Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Apprendimento con rinforzo applicato allo scheduling dei treni per la Flatland challenge

Relatore:
Chiar.mo Prof.
ANDREA ASPERTI

Presentata da:
DEVID FARINELLI

III Sessione
Anno Accademico 2018/2019

Abstract

La Flatland challenge è una competizione che ha come obiettivo incentivare la ricerca nell'ambito del reinforcement learning multi agente (MARL) applicato ai problemi di rescheduling (RSP). Lo scopo della sfida è sviluppare soluzioni per la gestione di una flotta di treni su una vasta rete ferroviaria, in modo che gli agenti si coordinino e collaborino per raggiungere ciascuno la propria destinazione nel minor tempo possibile, anche in caso si verifichino guasti temporanei. Il rescheduling è un problema complesso già affrontato in varie forme e con vari approcci, la Flatland challenge lo ripropone fornendo un ambiente per effettuare simulazioni del traffico ferroviario, con lo scopo di incentivare lo sviluppo di nuove soluzioni basate su reinforcement learning. In questo elaborato si affrontano i problemi di navigazione e rescheduling di treni posti dalla sfida, utilizzando un approccio basato sul reinforcement learning multi-agente (MARL). Viene descritto come, attraverso l'uso di tecniche di Deep Q-learning e una rappresentazione dello stato dell'ambiente sotto forma di bitmap, siano stati ottenuti risultati promettenti.

Indice

Introduzione **1**

1 Flatland **3**

1.1 La competizione . . . 3
1.2 Gli strumenti . . . 4
1.2.1 Ambiente . . . 4
1.2.2 Mappa . . . 5
1.2.3 Agenti . . . 6
1.2.4 Osservazioni . . . 8
1.2.5 Reward . . . 9

2 Stato dell'arte **11**

2.1 Vehicles Rescheduling Problem (VRSP) . . . 11
2.2 Reinforcement Leaning (RL) . . . 12
2.2.1 Q-Learning . . . 13
2.2.2 Deep Q-Learning . . . 13
2.2.3 Double Q-Learning . . . 14
2.2.4 Dueling Q-Learning . . . 14
2.3 RL applicato a VRSP . . . 15
2.3.1 Q-Learning . . . 16
2.3.2 Deep Q-Learning . . . 17
2.3.3 Pointer Network . . . 19

3 Baselines 21
3.1 Rappresentazione a Grafo 21
3.2 Bitmaps 22
3.3 Navigazione 24
3.3.1 Scelta delle Azioni 24
3.3.2 Prevenzione dei Crash 25
3.3.3 Gestione dei Ritardi 26
3.4 Reinforcement Learning 26
3.4.1 Agente 26
3.4.2 Generazione delle Osservazioni 26

4 Contributo Originale 29
4.1 Premesse 29
4.2 Contributo Algoritmico 30
4.2.1 Navigazione 30
4.2.2 Bitmaps 31
4.2.3 Altmaps 31
4.3 Reinforcement Learning 33
4.3.1 Heatmaps 34
4.3.2 Crash Penalty 35
4.3.3 Addestramento di Switch in Switch 35
4.3.4 Riordinamento dei Binari 36
4.4 Altro 36

5 Valutazione 39
5.1 Navigazione 40
5.2 Buffer Size 41
5.3 Crash Penalty 45
5.4 Switch in Switch 47
5.5 Velocità variabili 47
5.6 Malfunzionamenti 48
5.7 Numero di agenti 49

Conclusioni **53**

Elenco delle figure

1.1 Tipi di celle 6
1.2 Ingressi/uscite di una cella 6
1.3 Una mappa 20x20 7
1.4 Rappresentazione di un agente 8
1.5 Rappresentazione di un target 8
1.6 Le 3 osservazioni di Flatland 9

2.1 Interazione fra agente e ambiente [18] 12
2.2 Confronto delle architetture DQN e Dueling DQN 15
2.3 Layout di una rete a singolo binario con 3 stazioni e 5 sezioni 16
2.4 Schedule dei 3 treni sulla mappa in Figura 2.3. Ognuna delle righe colorate rappresenta il percorso di un treno, con il tempo sulle x e lo spazio sulle y. Le righe orizzontali (A, B, C) indicano 3 stazioni. 17
2.5 Interazione Agente (DQN) e Ambiente (PERT Graph) 18
2.6 Schedule rappresentato come grafico PERT 18
2.7 Modello della soluzione CVRP 20

3.1 Una bitmap 24
3.2 Rilevazione e risoluzione di un tamponamento usando le bitmap 25
3.3 Frontale su bitmap, l'agente 2 vuole andare sul binario in cui si trova l'agente 1 in direzione opposta 26

4.1 Una bitmap "bucata" a sinistra e una bitmap senza "buchi" a destra . . 33
4.2 Esempio di Heatmap generate per l'agente 1. Si nota che la bitmap dell'agente non viene considerata nella generazione delle heatmap 34

5.1 Due bitmap relative allo stesso percorso, con velocità diverse. Ad ogni bit $1/-1$ della bitmap sinistra, corrispondono 3 bit nella bitmap destra . . . 40
5.2 Media di agenti a destinazione, senza training, con epsilon decay a 0.9998 41
5.3 Media di agenti a destinazione, senza training e senza epsilon decay . . . 42
5.4 Valore di epsilon con decay 0.9998 42
5.5 Media di agenti a destinazione, variando la dimensione del buffer: 2k (in blu), 10k (in rosso), 25k (in grigio) interrotto dopo 5k episodi 43
5.6 Media di episodi conclusi, variando la dimensione del buffer: 2k (in blu), 10k (in rosso), 25k (in grigio) interrotto dopo 5k episodi 44
5.7 Media di agenti a destinazione, con crash penalty (in fucsia) e senza (in verde) interrotto dopo 17K episodi 45
5.8 Media di episodi conclusi, con crash penalty (in fucsia) e senza (in verde) interrotto dopo 17K episodi 46
5.9 Media di agenti a destinazione, approccio switch2switch (in arancione), senza switch2switch (in fucsia) e navigazione senza training (in blu) . . . 47
5.10 Media di agenti a destinazione, senza training (in fucsia), training con velocità diverse (in azzurro) 48
5.11 Media di agenti a destinazione, con malfunzionamenti (in verde) e senza (in blu) 49
5.12 Media di agenti a destinazione, con 50 agenti per 5K timestep 50
5.13 Esempio del crash più semplice fra due agenti (rosso e blu) non rilevabile dalle bitmap 51

Elenco delle tabelle

2.1 Legenda dello schedule rappresentato come grafico PERT in Figura 2.6 . 19

5.1 Confronto del variare del tempo e della percentuale di episodi conclusi, al variare delle dimensioni del buffer 44

5.2 Confronto del variare del tempo e della percentuale di episodi conclusi, in base all'utilizzo della crash penalty 46

5.3 Confronto del variare del tempo in base all'introduzione di malfunzionamenti 49

5.4 Confronto del variare del tempo e della percentuale di episodi conclusi al variare del numero di agenti 51

Introduzione

La Flatland challenge [15] è una competizione organizzata da AiCrowd e SBB nata per incentivare la ricerca nell'ambito del reinforcement learning multi agente (MARL) applicato ai problemi di rescheduling (RSP) [7]. La sfida affronta problematiche reali che le compagnie di trasporti e di logistica si trovano a risolvere ogni giorno nellla gestione dei flussi di veicoli. Nello specifico si vuole organizzare un gruppo di treni in modo da farli arrivare alla propria destinazione nel minor tempo possibile, anche nel caso in cui alcuni di essi subiscano dei guasti temporanei. Nonostante il focus della sfida fosse sull'utilizzare tecniche MARL, era ammesso qualunque tipo di approccio.

Fra le soluzioni in gara, quella basata su machine learning che ha raggiunto la posizione migliore non ha dato grandi risultati, classificandosi al 5 posto [11] con una media del 55% di agenti a destinazione. Anche lo stato dell'arte del reinforcement learning applicato a problemi di rescheduling di treni non offre molti spunti in quanto ancora molto limitato [10], [12], [16].

In questo elaborato si affrontano i problemi di navigazione e rescheduling di treni posposti dalla Flatland challenge, utilizzando un approccio basato sul reinforcement learning multi-agente (MARL).

Il Capitolo 1 è una introduzione alla Flatland challenge, sono descritti gli obiettivi della competizione, i concetti fondamentali e viene fatta una panoramica degli strumenti forniti a supporto della ricerca.

Nel Capitolo 2 è trattato lo stato dell'arte. Viene fatta un'introduzione al problema del rescheduling dei veicoli (VRSP), per poi passare alla descrizione dei principali algoritmi di reinforcement learning nell'ambito del Q-Learning (DQN, Double DQN e Dueling DQN). A conclusione del capitolo sono descritte le pubblicazioni esistenti riguardanti lo

stato dell'arte del reinforcement learning applicato a problemi di rescheduling.

Al Capitolo 3 è descritta la soluzione precedentemente sviluppata per la sfida, sulla quale si è basato il contributo originale descritto in questo elaborato. Sono approfonditi i dettagli sulla generazione della struttura a grafo della rete ferroviaria, le bitmap, la rete usata per prendere le decisioni e la prima versione dell'algoritmo di navigazione.

Il Capitolo 4 espone il contributo originale. Sono discusse le limitazioni della soluzione esistente e descritti gli algoritmi ed i concetti sviluppati. Nello specifico, la rivisitazione dell'algoritmo di navigazione e delle bitmap, l'introduzione delle altmap e delle heatmap, e l'implementazione di diverse strategie di addestamento (crash penalty, approccio switch to switch e riordinamento di binari)

Nel Capitolo 5 sono mostrati e discussi i risultati dei test effettuati.

Capitolo 1

Flatland

1.1 La competizione

La Flatland challenge è una competizione che ha come obiettivo incentivare la ricerca nell'ambito del reinforcement learning multi agente (MARL) applicato ai problemi di rescheduling (RSP). Lo scopo della sfida è quello di fare in modo che dei treni, su una vasta rete ferroviaria, si coordinino per da arrivare alla propria destinazione nel minor tempo possibile. La sfida pone diversi problemi da risolvere:

1. *Navigazione*, ogni agente deve essere in grado di navigare fino al proprio target

2. *Risoluzione dei conflitti*, gli agenti devono collaborare per evitare situazioni di crash o deadlock

3. *Rescheduling*, nel caso in cui un treno abbia un malfunzionamento, gli altri devono riorganizzarsi per minimizzare i ritardi

Ai partecipanti è stato messo a disposizione un ambiente 2D su cui fare simulazioni e sul quale sviluppare soluzioni per la gestione di un sistema complesso di trasporti. Nonostante il focus sia sul reinforcement learning è comunque ammesso qualunque approccio, come ad esempio soluzioni basate su machine learning o ricerca operativa.

1.2 Gli strumenti

Flatland è un set di strumenti che permette lo sviluppo di algoritmi di reinforcement learning multi agente su griglie bidimensionali, nel caso specifico della sfida l'ambiente utilizzato consente di simulare un rete ferroviaria su cui si muovono dei treni (agenti) che hanno come scopo raggiungere una stazione (target).

Come si vedrà successivamente l'ambiente di Flatland è completamente estensibile ed oltre ad esso sono disponibili varie utility di supporto allo sviluppo, come ad esempio strumenti per la visualizzazione delle simulazioni e funzioni di libreria come il calcolo dello shortest path.

Il codice di Flatland è interamente scritto in Python, open source e disponibile su Gitlab.

1.2.1 Ambiente

Il `RailEnv` è un ambiente 2D multi agente a tempo discreto, in cui gli agenti hanno l'obiettivo di navigare fino al proprio target nel minor tempo possibile, evitando incidenti e congestioni.

Alcuni parametri configurabili di un `RailEnv` sono:

- `width` e `height`, per impostare le dimensioni in celle della mappa
- `number_of_agents`, per definire il numero di agenti per ogni episodio

L'ambiente inoltre è molto estensibile, è infatti possibile sostituire o estendere componenti fondamentali riguardanti le simulazioni:

- `rail_generator`, è il componente che si occupa della generazione delle mappe, è quindi responsabile del layout della rete ferroviaria e delle posizioni degli agenti e dei target. Fra i generatori di default ci sono il `random_rail_generator` e lo `sparse_rail_generator` che genera layout realistici
- `schedule_generator`, lo schedule generator è il componente che, dato il layout della mappa, si occupa di assegnare a ciascun agente un target ed una velocità

- `obs_builder_object`, l'observation builder è il componente che si occupa di generare le osservazioni dell'environment
- `malfunction_generator_and_process_data`, è il componente che si occupa della generazione dei malfunzionamenti degli agenti

In Flatland il tempo è discreto, ossia avanza di un'unità di tempo (timestep) solo quando viene eseguita la funzione `env.step(...)` usata come segue:

```
obs, rewards, dones, info = env.step(railenv_action_dict)
```

La funzione `step` prende come parametro un dizionario che associa ad ogni agente l'azione da compiere, e restituisce diverse informazioni:

- `obs`, le osservazioni dell'ambiente generate dall'`observation_builder`
- `rewards`, un dizionario che associa una reward a ciascun agente
- `dones`, un dizionario che indica con un booleano quali agenti sono arrivati a destinazione. Una chiave aggiuntiva, `'__all__'`, viene impostata a `True` quando tutti gli agenti sono arrivati a destinazione
- `info`, un dizionario contenente informazioni sugli agenti, un dato particolarmente utile è la chiave `'action_required'` che ha valore `True` quando l'agente ha la possibilità di scegliere una azione

È inoltre disponibile una libreria per la visualizzazione delle simulazioni, in modo da poter vedere graficamente la mappa, i target e gli agenti che si spostano sulla rete ferroviaria. Un esempio del risultato è la Figura 1.3.

1.2.2 Mappa

La mappa è una griglia bidimensionale dove ad ogni posizione corrisponde una cella. Ogni cella ha una capacità unitaria, quindi in un determinato istante di tempo può ospitare al più un agente. Per costruire la rete ferroviaria, a ciascuna cella viene associato un dei tipi in Figura 1.1 che possono essere ruotati di 0, 90, 180 o 270 gradi.

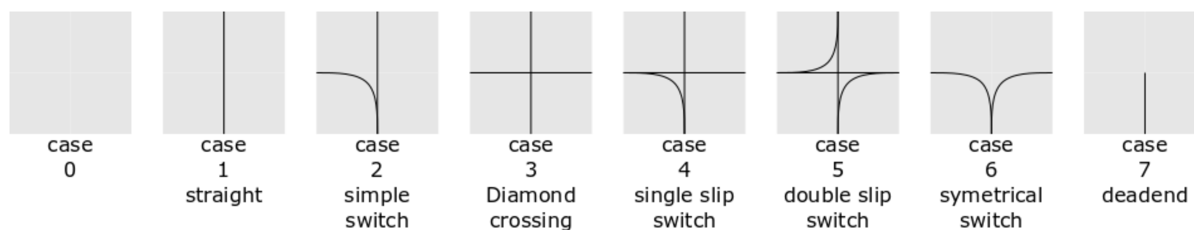

Figura 1.1: Tipi di celle

Una cella può avere al più 4 ingressi/uscite identificati con i 4 punti cardinali (vedi Figura 1.2). Questa informazione, insieme al tipo della cella, è utile per definire quali sono le transizioni possibili per un agente in base al punto cardinale di ingresso alla cella.

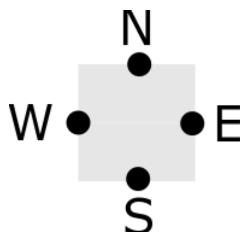

Figura 1.2: Ingressi/uscite di una cella

1.2.3 Agenti

Un agente è definito come un'entità che ha come scopo raggiungere il proprio target nel minor tempo possibile muovendosi sulla griglia. Ogni agente ha un id progressivo, una velocità, uno stato e un determinato insieme di azioni fra cui scegliere.

Azioni

I movimenti sulla griglia sono effettuati spostandosi fra celle adiacenti, limitatamente alle transazioni possibili, che sono definite dal tipo di cella e dal punto cardinale di ingresso. Ogni agente ha a disposizione 5 possibili azioni per muoversi nella mappa, le azioni non valide non hanno effetto:

- `DO_NOTHING` (0), se l'agente è fermo rimane fermo, se invece era in movimento continua a muoversi

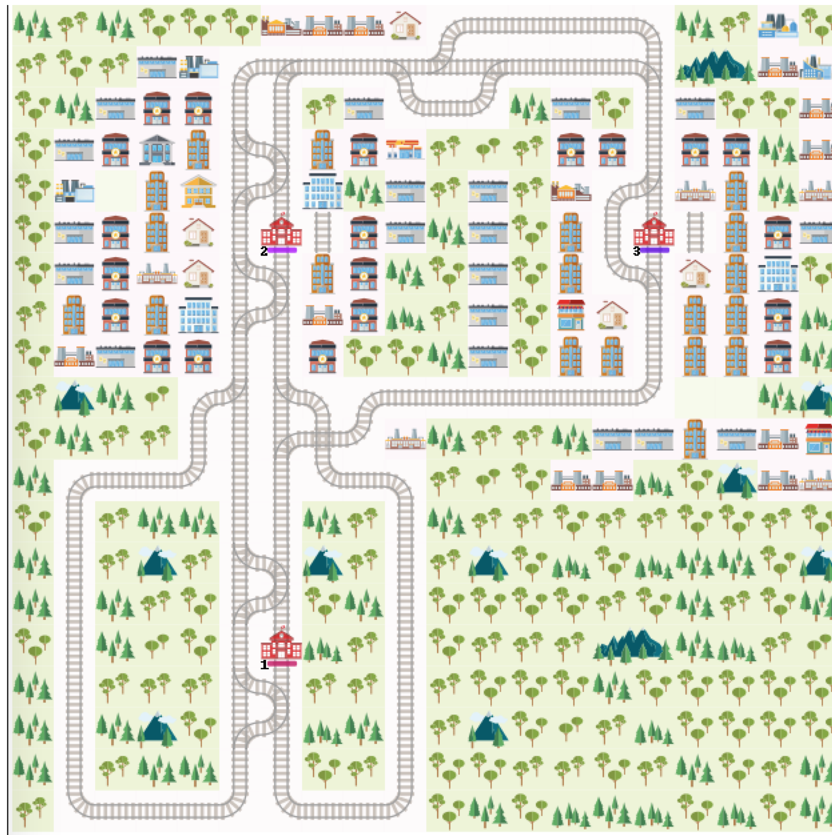

Figura 1.3: Una mappa 20x20

- `MOVE_LEFT` (1), allo switch curva a sinistra, se l'agente era fermo inizia a muoversi
- `MOVE_FORWARD` (2), allo switch vai dritto, se l'agente era fermo inizia a muoversi
- `MOVE_RIGHT` (3), allo switch curva a destra, se l'agente era fermo inizia a muoversi
- `STOP_MOVING` (4), ferma un agente in movimento

Stato

Ogni agente ha un attributo che conserva un'informazione sullo stato, i valori possibili sono:

- `READY_TO_DEPART` (0), l'agente non è ancora partito quindi non si trova ancora sul binario iniziale

- `ACTIVE` (1), l'agente è attivo e si trova su un binario
- `DONE` (2), l'agente è arrivato a destinazione, ma si trova ancora sul binario di arrivo
- `DONE_REMOVED` (3), l'agente è arrivato a destinazione ed è stato rimosso dalla mappa

Velocità

Per simulare la realtà delle diverse tipologie di treni, ciascun agente si muove ad una velocità variabile pre-assegnata che può essere unitaria (1 cella per timestep) o frazionaria (es. 1/2, 1/3, 1/4 di cella per timestep). Indipendentemente dalla velocità, gli agenti sono tenuti a scegliere quale azione eseguire al primo timestep di ingresso in una cella. Nel caso in cui un agente decida di fermarsi gli verrà permesso di scegliere una nuova azione al timestep successivo.

Figura 1.4: Rappresentazione di un agente

Figura 1.5: Rappresentazione di un target

1.2.4 Osservazioni

Le osservazioni sono delle rappresentazioni dell'ambiente, ossia un mezzo per codificare le informazioni rilevanti. Flatland mette a disposizione 3 tipi di osservazioni mostrate in Figura 1.6:

- *Globale*, un'osservazione che riguarda tutta la mappa. La rappresentazione è costituita da diverse strutture dati di dimensioni `map_height x map_width` e un numero di canali dipendente dalle informazioni da codificare (es. posizione degli agenti, direzione, ecc...)
- *Locale*, una osservazione che riguarda la porzione di mappa intorno ad un agente. La rappresentazione è costituita da diverse strutture dati di dimensioni

```
(2 * view_radius) x (2 * view_radius)
```

e un numero di canali dipendente dalle informazioni da codificare (es. posizione degli agenti, direzione, ecc...)

- *Albero*, un tipo di osservazione generata sfruttando la struttura a grafo della rete ferroviaria. Dalla posizione dell'agente si genera un albero di grado 4, dove i nodi corrispondono agli switch e gli archi corrispondono ai binari, uno per ciascun punto cardinale della cella contenente lo switch.

Per i dettagli specifici si rimanda a [13].

Il modo in cui sono generate e codificate le osservazioni è uno fra i punti più importanti della sfida, viene infatti messo in palio anche un premio per chi condivide con la comunity l'osservazione migliore. Le caratteristiche principali di una buona osservazione sono l'efficacia durante training e l'efficienza computazionale nel generarle.

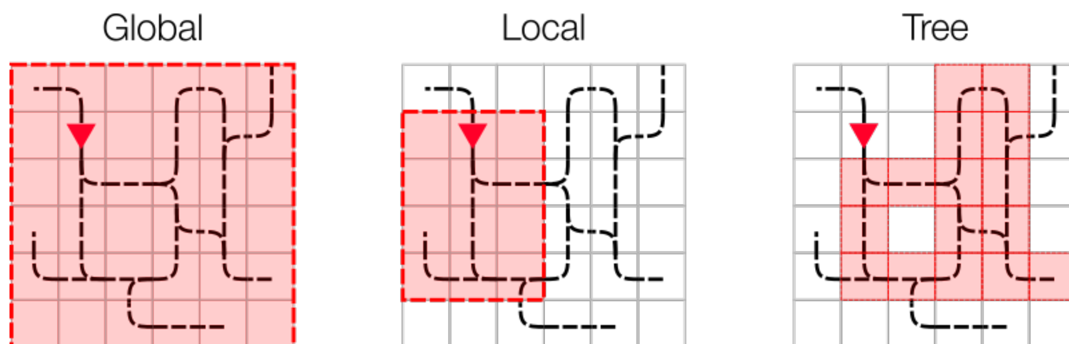

Figura 1.6: Le 3 osservazioni di Flatland

1.2.5 Reward

Le reward in Flatland sono parametrizzate con `alpha` e `beta` entrambe impostate a 1.0 come valore di default. Le altre possibili reward dell'ambiente sono le seguenti:

- `invalid_action_penalty = 0`, penalità per azioni non permesse, disabilitata di default

- `step_penalty = -1 * alpha`, la penalità ad per ogni step dell'ambiente
- `stop_penalty = 0`, penalità per aver fermato un agente in movimento, disabilitata di default
- `start_penalty = 0`, penalità per aver messo in movimento un agente fermo, disabilitata di default
- `global_reward = 1 * beta`, reward per gli agenti che arrivano a destinazione

Capitolo 2

Stato dell'arte

Il rescheduling di treni (VRSP) è un problema molto complesso e già affrontato in varie forme e con vari approccci, la Flatland challenge ripropone il problema fornendo un ambiente di simulazioni per incentivare lo sviluppo di soluzioni basate sul reinforcement leaning. In questo capitolo viene descritto il problema del rescheduling, viene fatta una breve introduzione al reinforcement learning e si analizza lo stato dell'arte di tecniche di RL applicate a problemi di rescheduling.

2.1 Vehicles Rescheduling Problem (VRSP)

Il problema del rescheduling di veicoli (VRSP) [7] è un problema di ottimizzazione che consiste nel riorganizzare un flusso di veicoli nel caso in cui si verifichino imprevisti, come un guasto o una congestione del traffico. Esistono diverse varianti del problema, generalmente si ha uno schedule iniziale che per qualche ragione viene invalidato e deve essere ricalcolato, minimizzando per esempio il ritardo dei veicoli. Diverse ricerche hanno evidenziato come il problema del rescheduling dei treni sia NP-arduo [14] e anche NP-completo [2], di conseguenza si ha che lo spazio delle soluzioni cresce esponenzialmente con con le dimensioni del problema, rendendo spesso non applicabili soluzioni basate su programmazione dinamica. Per questa ragione le principali soluzioni sfruttano metodi euristici come ad esempio approcci basati su: depth-first search [6], branch and bound

[3], algoritmi genetici [4], ecc. Diversamente, le soluzioni che sfruttano reinforcement learning sono ancora poche nonostante la tecnologia sia molto promettente.

2.2 Reinforcement Leaning (RL)

Il reinforcement learning è un'area del machine learning che consiste nell'elaborazione di modelli per la risoluzione di problemi di decision making. Diversamente dal supervised learning non si lavora con un dataset etichettato contenente le risposte al problema, ma si raffinano i modelli tramite tentativi ripetuti di un task, con lo scopo di ricavare una strategia che permetta di massimizzare i reward ottenuti. Il reinforcement learning è il risultato dell'interazione fra due elementi, l'agente e l'ambiente (Figura 2.1).

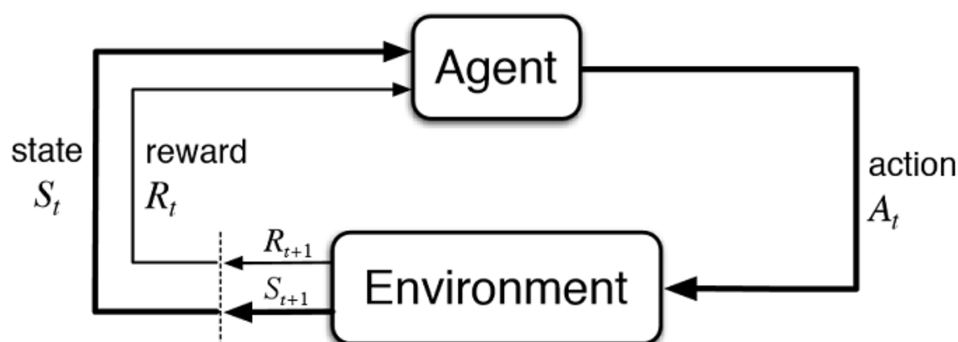

Figura 2.1: Interazione fra agente e ambiente [18]

L'ambiente guida l'agente verso l'apprendimento della strategia ottimale di scelta delle azioni, dando ricomponse o penalità che incoraggiano o disincentivano le azioni prese dall'agente.

L'agente invece è un algoritmo che tramite ripetute interazioni con l'ambiente elabora il modello decisionale, detto *policy* o *funzione decisionale*, solitamente implementata come rete neurale.

2.2.1 Q-Learning

Il Q-learning [18] è una tecnica di Reinforcement Learning che approssima la policy ottima q_* di un agente tramite una funzione action-value $Q(s, a)$. La funzione Q da una stima di quanto sia positivo eseguire una azione a quando ci si trova nello stato s e viene quindi usata per scegliere l'azione che si pensa essere la migliore. L'aggiornamento di Q viene effettuato come segue:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[y_i - Q(s_t, a_t)] \tag{2.1}$$

$$y_i = r_{t+1} + \gamma arg \max_a Q(s_{t+1}, a)) \tag{2.2}$$

dove:

- y_i, è il target dell'apprendimento
- r_t è la reward dell'ambiente all'istante t
- α detto *learning rate*, indica il peso dato alla nuove esperienza rispetto a quelle esistenti
- γ detto *discount factor*, bilancia l'importanza delle reward immediate e quelle future

Nel Q-Learning la funzione Q viene implementata come una tabella che contiene i Q-value di tutte le possibili combinazioni di s ed a, tuttavia se lo spazio degli stati è molto ampio, questo approccio diventa oneroso in termini di memoria, se non addirittura inapplicabile.

2.2.2 Deep Q-Learning

Il Deep Q-Learning (DQN) [9] sfrutta le Deep Neural Network, efficaci nell'approssimazione di funzioni, per approssimare Q. Si parametrizza quindi la funzione Q con una rete neurale θ:

$$y_i = r_{t+1} + \gamma arg \max_a Q(s_{t+1}, a; \theta)) \tag{2.3}$$

Per migliorare la stabilità dell'apprendimento si sfrutta l'*experience replay* [8], una tecnica in cui l'agente accumula un dataset di tuple di esperienza da vari episodi, e nell'aggiornare i pesi della rete viene fatto sampling dal dataset generato invece che utilizzare l'esperienza più recente. Questa tecnica è molto utile poiché consente di decuplicare la correlazione temporale che c'è fra gli stati successivi di una traiettoria.

Si usano anche strategie di scelta delle azioni ϵ-greedy per bilanciare l'esplorazione (*exploration*) di nuovi stati e l'utilizzo della policy (*exploitation*).

2.2.3 Double Q-Learning

Nella formula 2.2 si nota che *max* usa la funzione Q sia nella valutazione che nella scelta dell'azione, di conseguenza introducendo un bias che potrebbe portare a sovrastimare l'utilità di un'azione favorendone la scelta.

Per risolvere il problema si è dimostrato essere efficace utilizzare due reti separate. In Double DQN [5], viene utilizzata una rete per la valutazione e una rete diversa per la scelta dell'azione. Il target dell'addestramento viene riformulato come segue:

$$y_i = r_{t+1} + \gamma Q(s_{t+1}, arg\max_a Q(s_{t+1}, a; \theta_t); \theta_t^-) \tag{2.4}$$

Dove θ è la rete *online* usata anche in DQN per fare la scelta dell'azione, mentre θ^- è la rete target usata per stimare il valore corretto dell'azione.

2.2.4 Dueling Q-Learning

In Dueling DQN [20] la funzione Q viene scomposta nelle funzioni state-value V e action advantage A.

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|A|}\sum_{a=1}^{|A|} A(s,a) \tag{2.5}$$

La rete invece che imparare direttamente la funzione Q, calcola le funzioni V ed A, dove:

- $V(s)$ detta funzione *state-value*, indica quanto sia positivo trovarsi in un particolare stato s

- $Q(s, a)$, misura il valore di scegliere una determinata azione a quando ci si trova nello stato s

- $A(s, a)$ è la funzione *action advantage*, corrisponde a $Q - V$ e fornisce una misura relativa dell'importante dell'azione a

A differenza di DQN in cui la funzione Q viene aggiornata per una specifica coppia stato-azione, nel Dueling DQN si aggiorna la funzione V che può dare un beneficio ad altre azioni. Intuitivamente, questa architettura è in grado di apprendere quali sono gli stati rilevanti in cui è importante prendere una decisione, piuttosto che imparare l'effetto di ogni azione per ogni stato. Questa struttura è compatibile con le tecniche di learning esistenti (es. SARS) ed ha mostrato effettivi miglioramenti nelle performance.

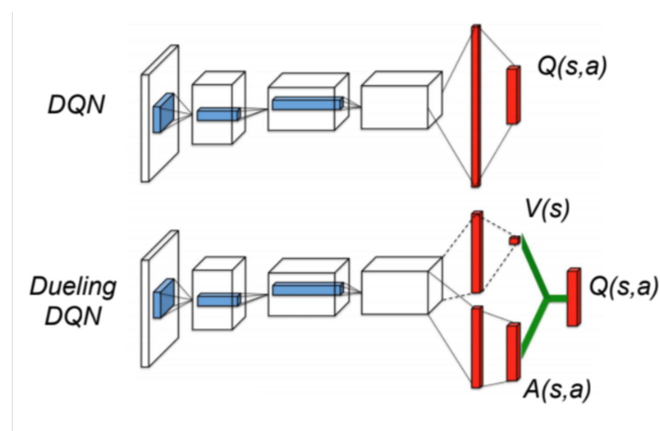

Figura 2.2: Confronto delle architetture DQN e Dueling DQN

2.3 RL applicato a VRSP

Nonostante il reinforcement learning abbia un grosso potenziale nell'ambito dei problemi di rescheduling, lo stato dell'arte attuale è molto limitato. Un motivo potrebbe essere la varietà dei problemi di rescheduling che potrebbe portare a favorire soluzioni ad-hoc per una precisa istanza del problema, piuttosto che soluzioni generalizzate. Un'altra limitazione potrebbe essere la mancanza di uno standard comune. Flatland, come Gym di OpenAI, permette ai ricercatori di avere un ambiente che prima non esisteva,

per sviluppare e testare le proprie soluzioni. In questo senso la challenge di Flatland ha dato un contributo portando l'attenzione sul problema e sviluppando tool per rendere più semplici e confrontabili le ricerche.

2.3.1 Q-Learning

In "Reinforcement learning approach for train rescheduling on a single-track railway" [16] viene utilizzato il Q-Learning per risolvere il problema del rescheduling.

Le caratteristiche dell'ambiente sono molto simili a quelle Flatland: ogni agente ha una velocità diversa e la rete ferroviaria è divisa in sezioni con capacità unitaria. Diversamente da Flatland, ogni mappa ha un solo binario che si dirama nei pressi delle stazioni come in Figura 2.3, e le azioni "Stop" e "GO" non vengono assegnate ai treni ma ai semafori in ingresso e in uscita da ogni sezione.

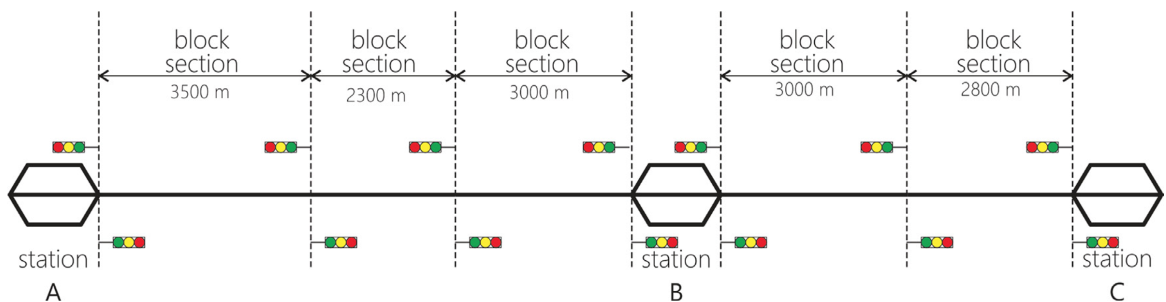

Figura 2.3: Layout di una rete a singolo binario con 3 stazioni e 5 sezioni

Viene usato il Q-Learning come descritto nella Sezione 2.2.1. Lo spazio degli stati corrisponde al prodotto cartesiano delle informazioni rilevanti ossia: la posizione del treno, espressa come il segmento in cui si trova, le sezioni disponibili su cui il treno può spostarsi ed il tempo. Lo spazio delle azioni ha dimensione 2, poichè le azioni possibili per ogni segmento sono "Stop" oppure "Go". La reward è calcolata alla fine dell'episodio ed è la somma di reward negative per il ritardo accumulato più una reward positiva per l'arrivo a destinazione. Inoltre le azioni che causano deadlock sono rilevate e ricevono una reward negativa molto alta.

Per fare il training viene considerato uno schedule come quello in Figura 2.4, viene aggiunto ritardo ad un agente ed eseguito per 50 step l'algoritmo di apprendimento.

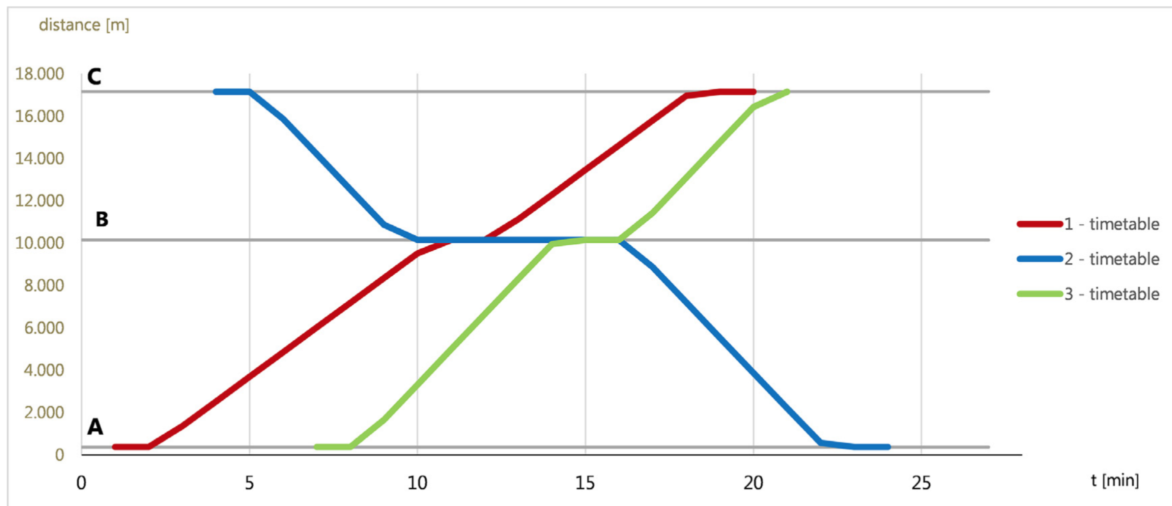

Figura 2.4: Schedule dei 3 treni sulla mappa in Figura 2.3. Ognuna delle righe colorate rappresenta il percorso di un treno, con il tempo sulle x e lo spazio sulle y. Le righe orizzontali (A, B, C) indicano 3 stazioni.

Per lo stato dell'arte attuale questa pubblicazione è la più simile al problema posto dalla Flatland challenge, nonostante sia relativa ad una versione molto semplificata del problema (single rail) e che non affronti la navigazione.

2.3.2 Deep Q-Learning

La pubblicazione "Deep Reinforcement Learning Approach for Train Rescheduling Utilizing Graph Theory" [12] affronta il problema del rescheduling usando DQN e una rappresentazione dell'ambiente come grafo PERT.

Come mostrato in Figura 2.5, l'agente è un classico agente DQN mentre l'osservazione (lo schedule) è rappresentato come grafo PERT, dove ogni nodo rappresenta l'estremo di un viaggio (partenza o arrivo) e ogni riga rappresenta una stazione (vedi Figura 2.6).

L'agente è implementato come DQN agent con un neurone per ogni nodo del grafo PERT. Il grafo in Figura 2.6 rappresenta lo schedule di due agenti su due binari, con 3 stazioni e gli eventuali ritardi. Intuitivamente la rete prende in input il grafico PERT a cui è stato aggiunto un delay e restituisce in output un grafo distorto che rappresenta

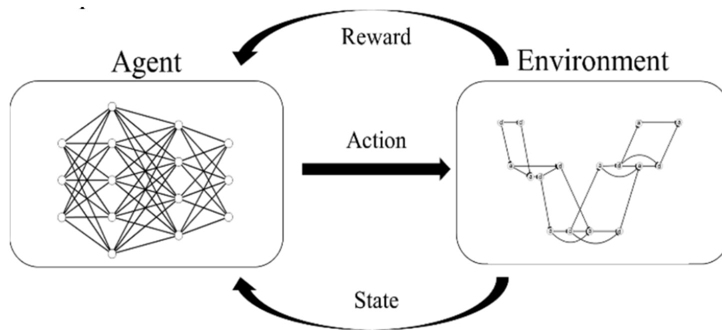

Figura 2.5: Interazione Agente (DQN) e Ambiente (PERT Graph)

il nuovo schedule. La valutazione della soluzione viene fatta attraverso la stima della soddisfazione dei passeggeri, ossia quanto ritardo fa ciascun treno.

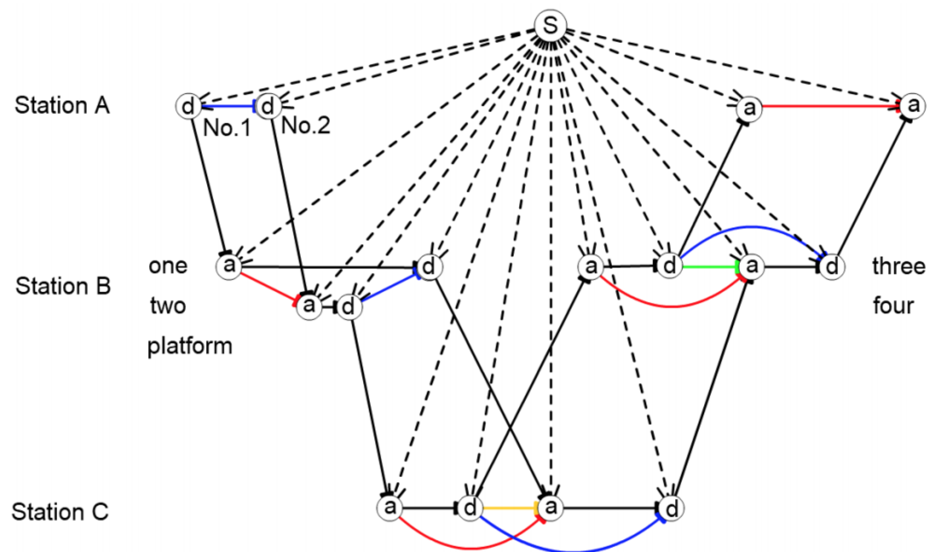

Figura 2.6: Schedule rappresentato come grafico PERT

La soluzione proposta in [12] non è applicabile alla Flatland challenge poichè tratta il rescheduling ma non la navigazione, ed inoltre non propone una soluzione scalabile poiché la rete neurale deve avere un neurone per ogni nodo del grafo PERT.

Edge Type	Weight and meaning	Edges in Fig. 3
Running	minimum time required for trains to run from one station to the next	Black (d to a)
Stoppage	minimum time required for the passengers to get on and off the train safely	Black (a to d)
Turning	minimum time required for trains to shuttle continuously	Black at last stations (a to d)
Departure order	minimum time required time for the departure of trains bound for the same station continuously	Blue
Arrival order	minimum time required for the arrival at the same station continuously	Red
Same platform	minimum time required for same-bound trains to use the same platform continuously	Green
Cross hindrance	minimum time required time for the departure or arrival at the last station continuously	Orange
Schedule	edges from starting node to each node – weight is schedule time of each event	Dotted black

Tabella 2.1: Legenda dello schedule rappresentato come grafico PERT in Figura 2.6

2.3.3 Pointer Network

In "Reinforcement Learning for Solving the Vehicle Routing Problem" [10] si affronta il problema del Capacited Vehicle Routing Problem (CVRP) [17], ossia si hanno uno o più agenti che si occupano di prelevare un carico da un deposito per consegnarlo ai rispettivi clienti. Non si tratta di un problema di rescheduling, ma la natura è molto simile, si vuole infatti generare uno schedule degli agenti in modo da ottimizzare i viaggi di ciascuno.

Per risolvere il problema viene usata una Pointer Network [19] con un meccanismo di attenzione. Il modello mostrato in Figura 2.7 prende come input X_t ossia lo stato dell'ambiente all'istante t. Ogni X_t è una sequenza di $x_i = (s_i, d_i)$, dove con s sono indicate le componenti statiche, ossia le coordinate di ciascuna consegna da effettuare, mentre con d le componenti dinamiche, ossia la domanda di merce che varia da step a step. Ad ogni passo viene generata in output una y_t, che sarà il puntatore ad uno degli input X nello step successivo. Il processo continua finché tutte le richieste di merce sono

state soddisfatte.

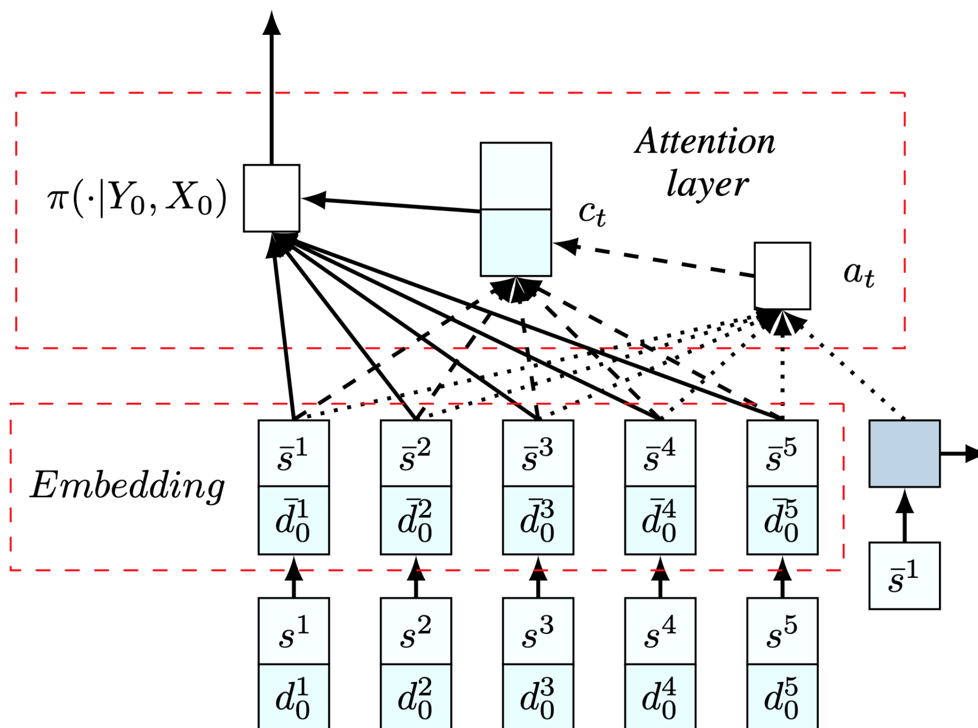

Figura 2.7: Modello della soluzione CVRP

Il modello in Figura 2.7 è implementato usato un layer per il Graph Embedding di X_t ad un vettore, ed un decoder RNN (sulla destra) che punta un input per ogni istante di tempo. Ad ogni step i viene utilizzato un meccanismo di attenzione che estrae le informazioni rilevanti dall'input scegliendo quali verrano utilizzate per lo step successivo. L'addestramento avviene con tecniche classiche di policy gradient.

La soluzione proposta in [19] non è tuttavia applicabile alla Flatland challenge poichè non tratta un problema di rescheduling, ma di un problema routing capacitivo e poichè non supporta scenari multi-agente.

Capitolo 3

Baselines

La prima versione del codice [1] ha definito la struttura base del progetto e implementato delle funzioni fondamentali, come la generazione della rappresentazione a grafo della rete ferroviaria e l'approccio basato su bitmap. Siccome il contributo originale descritto in questo elaborato si è basato sul prezioso materiale pre-esistente, viene descritto in questo capitolo tutto il codice ed i concetti prodotti precedentemente da altri per la risoluzione del problema posto dalla Flatland challenge.

3.1 Rappresentazione a Grafo

Per realizzare la soluzione basata su bitmaps, approfondita nella prossima sezione, è necessario avere una struttura dati che codifichi il layout della rete ferroviaria. Si vuole quindi ottenere una rappresentazione come grafo direzionato, in cui ogni nodo corrisponde ad uno switch ed ogni arco ad un binario che collega due switch (sorgente e destinazione):

1. Si identificano gli switch: partendo dalla mappa di un episodio rappresentata come tabella di transizione, si controllano tutte le celle le cui transizioni permettono ad un agente in ingresso di scegliere fra più percorsi in uscita

2. Per ciascuno dei nodi identificati si percorrono tutte le direzioni possibili e per ciascuna di esse si ricostruisce l'arco, percorrendo tutte le celle consecutive finché non si raggiunge un altro switch

Durante i passaggi descritti, l'algoritmo popola le seguenti strutture dati usate nell'implementazione delle soluzioni:

- `nodes`, una lista contenente tutti gli id dei nodi
- `edges`, un lista contenente tutti gli id degli archi
- `id_node_to_cell`, un dizionario che associa a ciascun `node_id` le coordinate `(x, y)` della cella a cui corrisponde nella mappa
- `cell_to_id_node`, un dizionario inverso al precedente, associa alle coordinate `(x, y)` di ciascun switch, l'id del nodo corrispondente nella rappresetazione a grafo
- `id_edge_to_cells`, un dizionario che associa a ciascun `edge_id` una lista contenente le coordinate di ciascuna cella di cui si compone l'arco
- `info`, un array in cui ogni elemento corrisponde ad un arco e ne codifica le informazioni fondamentali (sorgente, destinazione e lunghezza):

```
((src_id, cardinal_point), (dst_id, cardinal_point), edge_length)
```

3.2 Bitmaps

Una bitmap è una struttura dati che rappresenta il percorso che un agente farà per arrivare al proprio target. Si tratta di una matrice che ha per righe gli id delle rotaie e per colonne i timestep, ciascuna cella indica quindi su quale binario si troverà l'agente ad un determinato istante temporale. Le bitmap sono generate come segue:

1. Viene calcolato lo *shortest path* dell'agente, usando una funzione di libreria Flatland che restituisce il percorso più breve fra la posizione dell'agente e il proprio target. Lo shortest path è rappresentato da una lista in cui ogni elemento è un `WalkingElement` che si può riassumere come una tupla `(position, action)` che indica la posizione corrente dell'agente e l'azione da eseguire

2. Gli shortest path sono trasformati in *predictions*, ossia degli shortest path che tengono in considerazione la velocità dell'agente. Una prediction è una lista di elementi `(timestep, position, action)` che per agenti con velocità frazionarie vede il ripetersi di tuple consecutive in cui avanza il timestep ma la posizione dell'agente resta invariata.

3. Come ultimo passo le prediction sono trasformate in bitmap:

   (a) Si crea una tabella con tante righe quanto il numero massimo di binari (es. 100), tante colonne quante la profondità desiderata (es. 500) e tutte le celle inizializzate a 0

   (b) Per gli agenti che devono ancora partire si lascia la prima colonna vuota, quindi con tutti i valori a 0

   (c) Si prende il primo elemento della prediction e dalle coordinate della cella si risale all'id del binario

   (d) Se la cella considerata risulta non essere un binario ma uno switch si lascia vuota la colonna corrispondente, quindi con tutti i valori a 0

   (e) Si calcola la direzione in cui viene percorso il binario corrente e la si inserisce nella bitmap:

   ```
   bitmaps[agent_id, rail, timestep] = direction
   ```

   È possibile calcolare la direzione di percorrenza sfruttando la struttura a grafo presentata nella sezione 3.1. Ogni agente che attraversa un arco andando dal nodo sorgente al nodo destinazione avrà come direzione `1`, altrimenti avrà come direzione `-1`

La figura 3.1 mostra la bitmap di un agente in attesa di partire, che quindi ha tutta la prima colonna a 0. L'agente percorrerà 3 timesteps sulla rail 2, successivamente percorrerà la rail 1 per 2 timesteps in direzione contraria al senso del binario e infine si sposterà sulla rail 4 per 3 timesteps. Si noti che ai timestep 4 e 7, tutti i valori sono a 0, poiché l'agente si troverà su uno switch.

```
                    TIMESTEPS
               0  1  2  3  4  5  6  7  8  9 10
             _________________________________
          0 |  0  0  0  0  0  0  0  0  0  0  0
          1 |  0  0  0  0  0 -1 -1  0  0  0  0
RAILS     2 |  0  1  1  1  0  0  0  0  0  0  0
          3 |  0  0  0  0  0  0  0  0  0  0  0
          4 |  0  0  0  0  0  0  0  0  1  1  1
```

Figura 3.1: Una bitmap

3.3 Navigazione

Le bitmaps sono alla base dell'algoritmo di navigazione, in questa sezione si vedrà come sono usate per il training della rete e lo scheduling del traffico.

3.3.1 Scelta delle Azioni

Nel percorso dal target alla destinazione un agente deve effettuare delle scelte, tuttavia la maggior parte del tempo si limiterà a seguire il binario su cui si trova. Per capire quando è il momento di effettuare una scelta è possibile guardare la bitmap di un agente:

1. Quando le colonne `0` e `1` sono uguali, vuol dire che l'agente sta continuando sulla stessa rail, quindi l'azione da eseguire sarà `MOVE_FORWARD`

2. Quando le colonne `0` e `1` differiscono, vuol dire che l'agente sta per attraversare uno switch che lo porterà su un'altra rail, quindi deve scegliere che azione eseguire

La scelta dell'azione è effettuata calcolando lo shortest path dell'agente, usando la funzione di utility di Flatland e recuperando dalla tupla che si trova come primo elemento l'azione da eseguire. Se l'azione scelta prevede uno spostamento dell'agente, le bitmap sono aggiornate tramite shift di un bit verso sinistra scartando la prima colonna.

3.3.2 Prevenzione dei Crash

Una volta scelta l'azione da eseguire si controlla che essa non provochi un incidente (crash) con un altro treno. I crash possono essere di due tipi:

- *Tamponamento*, può avvenire quando un treno veloce si accoda sullo stesso binario ad un treno più lento. Se il timestep stimato di uscita dal binario del treno veloce, è minore o uguale a quello del treno più lento, si verificherà un tamponamento. Il treno più veloce viene quindi ritardato, allungando nella bitmap il timestep di uscita dal binario. Vedi figura 3.2.

- *Frontale*, avviene quando un agente fa una scelta che lo porta su un binario su cui si trova un treno in direzione opposta. Per evitare il frontale l'agente si ferma (`action = STOP_MOVING`) e riparte al timestep successivo a quello di uscita dell'altro treno. Vedi figura 3.3.

```
                AGENTE 1 (LENTO)                     AGENTE 2 (VELOCE)
            0  1  2  3  4  5  6  7  8  9 10      0  1  2  3  4  5  6  7  8  9 10
          _________________________________    _________________________________
        0 | 0  0  0  0  0  0  0  0  1  1  1  0 | 0  0  0  0  0  0  0  0  0  0  0
        1 | 0  0  0  0  0  0 -1  0  0  0  0  1 | 0  0  0  0  0  0  0  0  0  0  0
RAILS   2 | 1  1  1  1  1  0  0  0  0  0  0  2 | 0  1  1  0  0  0  0  0  0  0  0
        3 | 0  0  0  0  0  0  0  0  0  0  0  3 | 0  0  0  0 -1 -1 -1  0  0  0  0
        4 | 0  0  0  0  0  0  0  0  0  0  0  4 | 0  0  0  0  0  0  0  0  1  1  1

                                                    AGENTE 2 CON RITARDO
                                                 0  1  2  3  4  5  6  7  8  9 10
                                               _________________________________
                                             0 | 0  0  0  0  0  0  0  0  0  0  0
                                             1 | 0  0  0  0  0  0  0  0  0  0  0
                                             2 | 1  1  1  1  1  1  0  0  0  0  0
                                             3 | 0  0  0  0  0  0  0 -1 -1 -1  0
                                             4 | 0  0  0  0  0  0  0  0  0  0  0
```

Figura 3.2: Rilevazione e risoluzione di un tamponamento usando le bitmap

```
                AGENTE 1 (SUL BINARIO)                          AGENTE 2 (IN INGRESSO)
              0  1  2  3  4  5  6  7  8  9 10             0  1  2  3  4  5  6  7  8  9 10
            _________________________________           _________________________________
        0 |   0  0  0  0  0  0  0  0  0  0  0       0 |   0  0  0  0  0  0  0  0 -1 -1 -1
        1 |   0  0  0  0  0  0  0  0  0  0  0       1 |   0  0  0  0  1  1  1  0  0  0  0
RAILS   2 |   1  1  0  0  0  0  0  0  0  0  0       2 |   0 -1 -1  0  0  0  0  0  0  0  0
        3 |   0  0  0  1  1  1  1  0  0  0  0       3 |   0  0  0  0  0  0  0  0  0  0  0
        4 |   0  0  0  0  0  0  0  0 -1 -1 -1       4 |   0  0  0  0  0  0  0  0  0  0  0
```

Figura 3.3: Frontale su bitmap, l'agente 2 vuole andare sul binario in cui si trova l'agente 1 in direzione opposta

3.3.3 Gestione dei Ritardi

Gli agenti effettuano una scelta quando si trovano nella cella prima di uno switch, nel caso in cui l'azione decisa sia quella di fermarsi è necessario ritardare la programmazione dei treni che seguono. Per farlo si considerano tutti i treni che si trovano sullo stesso binario occupato dall'agente (che quindi avranno `1/-1` nella cella `rail x 0`) e si allunga la bitmap di un bit a tutti quelli che prevedono di uscire nello stesso istante o prima dell'agente che si è fermato.

3.4 Reinforcement Learning

3.4.1 Agente

L'agente è un DQNAgent fornito con le baselines di Flatland, la struttura è quella di una Dueling Double DQN con experience replay e strategia di scelta delle azioni ϵ-greedy.

L'agente ha un action space di dimensione 2, infatti ha a disposizione come azioni solamente 0 e 1 che corrispondono a Stop e Go.

3.4.2 Generazione delle Osservazioni

Per effettuare il training è necessario avere una rappresentazione dello stato della simulazione da passare alla rete. Queste rappresentazioni sono chiamate osservazioni e dato l'id di un agente sono generate come segue:

1. Si calcolano gli agenti più conflittuali, ossia quelli che secondo le bitmap, in un determinato istante di tempo, si troveranno sullo stesso binario con la stessa direzione dell'agente di cui si sta generando l'osservazione
2. Una volta ottenuto l'elenco, si ordinano gli agenti in base al numero di sovrapposizioni e si selezionano i 10 più conflittuali
3. Per costruire le osservazioni si concatenano, sull'asse delle x, le bitmap degli agenti più conflittuali alla bitmap dell'agente di cui si sta calcolando l'osservazione

Capitolo 4

Contributo Originale

In questo capitolo è descritto il contributo originale sviluppato partendo dai concetti e dal codice descritti nel capitolo precedente. Il contributo originale si può dividere in due parti:

- *Contributo all'Algoritmo*, ossia il lavoro svolto per migliorare l'algoritmo di navigazione sulla rete ferroviaria
- Contributo nell'ambito del *reinforcement learning*, ossia come si è sfruttato l'apprendimento a supporto dell'algoritmo di navigazione

4.1 Premesse

La soluzione descritta nel Capitolo 3 presentava diversi problemi:

- Il codice era stato sviluppato in maniera incrementale aggiungendo progressivamente concetti nuovi, era quindi necessario un refactoring della struttura e dei concetti per semplificare il codice, ottimizzarlo e renderlo meno incline a bug
- L'approccio iniziale si basava interamente sulle bitmap, esse infatti venivano usate sia per generare le osservazioni per la rete, sia per capire quando far scegliere un'azione ad un agente e sia per rilevare possibili crash. Tuttavia la complessità del problema e del codice non permettevano di garantire l'attendibilità delle bitmap,

che spesso perdevano di sincronizzazione con l'ambiente rendendo completamente inefficace la navigazione per un episodio

- Essendo una versione iniziale poco testata erano presenti alcuni bug che non consentivano la corretta esecuzione di una simulazione o di un training

4.2 Contributo Algoritmico

4.2.1 Navigazione

La navigazione è stata profondamente modificata, come è possibile vedere all'Algoritmo 2. Nella prima versione, la navigazione era basata sul confronto dei bit della colonna `0` ed `1` di una bitmap per capire quando l' agente dovesse fare una scelta. Questo approccio tuttavia era dipendente dalla sincronizzazione delle bitmap con l'effettiva posizione dell'agente, e non teneva in considerazione di vari casi particolari che andavano gestiti a parte. Nel corso dello sviluppo si è quindi reso necessario basare la navigazione sulle informazioni date dall'ambiente, in particolare:

- Lo *stato* dell'agente, ossia se deve ancora partire (`READY_TO_DEPART`) o se è arrivato a destinazione (`DONE` o `DONE_REMOVED`)

- La *posizione* della prossima cella, ossia la coppia di coordinate in cui si troverà l'agente al timestep successivo. Grazie a questa informazione è possibile capire se l'agente si trovi prima di uno switch

- `info[a]['action_required']`, è una informazione restituita dall'`env.step()` che indica se l'ambiente si aspetta che l'agente `a` scelga un'azione. Generalmente il valore sarà `True` al primo timestep in cui un agente entra in una nuova cella

Sfruttando le informazioni precedenti è possibile capire con precisione quando un agente deve prendere una decisione, indipendentemente dalla velocità a cui si muove.

Rilevamento di Crash

Una volta scelta l'azione è necessario verificare che possa essere eseguita senza causare incidenti. A questo scopo è stata sviluppata la funzione `check_crash` che rileva frontali e tamponamenti ed è implementata come segue:

Algorithm 1: Rilevamento di Crash

```
if status == READY_TO_DEPART then
    if agent.initial_position not occupata then
        Si controlla che non si provochi un frontale sul nuovo binario
    end if
else if l'agente si trova prima di uno switch then
    Si controlla che non si provochi un frontale sul nuovo binario
else
    Si controlla che la prossima cella non sia già occupata
    evitando un tamponamento
end if
```

4.2.2 Bitmaps

Nella versione originale le bitmap erano "bucate", ossia si avevano colonne piene di 0 in corrispondenza dei timestep in cui un agente si trovava su uno switch. Questa soluzione era la più semplice da implementare ma non la più corretta da utilizzare nel training della rete neurale. Si è quindi deciso di completare le bitmap, associando uno switch (o un percorso di switch) alla rail successiva come mostrato in figura 4.1

4.2.3 Altmaps

Una limitazione della versione iniziale era che gli agenti avevano a disposizione un solo percorso (lo shortest path) e questo implicava che in caso di conflitto (es. deadlock) non ci fosse alcuna soluzione. Inoltre risulta necessario che ciascun agente possa decidere fra più percorsi alternativi per consentire un'effettiva dinamicità dello scheduling.

Algorithm 2: Algoritmo di Navigazione

```
if status == DONE or status == DONE_REMOVED then
  action ← DO_NOTHING
else if status == READY_TO_DEPART then
  network_action ← la rete sceglie Go(1) o Stop(0)
  if network_action == STOP then
    action ← DO_NOTHING
  else
    if check_crash() then
      action ← DO_NOTHING
    else
      action ← MOVE_FORWARD
      shift della bitmap
    end if
  end if
else if agente si trova prima di uno switch and deve scegliere una azione then
  network_action ← la rete sceglie Go(1) o Stop(0)
  if network_action == STOP then
    action ← DO_NOTHING
  else
    if check_crash() then
      action ← STOP_MOVING
    else
      action ← azione prevista dallo shortest path
      shift della bitmap
    end if
  end if
else if agente deve scegliere una azione then
  if check_crash() then
    action ← STOP_MOVING
  else
    action ← azione prevista dallo shortest path
    shift della bitmap
  end if
else
  action ← DO_NOTHING
end if
```

BITMAP "BUCATA"

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	**-1**	**-1**	0	0	0	0
2	0	**1**	**1**	**1**	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	**1**	**1**	**1**

BITMAP SENZA "BUCHI"

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	**-1**	**-1**	**-1**	0	0	0	0
2	0	**1**	**1**	**1**	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**

Figura 4.1: Una bitmap "bucata" a sinistra e una bitmap senza "buchi" a destra

È stato quindi sviluppato l'approccio ad *Altmaps*, ossia quando un agente si trova davanti ad uno switch, viene generato lo shortest path per ciascuna delle transizioni possibili. Dagli shortest path alternativi, chiamati *altpath*, si generano le relative bitmap, chiamate altmap. Per ciascuna altmap si genera un'osservazione, che viene passata alla rete che restituisce dei Q-value (uno per stop ed uno per go). Viene quindi eseguita l'azione con il Q-value più alto e il relativo altpath viene scelto come percorso dell'agente.

Per generare gli altpath è stata modificata la funzione di Flatland `get_shortest_paths` in modo da rilevare tutti i possibili percorsi di uscita da un gruppo di switch e generare un percorso per ciascuno. Infatti non è sufficiente calcolare solamente i percorsi di uscita dal primo switch, ma è necessario considerare anche altri eventuali switch adiacenti, che potrebbero aumentare il numero di scelte possibili. Nel caso in cui l'azione scelta dall'agente sia Stop (0), al timestep successivo gli sarà permesso di rieffettuare la scelta, questo finché non decide di avanzare (1) su un determinato percorso. Per efficienza, gli altpath e le altmap per uno switch sono generate una sola volta e poi salvate per essere eventualmente riutilizzate in caso l'agente debba rieffettuare la scelta.

4.3 Reinforcement Learning

L'approccio originale all'apprendimento della prevenzione degli incidenti era basato sull'identificazione di un set di agenti con cui era più probabile avere un conflitto durante la navigazione. Questa soluzione aveva delle limitazioni, ossia che era basata sull'euristica di selezione degli agenti più conflittuali e forniva alla rete solo una parte delle informazioni

relative all'ambiente. Oltre a sviluppare un approccio diverso ad `heatmap` sono state implementate altre soluzioni con lo scopo di favorire l'apprendimento.

4.3.1 Heatmaps

Per fornire alla rete un'osservazione complessiva del percorso di tutti gli agenti di una simulazione sono state sviluppate le `heatmap`. Questo approccio si basa sul generare due heatmap da appendere alla bitmap di un agente, costruite sommando tutte le bitmap degli altri agenti. Siccome una bitmap ha componenti positive (1) e negative (-1) si è deciso di generare due heatmap, una per la somma di tutte le componenti positive, e una per quelle negative. Il concetto alla base è che si vogliono generare delle mappe che codifichino informazioni su come il traffico sia concentrato in base all'istante di tempo, al binario e alla direzione di percorrenza. L'obiettivo è quello di consentire alla rete di capire come distribuire gli agenti sui binari in maniera più efficace.

Figura 4.2: Esempio di Heatmap generate per l'agente 1. Si nota che la bitmap dell'agente non viene considerata nella generazione delle heatmap

4.3.2 Crash Penalty

Oltre che per la navigazione, la funzione di detection dei crash, descritta nella sezione sul contributo all'algoritmo, fornisce informazioni su quali siano le azioni da non eseguire. Quando viene rilevato un crash infatti, si può assegnare una reward negativa molto alta all'azione, penalizzando la scelta con lo scopo di renderla meno probabile in futuro. La penalità è assegnata in questo modo:

```
dqn.step(curr_obs, 1, -100, curr_obs, True)
```

La tupla aggiunta `(q_0, action, reward, q_1, done)`, ha come stati `q_0` e `q_1` due osservazioni uguali, `done = True` poiché l'episodio si considera concluso e una penalità molto alta `-100`.

Questo comportamento non è attivo di default, ma si può abilitare tramite il parametro `--crash-penalty`.

4.3.3 Addestramento di Switch in Switch

L'approccio granulare di switch in switch ha lo scopo di migliorare l'apprendimento uniformando gli stati che vengono passati alla rete, l'idea è quella che ogni bitmap sia relativa ad uno stato in cui l'agente si trova prima di uno switch. Normalmente la funzione `dqn.step` viene eseguita come segue:

```
dqn.step(curr_obs, network_action, reward, next_obs, done)
```

Dove:

- `curr_obs` è l'osservazione precedente alla scelta dell'azione
- `next_obs` è l'osservazione successiva alla scelta dell'azione, se l'azione scelta è Go, corrisponde alla bitmap precedente shiftata verso sinistra
- `network_action` è l'azione scelta dalla rete (0 o 1)
- `reward` e `done` sono le informazioni restituite dall'ambiente

Tuttavia in questo modo le due osservazioni passate sono molto simili e la rete potrebbe non essere in grado di capire quali siano gli effetti a lungo termine della scelta appena fatta. La soluzione sviluppata è stata quella di aggiungere esperienza in maniera più "granulare", considerando solo stati di switch in switch, ossia:

- `curr_obs` resta l'osservazione precedente alla scelta dell'azione
- Come `next_obs` viene utilizzata l'osservazione in cui l'agente si trova prima dello switch successivo
- `reward` diventa la ricompensa cumulata calcolata nel percorso fra i due switch

Questo comportamento non è attivo di default, ma si può abilitare tramite il parametro `--switch2switch`.

4.3.4 Riordinamento dei Binari

Ad ogni episodio viene generata una mappa che ha una rete ferroviaria diversa, e ad ogni layout corrisponde una numerazione diversa dei binari. Questo implica che ad ogni episodio le bitmap generate abbiano caratteristiche che dipendono dalla mappa in questione. La seguente soluzione è stata quindi sviluppata per facilitare alla rete la comprensione delle bitmap rimuovendo una proprietà non rilevante che è l'ordinamento dei binari.

L'algoritmo di riordinamento su due step: inizialmente ordina i binari relativamente all'ordine di attraversamento dell'agente. Successivamente riordina i restanti binari, poiché la permutazione va applicata anche alle heatmap. Resta tuttavia ancora da definire una strategia coerente per il riordinamento dei binari non compresi nel percorso dell'agente.

Questo comportamento non è attivo di default, ma si può abilitare tramite il parametro `--reorder-rails`.

4.4 Altro

Inoltre sono state aggiunte delle utility di supporto allo sviluppo:

- `--plot`, per generare i grafici per tensorboard
- `--profile`, per far stampare un profilo dell'esecuzione del codice, utile per capire dove il programma spende la maggior parte del tempo
- `--load-path`, per caricare i pesi di una rete preaddestrata

Capitolo 5

Valutazione

In questo capitolo vengono esposti i risultati ottenuti testando le varie soluzioni sviluppate. La strategia usata è stata quella di eseguire i primi test su una versione rilassata del problema, in modo da individuare gli approcci più efficaci per poi valutarli successivamente su simulazioni più realistiche.

Il range di parametri su cui testare ai fini della sfida è il seguente:

- *Dimensioni della mappa*, `width` e `height`: comprese fra 20 e 150
- *Numero di agenti*, `number_of_agents`: compreso fra 50 e 200
- *Numero di città*, `max_num_cities`: compreso fra 2 e 35
- *Massimo numero di binari in città*, `max_rails_between_cities`: compreso fra 2 e 4
- *Frequenza dei malfunzionamenti*, `malfunction_rate`: compreso fra 500 e 2000
- *Durata dei malfunzionamenti*, `min_duration` e `max_duration`: compreso fra 20 e 80
- *Velocità degli agenti*: equamente distribuite fra 1, 1/2, 1/3 e 1/4

Per la soluzione rilassata, oltre a ridurre sensibilmente le dimensioni della mappa e il numero di agenti, si è deciso di impostare la stessa velocità per tutti i treni. Come

mostrato in Figura 5.1, la velocità di un agente influisce sulla sua bitmap che, a parità di percorso, avrà una sequenza di $1/-1$ più lunga, minore è la sua velocità. Si è deciso quindi di usare solo velocità unitarie per evitare alla rete l'ulteriore onere di inferire la velocità dell'agente dalla bitmap. I parametri del problema semplificato sono:

- *Dimensioni della mappa*: $20x20$
- *Numero di agenti*: 4
- *Numero di città*: 3
- *Massimo numero di binari in città*: 2
- *Frequenza dei malfunzionamenti*: 0, non ci sono malfunzionamenti
- *Velocità degli agenti*: 1, tutti gli agenti hanno velocità unitaria

```
                   VELOCITÀ = 1                                VELOCITÀ = 1/3
          0  1  2  3  4  5  6  7  8  9 10            0  1  2  3  4  5  6  7  8  9 10
        _________________________________          _________________________________
      0 | 0  0  0  1  1  1  1  0  0  0  0       0 | 0  0  0  0  0  0  0  0  0  1  1
      1 | 1  0  0  0  0  0  0  0  0  0  0       1 | 1  1  1  0  0  0  0  0  0  0  0
RAILS 2 | 0 -1 -1  0  0  0  0  0  0  0  0       2 | 0  0  0 -1 -1 -1 -1 -1 -1  0  0
      3 | 0  0  0  0  0  0  0  0  0  0  0       3 | 0  0  0  0  0  0  0  0  0  0  0
      4 | 0  0  0  0  0  0  0 -1 -1 -1 -1       4 | 0  0  0  0  0  0  0  0  0  0  0
```

Figura 5.1: Due bitmap relative allo stesso percorso, con velocità diverse. Ad ogni bit $1/-1$ della bitmap sinistra, corrispondono 3 bit nella bitmap destra

5.1 Navigazione

Non essendoci grafici relativi alle performance dell'algoritmo di navigazione delle baseline, per avere un paragone per le valutazioni successive è stato eseguito un test senza training. Come si vede in Figura 5.2 la media degli agenti arrivati a destinazione si aggira intorno al 50% fino al timestep 4000. Successivamente il valore cala poichè a causa dell'epsilon decay (Figura 5.4) viene data priorità alle decisioni della rete, che non essendo addestrata prende scelte inefficaci.

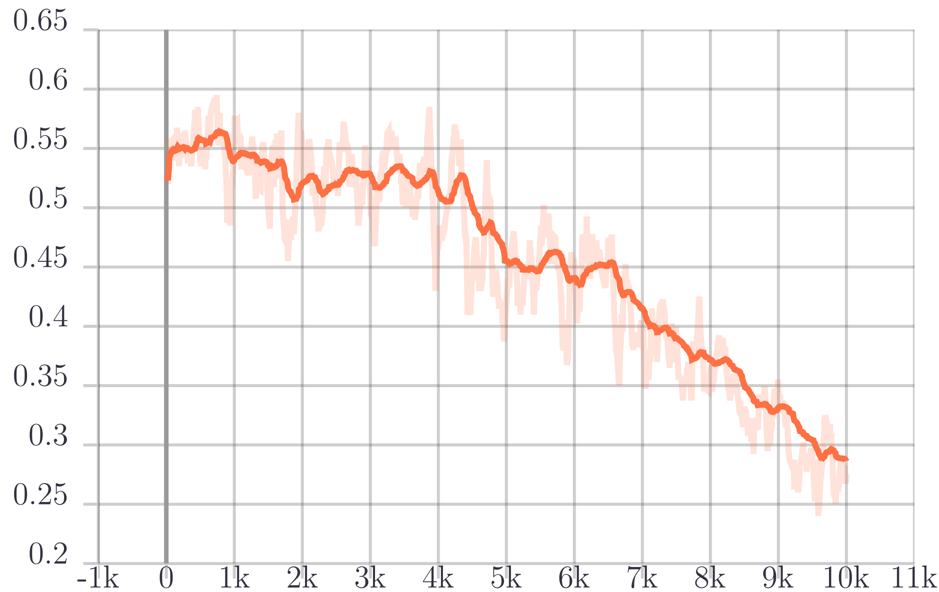

Figura 5.2: Media di agenti a destinazione, senza training, con epsilon decay a 0.9998

Nel test in Figura 5.3 è stato utilizzato un epsilon con valore 1 senza decay, tutte le scelte sono quindi casuali. Questa policy randomica ha dato un risultato stabile intorno al 55% di agenti a destinazione, e sarà utilizzata come base per valutare le soluzioni implementate.

5.2 Buffer Size

Sono stati effettuati diversi test per valutare l'impatto delle dimensioni del buffer sull'efficacia del training. Il risultato è mostrato in Figura 5.5 e i valori utilizzati sono:

- *2k*, il training non ha avuto alcun impatto rispetto al test senza training in Figura 5.2, la dimensione del buffer risulta probabilmente troppo limitata
- *10k*, la percentuale media di agenti a destinazione ha raggiunto circa il 70%, con picchi all'80%
- *25k*, il training ha dato i risultati peggiori, misurando addirittura un peggioramento rispetto alla soluzione senza training, con valori intorno al 40%

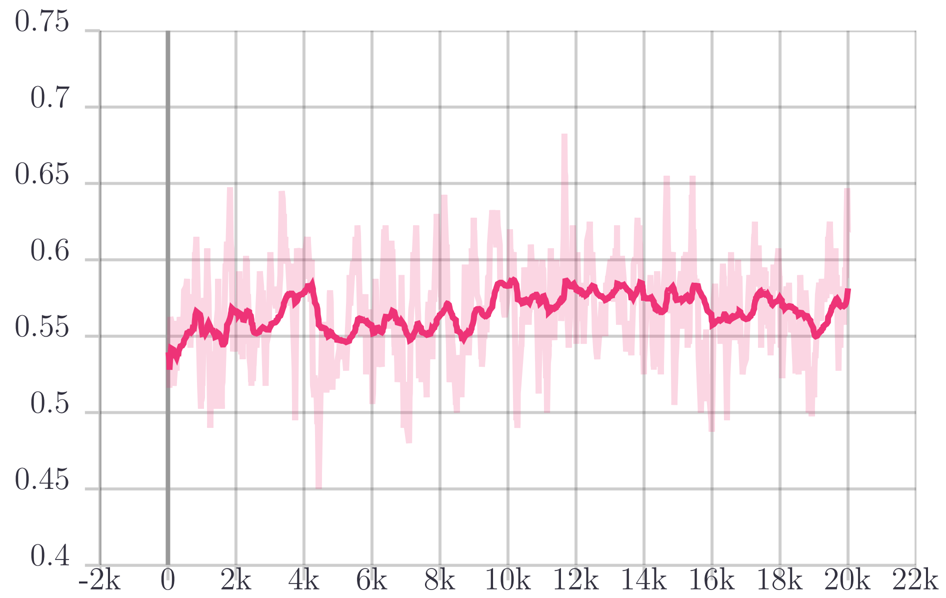

Figura 5.3: Media di agenti a destinazione, senza training e senza epsilon decay

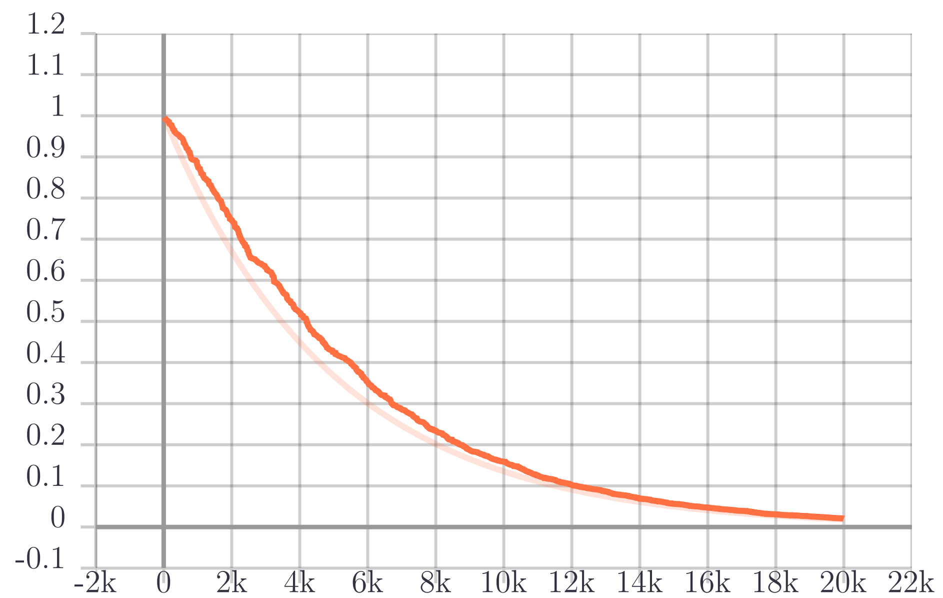

Figura 5.4: Valore di epsilon con decay 0.9998

- *1M* è il valore di default di dimensione del buffer. Non è stato possibile generare risultati poichè la memoria necessaria era superiore a quella dell'elaboratore, causando la terminazione dello script per `Out Of Memory` al superamento dei 50GB di memoria virtuale disponibili

La dimensione del buffer a 10k è risultata essere la più efficace, ed è quindi stata usata nei test successivi.

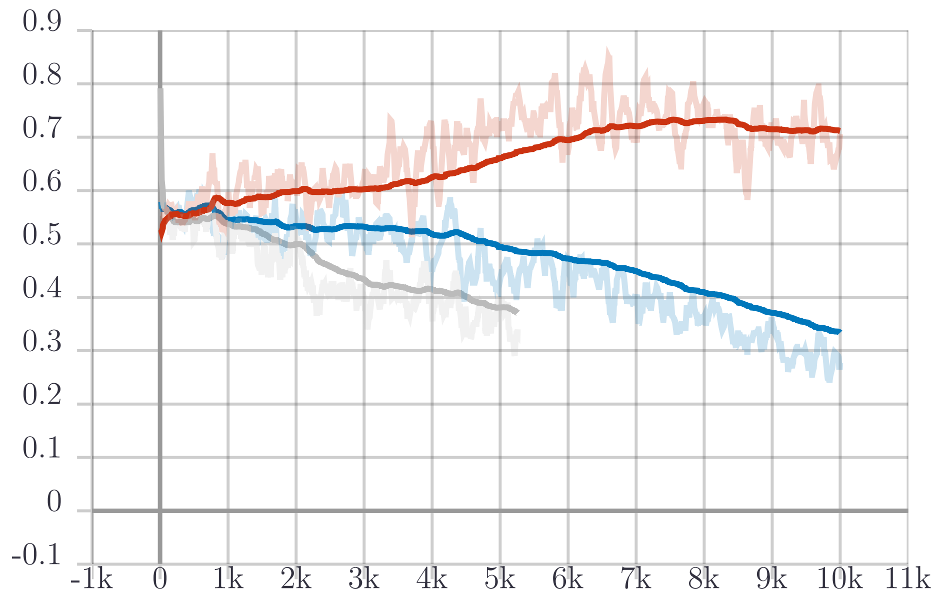

Figura 5.5: Media di agenti a destinazione, variando la dimensione del buffer: 2k (in blu), 10k (in rosso), 25k (in grigio) interrotto dopo 5k episodi

Si nota inoltre dalla Tabella 5.2, che all'aumentare del buffer size aumenta anche il tempo necessario per eseguire lo stesso numero di episodi (5K). L'aumento del tempo di computazione può essere dovuto a due fattori:

- Un aumento del tempo richiesto per la computazione (generazione altpaths, bitmap, navigazione e training)
- Una maggiore frequenza di episodi in cui uno o più agenti non arrivano a destinazione, terminando al raggiungimento del massimo numero di timesteps. Intuitivamente, se tutti gli agenti arrivano a destinazione l'episodio si conclude prima.

Buffer Size	Tempo per 5K episodi	Media Episodi Conclusi
2K	18h	45%
10K	19h	70%
25K	39h	37%

Tabella 5.1: Confronto del variare del tempo e della percentuale di episodi conclusi, al variare delle dimensioni del buffer

Tuttavia in Figura 5.6 si nota come la percentuale di episodi conclusi usando un buffer da 25K sia molto vicina a quella dei 2K, confermando quindi che la variazione di tempo richiesto per le simulazioni, sia effettivamente dovuta all'aumento della dimensione del buffer, e non alla durata degli episodi.

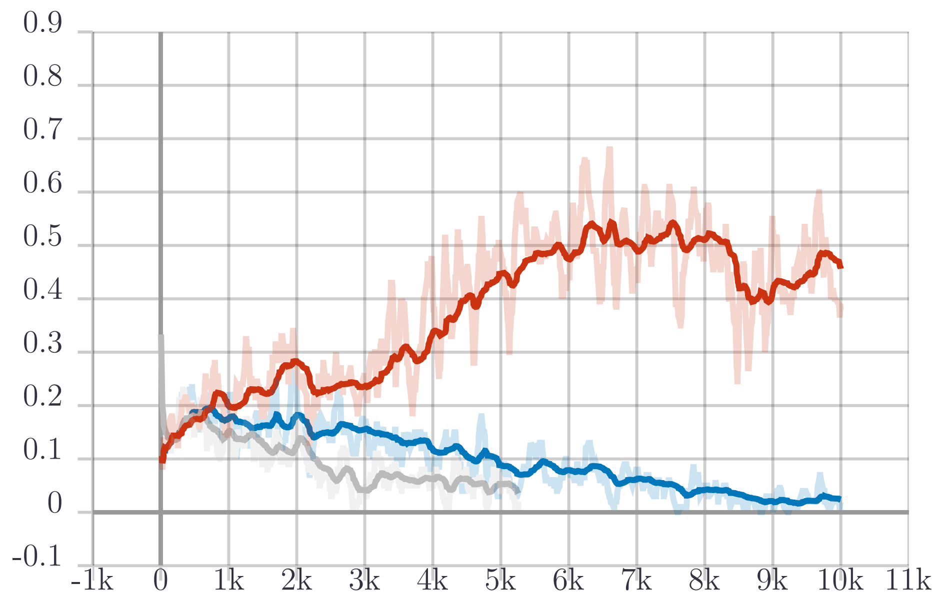

Figura 5.6: Media di episodi conclusi, variando la dimensione del buffer: 2k (in blu), 10k (in rosso), 25k (in grigio) interrotto dopo 5k episodi

5.3 Crash Penalty

Si è testato l'impatto della penalità in caso di crash, come si può osservare in Figura 5.7 aggiungere esperienza negativa in caso di crash ha dato risultati negativi. La valutazione senza penalità ha registrato il punteggio migliore, con una media di agenti a destinazione intorno al 90% dopo 4K timestep. Aggiungendo la penalità si ha inizialmente il 50% degli agenti a destinazioni, che diminuisce progressivamente insieme all'epsilon, risultato che potrebbe essere dovuto ad un overfitting di reward negative.

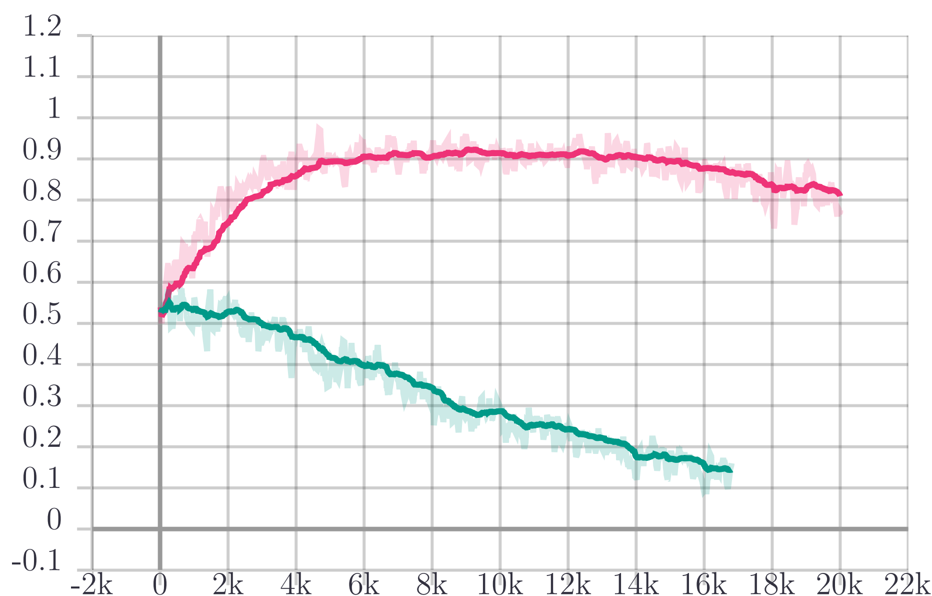

Figura 5.7: Media di agenti a destinazione, con crash penalty (in fucsia) e senza (in verde) interrotto dopo 17K episodi

Si nota dalla Tabella 5.3, che la soluzione senza crash penalty richiede circa 1/3 del tempo rispetto a quella con penalità. Tuttavia la differenza di tempo è probabilmente dovuta alla percentuale di episodi completata, che per il test con penalità è intorno al 2% (vedi Figura 5.8).

	Tempo per 10K episodi	Media Episodi Conclusi
Senza Crash Penalty	10h	83%
Con Crash Penalty	27h	2%

Tabella 5.2: Confronto del variare del tempo e della percentuale di episodi conclusi, in base all'utilizzo della crash penalty

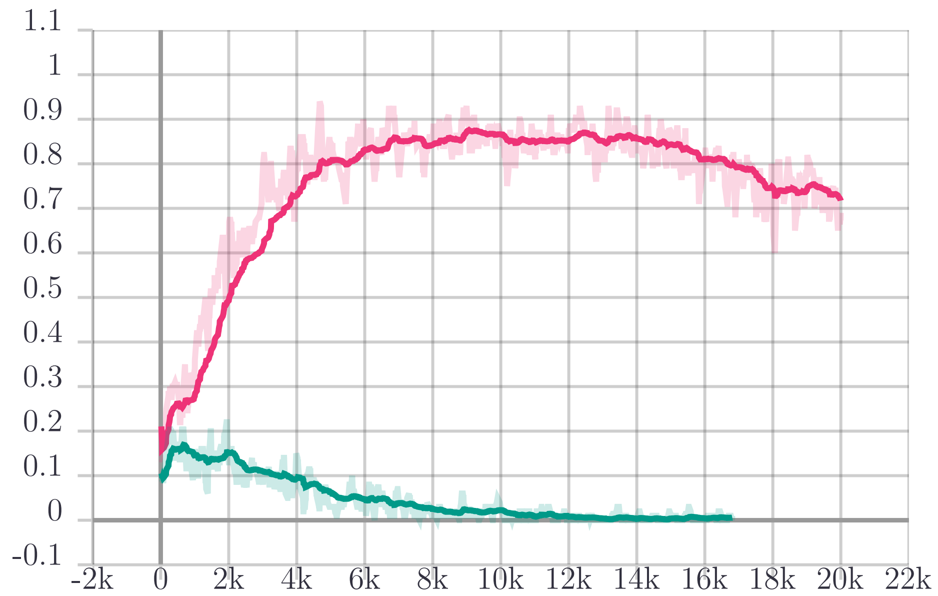

Figura 5.8: Media di episodi conclusi, con crash penalty (in fucsia) e senza (in verde) interrotto dopo 17K episodi

5.4 Switch in Switch

L'approccio granulare di switch in switch (detto anche switch2switch) con una media del 60% di agenti a destinazione, ha registrato un miglioramento rispetto alla navigazione senza training. Tuttavia come visibile in Figura 5.9, ha dato risultati decisamente più bassi della soluzione che non ne fa uso.

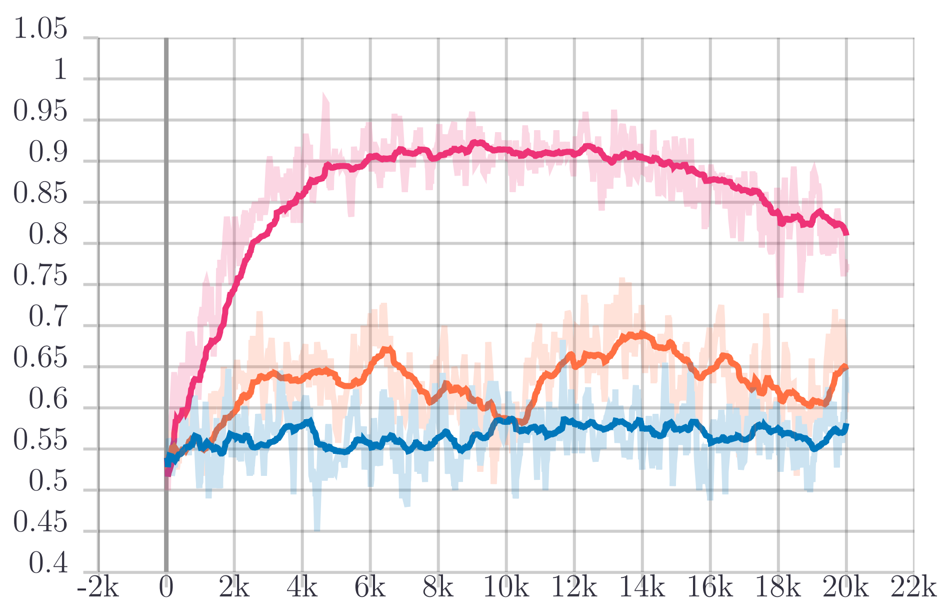

Figura 5.9: Media di agenti a destinazione, approccio switch2switch (in arancione), senza switch2switch (in fucsia) e navigazione senza training (in blu)

5.5 Velocità variabili

Si è testato come varia l'apprendimento nel caso in cui gli agenti possano avere velocità diverse. Dai risultati in Figura 5.10 si nota come il training abbia dato risultati peggiori rispetto all'esecuzione con policy casuale. Il motivo di questo risultato è che la rete non è in grado di inferire la velocità di un agente avendo a disposizione solamente la sua bitmap.

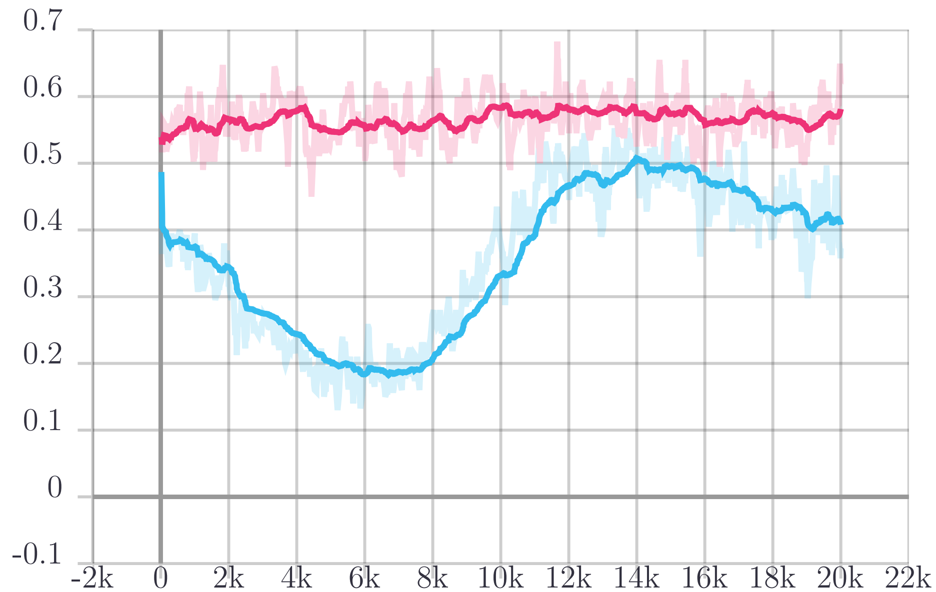

Figura 5.10: Media di agenti a destinazione, senza training (in fucsia), training con velocità diverse (in azzurro)

5.6 Malfunzionamenti

Si è testata la capacità dell'agente sviluppato, di reagire ai malfunzionamenti randomici dei treni. I parametri utilizzati per generare i guasti sono:

- Frequenza dei malfunzionamenti, `malfunction_rate`: 500, ossia in media un malfunzionamento ogni 500 timesteps
- Durata minima dei malfunzionamenti, `min_duration`: 20, ossia minimo 20 timesteps
- Durata massima dei malfunzionamenti, `max_duration`: 80

Come si può vedere in Figura 5.11, l'introduzione dei malfunzionamenti non ha influito sulla percentuale di agenti a destinazione, dimostrando l'efficacia dell'algoritmo di prevenzione dei crash nella gestione dei malfunzionamenti. Si registra tuttavia un aumento del tempo necessario per eseguire lo stesso numero di episodi (Tabella 5.6),

	Tempo per 10K episodi
Senza Malfunzionamenti	10h
Con Malfunzionamenti	17h

Tabella 5.3: Confronto del variare del tempo in base all'introduzione di malfunzionamenti

dovuto al fatto che i malfunzionamenti causano ritardi che influiscono sulla lunghezza degli episodi.

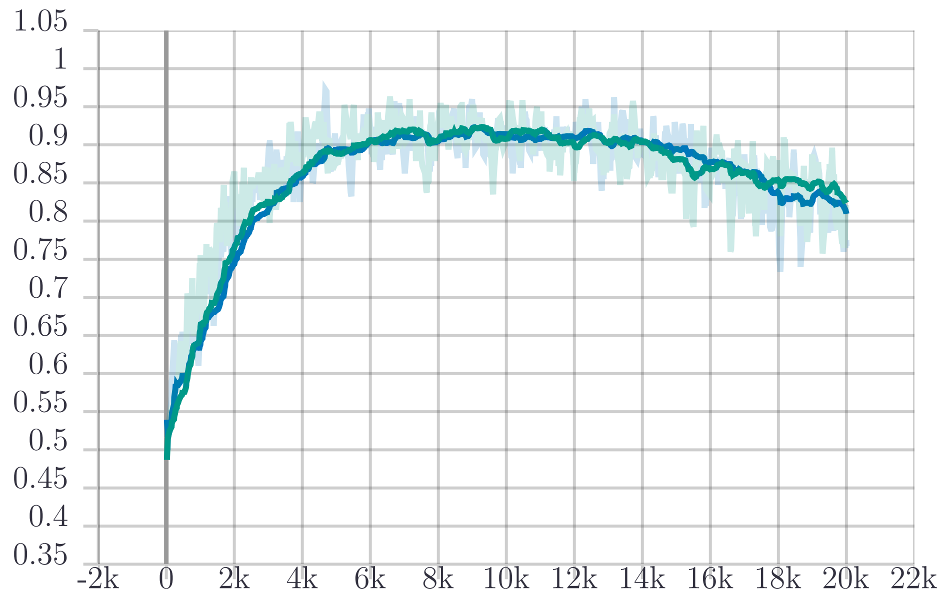

Figura 5.11: Media di agenti a destinazione, con malfunzionamenti (in verde) e senza (in blu)

5.7 Numero di agenti

Si è effettuato un test in cui è stato aumentato a 50 il numero di agenti, per riportarlo nel range dei parametri di Flatland. Come ci si aspettava i risultati sono peggiorati e il tempo richiesto per la simulazione è aumentato drasticamente. Come si vede in Figura

5.12 la percentuale di agenti a destinazione è molto bassa, intorno al 4%, ma l'andamento è promettente.

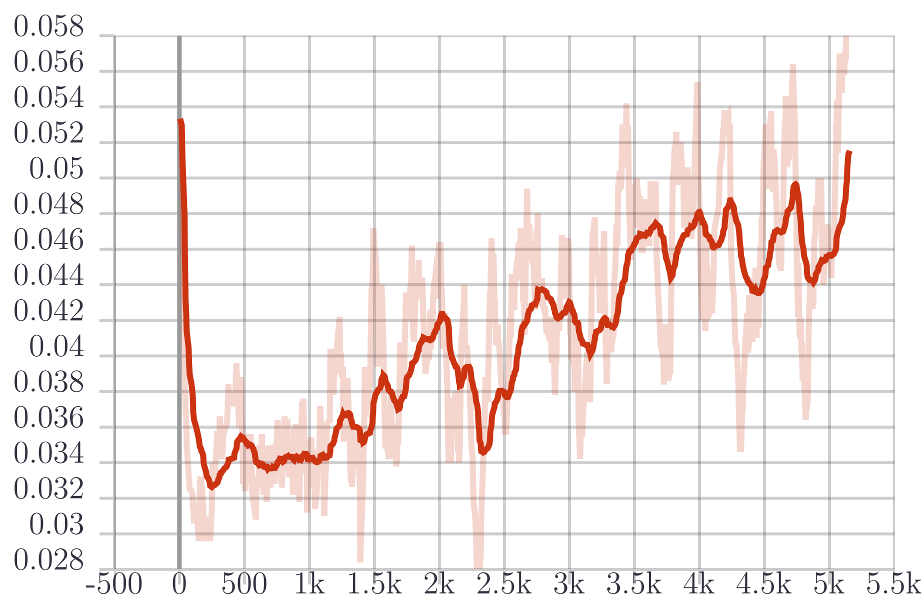

Figura 5.12: Media di agenti a destinazione, con 50 agenti per 5K timestep

Il crollo dello score è dovuto all'aumento del traffico. I conflitti più semplici sono rilevati dall'algoritmo di crash detection, ma all'aumentare del numero di agenti cresce anche la probabilità che si verifichino situazioni di deadlock complesse attualmente non gestite. Inoltre, per come sono rappresentati gli switch nelle bitmap, la rete non è in grado di prevenire alcuni tipi di incidenti su particolari percorsi di switch, vedi Figura 5.13

Per quanto riguarda il tempo richiesto, dalla Tabella 5.7 si nota come per 5K episodi aumentando il numero di agenti, il tempo necessario per una simulazione è aumentato di 30 volte arrivando a richiedere circa 8 giorni per 5K episodi. Questo è sicuramente dovuto al fatto che essendoci più treni, il numero di operazioni da svolgere è aumentato di circa 10 volte. Inoltre è cresciuta la complessità delle simulazioni, che terminano sempre al raggiungimento del limite di timestep.

Figura 5.13: Esempio del crash più semplice fra due agenti (rosso e blu) non rilevabile dalle bitmap

Numero di agenti	Tempo per 5K episodi	Media Episodi Conclusi
4	6h	80%
50	184h	0%

Tabella 5.4: Confronto del variare del tempo e della percentuale di episodi conclusi al variare del numero di agenti

Conclusioni

In questo elaborato è stata descritta la Flatland challenge e gli strumenti messi a disposizione della ricerca nell'ambito del reinforcement learning multi-agente (MARL) applicato a problemi di rescheduling (VRSP). Si è approfondita la letteratura riguardante gli algoritmi di Q-Learning e Deep Q-Learning (DQN), e si sono analizzate le ricerche riguardanti l'applicazione del reinforcement learning a problemi di rescheduling.

Dopo l'introduzione al problema e allo stato dell'arte, è stata discussa la prima soluzione, descrivendo i dettagli implementativi della generazione e dell'utilizzo delle bitmap ai fini del training e della navigazione. Successivamente sono state evidenziate le limitazioni della prima versione, come il fatto che l'utilizzo delle bitmap per training, navigazione e prevenzione di conflitti, portava a situazioni in cui si perdeva irreversibilmente la sincronizzazione fra le bitmap e l'ambiente. Sono poi state descritte le soluzioni sviluppate per risolvere i problemi della prima versione, come la nuova strategia di navigazione basata sulle informazioni dell'ambiente Flatland, e la rappresentazione degli switch nelle bitmap, che hanno reso la navigazione più stabile ed efficace. È stato poi esposto l'ulteriore lavoro ed i concetti sviluppati per migliorare i risultati dell'addestramento. In particolare sono stati descritti: l'approccio di training basato sulle heatmap per dare alla rete una visione globale scalabile, la generazione e l'utilizzo delle altmaps per consentire agli agenti di scegliere fra più percorsi, e l'utilizzo di reward molto basse in caso di crash per disincentivare azioni sbagliate. Sono stati anche descritti i metodi sviluppati con l'obiettivo di semplificare il lavoro della rete, rendendo più omogenee le osservazioni, ossia l'approccio di training granulare fra switch e il riordinamento del binari nelle heatmap.

Le soluzioni sono poi state testate su una versione molto rilassata del problema, con

una mappa piccola (20x20) e pochi agenti (4) che si muovevano tutti a velocità unitaria e non c'erano malfunzionamenti. Dai risultati è stato chiaro che l'approccio abbia un potenziale, siccome il numero di agenti a destinazione medio a seguito di 10 mila episodi di training è arrivato attorno al 90%.

È stato testato che la dimensione migliore del replay buffer sia 10K elementi, le altre prove effettuate hanno dato risultati negativi (2K e 25K elementi) o risultano troppo onerose per il consumo di memoria (1 milione di elementi).

Fra i risultati rilevanti è stato riscontrato che assegnare una reward negativa molto alta in caso di azioni che causano crash, porti probabilmente ad un overfitting di penalità, andando ad impattare negativamente sulla percentuale media di agenti che arrivano a destinazione.

È stato verificato quanto ipotizzato, ossia che con la rappresentazione attuale delle bitmap, la rete non abbia modo di inferire quale sia la velocità di un agente, dando risultati negativi nel caso di test con agenti con velocità variabili.

L'approccio granulare di switch in switch ha registrato un miglioramento rispetto alla policy random, dando però risultati sensibilmente più bassi rispetto al punteggio migliore ottenuto.

Per quanto riguarda i problemi posti dalla Flatland challenge è possibile concludere che:

- La navigazione è stata implementata con successo, un agente infatti è in grado di raggiungere il proprio target
- La prevenzione dei crash è stata implementata in una versione basilare molto efficace ma comunque limitata. I limiti dell'algoritmo infatti sono evidenti in situazioni di traffico intense
- Per quanto riguarda il rescheduling, nonostante tutte le soluzioni siano state sviluppate senza considerare il rischio di malfunzionamenti, l'algoritmo di prevenzione dei crash, necessario per la navigazione, è risultato efficace anche nella gestione delle situazioni di rescheduling in caso di guasti temporanei

Gli sviluppi futuri prevedono un miglioramento della capacità di prevenzione di deadlock non triviali. È infatti necessario migliorare l'algoritmo di crash detection, o trovare

una rappresentazione degli switch nelle bitmap, che consenta alla rete di imparare a rilevare gli incidenti.

Un altro punto fondamentale è aggiungere il supporto ad agenti con velocità diverse. Sarebbe opportuno passare la velocità dell'agente come parametro alla rete.

Inoltre è necessario migliorare l'efficienza degli algoritmi e di utilizzare elaboratori con prestazioni maggiori, in modo da poter testare le soluzioni su mappe più vaste (200x200) e con molti più agenti (150), come specificato nei requisiti della challenge.

Bibliografia

[1] G. Cantini. «Flatland: A study of Deep Reinforcement Learning methodologies applied to the vehicle rescheduling problem in a railway environment». In: mar. 2020.

[2] F. Corman et al. «Dispatching trains during seriously disrupted traffic situations». In: *2011 International Conference on Networking, Sensing and Control*. Apr. 2011, pp. 323–328. DOI: 10.1109/ICNSC.2011.5874901.

[3] A. D'Ariano, M. Pranzo e I. A. Hansen. «Conflict Resolution and Train Speed Coordination for Solving Real-Time Timetable Perturbations». In: *IEEE Transactions on Intelligent Transportation Systems* 8.2 (giu. 2007), pp. 208–222. ISSN: 1558-0016. DOI: 10.1109/TITS.2006.888605.

[4] M. Dotoli et al. «A real time traffic management model for regional railway networks under disturbances». In: *2013 IEEE International Conference on Automation Science and Engineering (CASE)*. Ago. 2013, pp. 892–897. DOI: 10.1109/CoASE.2013.6653977.

[5] Hado V. Hasselt. «Double Q-learning». In: *Advances in Neural Information Processing Systems 23*. A cura di J. D. Lafferty et al. Curran Associates, Inc., 2010, pp. 2613–2621. URL: http://papers.nips.cc/paper/3964-double-q-learning.pdf.

[6] Johanna Törnquist Krasemann. «Design of an effective algorithm for fast response to the re-scheduling of railway traffic during disturbances». In: *Transportation Research Part C: Emerging Technologies* 20.1 (2012). Special issue on Optimization in Public Transport+ISTT2011, pp. 62–78. ISSN: 0968-090X. DOI: https://doi.

org/10.1016/j.trc.2010.12.004. URL: http://www.sciencedirect.com/science/article/pii/S0968090X10001671.

[7] Jing-Quan Li, Pitu B. Mirchandani e Denis Borenstein. «The vehicle rescheduling problem: Model and algorithms». In: *Networks* 50.3 (2007), pp. 211–229. DOI: 10.1002/net.20199. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.20199. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.20199.

[8] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. Rapp. tecn. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[9] Volodymyr Mnih et al. «Human-level control through deep reinforcement learning». In: *Nature* 518.7540 (feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[10] Mohammadreza Nazari et al. «Reinforcement learning for solving the vehicle routing problem». In: *Advances in Neural Information Processing Systems*. 2018, pp. 9839–9849.

[11] Netcetera. *Solving disruptions in networks with Machine Learning*. 2019. URL: https://www.netcetera.com/home/stories/news/20200109-sbb-flatland-challenge-ai.html (visitato il 23/02/2020).

[12] Mitsuaki Obara, Takehiro Kashiyama e Yoshihide Sekimoto. «Deep Reinforcement Learning Approach for Train Rescheduling Utilizing Graph Theory». In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 4525–4533.

[13] *Observation Spaces*. 2020. URL: http://flatland-rl-docs.s3-website.eu-central-1.amazonaws.com/intro_observation_actions.html#observation-spaces (visitato il 23/02/2020).

[14] A. Sajedinejad et al. «SIMARAIL: Simulation based optimization software for scheduling railway network». In: *Proceedings of the 2011 Winter Simulation Conference (WSC)*. Dic. 2011, pp. 3730–3741. DOI: 10.1109/WSC.2011.6148066.

[15] AICrowd & SBB. *Flatland Challenge*. 2020. URL: https://www.aicrowd.com/challenges/flatland-challenge (visitato il 23/02/2020).

[16] D. Šemrov et al. «Reinforcement learning approach for train rescheduling on a single-track railway». In: *Transportation Research Part B: Methodological* 86 (2016), pp. 250–267. ISSN: 0191-2615. DOI: https://doi.org/10.1016/j.trb.2016.01.004. URL: http://www.sciencedirect.com/science/article/pii/S0191261516000084.

[17] Remy Spliet, Adriana F. Gabor e Rommert Dekker. «The vehicle rescheduling problem». In: *Computers & Operations Research* 43 (2014), pp. 129–136. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2013.09.009. URL: http://www.sciencedirect.com/science/article/pii/S0305054813002670.

[18] R. S. Sutton e A. G. Barto. «Reinforcement Learning: An Introduction». In: *IEEE Transactions on Neural Networks* 9.5 (set. 1998), pp. 1054–1054. ISSN: 1941-0093. DOI: 10.1109/TNN.1998.712192.

[19] Oriol Vinyals, Meire Fortunato e Navdeep Jaitly. «Pointer Networks». In: *arXiv e-prints*, arXiv:1506.03134 (giu. 2015), arXiv:1506.03134. arXiv: 1506.03134 [stat.ML].

[20] Ziyu Wang et al. «Dueling Network Architectures for Deep Reinforcement Learning». In: *arXiv e-prints*, arXiv:1511.06581 (nov. 2015), arXiv:1511.06581. arXiv: 1511.06581 [cs.LG].