali della simulazione, quando un'ambulanza è pronta per essere assegnata a un punto di spedizione, l'ambiente di simulazione *restituisce all'agente*: osservazioni, ricompensa, stato terminale e informazioni.

  In particolare, le *osservazioni* contengono dati che descrivono lo stato attuale dell'ambiente di simulazione. Le osservazioni vengono passate dall'ambiente di simulazione all'agente all'avvio dell'ambiente di simulazione e a ogni passo del modello e vengono restituite come array monodimensionale, la cui lunghezza è pari al numero di punti di spedizione + tre. La prima parte dell'osservazione è un array pari al numero di punti di spedizione e al numero di ambulanze attualmente assegnate a ciascun punto di spedizione. Seguono le coordinate x e y della posizione dell'ambulanza a cui l'agente deve assegnare un punto di spedizione. L'ultimo elemento dell'array di osservazione è l'ora del giorno, espressa come frazione tra 0 e 1.

  Ogni volta che un paziente viene trasportato in ospedale, all'agente viene restituita una *ricompensa*. La ricompensa è il quadrato negativo del tempo impiegato dalla chiamata all'arrivo dell'ambulanza sul luogo dell'incidente per quel paziente.

  Se la simulazione ha raggiunto un determinato tempo massimo di esecuzione, il caso terminale sarà *True*.

  Infine, il dizionario informativo contiene tutti i tempi di call-to-arrival (tempo tra la chiamata del paziente e l'arrivo dell'ambulanza sulla scena), assignment-to-arrival (tempo tra l'assegnazione di un'ambulanza a un incidente e l'arrivo sulla scena), il numero totale di chiamate effettuate fino a quel momento e la frazione di domanda soddisfatta fino a quel momento (il numero di chiamate in cui un'ambulanza è arrivata sulla scena).

Ogni agente è stato addestrato con 50 periodi di un anno simulati. I primi 10 anni hanno utilizzato una selezione di azioni completamente casuale. Successivamente, gli agenti sono passati all'esplorazione epsilon-greedy,

una probabilità decrescente di scegliere azioni a caso piuttosto che intraprendere l'azione raccomandata dalla rete neurale, oppure hanno utilizzato reti noisy e/o bagging per aiutare l'esplorazione. Al termine della simulazione, l'ambiente restituisce Terminal=True. L'agente con le migliori prestazioni, giudicate in base alla massima ricompensa totale, è stato salvato per il test. Ogni agente è stato testato in 30 esecuzioni indipendenti del modello della durata di un anno.

Agenti Deep Reinforcement Learning

Sono stati costruiti diversi agenti Deep RL, basati su reti Deep Q, utilizzando PyTorch.

In particolare gli agenti testati sono stati i seguenti:

1. **Random assignment**: i punti di spedizione sono selezionati a caso. (`00_random_action.ipynb`).

2. **Double Deep Q Network (ddqn)**: rete Deep Q standard, con reti di riferimento e reti di destinazione. (`01_qambo_ddqn.ipynb`).

3. **Duelling Deep Q Network (3dqn)**: le reti di policy e target calcolano Q dalla somma del *valore* dello stato e *vantaggio* (il valore aggiunto di un'azione rispetto al valore medio di tutte le azioni) di ogni azione. (`02_qambo_d3qn.ipynb`).

4. **Noisy Duelling Deep Q Network (noisy 3dqn e ddqn)**: le reti hanno strati che aggiungono rumore gaussiano per facilitare l'esplorazione. (`03_qambo_noisy_3dqn.ipynb` e `04_qambo_noisy_ddqn.ipynb`).

5. **Prioritised Replay Duelling Deep Q Network (pr 3dqn)**: durante l'addestramento della rete di policy, i passi vengono campionati dalla memoria utilizzando un metodo che dà priorità ai passi in cui la rete ha avuto il maggiore errore nella previsione di Q. (`05_qambo_prioritised_replay_d3qn.ipynb`).

6. **Prioritised Replay Noisy Duelling Deep Q Network (pr noisy 3dqn)**: combinazione di replay prioritario e strati rumorosi. (`06_qambo_prioritised_replay_noisy_3dqn.ipynb`).

7. **Bagging Deep Q Network (bagging ddqn)**: le reti multiple (cinque) sono addestrate a partire da diversi campioni bootstrap della memoria. L'azione può essere campionata a caso dalle reti, o a maggioranza. (`07_qambo_bagging_ddqn.ipynb`).

8. **Bagging Duelling Deep Q Network (bagging 3dqn)**: combinazione dell'approccio bagging multi-network con l'architettura duelling. (`08_qambo_bagging_d3qn.ipynb`).

9. **Bagging Noisy Duelling Deep Q Network (bagging noisy 3dqn e ddqn)**: combinazione dell'approccio bagging multi-network con l'architettura duelling e gli strati rumorosi. (`09_qambo_noisy_bagging_ddqn.ipynb` `10_qambo_noisy_bagging_3dqn.ipynb`).

10. **Bagging Prioritised Replay Noisy Duelling Deep Q Network (bagging pr noisy 3dqn)**: combinazione dell'approccio bagging multi-network con l'architettura duelling, gli strati rumorosi e il replay prioritario. (`11_qambo_noisy_bagging_prioritised_3dqn.ipynb`).

Tutti gli agenti hanno migliorato le prestazioni rispetto all'assegnazione casuale ai punti di spedizione. L'agente con le migliori prestazioni in questo test è stato la rete *Bagging Noisy Duelling Deep Q Network*, che combina i vantaggi delle reti duelling, delle reti rumorose e dell'addestramento di più reti con la tecnica del bagging.

Un'osservazione che ha riguardato tutti gli agenti è stato il degrado delle prestazioni con l'allenamento prolungato. Questo è particolarmente rilevante per la sfida di passare da ambienti simulati ad ambienti reali. Una soluzione potenziale potrebbe essere quella di utilizzare un gemello digitale dell'ambiente reale (in grado di riprodurre tutti i luoghi e le tempistiche degli incidenti) per tutto l'addestramento, e utilizzare l'agente risultante nel mondo reale, aggiornandolo solo con il trasferimento di nuovi dati al gemello digitale per l'ulteriore addestramento.

Approccio Proposto

Il nostro contributo al lavoro sopracitato, disponibile nella repository GitHub `qambo_modified`, è riassumibile in tre punti principali:

1. l'incidente può coinvolgere una o più parti del corpo del paziente;

2. gli ospedali sono specializzati per la cura di una o più parti del corpo;

3. l'obiettivo dell'ambulanza è portare il paziente all'ospedale più vicino che sia in grado di curare la maggior parte degli infortuni subiti.

Per consentire tali aggiunte, è necessario modificare l'environment `amboworld`, in particolare i file `patient.py` e `environment.py`.

Affinchè l'incidente coinvolga una o più parti del corpo del paziente, andremo ad aggiungere a `patient.py` le parti del corpo (bodyparts) sottoforma di dizionario.

In `environment.py` andremo a settare le parti del corpo che vogliamo considerare 1.

Snippet 1: environment.py Patient Body Parts

```
self.POSSIBLE_BODY_PARTS = ["head", "arm", "leg", "chest", "abdomen",
↪  "back"]
```

Successivamente è stata implementata una funzione per generare danni casuali alle parti del corpo 2. In particolare, genera un dizionario casuale di parti del corpo e livelli di danno (green, yellow, red) che si riferiscono ai '*codici*' utilizzati negli ospedali (codice verde, codice giallo, codice rosso) in relazione alla gravità delle ferite.

Snippet 2: environment.py Generate random Body Parts

```
def _generate_random_injuries(self):
        num_parts = random.randint(1, len(self.POSSIBLE_BODY_PARTS))
        selected_parts = random.sample(self.POSSIBLE_BODY_PARTS,
        ↪  num_parts)
        injuries = {part: random.choice(["green", "yellow", "red"]) for
        ↪  part in selected_parts}
        return injuries
```

Invece, affinchè gli ospedali siano specializzati per la cura di una o più parti del corpo, si aggiunge, sempre in `environment.py`, il metodo `_set_hospital_specialization(self)` che permette di generare in modo casuale le specializzazioni per ciascun ospedale (almeno una).

Snippet 3: environment.py Hospital specialization

```
def _set_hospital_specialization(self):
        for h_index in range(self.number_hospitals):
            specialized_parts =
            ↪  random.sample(self.POSSIBLE_BODY_PARTS,random.randint(1,
            ↪  len(self.POSSIBLE_BODY_PARTS)))
            self.hospitals_specializations.append(specialized_parts)
```

Infine, poichè l'obiettivo dell'ambulanza è portare il paziente all'ospedale più vicino che sia in grado di curare la maggior parte degli infortuni subiti dobbiamo considerare questi due criteri per effettuare l'assegnazione dell'ospedale al paziente. I due criteri sono:

- **Massima copertura** delle necessità del paziente: ogni paziente ha delle esigenze (rappresentate da patient.body_parts) che devono essere coperte dalle specializzazioni di un ospedale (contenute in self.hospitals_specializations).
- **Minima distanza**: se più ospedali offrono lo stesso livello di copertura, il paziente verrà assegnato a quello più vicino.

Per effettuare tali controlli è stato implementato il codice riportato in 4.

Snippet 4: environment.py Max Coverage-Min Distance

```
max_coverage = 0

    for hospital_index in range(self.number_hospitals):
        distance = get_distance(
            patient.incident_x, patient.incident_y,
            self.hospitals[hospital_index][0],
            self.hospitals[hospital_index][1])

        coverage = sum(1 for part in patient.body_parts if part in
        ↪  self.hospitals_specializations[hospital_index])

        if coverage > max_coverage or (coverage == max_coverage and
        ↪  distance < best_distance):
            best_distance = distance
            #print(f'Ho cambiato {best_hospital} \tche aveva coverage
            ↪  {max_coverage} \tcon {hospital_index} \tche ha coverage:
            ↪  {coverage}')
            best_hospital = hospital_index
            max_coverage = coverage

        self.results_coverage.append(max_coverage)

    patient.allocated_hospital = best_hospital
```

In particolare, avremo:

- **self.number_hospitals**: numero totale di ospedali disponibili.
- **self.hospitals**: lista delle coordinate degli ospedali, dove ogni ospedale è rappresentato da un tupla (x, y).
- **self.hospitals_specializations**: lista delle specializzazioni disponibili per ciascun ospedale.

- **patient.incident_x, patient.incident_y**: coordinate dell'incidente del paziente.
- **patient.body_parts**: lista delle esigenze mediche del paziente, rappresentate come parti del corpo che richiedono di essere curate.
- **best_distance** e **max_coverage**: variabili che memorizzano rispettivamente la minima distanza trovata e il massimo livello di copertura ottenuto.
- **best_hospital**: indice dell'ospedale che soddisfa al meglio i due criteri.
- **self.results_coverage**:lista in cui si memorizzano i valori di max_coverage per analisi successive.

Il ciclo scorre tutti gli ospedali per confrontarli e trovare quello più adatto al paziente.

La distanza tra il luogo dell'incidente del paziente e l'ospedale corrente viene calcolata usando una funzione `get_distance`.

Per calcolare la copertura, il codice conta quante delle esigenze del paziente (`patient.body_parts`) sono soddisfatte dalle specializzazioni dell'ospedale corrente (`self.hospitals_specializations[hospital_index]`).

L'ospedale corrente viene scelto come "migliore" se:

- la copertura è maggiore rispetto al massimo trovato finora (`coverage > max_coverage`), oppure
- la copertura è uguale al massimo trovato, ma la distanza è inferiore (`coverage == max_coverage and distance < best_distance`).

Se l'ospedale corrente soddisfa uno di questi criteri, vengono aggiornati i valori di `best_distance, best_hospital e max_coverage`.

Il valore di copertura massimo per il paziente corrente viene aggiunto a una lista per registrare i risultati.

Alla fine del ciclo, l'ospedale con la migliore combinazione di copertura e distanza viene assegnato al paziente.

Inoltre, a differenza del lavoro originale che considera un solo ospedale, andremo ad addestrare e testare gli algoritmi su quattro ospedali (`number_hospitals=4`). Gli esperimenti sono stati effettuati considerando 3 incidenti (parametro `NUMBER_INCIDENT_POINTS`) e 9 ambulanze (parametro `NUMBER_AMBULANCES`.

I parametri principali di output prodotti da tali agenti sono:

- **call_to_arrival**: il tempo tra la chiamata e l'arrivo della risposta ;

- **assign_to_arrival**: il tempo tra l'assegnazione dell'agente e il suo arrivo;
- **demand_met**: la proporzione di richieste soddisfatte (da 0 a 1).

Call_to_arrival e assign_to_arrival sono calcolati a partire dall'ora corrente della simulazione tramite la proprietà `Environment.now`. Il tempo di simulazione è un numero senza unità e viene incrementato tramite eventi di Timeout.

I grafici riportati rappresentano l'andamento delle seguenti metriche: esplorazione (linea verde tratteggiata), call to arrival (linea rossa continua) e assignment to arrival (linea blu tratteggiata). L'asse x rappresenta il numero di run (episodi), l'asse y di sinistra misura il livello di esplorazione, mentre l'asse y di destra misura i tempi di risposta.

Infine sono riportati i valori medi di call to arrival e assignment to arrival risultanti dal test in uno scenario simulato.

Random Action

L'agente random sceglie la prossima azione, nel nostro caso i punti di smistamento, in modo casuale, senza prestare attenzione allo stato attuale dell'ambiente.

I valori medi di call_to_arrival e assign_to_arrival risultanti dal test in un ambiente simulato sono i seguenti:

- call_to_arrival: $\simeq 18.9$ vs $\simeq 19.2$
- assign_to_arrival: $\simeq 18.5$ vs $\simeq 18.7$.

Double DQN Learning

L'obiettivo generale delle reti '$\mathcal{Q}$ learning' è quello di creare una rete neurale che preveda e massimizza $\mathcal{Q}$, dove $\mathcal{Q}$ sono le reward future attese scontate nel tempo. $\mathcal{Q}$ viene appreso attraverso l'equazione di Bellman. La rete deve prendere in considerazione le osservazioni sullo stato attuale e raccomandare l'azione con il massimo $\mathcal{Q}$.

Il **double DQN** [2], in particolare, contiene due reti. Questa modifica, rispetto al DQN semplice, consiste nel disaccoppiare l'addestramento del $\mathcal{Q}$ per lo stato corrente e il $\mathcal{Q}$ *target* derivato dallo stato successivo, che sono strettamente correlati quando si confrontano le caratteristiche di input.

La rete di policy viene utilizzata per selezionare l'azione (l'azione con la migliore $\mathcal{Q}$ prevista) durante il gioco.

Durante l'addestramento, l'azione migliore prevista viene presa dalla rete di criteri, ma la rete di criteri viene aggiornata utilizzando il valore $\mathcal{Q}$ previsto per lo stato successivo dalla rete target, che viene aggiornata dalla rete di criteri meno frequentemente.

In questo modo, durante l'addestramento, l'azione viene selezionata utilizzando i valori $\mathcal{Q}$ della rete di policy, ma la rete di policy viene aggiornata per prevedere meglio il valore $\mathcal{Q}$ di quell'azione dalla rete target. La rete di criteri viene copiata nella rete target ogni n passi.

I risultati dell'addestramento sono riportati nei grafici 1 e possiamo notare che le prestazioni dei due modelli sono simili.

Il **tempo medio dalla chiamata all'arrivo dell'ambulanza** (call to arrival, linea rossa continua) inizia con un valore di circa 20 in entrambi i progetti e subisce una drastica riduzione nei primi 10 episodi, raggiungendo valori prossimi ai 15.

Anche il **tempo medio dall'assegnazione all'arrivo** (assignment to arrival, linea blu tratteggiata) inizia con valori elevati, circa 18-19 e si riduce stabilizzandosi intorno ai 14.

Il primo grafico, presenta oscillazioni più marcate, indicando che l'agente non ha completamente ottimizzato le sue decisioni mentre nel secondo le oscillazioni sono meno evidenti, indicando una maggiore consistenza nei risultati. Nel secondo grafico, dunque, l'agente appare più stabile, con meno variazioni nei tempi di risposta.

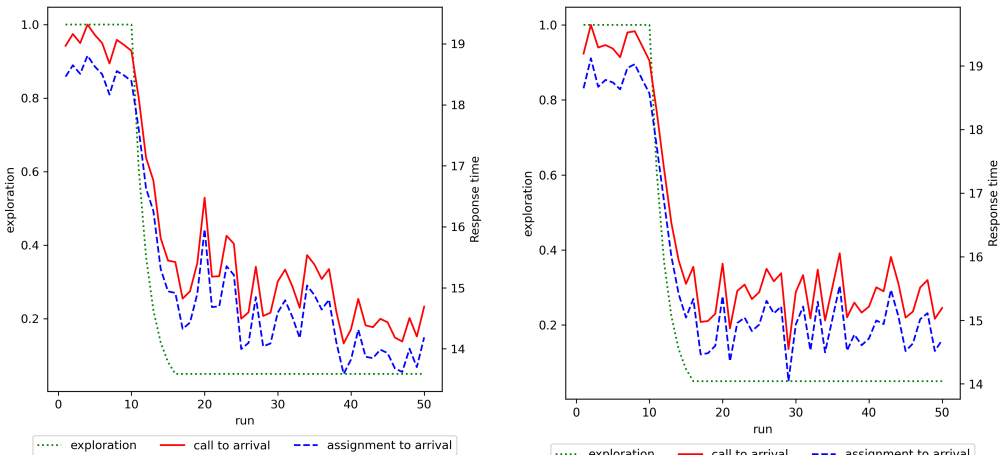

Figura 1: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente DDQN e NUMBER_INCIDENT_POINTS=3.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 16.8 vs 14.0
- assignment_to_arrival: $\simeq$ 16.3 vs 13.5

Duelling Deep Q Learning

Il **duelling** [3] è molto simile al Double DQN, tranne per il fatto che la rete di policy si divide in due. Una componente si riduce a un singolo valore, che modella il valore dello stato; l'altro componente modella il vantaggio, la differenza in $\mathcal{Q}$ tra le diverse azioni. Il valore medio viene sottratto da tutti i valori, in modo che il vantaggio sia sempre pari a zero. Questi valori vengono aggregati per produrre $\mathcal{Q}$ per ogni azione.

I risultati dell'addestramento sono riportati nei grafici 2. I valori risultano molto simili in quanto in entrambi i grafici i valori **call_to_arrival** e **assignment_to_arrival** inizialmente sono circa 18-19 per poi stabilizzarsi a circa 12-14 dopo 20 episodi.

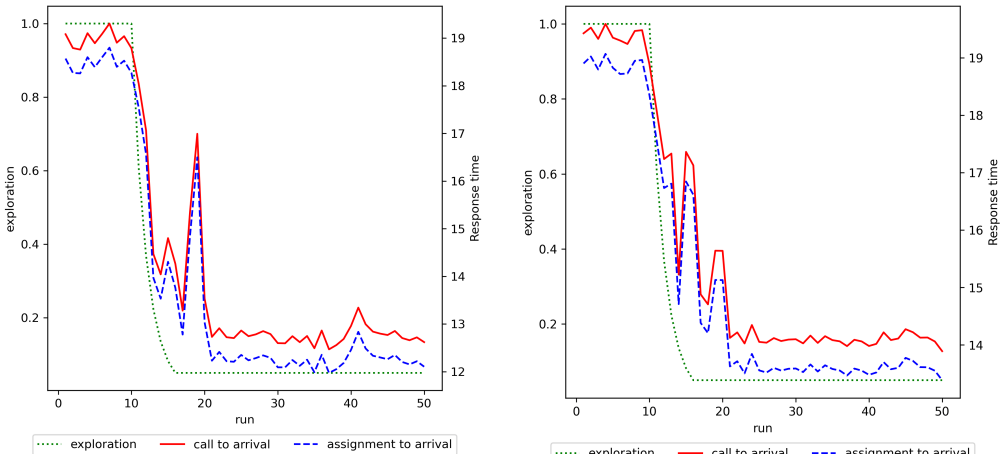

Figura 2: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente D3QN e NUMBER_INCIDENT_POINTS=3.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 12.6 vs 13.8
- assignment_to_arrival: $\simeq$ 12.2 vs 13.3.

Noisy Double DQN Learning

Nell'agente *Noisy Double DQN Learning* sono aggiunti i layer di rumore [4], un'alternativa all'esplorazione epsilon-greedy.

Per ogni peso dello strato abbiamo un valore casuale che estraiamo dalla distribuzione normale. Questo valore casuale viene utilizzato per aggiungere rumore all'output. I parametri per l'entità del rumore per ogni peso, σ, sono memorizzati all'interno dello strato e vengono addestrati come parte della back-propogation standard.

Una modifica ai normali strati nosiy consiste nell'utilizzare strati con *rumore gaussiano fattorizzato*. Questo riduce la quantità di numeri casuali da campionare e quindi è meno costoso dal punto di vista computazionale. Ci sono due vettori casuali, uno con la dimensione dell'input e l'altro con la dimensione dell'output. Infine viene creata una matrice casuale calcolando il prodotto esterno dei due vettori.

I risultati dell'addestramento sono riportati nei grafici 3. Nel progetto originale (grafico a sinistra) il tempo delle metriche **call_to_arrival** e **assignment_to_arrival** diminuisce significativamente durante i primi 10 episodi, successivamente, i tempi oscillano leggermente ma rimangono bassi. Ciò suggerisce un miglioramento rapido e che il modello diventa stabile abbastanza presto. Anche nel progetto modificato (grafico a destra) si osserva una diminuzione dei tempi della metrica **call_to_arrival** nei primi episodi, ma i tempi finali sono leggermente più bassi rispetto al progetto originale. Questo potrebbe indicare una migliore ottimizzazione o tempi di risposta più rapidi. Inoltre, sono presenti minori oscillazioni rispetto al progetto originale. Il progetto modificato,quindi, sembra più efficiente, con tempi di risposta inferiori in entrambe le metriche principali e le oscillazioni sono meno marcate, suggerendo un comportamento più robusto.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 12.2 vs 12.5
- assignment_to_arrival: $\simeq$ 11.7 vs 12.0.

Noisy Duelling Double Deep Q Learning

Anche nell'agente *Noisy Duelling Double Deep Q Learning* sono aggiunti i layer di rumore [4].

I risultati dell'addestramento sono riportati nei grafici 4. In entrambi possiamo notare valori iniziali di **call_to_arrival** e **assignment_to_arrival** di 18-19 per poi scendere a 15-16 nei primi 10 episodi.

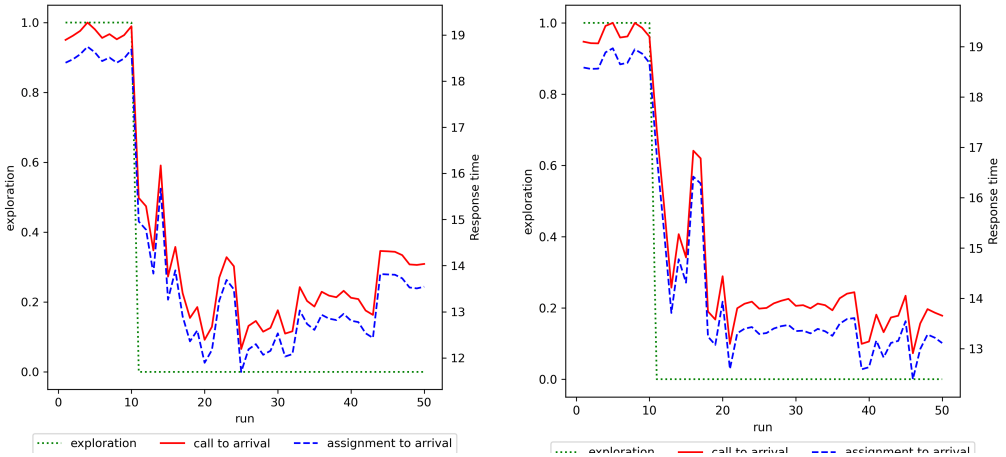

Figura 3: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente Noisy Double DQN e NUMBER_INCIDENT_POINTS=3.

Nel grafico di destra (progetto originale), la metrica call to arrival sembra più stabile rispetto al grafico di sinistra, specialmente dopo l'episodio 20, mentre assignment to arrival oscilla maggiormente.

Entrambi i grafici mostrano un calo rapido dell'exploration, ma nel grafico di destra la stabilizzazione è leggermente più rapida.

Anche il tempo di risposta mostra una tendenza al ribasso, ma nel grafico di destra si percepisce un miglioramento più evidente negli episodi iniziali.

Le modifiche, dunque, sembrano aver portato a maggiore stabilità nella metrica call to arrival, maggiore variabilità nella metrica assignment to arrival e una convergenza più rapida dell'exploration e miglioramento iniziale del tempo di risposta.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 11.7 vs 13.8
- assignment_to_arrival: $\simeq$ 11.2 vs 13.3.

Prioritised Replay Duelling Double Deep Q Learning

Nel **replay prioritario**[5], a differenza del DQN standard dove i campioni vengono prelevati in modo casuale dalla memoria (buffer di replay), i campioni vengono prelevati in proporzione alla loro perdita durante l'addestramento

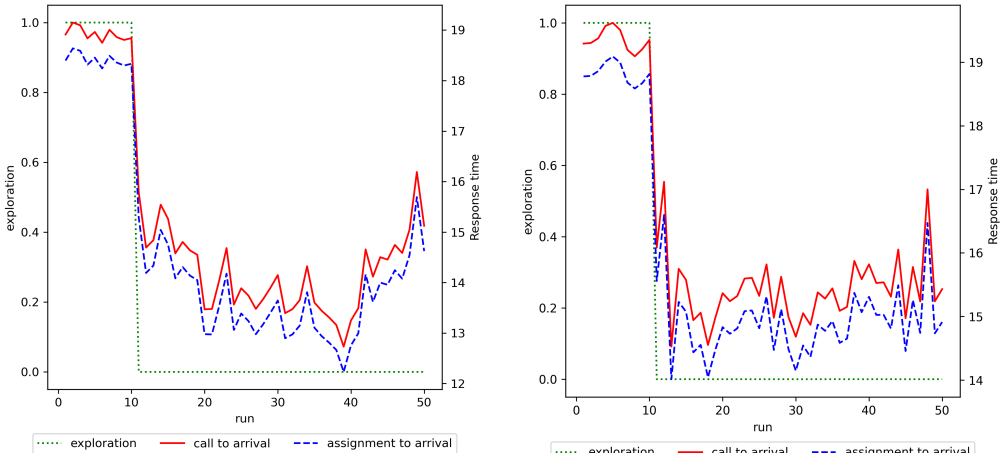

Figura 4: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente Noisy Duelling Double Deep Q e NUMBER_INCIDENT_POINTS=3.

della rete; se la rete ha un errore maggiore nel predire il valore target di uno stato/azione, quei campioni verranno campionati più frequentemente e ciò ridurrà l'errore della rete fino a quando il campione non sarà prioritario.

L'addestramento, quindi, si concentra maggiormente sui campioni che sbagliano di più e dedica meno tempo all'addestramento dei campioni che è già in grado di prevedere correttamente.

Quando è utilizzata la perdita come priorità, aggiungiamo un piccolo valore (1e-5) alla perdita in modo da evitare che un campione abbia priorità zero e non abbia mai la possibilità di essere campionato. Per la frequenza di campionamento, si aumenta anche la perdita alla potenza di α (valore predefinito di 0,6). Valori più piccoli di alfa comprimeranno le differenze tra i campioni, rendendo la ponderazione della priorità meno significativa nella frequenza di campionamento.

I risultati dell'addestramento sono riportati nei grafici 5. In entrambi i casi si parte da valori iniziali delle metriche **call_to_arrival** e **assignment_to_arrival** di 18-19 per poi calare a valori che si aggirano tra 14 e 17. Nel grafico di sinistra (progetto originale), le linee call to arrival e assignment to arrival appaiono più regolari e parallele; invece, nel grafico di destra, entrambe le linee mostrano oscillazioni più significative e meno coordinate.

Il progetto modificato sembra avere un tempo di risposta complessivamente migliore nei primi episodi rispetto al progetto originale. Tuttavia, i picchi di call to arrival potrebbero indicare inefficienze occasionali. Il grafico

di sinistra, invece, mostra un comportamento più stabile e prevedibile, con le metriche che si muovono in modo più coordinato.

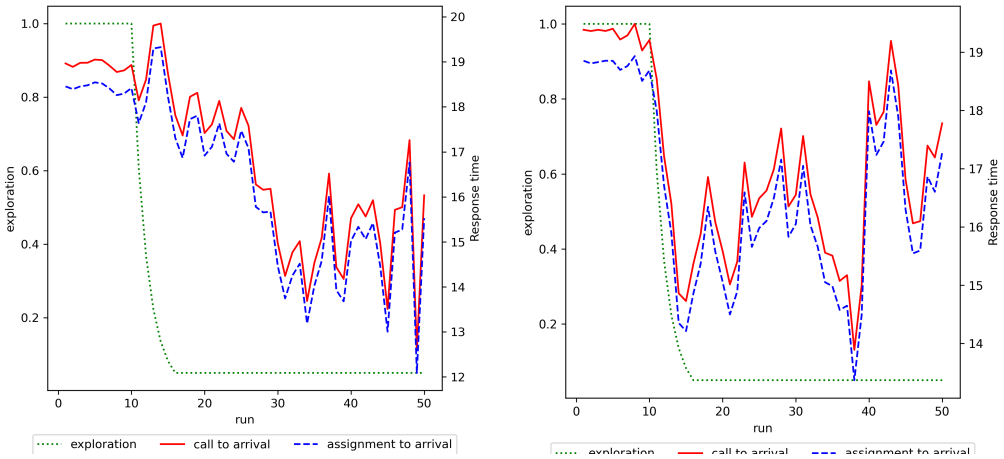

Figura 5: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente Prioritised Replay Duelling Double Deep Q e NUMBER_INCIDENT_POINTS=3.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 13.1 vs 16.3
- assignment_to_arrival: $\simeq$ 12.6 vs 15.8.

Prioritised Replay Noisy Duelling Double Deep Q Learning

Nel modello *Prioritised Replay Noisy Duelling Double Deep Q Learning* sono stati aggiunti i layer di rumore all'algoritmo *Prioritised Replay Duelling Double Deep Q Learning*.

I risultati dell'addestramento sono riportati nei grafici 6. Nel grafico di sinistra (progetto originale) entrambe le metriche **call_to_arrival** e **assignment_to_arrival** mostrano un andamento variabile durante i primi episodi, con fluttuazioni significative partendo da valori di circa 18-19. Dopo l'iniziale fase di esplorazione, i tempi diventano più stabili, con una riduzione generale a valori prossimi a 15-16. Nel grafico di destra (progetto modifico) le metriche mostrano una riduzione più consistente e meno fluttuazioni rispetto al grafico originale, passando da valori di circa 18-19 a 13-14. Sembra esserci un

miglioramento nel raggiungere tempi di risposta medi più bassi. In entrambi i casi, l'exploration rate diminuisce rapidamente, ma nel grafico modificato sembra esserci una transizione più controllata, che potrebbe favorire un miglior equilibrio tra esplorazione ed exploitation.

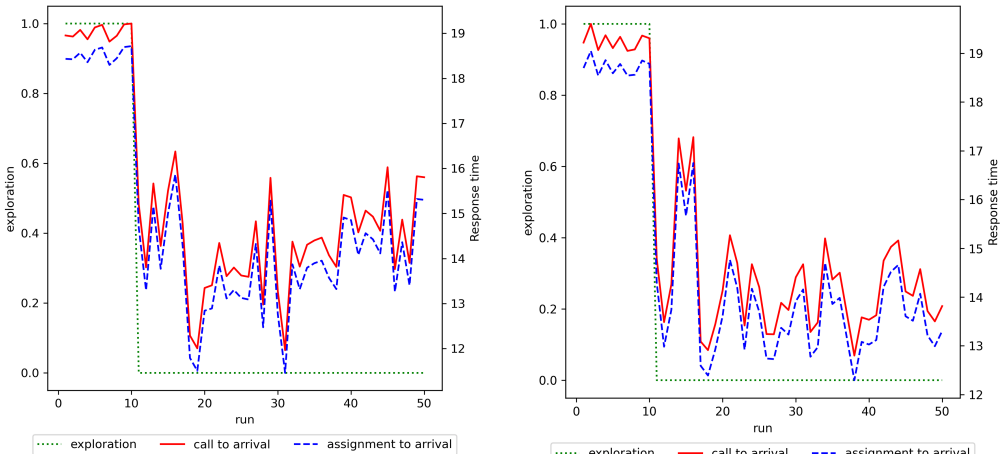

Figura 6: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente Prioritised Replay Noisy Duelling Double Deep Q e NUMBER_INCIDENT_POINTS=3.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 12.3 vs 12.4
- assignment_to_arrival: $\simeq$ 11.8 vs 11.9.

Bagging Double Deep Q Learning

Nel *Bagging Double Deep Q Learning* [6] ogni rete è addestrata a partire dalla stessa memoria, ma ha pesi iniziali diversi e viene addestrata su campioni bootstrap diversi da quella memoria. In questo esempio, le azioni sono scelte in modo casuale da ciascuna rete. Questo metodo di bagging può essere utilizzato anche per avere una misura dell'incertezza dell'azione, osservando la distribuzione delle azioni raccomandate dalle diverse reti. Il bagging può anche essere utilizzato per aiutare l'esplorazione durante le fasi in cui le reti forniscono diverse azioni suggerite.

I risultati dell'addestramento sono riportati nei grafici 7. Nel grafico a sinistra (progetto originale) i tempi di risposta calano in modo netto durante

i primi 20 episodi, con una stabilizzazione progressiva. La variabilità delle metriche è minima dopo i primi episodi, indicando che l'agente ha raggiunto un comportamento stabile e valori di circa 13-14. Nel grafico a destra (progetto modificato) il comportamento è simile al grafico di sinistra, ma le curve dei tempi di risposta sembrano essere più ordinate e con fluttuazioni ridotte. La stabilizzazione dei tempi di risposta avviene più rapidamente, dopo circa 15 episodi a valori di 12-13. Il miglioramento sembra più marcato nel grafico a destra dove le metriche convergono a valori ottimali più rapidamente.

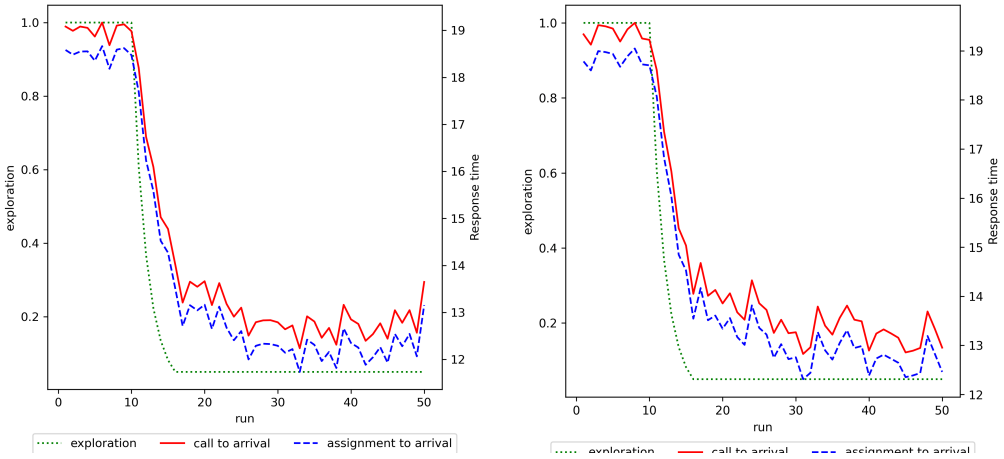

Figura 7: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente Bagging Double Deep Q e NUMBER_INCIDENT_POINTS=3.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 9.9 vs 13.3
- assignment_to_arrival: $\simeq$ 9.4 vs 12.8.

Bagging Duelling Double Deep Q Learning

Nell'agente *Bagging Duelling Double Deep Q Learning* avremo una rete duelling e non più standard dqn.

I risultati dell'addestramento sono riportati nei grafici 8. Nel grafico a sinistra (progetto originale) l'exploration rate scende rapidamente intorno alla decima iterazione e si stabilizza vicino a zero. I valori iniziali della metrica **call_to_arrival** sono alti, circa 18-19, e mostrano oscillazioni, ma

nel tempo si stabilizzano su un valore più basso, intorno a 12-13. I valori della metrica **assignment_to_arrival** mostrano un comportamento simile alla curva call_to_arrival, stabilizzandosi intorno al valore 11. Nel grafico di destra (progetto originale) i valori iniziali della metrica **call_to_arrival** sono anche più alti del progetto originale, circa 20, ma il calo è meno netto e le oscillazioni continuano più a lungo. Alla fine, si stabilizzano intorno a 12-13, ma il calo risulta meno uniforme rispetto al grafico sinistro. I valori della metrica **assignment_to_arrival** oscillano maggiormente nelle prime iterazioni, ma si stabilizzano su un valore di circa 12. Tale modello, quindi, esplora più a lungo prima di fissarsi su una politica stabile. Questo approccio potrebbe essere utile in ambienti complessi o dinamici, ma richiede più tempo per ottenere benefici comparabili.

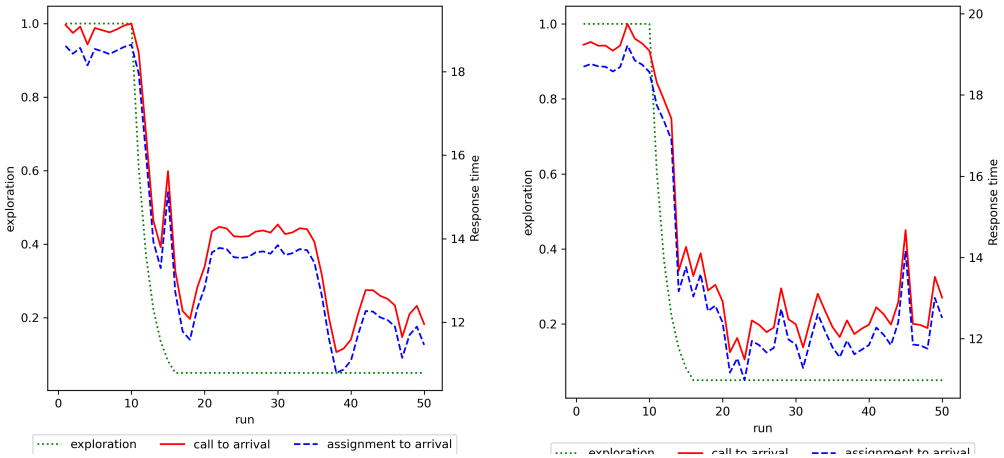

Figura 8: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente Bagging Duelling Double Deep Q e NUMBER_INCIDENT_POINTS=3.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 11.1 vs 11.9
- assignment_to_arrival: $\simeq$ 10.6 vs 11.4.

Noisy Bagging Double Deep Q Learning

Nel *Noisy Bagging Double Deep Q Learning* sono stati aggiunti i layer di rumore.

I risultati dell'addestramento sono riportati nei grafici 9. Nel grafico a sinistra (progetto originale) i tempi di risposta calano in modo netto durante i primi 10 episodi passando da valori di circa 19 a 12-13, con una stabilizzazione progressiva. La variabilità delle metriche è minima dopo i primi episodi, indicando che l'agente ha raggiunto un comportamento stabile. Nel grafico a destra (progetto modificato) il comportamento è simile al grafico di sinistra, con delle curve più pronunciate.

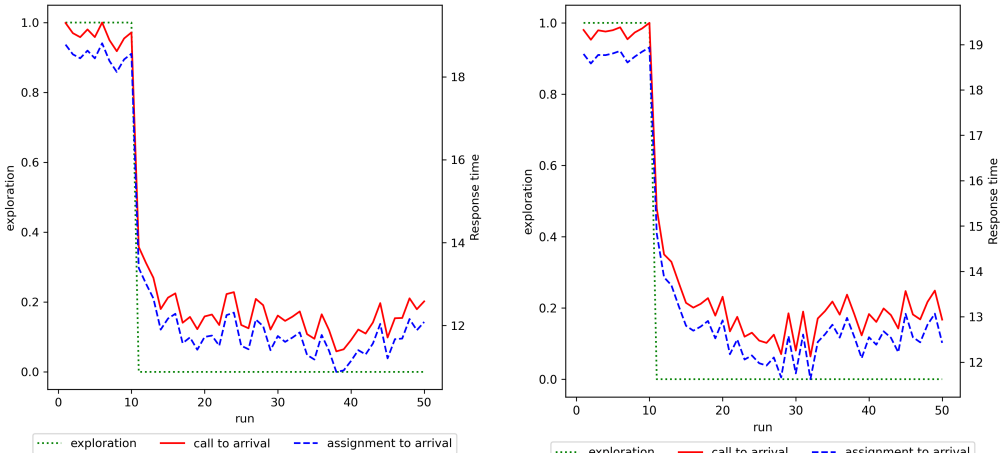

Figura 9: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente Noisy Bagging Double Deep Q e NUMBER_INCIDENT_POINTS=3.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 13.8 vs 13.6
- assignment_to_arrival: $\simeq$ 13.3 vs 13.1.

Noisy Bagging Duelling Double Deep Q Learning

L'agente *Noisy Bagging Duelling Double Deep Q Learning* utilizza il duelling dqn e i layer di rumore.

I risultati dell'addestramento sono riportati nei grafici 10. Nel grafico a sinistra (progetto originale) l'exploration rate scende rapidamente intorno alla decima iterazione e si stabilizza vicino a zero. I valori iniziali della metrica **call_to_arrival** sono alti, circa 18-19, e mostrano oscillazioni, ma nel tempo si stabilizzano su un valore più basso, intorno a 10-11. I valori

della metrica **assignment_to_arrival** mostrano un comportamento simile alla curva call_to_arrival, stabilizzandosi intorno al valore 11. Nel grafico di destra (progetto originale) i valori iniziali delle metriche **call_to_arrival** e **assignment_to_arrival** sono anche più alti del progetto originale, circa 19-20, ma il calo è meno netto. In questo caso abbiamo eseguito solo 25 episodi in relazione alle limitazioni dei nostri strumenti. Possiamo comunque notare che dopo l'episodio 20 inizia una fase di stabilizzazione verso i valori 11-12.

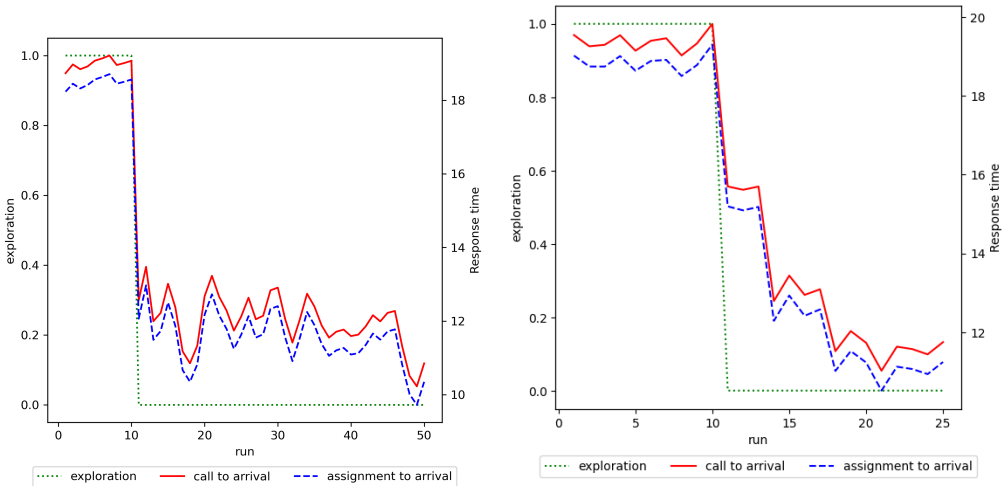

Figura 10: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente Noisy Bagging Duelling Double Deep Q e NUMBER_INCIDENT_POINTS=3.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 10.5 vs 10.8
- assignment_to_arrival: $\simeq$ 9.9 vs 10.5.

Noisy Bagging Duelling Prioritised Replay Double Deep Q Learning

Nel *Noisy Bagging Duelling Prioritised Replay Double Deep Q Learning* avremo l'aggiunta del replay prioritario.

I risultati dell'addestramento sono riportati nei grafici 11.

Nel grafico a sinistra (progetto originale) i valori iniziali delle metriche **call_to_arrival** e **assignment_to_arrival** sono inizialmente relativamente bassi, circa 18-19, per poi avere un calo tra gli episodi 10 e 20 a valori di

circa 10-12 e infine aumentare nuovamente fino a valori di circa 20-22. Nel grafico di destra (progetto originale), invece, i valori iniziali delle metriche **call_to_arrival** e **assignment_to_arrival** sono più alti del progetto originale, circa 18-19, ma abbiamo un calo netto nei primi 10 episodi a valori di circa 13-15. In questo caso abbiamo eseguito solo 25 episodi in relazione alle limitazioni dei nostri strumenti.

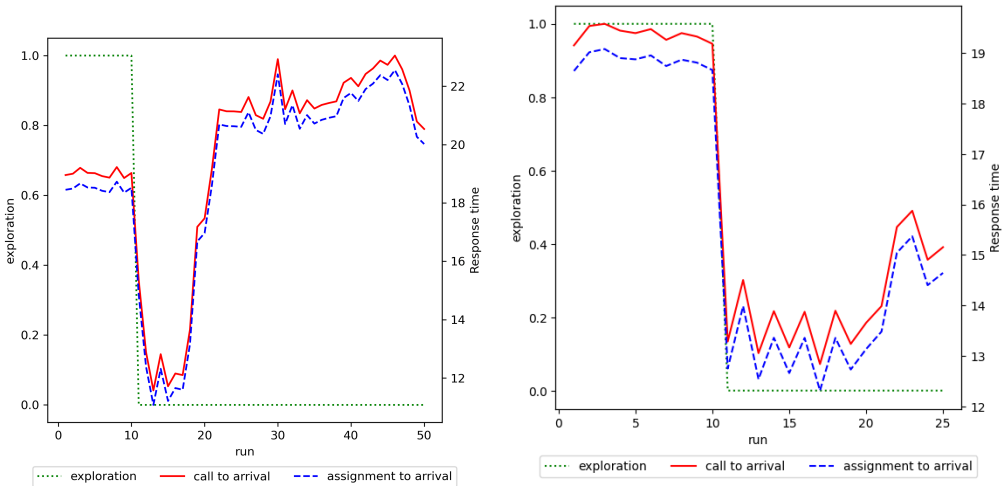

Figura 11: Rispettivamente i grafici relativi agli esperimenti del progetto originale e del progetto modificato con agente Noisy Bagging Duelling Prioritised Replay Double Deep Q e NUMBER_INCIDENT_POINTS=3.

Una volta completato l'addestramento, i modelli sono testati in uno scenario simulato. I risultati medi sono i seguenti:

- call_to_arrival: $\simeq$ 11.0 vs 12.4
- assignment_to_arrival: $\simeq$ 10.5 vs 11.9.

Discussione

Per osservare e analizzare le performance di tutti gli agenti testati, è stato prodotto un **boxplot** delle due metriche principali del sistema 12: call_to_arrival e assignment_to_arrival. Il boxplot è un tipo di visualizzazione che sintetizza la distribuzione di un insieme di dati attraverso cinque principali indicatori: linea centrale (mediana), box, 'whiskers' (baffi), outlier. Ogni boxplot mostra i tempi medi di risposta per ciascuna metrica suddivisi per i diversi agenti.

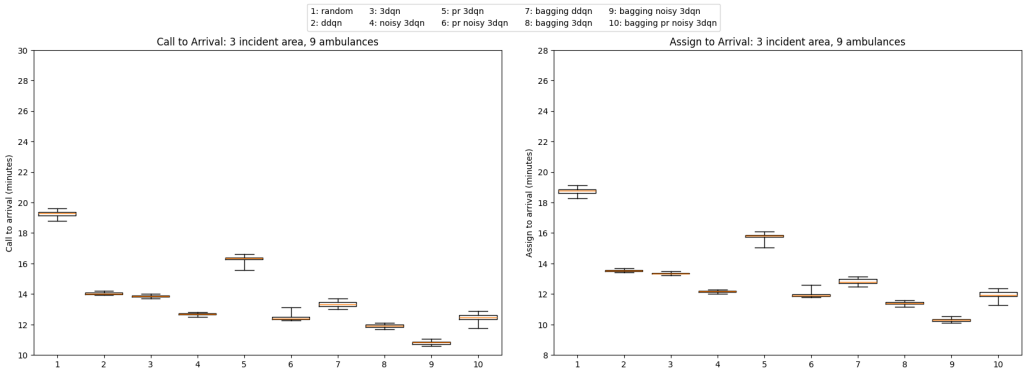

Figura 12: Performance degli agenti di RL testati nel progetto modificato.

La metrica call_to_arrival, ovvero il tempo dalla chiamata iniziale all'arrivo dell'ambulanza, evidenzia differenze significative tra gli agenti. L'agente 1, basato su un approccio casuale (random), ottiene le performance peggiori, con un tempo medio che si aggira attorno ai 20 minuti. Questo sottolinea l'inefficienza di un sistema senza logica o apprendimento. Al contrario, gli agenti basati su reinforcement learning (dal 2 al 10) mostrano un netto miglioramento, riducendo i tempi medi al di sotto dei 16 minuti. Tra questi, gli agenti 7, 9 e 10 si distinguono come i migliori, con tempi medi intorno ai 12-13 minuti e una distribuzione molto ristretta, che suggerisce una maggiore consistenza delle prestazioni. É interessante notare che l'agente 6, pur avendo un modello avanzato, ottiene un tempo leggermente più alto, intorno ai 15-16 minuti, suggerendo che l'introduzione del rumore nei dati potrebbe aver compromesso le prestazioni.

Per quanto riguarda la metrica assignment_to_arrival, ovvero il tempo tra l'assegnazione dell'ambulanza e il suo arrivo sul posto, lo schema è simile. Anche qui, l'agente random (1) presenta i tempi peggiori, con una media vicina ai 18 minuti, confermando la scarsa affidabilità di un sistema non ottimizzato. Gli agenti RL migliorano sensibilmente i risultati, portando i tempi al di sotto dei 14 minuti nella maggior parte dei casi. Ancora una volta, gli agenti 7, 9 e 10 emergono come i più performanti, con tempi medi di circa 12 minuti e una variabilità contenuta, che riflette una combinazione ottimale di velocità e stabilità operativa.

Nel complesso, i risultati indicano che gli agenti 7, 9 e 10 sono chiaramente i più efficaci per entrambe le metriche, risultando i migliori nel bilanciare la riduzione dei tempi con una consistenza nelle performance. Questo suggerisce che le strategie di bagging e l'uso di tecniche avanzate di apprendimento come il noisy 3dqn siano particolarmente promettenti.

In sintesi, mentre tutti gli agenti RL superano nettamente l'approccio

casuale, gli agenti che combinano tecniche avanzate (7, 9 e 10) rappresentano l'opzione migliore per ottimizzare il sistema di risposta alle emergenze.

Riferimenti bibliografici

[1] Michael Allen, Kerry Pearn e Tom Monks. «Developing an OpenAI Gym-compatible framework and simulation environment for testing Deep Reinforcement Learning agents solving the Ambulance Location Problem». In: *arXiv preprint arXiv:2101.04434* (2021).

[2] Hado Van Hasselt, Arthur Guez e David Silver. «Deep reinforcement learning with double q-learning». In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[3] Ziyu Wang et al. «Dueling network architectures for deep reinforcement learning». In: *International conference on machine learning*. PMLR. 2016, pp. 1995–2003.

[4] Meire Fortunato et al. *Noisy Networks for Exploration*. 2019. arXiv: 1706.10295 [cs.LG]. URL: https://arxiv.org/abs/1706.10295.

[5] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: 1511.05952 [cs.LG]. URL: https://arxiv.org/abs/1511.05952.

[6] Ian Osband et al. *Deep Exploration via Bootstrapped DQN*. 2016. arXiv: 1602.04621 [cs.LG]. URL: https://arxiv.org/abs/1602.04621.