

UNIVERSITÀ
DEGLI STUDI
DI PADOVA



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Department of Information Engineering
Master's degree in "Control Systems Engineering"

Fully Dynamic Rebalancing of Dockless Bike Sharing Systems using Deep Reinforcement Learning

MASTER CANDIDATE

Edoardo Scarpel

Student ID 2089096

SUPERVISOR

Professor Gian Antonio Susto

University of Padova

CO-SUPERVISOR

Matteo Cederle

Federico Chiarotti

Marco Fabris

University of Padova

ACADEMIC YEAR
2024/2025

In memoria di mio zio Fabio,
il cui sorriso continuerà sempre a ispirarci.

Abstract

Cities worldwide are increasingly embracing sustainable solutions to tackle urban challenges such as congestion, pollution, and reliance on private cars. Central to these efforts is improving last-mile connectivity, enabling travelers to easily move from main transportation hubs to their final destinations. *Dockless* Bike Sharing Systems (BSSs) offer a convenient and eco-friendly solution for last-mile mobility, reducing car use and emissions. However, their flexibility introduces operational challenges, as bikes often cluster in popular areas, causing shortages elsewhere. Effectively managing these imbalances, especially under unpredictable demand, remains challenging. This thesis proposes a fully dynamic rebalancing framework for *Dockless* BSSs using a Dueling Deep Q-Network (DDQN). The study employs an event-driven simulation environment based on real-world data from Cambridge, Massachusetts, including traffic patterns from TomTom. This environment realistically models bike-sharing dynamics in real-time, offering a robust platform to test adaptive rebalancing strategies. The DDQN agent successfully learned rebalancing policies that adapt to demand fluctuations, reduce service failures, and perform competitively even in complex, fully dynamic scenarios. While it did not outperform all simplified baselines, it demonstrated the viability of dynamic, data-driven approaches. The framework is scalable to larger urban areas and highlighted critical challenges in reward design and model tuning, setting a solid foundation for future research in realistic, city-scale mobility systems.

Sommario

Le città di tutto il mondo stanno adottando soluzioni sempre più sostenibili per affrontare sfide urbane quali congestione, inquinamento e dipendenza dall'automobile privata. Un elemento fondamentale di questi interventi riguarda il miglioramento della mobilità nell'ultimo tratto del tragitto urbano, ovvero la cosiddetta mobilità dell'ultimo miglio (*last-mile connectivity*), che permette agli utenti di spostarsi agevolmente dai principali nodi di trasporto alle proprie destinazioni finali. I sistemi di bike sharing senza stazioni fisse (*dockless Bike Sharing Systems*, BSS) rappresentano una soluzione pratica ed ecologica a questo problema, contribuendo a ridurre il traffico automobilistico e le emissioni inquinanti. Tuttavia, la loro natura flessibile comporta una sfida significativa: la tendenza delle biciclette a concentrarsi in zone ad alta richiesta, lasciando scoperte altre aree della città. Riuscire a bilanciare efficacemente la distribuzione delle biciclette, specialmente in presenza di una domanda imprevedibile, resta una problematica complessa. Questa tesi propone un modello di ribilanciamento completamente dinamico per sistemi di bike sharing senza stazioni, basato su una rete neurale Dueling Deep Q-Network (DDQN). Il metodo è stato validato attraverso un ambiente simulativo basato su eventi reali (*event-driven*), costruito su dati concreti della città di Cambridge, Massachusetts, e arricchito da modelli di traffico forniti da TomTom. Tale scenario consente di riprodurre accuratamente le dinamiche del bike sharing in tempo reale, offrendo una base solida per sviluppare strategie adattive di ribilanciamento. L'agente DDQN ha dimostrato la capacità di apprendere strategie efficaci per gestire il ribilanciamento delle biciclette, adattandosi alle variazioni nella domanda e riducendo significativamente i casi di servizio non soddisfatto, anche in situazioni complesse e mutevoli. Sebbene non abbia superato tutte le prestazioni di riferimento, l'approccio proposto ha dimostrato chiaramente la fattibilità di un ribilanciamento dinamico e guidato dai dati. Il modello è inoltre scalabile a contesti urbani più ampi, e ha messo in luce sfide importanti riguardo alla definizione ottimale della funzione di ricompensa e alla scelta dell'architettura neurale, aprendo così la strada a futuri sviluppi nel campo dei sistemi di mobilità intelligenti e realistici.

Contents

List of Figures	xi
List of Algorithms	xv
List of Acronyms	xvii
1 Introduction	1
2 Theoretical Background	5
2.1 Markov Decision Processes	5
2.2 Elements of Reinforcement Learning	7
2.3 Deep RL and Deep Q-Learning	12
3 Literature Review	17
3.1 Overview of Bike Sharing Systems	17
3.2 Static vs. Dynamic Rebalancing Methods	20
3.3 Deep Reinforcement Learning in Mobility Optimization	23
4 System Model	25
4.1 Mathematical Representation of the System	25
4.2 Modeling Demand and Arrival Patterns with a Continuous-Time Markov Chain	28
5 Proposed Framework	35
5.1 Fully Dynamic Framework	35
5.2 MDP	43
5.3 Double DQN with Graph Neural Networks	51
6 Simulations & Results	57
6.1 Simulation Environment and Setup	57

CONTENTS

6.2	State-of-the-Art Approaches	67
6.3	Results and Analysis	69
7	Conclusions and Future Work	79
7.1	Conclusions	79
7.2	Future Work	80
	Acknowledgments	85
	References	87

List of Figures

1.1	A 2018 news article from <i>Il Post</i> discussing the failure of BSSs in Rome, Italy highlighting the operational challenges and high costs that have prevented their success in the city [17].	2
2.1	The agent–environment interaction in a Markov Decision Process.	7
3.1	Carrying capacity of different modes, 3.5 m carriageway [12]. . . .	19
5.1	Graph representation of Cambridge, MA, used for the BSS environment. Data are extracted from OpenStreetMap.	37
5.2	Flowchart of the event-driven simulator for the BSS: this diagram outlines the sequence of operations, from departure event generation to bike availability checks, trip execution, and arrival processing. The event buffer ensures orderly execution of events in the system.	40
5.3	Grid representation of Cambridge, MA, used for the truck environment.	42
5.4	Illustration of the DQN Module with Graph Neural Networks. The agent leverages GATs to process subgraph information and extracts state representations through FFNNs.	52
5.5	Training workflow of the DDQN. The agent interacts with the environment, storing experiences (s, a, r, s', n) in the replay buffer. The online network $Q(s, a; \theta)$ is updated using a loss function computed from target values generated by the target network $Q(s', a'; \theta')$. A soft update with parameter τ is applied to periodically update the target network, ensuring training stability. . .	55
6.1	Graph representation of the selected sub-area ($\sim 2.3 \text{ km}^2$) around MIT, used for the preliminary analysis of dynamic bike rebalancing.	58

LIST OF FIGURES

6.2	Visual summary of the two-step interpolation process using 500m-radius neighborhoods.	62
6.3	Interpolated trip demand graph, representing estimated trip rates for the <i>dockless</i> system, based on data from Monday, 07:00–10:00 AM. The colors indicate the intensity of the demand, with higher values corresponding to areas experiencing greater trip activity. .	63
6.4	Sample from TomTom’s velocity matrix for the Boston, MA area [38], showing average travel speeds by hour and day of the week.	64
6.5	Benchmark comparison of failures in static and dynamic rebalancing simulations (Monday to Sunday).	68
6.6	Benchmark comparison of rebalance time in static and dynamic rebalancing simulations (Monday to Sunday).	68
6.7	Evolution of the reward per time slot over training episodes. Each point corresponds to the time slot average reward, while the red line indicates its cumulative mean. The steadily increasing trend and reduced fluctuation range suggest that the learning process is gradually converging toward a higher overall reward.	70
6.8	Number of service failures per time slot across training episodes, with a cumulative mean shown by the red line. The declining trajectory indicates that the agent increasingly learns to meet demand, thus reducing failures as training progresses.	70
6.9	Mean Q-value per time slot, tracked across time slots and averaged cumulatively (red line). The stabilization and smoother profile at later episodes imply that the agent’s policy converges, reflecting consistent decision-making in selecting optimal rebalancing actions.	71
6.10	Failures in the baseline scenario without rebalancing. The red line depicts the cumulative mean, which stabilizes at about 19 failures per time slot.	72
6.11	Failures over the validation days. The red line marks the time slot mean.	72
6.12	Number of deployed bikes per day. The red line shows the time slot mean.	72
6.13	Frequency of each action in the validation episode.	73
6.14	Mean reward associated with each action.	73

6.15	Heat map of the rebalancing area's zones, where each color indicates the number of visits made by the rebalancing truck. Higher intensity suggests more frequent visits.	74
6.16	Average risk score of each zone over one validation episode. Negative scores represent a surplus of bikes relative to demand, whereas positive scores indicate a potential shortage.	75
6.17	Distribution of bikes across zones at the end of the validation episode. Higher intensity indicates a greater number of bikes in that zone.	76

List of Algorithms

1	Q-Learning	10
2	Deep Q-learning with Experience Replay	14
3	Double Q-learning (Hassel et al., 2016)	15

List of Acronyms

BDPs Birth-Death Processes

BSSs Bike Sharing Systems

CTMC Continuous-Time Markov Chain

DBRPs Dynamic Bike Repositioning Problems

DDQN Double Deep Q-Network

DPSO Discrete Particle Swarm Optimization

DRL Deep Reinforcement Learning

DQN Deep Q-Network

DTMC Discrete-Time Markov Chain

FFNN Feed Forward Neural Network

GATs Graph Attention Networks

GCNs Graph Convolutional Networks

GDP Gross Domestic Product

GNN Graph Neural Network

GPS Global Positioning System

IoT Internet of Things

MAE Mean Absolute Error

MARL Multi-Agent Reinforcement Learning

LIST OF ALGORITHMS

MDP Markov Decision Process

MILP Mixed-Integer Linear Programming

MINP Mixed-Integer Non-linear Programming

MIP Mixed-Integer Program

MRP Markov Reward Process

MP Markov Process

MSE Mean Squared Error

PMF Probability Mass Function

ReLU Rectified Linear Unit

RFID Radio-Frequency Identification

RL Reinforcement Learning

SBRPs Static Bike Repositioning Problems

SGD Stochastic Gradient Descent

TD Temporal Difference

1

Introduction

Cities worldwide are undergoing significant transformations aimed at promoting sustainability, healthier lifestyles, and lively community spaces. Projects such as car-free streets, increased green spaces, and better public transport systems are designed to solve persistent urban problems like traffic congestion, air pollution, and dependency on private vehicles. A crucial part of these initiatives is improving last-mile connectivity, helping travelers easily move between main transport hubs and their final destinations.

In this scenario, Bike Sharing Systems (BSSs), especially *dockless* ones, represent an attractive and environmentally friendly last-mile mobility solution. These systems provide flexible short-distance trips, reducing car use and emissions. However, the convenience of *dockless* systems leads to substantial operational challenges: bicycles often cluster in specific areas, while other locations face shortages, affecting usability and causing inefficient use of resources. Managing these imbalances, particularly under unpredictable conditions, remains a key challenge in operating BSSs effectively.

Despite the growing adoption of BSSs across major cities, operational challenges have led to repeated failures in certain urban areas. As of 2018, Rome, for instance, was one of the few European capitals without a functioning BSS due to logistical issues, high costs, and ineffective rebalancing strategies (Figure 1.1) [17]. This highlights the necessity for more advanced, dynamic approaches to managing *dockless* BSSs efficiently.

Most existing methods for bike rebalancing simplify the problem, often using fixed time steps or averaged demand patterns to make it more tractable. Many

L'unica grande capitale europea senza bike sharing

È Roma, e dopo anni di tentativi e fallimenti non ci sono grandi piani per cambiare le cose



(ANSA)

Figure 1.1: A 2018 news article from *Il Post* discussing the failure of BSSs in Rome, Italy highlighting the operational challenges and high costs that have prevented their success in the city [17].

approaches also decouple departures and arrivals, treating them as independent processes, or rely on instantaneous rebalancing decisions without fully modeling their impact over time. However, these simplifications struggle to capture the complexities of real urban environments, where demand, traffic conditions, and energy consumption fluctuate dynamically throughout the day. To address these limitations, this research presents an event-driven simulation environment that realistically models BSS dynamics, including user arrivals, bike trips, battery usage, and traffic conditions, without imposing restrictive assumptions. This simulator, a notable improvement over existing solutions, integrates detailed urban variability, creating a more accurate representation of a *dockless* BSS. It provides a robust platform for testing rebalancing strategies that have not yet been explored under realistic, real-time conditions.

Using this fully dynamic environment, the thesis introduces a Reinforcement Learning agent based on a Double Deep Q-Network architecture to manage rebalancing operations. By continuously observing the system's state, including bike locations, user arrivals, battery levels, and possible congestion points, the agent determines optimal times and methods for repositioning or recharging bicycles. Although the research presented is preliminary, the ultimate goal is to develop a flexible, easily adaptable solution that can scale to different neighborhoods or larger areas, such as districts comparable in size to Cambridge,

Massachusetts, simply by changing local data inputs. The findings underscore the potential of this approach in meeting the real-time requirements of *dockless* BSSs, while also highlighting the need for further refinements to realize fully scalable, context-aware rebalancing across entire cities.

Preliminary results show that the Double Deep Q-Network agent is capable of learning effective rebalancing policies that reduce service failures, adapt to demand fluctuations, and operate efficiently in a fully dynamic environment. Although it does not outperform all existing methods designed for simplified or static scenarios, it demonstrates the feasibility of integrating Reinforcement Learning into realistic, real-time bike sharing operations. The framework also proves to be easily scalable, allowing extension to larger urban areas by simply adjusting local data inputs collected via sensor networks (see, e.g., [11]). In doing so, this work highlights several open challenges, such as the complexity of balancing multiple objectives within the reward function and the computational demands of tuning deep neural architectures. Despite these limitations, the proposed approach represents a meaningful step forward in the development of adaptive, city-scale rebalancing strategies for *dockless* BSSs, offering both a novel methodology and a solid foundation for future research in the field of smart urban mobility.

THESIS OUTLINE

The remainder of the thesis unfolds as follows.

- Chapter 2: **Theoretical Background** presents the fundamentals of Reinforcement Learning, including Markov Decision Processes and Q-learning, as well as the basics of Deep Reinforcement Learning, particularly Deep Q-Networks.
- Chapter 3: **Literature Review** provides a summary of existing methods for BSS rebalancing and highlights the challenges involved in dynamic management under varying demand.
- Chapter 4: **System Model** describes the mathematical framework, including modeling demand in continuous time, battery consumption, and vehicle movements between nodes.
- Chapter 5: **Proposed Framework** presents the event-driven simulator, emphasizing its ability to realistically represent actual conditions without relying on standard simplifying assumptions. The chapter formally defines

the problem as a Markov Decision Process, clearly specifying states, actions, rewards, and transition dynamics that characterize the environment. The implementation of a Deep Reinforcement Learning approach based on the Double Deep Q-Network algorithm is also detailed, focusing on its network architecture, experience replay mechanism, and the specific challenges posed by operating in a dynamic, continuously evolving urban environment.

- Chapter 6: **Simulations & Results** explains the experimental settings, compares different strategies, and demonstrates how the proposed approach manages the inherent uncertainty of *dockless* BSSs.
- Chapter 7: **Conclusions and Future Work** summarizes the main contributions of the thesis, identifies remaining challenges, and outlines directions for future improvements and potential developments of the proposed approach.

2

Theoretical Background

This section provides the theoretical foundation necessary to understand the tools and concepts explored in the following chapters. It begins with the fundamentals of Markov Decision Processes, a mathematical framework for sequential decision-making under uncertainty. Then, it introduces Reinforcement Learning as a method for solving Markov Decision Processes when system dynamics are unknown or complex. Finally, the section covers advancements in Reinforcement Learning, setting the stage for the techniques and methodologies discussed throughout the thesis. This background is essential for comprehending the core principles underlying the work and applications presented later.

2.1 MARKOV DECISION PROCESSES

A Markov Decision Process (MDP) is a mathematical framework used to model decision-making in situations where outcomes are influenced by both chance and control. It builds upon Markov Reward Process (MRP) and Markov Process (MP), which describe probabilistic systems where the future state depends only on the present state, satisfying the Markov property [34, 37].

A Markov Process describes a sequence of states where transitions occur based on a predefined probability distribution. Formally, it is defined as a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$, where:

- $\mathcal{S}$ is a finite set of states.

2.1. MARKOV DECISION PROCESSES

- $\mathcal{P}$ is the state transition probability matrix, with entries:

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s] \quad (2.1)$$

A Markov Reward Process extends this definition by introducing rewards and discounting. An MRP is defined as a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where:

- $\mathcal{R}$ is a reward function, with $\mathcal{R}_s = \mathbb{E}[R_{t+1} \mid S_t = s]$, representing the expected immediate reward for being in state s .
- γ is a discount factor, $\gamma \in [0, 1]$, which balances the importance of future rewards relative to immediate ones (0=myopic, 1=far-sighted).

The goal in such settings is often to maximize the return G_t , which represents the cumulative discounted rewards from time t onwards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (2.2)$$

From this, the state-value function $v(s)$ can be defined as the expected return starting from state s :

$$v(s) = \mathbb{E}[G_t \mid S_t = s] \quad (2.3)$$

This function satisfies a recursive relationship, known as the Bellman equation:

$$v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] \quad (2.4)$$

The Markov Decision Process further extends MRPs by introducing a decision-making agent that can influence outcomes through actions. An MDP is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{A}$ is a finite set of actions, and:

- The state transition probabilities become action-dependent:

$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a] \quad (2.5)$$

- The reward function also depends on actions:

$$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a] \quad (2.6)$$

To make decisions, a policy π is defined as a mapping from states to a

probability distribution over actions:

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s] \quad (2.7)$$

In addition to the state-value function $v(s)$, the action-value function $q_\pi(s, a)$ is introduced. This function represents the expected return starting from state s , taking action a , and then following policy π :

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \quad (2.8)$$

Solving an MDP typically involves finding an optimal policy π^* that maximizes expected rewards over time. Various exact and approximate methods exist to find π^* , depending on whether the transition dynamics are fully known or need to be learned.

2.2 ELEMENTS OF REINFORCEMENT LEARNING

Reinforcement Learning (RL) is a branch of machine learning and optimal control that focuses on training intelligent agents to make decisions in dynamic environments, combining learning with long-term decision making [37]. Unlike supervised and unsupervised learning, RL does not require predefined labels or structures in the data. Instead, it enables an agent to learn by interacting with the environment, taking actions, and receiving feedback in the form of rewards or penalties. Over time, the agent refines its strategy to maximize cumulative rewards, striving to discover the optimal policy for achieving the best outcomes.

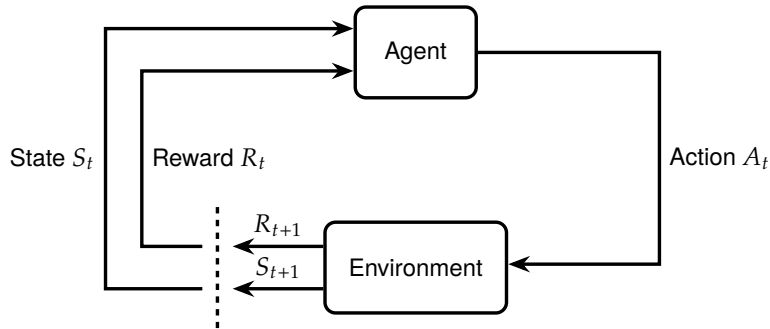


Figure 2.1: The agent–environment interaction in a Markov Decision Process.

As shown in Figure 2.1, the agent observes the current state S_t , chooses an action A_t according to a policy π , and then receives a reward R_{t+1} from

the environment along with the next state S_{t+1} . By repeatedly updating its internal estimates of value (i.e., expected cumulative reward) for states or state-action pairs, the agent converges on a policy that maximizes long-term returns. Crucially, this process does not require a known transition function $\mathcal{P}$; the agent learns it implicitly from experience. Key components in an RL setup include:

- Policy π : Specifies how the agent selects actions based on states. It can be deterministic ($\pi(s) = a$) or stochastic ($\pi(a | s)$).
- Value function $v_\pi(s)$ and action-value function $q_\pi(s, a)$: Measure the expected return starting from state s (or state-action pair (s, a)) under policy π . They guide the agent toward actions that yield higher long-term rewards.
- Model of the environment (optional): If available (or learned), a model of the environment's dynamics can be used to plan ahead. However, many RL algorithms (model-free methods) bypass explicit modeling of $\mathcal{P}$ and $\mathcal{R}$.

This paradigm is particularly well-suited to the problem explored in this thesis, as it provides an effective approach for dynamic decision-making under uncertainty. RL has achieved remarkable success across various domains where system dynamics are difficult to model or rapidly changing, such as resource rebalancing, robotics, game-playing, finance, and recommendation systems.

EXPLORATION VS. EXPLOITATION

One of the fundamental challenges in RL is managing the exploration-exploitation trade-off. During training, an agent must explore by taking actions it hasn't tried before to discover effective strategies for solving the problem. However, relying solely on random exploration can be inefficient, as it may lead to actions that yield little to no progress. At the same time, the agent must also exploit its existing knowledge, leveraging what it has learned to optimize its performance. Striking the right balance between exploration and exploitation is essential, ensuring the agent continues to learn while making meaningful progress toward its goal.

A direct solution to the exploration-exploitation trade-off does not exist, but various approaches have been proposed to address it. One of the most popular and simple methods is the ϵ -Greedy Policy [37]. In this approach, with probability $1-\epsilon$, the agent selects the optimal action, ensuring exploitation of its current knowledge. With probability ϵ , it instead chooses a random action excluding the optimal one, promoting exploration. This formulation ensures

that the probability distribution remains correctly normalized, preventing the optimal action from being selected more frequently than intended. By balancing exploration and exploitation, the agent can gradually refine its strategy while continuing to discover potentially better solutions.

Beyond ϵ -Greedy, there are several other methods designed to handle the trade-off. These include Optimistic Initial Values [37], which encourage exploration by initially assigning high values to all actions, Upper-Confidence-Bound (UCB) Action Selection [1], which selects actions based on both their expected reward and the uncertainty around that estimate, and Gradient Bandit Algorithms (GBA) [37], which adjust action probabilities based on performance feedback. These and other strategies offer alternative ways to balance exploration and exploitation, each with its strengths and weaknesses depending on the context.

Q-LEARNING

Q-learning is a foundational RL algorithm designed to solve MDPs by determining an optimal policy that maximizes the expected return from any state. Central to learning in MDPs is balancing the exploration-exploitation trade-off: exploring new actions to discover their effects while exploiting known actions to maximize rewards. Q-learning, a model-free, value-based, and off-policy algorithm, addresses this challenge by iteratively updating state-action values, or Q-values, to converge toward an optimal policy.

Earlier learning methods, such as Monte Carlo approaches and Temporal Difference (TD) learning, laid the groundwork for estimating value functions and policies. Monte Carlo methods rely on complete episodes to compute value updates, while TD learning uses estimated values of successor states to enable learning from incomplete episodes. Despite their utility, these methods focus primarily on state or state-action value estimation, without directly optimizing policies.

Q-learning extends TD learning by explicitly learning the value of taking specific actions in specific states. It updates Q-values using immediate rewards and the discounted value of subsequent states under the current policy, enabling it to evaluate and improve policies simultaneously. As a model-free algorithm, Q-learning does not require explicit knowledge of transition or reward functions, allowing it to learn optimal policies purely through experience, even in deterministic or stochastic environments. This combination of efficiency and

2.2. ELEMENTS OF REINFORCEMENT LEARNING

flexibility makes Q-learning a cornerstone of RL.

The Q-learning algorithm relies on the following update rule, which incrementally refines the state-action value estimates:

$$Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma \max_a Q(S', a) - Q(S, A)) \quad (2.9)$$

Here:

- $Q(S, A)$: The current estimate of the value for taking action A in state S .
- α : The learning rate, which controls how much the new information affects the current estimate.
- R : The immediate reward received for taking action A in state S .
- γ : The discount factor, which determines the importance of future rewards relative to immediate ones.
- $\max_a Q(S', a)$: The maximum estimated value of the next state S' , evaluated over all possible actions.

Q-learning pseudo code is presented in Algorithm 1.

Algorithm 1 Q-Learning

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
for each episode **do**
 Initialize S
 while S is not terminal **do**
 Choose A from S using policy derived from Q (e.g., ε -greedy)
 Take action A , observe R, S'
 $Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma \max_a Q(S', a) - Q(S, A))$
 $S \leftarrow S'$
 end while
end for

A key innovation of Q-learning is the use of the TD target, $R + \gamma \max_a Q(S', a)$, which serves as an estimate of the true value of $Q(S, A)$. This approach is called bootstrapping because it uses current Q -values to help update other Q -values. The method incorporates both observed data, such as the reward R and the transition $S \rightarrow S'$, and predictions from the model.

Q-learning is off-policy because the behavior policy, which determines the actions taken during exploration, differs from the updated policy. Specifically,

Q-learning estimates the value of the optimal policy by appending a value to the new state based on a greedy action (maximizing $Q(S', a)$) without necessarily following that policy. This separation allows the algorithm to evaluate future rewards independently of the exploration strategy, enabling efficient learning even with exploratory or random actions.

While bootstrapping accelerates learning by leveraging current estimates, it introduces bias. The algorithm's initial $Q(S, A)$ values are likely inaccurate, which can lead to instability in the learning process, especially when combined with off-policy updates and neural networks. This instability arises because the system may become overly self-referential, relying too much on its own incorrect estimates.

On the other hand, methods without bootstrapping, such as Monte Carlo approaches, often exhibit higher variance. They require longer trajectories and more samples for estimates to converge. Despite these challenges, bootstrapping methods like Q-learning are frequently preferred because they converge faster in practice when managed correctly.

Q-learning is guaranteed to converge to the optimal policy, provided certain conditions—such as sufficient exploration, decaying learning rates, and finite state-action spaces—are met, as established by the convergence theorem.

Theorem 1 (Convergence of Q-Learning [23]) *Given a finite MDP, the Q-Learning algorithm given by the update rule*

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha_t(s_t, a_t) \left(R_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t) \right) \quad (2.10)$$

converges with probability 1 to the optimal state-action value function as long as

$$\sum_t \alpha_t(s, a) = \infty \quad \wedge \quad \sum_t \alpha_t^2(s, a) < \infty \quad (2.11)$$

for all $(s, a) \in \mathcal{S} \times \mathcal{A}$.

For instance, a commonly used learning rate schedule is $\alpha_t(s, a) = \frac{1}{t+1}$, which satisfies the convergence conditions, or alternatively, $\alpha_t(s, a) = \frac{1}{\sqrt{t}}$ can also be employed for similar convergence behavior.

Q-learning is a versatile and widely adopted RL algorithm that empowers agents to discover optimal actions through trial and error. Its model-free design makes it particularly well-suited for complex environments with unknown dynamics. By iteratively improving its action-selection strategy, Q-learning enables

2.3. DEEP RL AND DEEP Q-LEARNING

agents to maximize cumulative rewards over time, proving its effectiveness in diverse applications such as robotics, game AI, autonomous vehicles, and financial modeling.

2.3 DEEP RL AND DEEP Q-LEARNING

DEEP REINFORCEMENT LEARNING

Deep Reinforcement Learning (DRL) is a revolutionary branch of Artificial Intelligence that combines the decision-making framework of RL with the representational power of deep neural networks. By iteratively interacting with an environment and optimizing cumulative rewards, DRL enables agents to learn complex strategies directly from raw sensory data, leveraging deep learning's capability to extract intricate features from unstructured inputs. Central to DRL are methods such as Q-learning, policy gradients, and actor-critic approaches, which rely on value networks, policy networks, and a careful balance between exploration and exploitation [26].

In DRL, deep neural networks are used to approximate key components of RL, such as the value function $\hat{v}(s; \theta)$ or $\hat{q}(s, a; \theta)$, the policy $\pi(a|s; \theta)$, and in some cases, the model itself. Here, θ represents the parameters of the neural network, typically updated using stochastic gradient descent [19].

The applications of DRL span diverse fields, including robotics, gaming, finance, and healthcare. Its evolution, from mastering Atari games to addressing complex real-world problems, highlights its adaptability and immense potential to revolutionize decision-making systems across domains.

DEEP Q-LEARNING

Deep Q-Learning, or Deep Q-Network (DQN), extends the traditional Q-Learning algorithm by leveraging deep neural networks to approximate Q-values, making it suitable for environments with large or continuous state spaces. Traditional Q-Learning, which relies on a Q-table to store the Q-values for each state-action pair, becomes computationally infeasible in such cases due to the exponential growth of the state space. DQNs address this limitation by using a neural network to approximate the Q-function, significantly enhancing scalability [25].

In DQN, the agent learns to estimate the Q-value $Q(s, a)$. This learning process involves training a neural network to minimize the following loss function:

$$\mathcal{L} = \mathbb{E} \left[(y - Q(s, a; \theta))^2 \right] \quad (2.12)$$

where:

- $y = r + \gamma \max_{a'} Q(s', a'; \theta_t)$ is the target value
- θ are the weights of the policy network, which estimates the Q-values.
- θ_t are the weights of the target network, used to compute stable target Q-values.

The neural network is trained by comparing its current estimate $Q(s, a; \theta)$ to the target y , which represents a slightly improved guess based on subsequent information.

A critical innovation in DQN is the use of two separate neural networks:

- Policy Network: Responsible for estimating the Q-values $Q(s, a; \theta)$ for the current state-action pair. This is the network being actively trained.
- Target Network: Provides stable target Q-values $Q(s', a'; \theta_t)$ to calculate the loss. Its weights θ_t are updated less frequently, either by copying the policy network's weights every N steps or through a soft update mechanism.

Without this two-network approach, training would become unstable. Using the same network for both estimating and optimizing Q-values creates a feedback loop: errors in initial estimates reinforce themselves, leading to divergence and poor performance. By decoupling the target calculation from the updates, the target network ensures more stable training.

Another important innovation in Deep Q-Learning is the introduction of experience replay and replay memory. Instead of training the neural network on each state-action pair immediately after it is encountered, the agent stores these experiences—containing the state, the action taken, the reward received, and the next state—in a replay memory. This memory accumulates a history of interactions, which can then be randomly sampled during training. The purpose of this random sampling is to break the temporal correlation between consecutive experiences, which could bias the learning process. By replaying experiences from different points in time, the agent can objectively learn the relationships between actions and their long-term rewards, leading to more effective and stable training.

2.3. DEEP RL AND DEEP Q-LEARNING

Deep Q-learning pseudo code from the original paper [25] is presented in Algorithm 2.

Algorithm 2 Deep Q-learning with Experience Replay

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
  Initialise sequence  $s_1 = x_1$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
  for  $t = 1, T$  do
    With probability  $\varepsilon$  select a random action  $a_t$ 
    otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
    Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
    Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
    Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
    Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
    Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non terminal } \phi_{j+1} \end{cases}$ 
    Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$ 
  end for
end for

```

DOUBLE DEEP Q-NETWORK

In their work, [39], Hasselt *et al.* (2016) introduced Double Deep Q-Network (DDQN) as a significant improvement to the vanilla DQN algorithm to address a key limitation. In traditional Q-learning, selecting the maximum over estimated Q-values, as shown in Eq. 2.9, implicitly estimates the maximum expected value. However, this approach systematically introduces a maximization bias, as over-estimations are amplified during the learning process. When combined with bootstrapping, where learning relies on previous estimates, this bias can lead to instability and inaccuracies in value estimation.

The proposed solution, known as Double Q-learning, employs two separate Q-value estimators that update each other independently. By leveraging these independent estimators, the algorithm reduces bias in Q-value estimates. This is achieved by using two separate networks, which decouple action selection and evaluation. The first network, called *Online* network, selects the best action a^* for the next state s' , while the second network, called the *Target* network, evaluates

Algorithm 3 Double Q-learning (Hassel et al., 2016)

```

Initialize primary network  $Q_\theta$ , target network  $Q_{\theta'}$ , replay buffer  $\mathcal{D}$ ,  $\tau \ll 1$ 
for each iteration do
  for each environment step do
    Observe state  $s_t$  and select  $a_t \sim \pi(a_t, s_t)$ 
    Execute  $a_t$  and observe next state  $s_{t+1}$  and reward  $r_t = R(s_t, a_t)$ 
    Store  $(s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$ 
  end for
  for each update step do
    sample  $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$ 
    Compute target Q value:
       $Q^*(s_t, a_t) \approx r_t + \gamma Q_\theta(s_{t+1}, \arg \max_{a'} Q_{\theta'}(s_{t+1}, a'))$ 
    Perform gradient descent step on  $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$ 
    Update target network parameters:
       $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$ 
  end for
end for

```

that action using the following equation:

$$Q_o(s, a) \leftarrow Q_o(s, a) + \alpha \left[r + \gamma Q_t(s', \arg \max_{a'} Q_o(s', a')) - Q_o(s, a) \right] \quad (2.13)$$

where Q_o and Q_t represent the online and target networks, respectively. Instead of directly taking the maximum Q-value, DDQN uses the online network for action selection and the target network for evaluation. This approach helps mitigate overestimation, leading to more stable training.

To further enhance stability, the target network is updated gradually using a soft update mechanism rather than being replaced entirely at each step. While the online network is updated at every training iteration, the target network is updated slowly to track the online network's parameters over time. This is done using an exponential moving average with a small update factor τ , following the equation:

$$\theta_t \leftarrow \tau \theta_o + (1 - \tau) \theta_t \quad (2.14)$$

where θ_o and θ_t represent the parameters of the online and target networks, respectively. This approach prevents drastic changes in the target values, reducing instability and improving convergence. By ensuring that the target network evolves smoothly, the agent learns from a more consistent objective, further mitigating overestimation bias and improving policy learning.

2.3. DEEP RL AND DEEP Q-LEARNING

The pseudo code for Double Q-learning is presented in Algorithm 3. Here, $Q_{\theta'}$ is utilized for selecting actions, while Q_{θ} is responsible for evaluating them. The mean squared error between the Q-values (Q_{θ} and Q^*) is minimized, and the parameters θ are periodically or gradually copied into θ' .

3

Literature Review

This chapter presents a comprehensive literature review to outline the current state of research and technological advancements in the field. It begins with an overview of the foundations of Bike Sharing Systems, followed by a comparison of static and dynamic rebalancing approaches. The discussion then transitions to the latest advancements in DRL solutions for mobility optimization to address the challenges explored in this study.

3.1 OVERVIEW OF BIKE SHARING SYSTEMS

In 2007, urbanization reached a significant moment, with over half of the global population living in cities for the first time [30]. This proportion is expected to rise to 70% by 2050, defining the Urban Age, a time when cities serve as economic centers, driving population growth and contributing substantially to national Gross Domestic Product (GDP) [12]. The urgency to transition toward a more sustainable world is more critical than ever. The growing need to reduce carbon emissions, minimize reliance on fossil fuels, and embrace more sustainable, efficient, and healthier lifestyles is pushing governments to adopt greener solutions. Historically, car-dependent urban planning has led to high levels of environmental pollution, sedentary behaviors, and heightened vulnerability to the impacts of climate change [28]. This car-centric model of urban development has exacerbated issues like air pollution, traffic congestion, and poor public health, making it imperative to reimagine cities with a focus on

3.1. OVERVIEW OF BIKE SHARING SYSTEMS

sustainable mobility.

An example of such a reimagining is the Barcelona Superblock model, which aims to reclaim public space, reduce motorized transport, promote sustainable mobility, and encourage active lifestyles. The Superblock model also enhances urban greening and seeks to mitigate the effects of climate change [27]. By restructuring urban layouts to prioritize pedestrians and cyclists over cars, the model contributes to a healthier, more livable city and fosters greater social interaction within neighborhoods.

Nevertheless, urban mobility remains essential for modern society, and individuals still need efficient and fast transportation options to commute, particularly for work. Traditional solutions, such as building subways, trams, and improving public services, have been employed in many cities to address these needs. However, these large-scale infrastructure solutions are not always feasible in smaller, densely built environments or in cities with complex, historic layouts. These scenarios face unique challenges that often make traditional transport solutions difficult to implement [5].

Given these constraints, alternative mobility solutions must be considered to optimize urban transport efficiency. As highlighted by Fishman in Bike Share [12], one of the key advantages of bicycles is their superior space efficiency relative to other transport modes. Figure 3.1 illustrates the space efficiency of various vehicles within a 3.5-meter lane: while cars can accommodate around 2,000 people per hour, buses can transport up to 9,000, and bicycles can carry as many as 14,000. Although public transportation remains the most efficient overall, bicycles offer the highest efficiency among individually operated vehicles, providing flexibility without the constraints of fixed schedules.

In response to these challenges, BSSs have surged in popularity in recent years. These systems provide an innovative and efficient form of urban mobility, acting as a missing "last-mile" link between public transportation hubs and final destinations. They offer both conventional and electric bikes, often equipped with smart features such as Global Positioning System (GPS) for real-time tracking, Radio-Frequency Identification (RFID) tags, and Internet of Things (IoT)-enabled devices for locking and usage monitoring. Designed for durability and low maintenance, the bikes commonly include features like puncture-resistant tires, adjustable seats, and robust frames. Most BSSs integrate seamlessly with user-friendly mobile applications, enabling users to locate, unlock, and pay for bikes, as well as providing information on nearby docking stations, trip history,

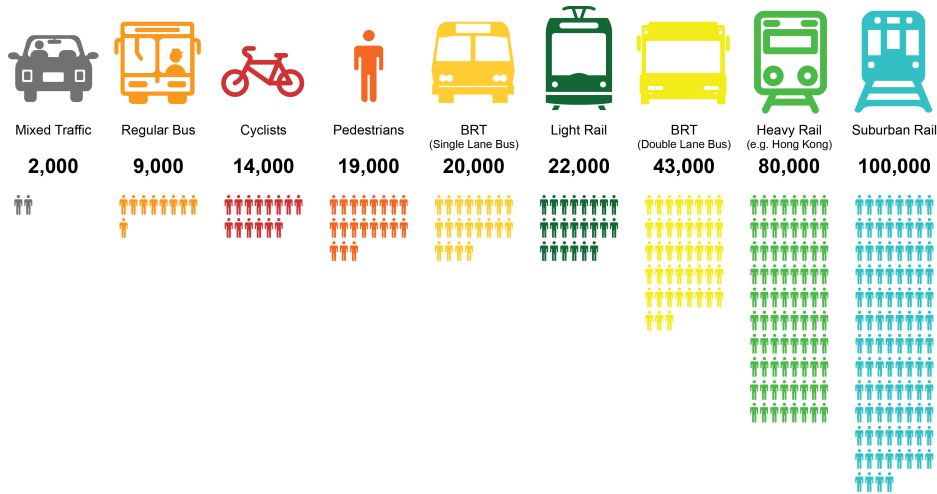


Figure 3.1: Carrying capacity of different modes, 3.5 m carriageway [12].

and route suggestions.

BSSs are categorized into two main types: *docked* and *dockless* systems. *Docked* systems require users to pick up and return bikes at designated stations, ensuring organized parking and simplifying management for service providers. However, uneven station distribution may limit convenience and discourage usage in underserved areas. *Dockless* systems, on the other hand, allow users to pick up and drop off bikes anywhere within a designated area, offering greater flexibility. This approach introduces challenges such as rebalancing bike availability, preventing improper parking, and maintaining urban order. Both systems play vital roles in reducing reliance on private vehicles, supporting public transport integration, and fostering greener, healthier cities.

By offering a convenient and sustainable mode of transport, BSSs complement existing public transportation networks, providing users with a seamless and accessible experience. Their ease of use, 24/7 availability, and integration with modern technologies such as smartphones and contactless payment systems have made them an increasingly popular choice, particularly in cities where other mobility infrastructure may be limited or underdeveloped [24]. Beyond convenience, BSSs contribute to reducing traffic congestion, improving air quality, and supporting public health by encouraging physical activity. As cities integrate these systems into broader urban mobility strategies, they not only provide an eco-friendly alternative to private car use but also enhance equitable access to transportation, fostering more sustainable, livable environments [33].

3.2 STATIC VS. DYNAMIC REBALANCING METHODS

The Meddin Bike-Sharing World Map [22] is a comprehensive global directory of BSSs, tracking short-term bike and e-bike rental networks across various cities. This platform provides regularly curated data to reflect the industry's rapid expansion. As of January 2025, it reports over 3,000 active BSSs worldwide, with approximately 10 million shared bikes deployed.

One of the most critical challenges for BSS operators is rebalancing—the process of redistributing bikes across stations to ensure availability and meet demand while minimizing service failures, such as users arriving at an empty station. Rebalancing strategies generally fall into two categories: Static Bike Repositioning Problems (SBRPs) and Dynamic Bike Repositioning Problems (DBRPs).

- **SPRB** involves scheduled redistribution, where service vehicles collect, recharge, and relocate bikes at predetermined times throughout the day. Typically, these operations take place overnight, with limited intervention during the day. Since static rebalancing does not account for real-time fluctuations in demand, it focuses on preparing station inventories in advance to accommodate expected usage patterns for the following day.
- **DBRP**, in contrast, is demand-driven and operates in real time. Because demand satisfaction depends on the ever-changing status of stations and the stochastic nature of customer rentals and returns, DBRP presents greater operational challenges. This approach requires a deep understanding of system dynamics to anticipate demand fluctuations throughout the day. Service vehicles continuously monitor system conditions and actively pick up, recharge, and drop off bikes as needed, ensuring a more responsive and adaptive redistribution strategy.

While dynamic rebalancing offers greater efficiency and adaptability, it also demands advanced data analysis, predictive modeling, and real-time operational adjustments to be effectively implemented.

STATIC REBALANCING

Early research on the SBRP includes the work of Raviv *et al.* (2013) [35], one of the first attempts to formally model the problem. They formulated the static repositioning problem as a Mixed-Integer Linear Programming (MILP), focusing

on minimizing the cost of repositioning while ensuring stations meet demand constraints. Their approach incorporated practical constraints such as vehicle capacity limits and station imbalances, setting a foundation for optimization-based solutions. Around the same time, Chemla *et al.* (2013) [6] introduced the static rebalancing problem as a generalization of the many-to-many pickup and delivery problem, framing it within the broader logistics and operations research domain. Following these foundational works, O’Mahony and Shmoys (2015) [29] presented a data-driven approach tailored to New York City’s Citibike system. They analyzed historical usage data to identify usage patterns, classifying stations as producers, consumers, or balanced, and formulated two rebalancing strategies. The mid rush-hour rebalancing strategy employed a covering problem approach with bike trailers, while the overnight rebalancing was modeled as a bipartite graph covering problem to optimize full truck-load routes. Building on these early models, Zhang *et al.* (2018) [42] extended static rebalancing to include the collection of bikes requiring maintenance, employing an Integer Linear Programming (ILP) formulation and a hybrid Discrete Particle Swarm Optimization (DPSO) algorithm to optimize both rebalancing and repair processes. Further refining the problem, Bruck *et al.* (2019) [3] introduced a variant where temporary storage or provision of bikes at stations was prohibited, reflecting a more constrained yet realistic operational environment and leading to the development of three exact algorithms. Another significant contribution came from Liu *et al.* (2016) [21], who proposed a data-driven approach integrating multiple sources—such as weather conditions, station connectivity, and historical demand patterns—to predict bike flows. Their hierarchical optimization model, leveraging clustering techniques and a Mixed-Integer Non-linear Programming (MINP) framework, demonstrated that intelligent demand forecasting combined with optimized vehicle routing can significantly enhance system performance.

DYNAMIC REBALANCING

One of the earliest attempts at real-time rebalancing was made by Conrado *et al.* (2012) [9], who formulated the problem as a Mixed-Integer Program (MIP) and developed heuristic-based solutions to improve computational feasibility. However, their approach still relied on precomputed repositioning plans rather than enabling immediate, on-the-fly decision-making. A more

3.2. STATIC VS. DYNAMIC REBALANCING METHODS

adaptive approach came with Schuijbroek *et al.* (2017) [36], who introduced a demand-driven clustering model to dynamically allocate service vehicles. While incorporating spatial dependencies and demand uncertainty, their model still structured decisions around predefined inventory balancing rather than offering truly dynamic responses. Ghosh *et al.* (2017) [14] advanced the field by applying RL to optimize repositioning based on historical demand patterns, but their model was still limited to periodic updates rather than true real-time adjustments. A significant step forward came with Chiariotti *et al.* (2018) [8], who introduced a dynamic rebalancing strategy that used historical data to predict network conditions and employed Birth-Death Processes (BDPs) to estimate when rebalancing was necessary. By applying graph theory to determine optimal rebalancing paths, they demonstrated that a dynamic approach, guided by time-varying demand, was more effective than static strategies. This shift in focus from pre-scheduled to demand-driven rebalancing marked an important advance in real-time adaptation. Building on this, Chiariotti *et al.* (2020) [7] proposed a framework combining traditional dynamic rebalancing with user incentives like gamification. By jointly optimizing operator-driven rebalancing (using trucks) and user behavior through incentives, they achieved better service quality at a lower cost than relying solely on rebalancing efforts. More recently, Lian *et al.* (2024) [20] introduced a spatio-temporal RL algorithm for dynamic rebalancing with multiple vehicles, treating the problem as a Multi-agent MDP in a continuous-time framework. This approach allows for both independent and cooperative vehicle rebalancing, significantly improving flexibility and adaptability by enabling the model to learn optimal actions based on the current state. It addresses the limitations of time-discretized and synchronous methods used in earlier models. Meanwhile, Cederle *et al.* (2024) [4] proposed an RL approach for managing shared micromobility services, specifically focusing on fairness. Their model learns to balance efficiency with equitable access to bikes across locations and user groups, emphasizing the importance of fairness in rebalancing.

Despite these advancements, many existing dynamic bike rebalancing solutions still introduce simplifications, relying on periodic updates, clustering techniques, or predefined heuristics, which limit their ability to respond instantaneously to real-time demand fluctuations.

3.3 DEEP REINFORCEMENT LEARNING IN MOBILITY OPTIMIZATION

While the previous section explored traditional methods for solving the rebalancing problem, such as MILP, ILP, MINP, MIP and RL, this section focuses on the latest advancements in the rapidly growing field of DRL for BSS rebalancing. As an emerging approach, DRL has quickly gained traction due to its ability to optimize complex, real-time decision-making in dynamic environments like BSSs. DRL has the potential to significantly improve rebalancing strategies, enabling systems to autonomously adapt to fluctuations in demand, handle large-scale operations, and make near-instantaneous decisions, offering more practical and responsive solutions for BSSs.

An early attempt to apply DRL to the rebalancing problem was made by Pan *et al.* (2019) [32], who introduced a DRL framework focused on incentivizing users to rebalance *dockless* BSSs. Their model used DRL to learn optimal incentive prices for influencing user behavior, suggesting alternative pickup and drop-off locations to improve system balance. While this approach did not tackle all operational complexities, such as vehicle routing, it demonstrated the potential of DRL in optimizing shared micromobility management through user incentives. Building on this, Yin *et al.* (2023) [41] proposed a DQN to optimize dynamic rebalancing policies in large-scale systems with multiple repositioning vehicles. Their model incorporated real-time spatiotemporal data to generate intertwined rebalancing actions, including both repositioning and routing decisions. The use of Divvy Bike data from Chicago demonstrated the model's ability to respond effectively to fluctuating demand, reinforcing the viability of DRL for large-scale BSSs. Further advancements came with Pan *et al.* (2023) [31], who introduced BRIS, a DRL framework designed to handle the complexities of rebalancing in *dockless* BSSs by leveraging user incentives rather than vehicle-based rebalancing. This work emphasized the explicit incorporation of spatiotemporal data into the DRL framework through two key components: Bike Gym, a simulation environment for training and evaluation, and STRPA, a DRL agent that learns optimal repositioning strategies while managing large state spaces through a hierarchical learning approach. By framing the rebalancing problem as an MDP, this study demonstrated how hierarchical DRL can enhance decision-making efficiency in highly dynamic urban environments.

3.3. DEEP REINFORCEMENT LEARNING IN MOBILITY OPTIMIZATION

The studies mentioned above represent some of the few existing works in the application of DRL to BSS rebalancing, highlighting how this research area remains largely unexplored. While these approaches introduce promising methodologies, they still fail to capture the full complexity of real-world operations, where factors such as traffic, user unpredictability, and operational constraints play a crucial role. This gap in the literature leaves room for further advancements, opening opportunities to develop more sophisticated and adaptable solutions that better reflect the dynamic and uncertain nature of BSSs.

4

System Model

This chapter formally defines the BSS model, providing the foundation for the proposed framework. The system is represented as a directed, weighted graph, where nodes correspond to stations and edges capture bike movements. Key elements such as bike availability, capacity constraints, and battery dynamics are introduced. To model demand and arrivals, a Continuous-Time Markov Chain is used, capturing the stochastic nature of user-driven bike flows. While a rebalancing truck operates in the system, its actions are treated separately from the Continuous-Time Markov Chain model. This structured approach enables a realistic and adaptable representation of the system for subsequent analysis. For better readability and reference, all mathematical symbols used in this chapter are reported in Tables 1 and 2 in the appendix.

4.1 MATHEMATICAL REPRESENTATION OF THE SYSTEM

The BSS is modeled as a connected, directed, and weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ [16], where $\mathcal{V}$ represents the set of nodes, each corresponding to a station. In this *dockless* BSS, every node $n \in \mathcal{V}$ is treated as a station and is characterized by its current bike occupancy. The state of a station at time $t \in \mathbb{R}^+$ is represented by $S_n(t)$, denoting the number of available bikes at node n . The system-wide bike availability at time t is given by the vector:

$$\mathbf{S}(t) = (S_1(t), S_2(t), \dots, S_N(t)), \quad N = |\mathcal{V}| \quad (4.1)$$

4.1. MATHEMATICAL REPRESENTATION OF THE SYSTEM

where N is the total number of nodes in the network. Each node has an associated capacity $S_n^{\max} \in \mathbb{N} \cup \infty$, representing the maximum number of bikes it can accommodate. In a *docked* system, $S_n^{\max}$ is finite, while in a *dockless* system, it is unbounded. The set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ represents the directed edges between pairs of stations, where an edge $(n, m) \in \mathcal{E}$ exists if there is a direct travel route from station n to station m . Each edge (n, m) is weighted by a non-negative real value $w_{nm} \in \mathbb{R}^+$, which encodes the distance, travel time, or any relevant cost of traveling between the two stations. The direction of the edge specifies the permitted direction of travel within the system, ensuring that the graph $\mathcal{G}$ accurately represents both the physical layout of the stations and the constraints governing bike movement.

A matrix $L \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ is used to represent the distances between pairs of stations in the BSS, where each element ℓ_{nm} denotes the distance between nodes $(n, m) \in \mathcal{V}$. Since the system is directed, the matrix L is generally non-symmetric, meaning that the distance from node n to node m may differ from the distance in the reverse direction. The travel time between two nodes is determined by multiplying the weight of the corresponding edge by a given travel speed, reflecting the time required to traverse the distance. To capture temporal variations, the day is divided into T equally sized time intervals, referred to as time slots, indexed by $\tau \in \{1, 2, \dots, T\}$. Each time slot corresponds to a fixed time interval $\mathcal{I}_\tau = [t_\tau^{\text{start}}, t_\tau^{\text{end}})$, allowing the system to model dynamic behavior throughout the day. The system evolves in continuous time, represented by $t \in \mathbb{R}^+$, where the state of the system changes continuously to reflect dynamic events such as bike movements, arrivals, and departures.

The set of all bikes in the system is denoted by $\mathcal{B}$, and each bike $b \in \mathcal{B}$ has an associated position at time t denoted by $p_b(t) \in \mathcal{V}$, where $p_b(t)$ indicates the station where the bike is located. The battery charge state of bike b at time t is represented by $c_b(t) \in [0, C_{\max}^b]$, where $C_{\max}^b$ is the maximum battery capacity. For simplicity, both the remaining battery charge and the maximum capacity are measured in units of distance (e.g., kilometers). If a bike b travels from station n to station m and have enough charge to complete the trip, i.e. $c_b(t) - \ell_{nm} > 0$, the remaining battery charge at time $t + \varepsilon$ is computed as

$$c_b(t + \varepsilon) = c_b(t) - \varepsilon \cdot v, \quad \varepsilon \in \left[0, \frac{\ell_{nm}}{v}\right] \quad (4.2)$$

where ℓ_{nm} is the distance between nodes n and m as defined in the distance

matrix L , and v is the mean travel velocity. This equation ensures that the battery consumption is proportional to the distance traveled.

The velocity v is sampled from a truncated Gaussian distribution characterized by a mean μ_v , variance σ_v^2 , and lower and upper bounds $v_{\min}$ and $v_{\max}$. The probability density function of a truncated Gaussian distribution is given by:

$$f(v \mid \mu_v, \sigma_v, v_{\min}, v_{\max}) = \frac{1}{\sigma_v} \frac{\varphi\left(\frac{v - \mu_v}{\sigma_v}\right)}{\Phi\left(\frac{v_{\max} - \mu_v}{\sigma_v}\right) - \Phi\left(\frac{v_{\min} - \mu_v}{\sigma_v}\right)} \quad (4.3)$$

for $v_{\min} \leq v \leq v_{\max}$ where $\varphi(x)$ is the probability density function of the standard Gaussian distribution, and $\Phi(x)$ is its cumulative distribution function. The truncated Gaussian ensures that the velocity remains within realistic limits while capturing variability around the mean.

In addition to the bikes, the system includes a truck responsible for rebalancing and recharging the bikes across the network. The truck is treated as a mobile agent that moves along the edges of the graph $\mathcal{G}$, similar to the bikes. The truck's position at time t is denoted by $p_{\text{truck}}(t) \in \mathcal{V}$, representing the node where the truck is currently located. The truck has an associated load $K_{\text{truck}}(t) \in \{0, \dots, K_{\max}\}$, which denotes the number of bikes it is carrying at time t . The maximum capacity $K_{\max}$ restricts the number of bikes the truck can carry, ensuring that $K_{\text{truck}}(t) \leq K_{\max}$ at all times. The truck also has a battery charge state $c_{\text{truck}}(t) \in [0, C_{\max}^{\text{truck}}]$, with a maximum capacity of $C_{\max}^{\text{truck}}$, both measured in units of distance (e.g., kilometers). The battery consumption of the truck is proportional to the distance traveled, similar to the bikes in Eq. (4.2). If the truck travels from node n to node m , the battery charge at time $t + \varepsilon$ is given by

$$c_{\text{truck}}(t + \varepsilon) = c_{\text{truck}}(t) - \varepsilon \cdot v, \quad \varepsilon \in \left[0, \frac{\ell_{nm}}{v}\right] \quad (4.4)$$

where ℓ_{nm} and v are defined as above for the bikes.

The dynamics of the truck include three main operations: loading bikes, unloading bikes, and recharging bikes at stations. When the truck arrives at a station n , it can perform the following operations:

1. **Load bikes:** The truck picks up $\delta_{\text{load}}(t)$ bikes from station n , updating the state as

$$S_n(t) \leftarrow S_n(t) - \delta_{\text{load}}(t) \quad \text{and} \quad K_{\text{truck}}(t) \leftarrow K_{\text{truck}}(t) + \delta_{\text{load}}(t)$$

subject to the constraints

$$\begin{cases} S_n(t) \geq \delta_{\text{load}}(t), \\ K_{\text{truck}}(t) + \delta_{\text{load}}(t) \leq K_{\text{max}} \end{cases}$$

2. **Unload bikes:** The truck drops off $\delta_{\text{unload}}(t)$ bikes at station n , updating the state as

$$S_n(t) \leftarrow S_n(t) + \delta_{\text{unload}}(t) \quad \text{and} \quad K_{\text{truck}}(t) \leftarrow K_{\text{truck}}(t) - \delta_{\text{unload}}(t)$$

subject to the constraints

$$\begin{cases} K_{\text{truck}}(t) \geq \delta_{\text{unload}}(t), \\ S_n(t) + \delta_{\text{unload}}(t) \leq S_n^{\text{max}} \end{cases}$$

3. **Recharge bikes:** The truck can instantly recharge any bike b in its load or at the current station. If the truck recharges a bike b with an initial battery state $c_b(t)$, the battery is brought instantly to full capacity,

$$c_b(t + \varepsilon) = C_{\text{max}}^b$$

These operations are performed immediately and require no delay in time.

4.2 MODELING DEMAND AND ARRIVAL PATTERNS WITH A CONTINUOUS-TIME MARKOV CHAIN

An accurate representation of demand is important for understanding the dynamics of a *dockless* BSS. In this work, the system is modeled as a piecewise-homogeneous Continuous-Time Markov Chain (CTMC), where state transitions occur at rate-dependent intervals, and parameters may vary across different time slots. This formulation provides a rigorous yet tractable framework for capturing the stochastic movement of bikes.

Although the system physically includes a rebalancing truck, its actions are not incorporated into the CTMC in this work. The primary goal is to build an environment model from the bike's perspective, capturing user-driven demand and usage patterns, i.e., how bikes move through the network, become

depleted, and get used by customers. The truck's operations, such as picking up or recharging bikes, may still occur in the real system but are treated as exogenous events here. This simplifies the model so that it focuses on the stochastic processes directly driven by users.

To characterize the system state more precisely, the previously defined bike availability vector $\mathbf{S}(t)$ is refined by distinguishing between **active** and **depleted** bikes. Each station n keeps track of two separate counts: $S_n^a(t)$ for active bikes and $S_n^d(t)$ for depleted bikes. In practice, a threshold of 20% of the bike's maximum battery is adopted. Specifically, a bike b with battery $c_b(t)$ is considered active if $c_b(t) > 0.2 \cdot C_{\max}^b$ and depleted otherwise. Hence, $S_n^a(t)$ counts the bikes at station n with sufficient charge, while $S_n^d(t)$ counts those below the threshold. The total number of bikes at station n remains $S_n(t) = S_n^a(t) + S_n^d(t)$, though these categories are handled separately in the Markov chain.

The number of bikes currently moving between stations n and m is given by $\mathbf{T}(t) = \{T_{(n,m)} : n, m = 1, \dots, N, n \neq m\}$. Since only sufficiently charged bikes can start a trip, $T_{(n,m)}$ includes only those with enough charge. Once a bike enters transit, it keeps enough charge to reach its destination, becoming uncharged upon arrival only if the trip fully drains the battery.

To account for interactions with the external region, let $S_{\text{ext}}(t)$ represent bikes outside the system, assuming they are always fully charged. As previously defined, each station has a limited capacity. In contrast, the external region has infinite capacity, meaning $\mathcal{S}_{\text{ext}} = \mathbb{N}_0$, allowing an unbounded number of bikes to exist outside the predefined stations. The flows to and from this external region are tracked using $T_{(n,\text{ext})}(t)$ and $T_{(\text{ext},n)}(t)$. The complete state description at time t is then given by:

$$\mathbf{X}(t) = \left(\underbrace{S_1^a(t), S_1^d(t), \dots, S_N^a(t), S_N^d(t), S_{\text{ext}}(t)}_{\mathbf{S}(t)}, \underbrace{T_{(1,2)}(t), \dots, T_{(\text{ext},n)}(t)}_{\mathbf{T}(t)} \right). \quad (4.5)$$

The state space of the CTMC is given by $\mathcal{S} \times \mathcal{T}$. The set $\mathcal{S}$ represents all possible configurations of bikes at the stations and is defined as:

$$\mathcal{S} = \{(\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_N, \mathcal{S}_{\text{ext}}) \in \mathbb{N}_0^{N+1}\}. \quad (4.6)$$

where the notation $\mathbb{N}_0^{N+1}$ refers to the $(N+1)$ -dimensional space of non-negative

integers, this means that $\mathcal{S}$ consists of all possible ways to distribute bikes among N stations plus the additional external region.

The transit state space $\mathcal{T}$ accounts for all possible bike movements between stations and is defined as:

$$\mathcal{T} = \left\{ (\mathcal{T}_{(1,2)}, \mathcal{T}_{(1,3)}, \dots, \mathcal{T}_{(n,\text{ext})}, \mathcal{T}_{(\text{ext},n)}) \in \mathbb{N}_0^{N(N+1)/2} \right\}. \quad (4.7)$$

Each transit variable $\mathcal{T}_{(n,m)}$ represents the number of bikes currently traveling from station n to station m , constrained by $\mathcal{T}_{(n,m)} \in \{0, 1, \dots, B\}$, where $B = |\mathcal{B}|$ is the total number of bikes in the system. A value of zero indicates no active trips between the stations, ensuring that all bike transitions are properly accounted for within the Markovian framework.

STOCHASTIC DEMAND AND POISSON PROCESSES

BSS demand is inherently random, as users request bikes at unpredictable times throughout the day. To mathematically represent this randomness, Poisson point processes provide a well-suited framework for modeling event occurrences over time. A homogeneous Poisson point process is a type of stochastic counting process $X(t), t \geq 0$ that tracks the number of events occurring up to time t . A counting process is said to follow a homogeneous Poisson distribution with rate $\lambda > 0$ if it satisfies the following properties:

- $X(0) = 0$, meaning no events occur at time zero.
- The process has independent increments, meaning that the number of events in disjoint time intervals are independent random variables.
- The number of events occurring in any interval of length t follows a Poisson distribution with mean λt , such that

$$\Pr\{X(t) = x\} = \frac{(\lambda t)^x}{x!} e^{-\lambda t}. \quad (4.8)$$

This formulation implies that interarrival times, the time intervals between consecutive events, are independent and exponentially distributed with mean $1/\lambda$. Given that arrivals and departures in the CTMC model are governed by exponential distributions, the Poisson process naturally aligns with the stochastic structure of bike demand. Its properties also allow for easier simulation, as we will describe in Chapter 5.

Departures in the system are categorized into *internal* and *external* departures. The total departure rate from station n during time slot τ is given by

$$\lambda_n(\tau) = \lambda_n^{\text{int}}(\tau) + \lambda_n^{\text{ext}}(\tau), \quad (4.9)$$

where $\lambda_n^{\text{int}}(\tau)$ represents the rate of departures from station n to other stations within the system, while $\lambda_n^{\text{ext}}(\tau)$ accounts for trips that leave the modeled area. The internal departure rate is defined as

$$\lambda_n^{\text{int}}(\tau) = \lambda_n(\tau) \sum_{m \in \mathcal{V}} p_{n \rightarrow m}(\tau), \quad (4.10)$$

where $p_{n \rightarrow m}(\tau)$ denotes the probability that a trip originating at station n in time slot τ is destined for station m . Similarly, external departures are given by

$$\lambda_n^{\text{ext}}(\tau) = \lambda_n(\tau) p_{n \rightarrow \text{ext}}(\tau), \quad (4.11)$$

where $p_{n \rightarrow \text{ext}}(\tau)$ is the probability that a departing bike is headed outside the system. These probabilities satisfy the normalization condition

$$\sum_{m \in \mathcal{V}} p_{n \rightarrow m}(\tau) + p_{n \rightarrow \text{ext}}(\tau) = 1. \quad (4.12)$$

A user at station n initiates a trip at the rate $\lambda_n(\tau)$, provided that there is at least one charged bike available ($S_n^a(t) > 0$). A successful departure from n to m results in a transition from state

$$\mathbf{x} = (S_1^a(t), S_1^d(t), \dots, S_n^a(t), S_n^d(t), \dots, T_{(n,m)}(t), \dots)$$

to a new state $\mathbf{x}'$, where

$$S_n^a(t) \rightarrow S_n^a(t) - 1, \quad T_{(n,m)}(t) \rightarrow T_{(n,m)}(t) + 1, \quad \text{for } n \neq m. \quad (4.13)$$

To fully characterize the system's evolution, the infinitesimal generator matrix $Q(\tau)$ is defined, governing the transition rates between states. The corresponding transition rate at time t is then

$$Q(\tau)[\mathbf{x}, \mathbf{x}'] = \begin{cases} \lambda_n(\tau) p_{n \rightarrow m}(\tau), & \text{if } S_n^a(t) > 0, \mathbf{x}' = \mathbf{x} - e_{S_n^a}^a + e_{T_{(n,m)}} \\ 0, & \text{otherwise.} \end{cases} \quad (4.14)$$

Arrivals, just as departures, are divided into *internal* and *external* categories. The total arrival rate at station n during time slot τ is given by

$$\mu_n(\tau) = \mu_n^{\text{int}}(\tau) + \mu_n^{\text{ext}}(\tau). \quad (4.15)$$

Internal arrivals originate from other stations within the system and are given by

$$\mu_n^{\text{int}}(\tau) = \sum_{m \in \mathcal{V}} \lambda_m(\tau) p_{m \rightarrow n}(\tau), \quad (4.16)$$

while external arrivals, which represent bikes entering the system from an external region, are defined as

$$\mu_n^{\text{ext}}(\tau) = \lambda_{\text{ext}}(\tau) p_{\text{ext} \rightarrow n}(\tau). \quad (4.17)$$

As before, the probabilities governing arrivals from the external region satisfy the normalization constraint

$$\sum_{n \in \mathcal{V}} p_{\text{ext} \rightarrow n}(\tau) = 1. \quad (4.18)$$

Once a bike departs from station n to station m , it remains in transit for an exponentially distributed duration with rate

$$\gamma_{n \rightarrow m} = \frac{1}{\text{mean travel time from } n \rightarrow m} \approx \frac{\frac{1}{K} \sum_{k=1}^K v_k}{\ell_{nm}}, \quad (4.19)$$

where $\{v_k\}_{k=1}^K$ are sampled travel speeds from the truncated Gaussian distribution in Eq. (4.3) and ℓ_{nm} is the distance between stations n and m , obtained from the distance matrix L .

In the CTMC framework, transition rates should reflect the expected time required for a trip while maintaining the Markov property. Since travel speeds v are drawn from a truncated Gaussian distribution, the travel time ℓ_{nm}/v is a non-exponential random variable. However, rather than explicitly modeling the full distribution of travel times, the transition rate are approximated using the reciprocal of the mean travel time as in Eq (4.19). This formulation ensures that the expected time until transition from n to m remains consistent with empirical travel speeds while preserving the Markov assumption at the state level.

If $T_{(n,m)}(t) > 0$ bikes are currently in transit, the total rate at which bikes

complete their journey is given by $T_{(n,m)}(t) \cdot \gamma_{n \rightarrow m}$. Upon arrival, a bike is added to station m , leading to a transition from state

$$\mathbf{x} = (S_1^a(t), S_1^d(t), \dots, S_n^a(t), S_n^d(t), \dots, T_{(n,m)}(t), \dots)$$

to a new state $\mathbf{x}''$, where

$$T_{(n,m)}(t) \rightarrow T_{(n,m)}(t) - 1, \quad \text{for } n \neq m \quad (4.20)$$

and

$$\begin{cases} S_m^a(t) \rightarrow S_m^a(t) + 1 & \text{if } c_b(t) > 0.2 \cdot C_{\max}^b \\ S_m^d(t) \rightarrow S_m^d(t) + 1 & \text{otherwise.} \end{cases} \quad (4.21)$$

where $c_b(t) > 0.2 \cdot C_{\max}^b$ means that the battery is not depleted under 20% after the trip. The corresponding transition rate at time t in the matrix $Q(\tau)$ is then

$$Q(\tau)[\mathbf{x}, \mathbf{x}''] = \begin{cases} T_{(n,m)}(t) \gamma_{n \rightarrow m}, & \text{if } T_{(n,m)}(t) > 0, \mathbf{x}'' = \mathbf{x} + e_{S_m^a}^a - e_{T_{(n,m)}} \\ 0, & \text{otherwise.} \end{cases} \quad (4.22)$$

To facilitate analysis and computation, the embedded Markov chain is considered, which discretizes the continuous-time process by sampling states only at transition events. This transformation allows us to work with a Discrete-Time Markov Chain (DTMC) while preserving the underlying stochastic behavior of the original system. The transition probabilities for the embedded DTMC are derived from the generator matrix $Q(\tau)$ and are given by:

$$p(i, j; \tau) = \begin{cases} \frac{q(i, j; \tau)}{r_i(\tau)}, & \text{if } r_i(\tau) > 0 \text{ and } j \neq i, \\ 0, & \text{otherwise,} \end{cases} \quad (4.23)$$

where $r_i(\tau) = -q(i, i; \tau)$ represents the sojourn rate, i.e., the total rate at which the process exits state i . If $r_i(\tau) = 0$, the state is absorbing, meaning the process remains there indefinitely. In such cases, the embedded chain assigns $p(i, i; \tau) = 1$ to reflect the system's inability to transition out of the absorbing state. The transition matrix $P(\tau)$ for the embedded chain is then defined as:

$$P(\tau) = [p(i, j; \tau)]_{i, j \in \mathcal{X}}. \quad (4.24)$$

5

Proposed Framework

This chapter presents the implementation of the proposed system for dynamic bike rebalancing, building upon the concepts introduced in the previous chapters. The framework is structured around a fully dynamic environment that captures the real-time evolution of the BSS, serving as the foundation for the RL approach. Within this environment, the problem is formulated as an MDP, with particular attention given to the design of the reward function, which plays a crucial role in shaping the agent’s behavior. The decision-making process itself is governed by a DDQN, whose implementation details define how the agent learns and adapts to optimize rebalancing actions. By ensuring a realistic and adaptable simulation, this work aims to provide a robust framework for optimizing bike rebalancing in dynamic urban settings.

5.1 FULLY DYNAMIC FRAMEWORK

Designing a well-structured environment is crucial for the effectiveness of an agent’s decision-making process. The objective of this thesis is to develop a fully dynamic simulation that closely mirrors the real-world dynamics of a BSS. This environment must accurately represent trip patterns, capturing how bikes move between stations based on historical demand and arrival data. Additionally, it must incorporate real-world complexities such as system failures and stochastic variations in user behavior. Beyond serving as a foundation for RL, the environment also facilitates the development and analysis of key performance metrics,

5.1. FULLY DYNAMIC FRAMEWORK

enabling a deeper understanding of system behavior and the effectiveness of different rebalancing strategies.

GRAPH CONSTRUCTION AND PREPROCESSING

The first fundamental step in building the environment is determining how to represent the city's infrastructure. For this purpose, an accurate graph of Cambridge, Massachusetts, has been chosen, extracted from OpenStreetMap using the OSMNx library. The city infrastructure is represented as a weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, treating it similarly to what done in Chapter 4, where:

- $\mathcal{V}$ is the set of nodes representing intersections or significant waypoints,
- $\mathcal{E}$ is the set of edges representing road segments connecting these nodes,
- Each edge $e \in \mathcal{E}$ is assigned a weight $w(e)$, corresponding to the actual travel distance between its connected nodes.

To improve computational efficiency, the raw graph undergoes preprocessing. First, redundant complexities such as parallel streets for traffic separation are removed. The graph is then converted into an undirected representation, ensuring that every node remains accessible from any other. This simplification is justified by the observation that most streets in dense urban areas are bidirectional, and the approximation introduces negligible error while significantly reducing computational complexity.

Each node $n \in \mathcal{V}$ is treated as a bike station, meaning that bike pickups and drop-offs occur only at intersections. Trips that originate or terminate along a street are assigned to the nearest node, effectively consolidating bike movements at key locations in the network. Each station stores the following attributes:

- A unique ID n ,
- Geographic coordinates $(\text{lat}_n, \text{lon}_n)$,
- Maximum capacity C_n ,
- Request rate λ_n and arrival rate μ_n ,
- A list of available bikes.

Each edge is weighted by the shortest travel distance between the two connected nodes, forming a distance matrix L , similarly introduced in Chapter 4, where:

$$\ell_{nm} = \text{ShortestPath}_{\mathcal{G}}(n, m) \quad (5.1)$$

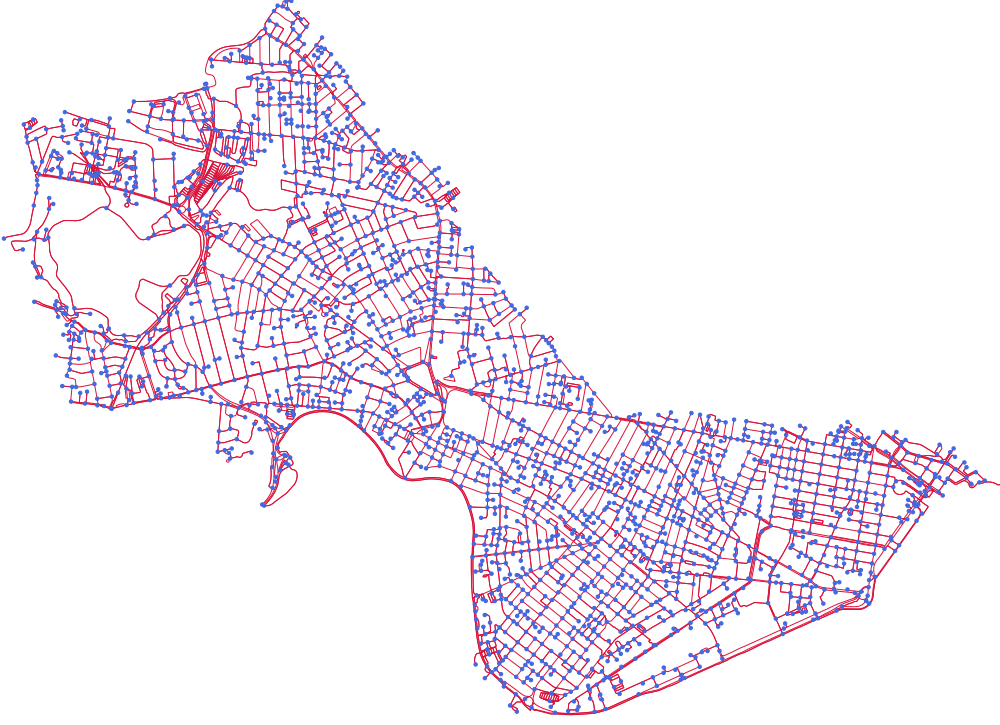


Figure 5.1: Graph representation of Cambridge, MA, used for the BSS environment. Data are extracted from OpenStreetMap.

and L is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix storing precomputed pairwise shortest path distances, enhancing routing efficiency.

The final processed graph covers approximately 16.57 km^2 and consists of more than 2,400 nodes, providing a detailed yet computationally efficient representation of the city. Figure 5.1 illustrates the resulting graph used in the environment, where the dots represent intersections and the lines depict the streets connecting them. This choice of graph structure is crucial, as it determines how accurately the environment reflects real-world conditions, directly influencing the feasibility and effectiveness of any rebalancing strategy.

EVENT-BASED TRIP SIMULATOR

The event simulator is a fundamental component of the fully dynamic framework, enabling the system to model trip departures and arrivals as discrete events in time. This approach closely mirrors the real-world behavior of a BSS, where trips are initiated and completed dynamically in response to fluctuating

5.1. FULLY DYNAMIC FRAMEWORK

user demand. Rather than relying on fixed intervals or artificial batching, the simulator ensures that trips are generated organically, reflecting the natural ebb and flow of requests throughout the day.

At its core, the event simulator relies on a Poisson point process, as described in Eq. (4.8), to model departure events based on the demand rate at each node. The Poisson process is widely used to represent random occurrences over time, making it well suited for modeling trip requests in a BSS. It is a common tool in queueing theory and has been applied in various domains, such as modeling customer arrivals at stores, call center traffic, and even seismic activity. Its ability to capture independent, randomly occurring events makes it an ideal choice for simulating user demand.

Given that trip requests occur between pairs of nodes, the natural approach would be to model each origin-destination pair ($n \rightarrow m$) as an independent Poisson process with rate $\lambda_{n \rightarrow m}$. This would mean simulating a large number of Poisson processes simultaneously, one for each possible trip in the system. However, this approach quickly becomes computationally intractable as the number of nodes increases, since the total number of processes grows quadratically with the number of stations.

To address this, a computationally efficient alternative is adopted using the merging property of Poisson processes [13]. Let $X_1(t), X_2(t), \dots, X_m(t)$ be independent Poisson processes with rates $\lambda_1, \lambda_2, \dots, \lambda_m$. The sum of these independent processes,

$$X(t) = X_1(t) + X_2(t) + \dots + X_m(t), \quad \text{for all } t \in [0, \infty),$$

is itself a Poisson process with rate equal to the sum of the individual rates, i.e.,

$$\lambda = \lambda_1 + \lambda_2 + \dots + \lambda_m.$$

Leveraging this property, instead of sampling from m independent Poisson processes, the simulator models a single Poisson process with a global rate equal to the sum of all node-to-node departure rates. This significantly reduces computational complexity while preserving the correct statistical distribution of trip departures. Each sample from this global Poisson process corresponds to a departure event, which is then assigned to a specific origin-destination pair according to the probability distribution of trips. In this way, a global departure

event is generated, encapsulating all trip requests into a single event-driven framework. This process, where a global departure event is generated and assigned to a specific trip, is illustrated in the "Departure events generation" step of Figure 5.2.

To account for trips originating from or heading toward external regions beyond the modeled city graph, these regions are treated as an additional virtual node. This ensures that trips involving external areas are integrated into the same probabilistic framework as internal trips. The external node has its own departure and arrival rates, which contribute to the overall Poisson process. By modeling external regions in this way, the simulator captures real-world scenarios where users enter or exit the BSS at peripheral locations.

Once a global departure event is generated, the simulator must determine both the origin and destination of the trip. Since the system tracks individual departure rates for each pair of nodes (n, m) , the probability of a trip occurring from node n to node m is obtained by normalizing the pairwise departure rates by the total system-wide departure rate λ_{global} :

$$p_{n \rightarrow m} = \frac{\lambda_{n \rightarrow m}}{\lambda_{\text{global}}}, \quad \sum_{n=1}^m \sum_{m=1}^m p_{n \rightarrow m} = 1. \quad (5.2)$$

This defines a joint Probability Mass Function (PMF) over all possible trips, where each departure-destination pair $(n \rightarrow m)$ has a probability proportional to its historical demand. A random sample from this PMF determines both the departure station and its corresponding destination, ensuring that busier travel routes are more likely to be selected. The assignment of origin and destination nodes, based on the joint PMF, is depicted in the "Origin and destination node assignment" step in Figure 5.2.

Since real-world trip demand fluctuates throughout the day, the simulator is designed to adapt over time. As introduced in Chapter 4, the day is divided into time slots, each associated with different demand patterns. To reflect this variation, a new Poisson process is sampled at the beginning of each time slot, adjusting the global rate and PMF according to the historical demand data for that specific period. This ensures that trip generation dynamically follows the expected temporal distribution of real-world demand.

Once the departures are generated for a specific time slot, the corresponding arrival times are determined based on the travel time between nodes n and m ,

5.1. FULLY DYNAMIC FRAMEWORK

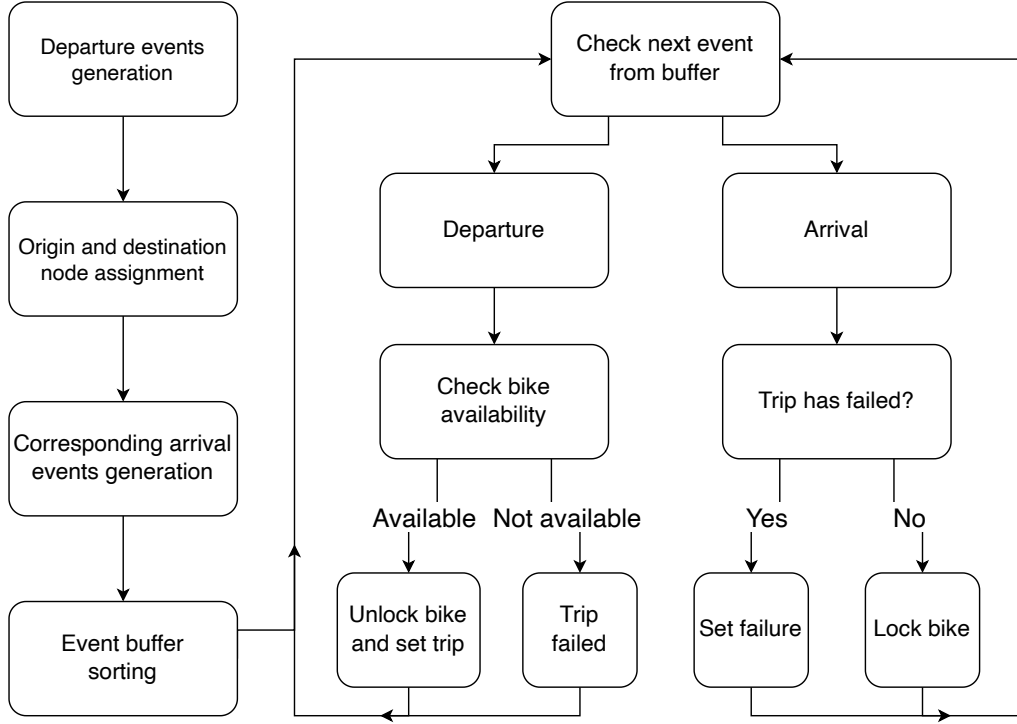


Figure 5.2: Flowchart of the event-driven simulator for the BSS: this diagram outlines the sequence of operations, from departure event generation to bike availability checks, trip execution, and arrival processing. The event buffer ensures orderly execution of events in the system.

computed as

$$t_{nm} = \frac{\ell_{nm}}{v}, \quad (5.3)$$

where ℓ_{nm} is the precomputed distance between the two nodes from L , and v is the trip velocity, sampled from the truncated Gaussian distribution described in Eq. (4.3). Regarding the velocity v , its distribution is centered at 15 km/h, with a standard deviation of 5 km/h, a minimum speed of 5 km/h, and a maximum speed of 25 km/h, ensuring realistic variability in travel times. Once all departures and their corresponding arrivals are generated, the events are sorted chronologically to enable efficient sequential processing. The determination of arrival times based on precomputed travel distances and the subsequent chronological sorting of events are shown in the "Corresponding arrivals generation" and "Event buffer sorting" steps of Figure 5.2.

With the event buffer fully populated and ordered, the simulation iterates through events in sequence. When a departure event is encountered, the sim-

ulator first checks the availability of a charged bike at the departing station. If no bikes are available, the system searches within a 250-meter radius for an alternative station with an available bike, simulating the real-world behavior of users willing to walk a short distance to find a bike. If a bike is found, it is removed from the starting station, marked as unavailable, and assigned to a newly created trip. The trip is then linked to both the departure and arrival events and marked as successful. If, instead, no bike was available at the time of departure and no nearby alternative was found, a trip is created and marked as failed.

When an arrival event is encountered in the event buffer, the corresponding trip is checked. If the trip was successful, the bike is assigned to the destination station, marked as available again, and its battery level is depleted based on the distance traveled. Otherwise, if the trip was unsuccessful a failure event is recorded. The simulation iterates through events in sequence, handling trip departures, bike availability checks, and arrivals as shown in the "Check next event from buffer" loop in Figure 5.2.

The event-driven nature of the simulator allows it to efficiently handle large-scale scenarios without requiring unnecessary computations at every time step. By aggregating individual Poisson processes into a single global process and leveraging precomputed distance matrices, the approach significantly reduces computational overhead. The event queue ensures that the system processes only relevant changes in state, making the framework scalable to larger networks while maintaining computational efficiency.

TRUCK SIMULATOR

Another key component of the simulator is the truck simulator, which operates alongside the bike simulator but is built on a specialized subgraph derived from the original city graph of Cambridge. The truck simulator operates on a subgraph $\mathcal{G}_T = (\mathcal{V}_T, \mathcal{E}_T)$, derived from $\mathcal{G}$ by dividing the city into a grid of zones Z of size 300×300 meters. Each zone Z_i is associated with a center node z_i , chosen as the closest node in $\mathcal{V}$ to the geometric center of the zone. The set of all zones is:

$$\mathcal{Z} = \{Z_1, Z_2, \dots, Z_J\}$$

where $J = |\mathcal{V}_T|$ denotes the total number of zones in the city.

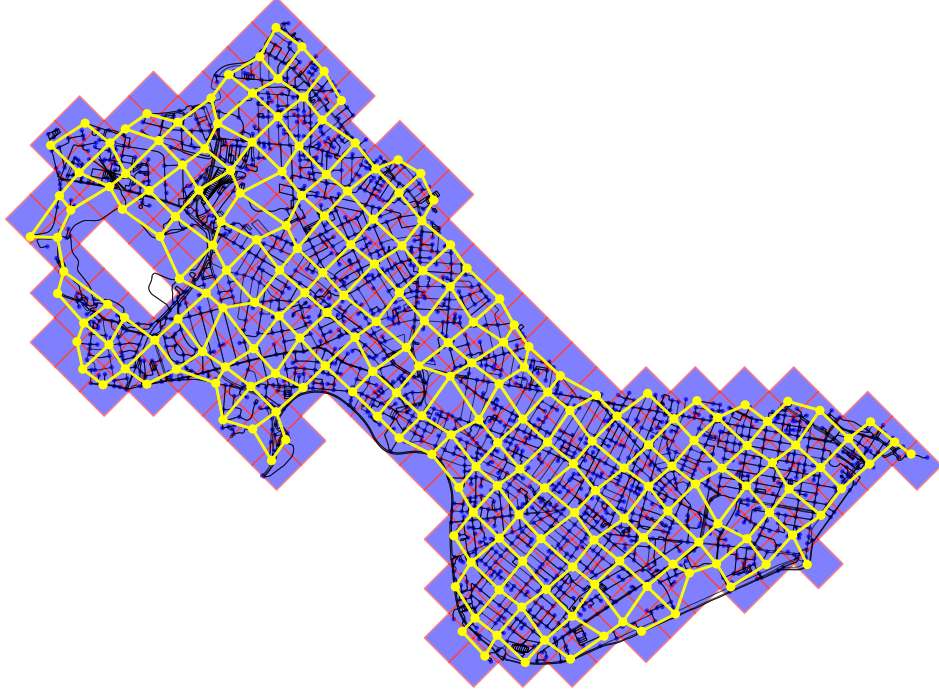


Figure 5.3: Grid representation of Cambridge, MA, used for the truck environment.

The corresponding center nodes are assigned following:

$$z_i = \arg \min_{v \in \mathcal{V}} \text{dist}(z_i, \text{center}(Z_i))$$

The edges connecting these central nodes reflect adjacency between zones:

$$\mathcal{E}_T = \{(z_i, z_j) \mid Z_i, Z_j \text{ are adjacent}\}.$$

To model realistic travel between zones, each edge (z_i, z_j) is assigned a weight $w_T(z_i, z_j)$, defined as the shortest path distance between z_i and z_j in the original graph $\mathcal{G}$:

$$w_T(z_i, z_j) = l_{z_i z_j} = \text{ShortestPath}_{\mathcal{G}}(z_i, z_j)$$

This forms a grid-like structure, allowing the truck to navigate efficiently between zones. However, the truck is also capable of moving along the original city graph $\mathcal{G}$ for fine-grained routing. The subgraph representing the truck network in Cambridge is illustrated in Fig. 5.3.

Truck trips are modeled similarly to bike trips. Given a start node $n \in \mathcal{V}$ and

a destination node $m \in \mathcal{V}$, the distance is given by ℓ_{nm} and the travel time t_{nm} is computed as in Eq. (5.3) with the velocity v following a truncated Gaussian distribution as previously described, with a mean velocity dependent on the traffic data, and with additional parameters including a minimum velocity of 10 km/h, a maximum velocity of 70 km/h, and a standard deviation of 5 km/h.

5.2 MDP

In this section, the control problem of dynamically rebalancing bikes is formulated through an MDP. This formulation enables capturing the evolving conditions of the BSS and guides the agent in selecting rebalancing actions that effectively redistribute bikes. The following subsections describe the primary components of this framework in detail.

The simulator, which models the real-world behavior of the BSS, operates through a continuous sequence of discrete events, such as user requests, arrivals, and departures, each unfolding dynamically according to their own occurrence times. However, the MDP is structured differently. Here, time is discretized into fixed intervals of 30 seconds, forming the temporal backbone of the decision-making process. When the agent selects an action, this action may span multiple discrete steps, depending on its real-world duration. Throughout the execution of these steps, the simulator continues evolving independently, processing all events sequentially in the background. Thus, the action chosen by the agent governs only its own operational timeline, while the broader environment simultaneously advances event-by-event. Formally, the number of discrete steps required to execute an action a with real-world duration t_a (in seconds) is computed as:

$$s_a = \left\lceil \frac{t_a}{30} \right\rceil$$

where s_a indicates the discrete time steps associated with action completion. This method ensures realistic representation of action durations while maintaining consistency within the MDP's structured temporal evolution.

STATE REPRESENTATION

The state representation is designed to capture the current configuration of the BSS and provide the agent with all relevant information for decision-making.

5.2. MDP

Each reward is uniquely associated with a specific system state, a property that is crucial for maintaining convergence. By organizing state information in this manner, the agent can learn to identify beneficial actions and avoid suboptimal ones. The agent's state is divided into two main components: one describing the broader environment and one focusing on the truck's position, status, and relevant temporal data.

For a broader view of the system, the state includes the subgraph $\mathcal{G}_T$, which represents the operational area of the truck as described in the previous section. Each node is assigned a set of features that collectively describe the zone's state. These features include:

- **Risk score:** A scalar reflecting the zone's overall health by quantifying the expected shortage of bikes over the next few hours. Let A be the predicted net demand (e.g., over three hours), and let U be the current supply of charged bikes in zone Z_i . The risk function $f(A, U)$ is defined as

$$f(A, U) = \frac{A - U}{A + U}. \quad (5.4)$$

This formulation ensures that the risk score remains bounded within $(-1, 1]$, where positive values indicate a potential shortage (i.e., demand exceeding supply), and negative values indicate a surplus. A value close to 1 signifies a severe shortage, while a value close to -1 reflects an excess of available bikes.

- **Recent Visit (Eligibility Trace):** A measure indicating how recently the truck has been present in a given zone. Unlike a simple binary flag that activates only upon entry, this score is set to 1 whenever the truck is currently in the zone, ensuring it remains at its maximum as long as the truck stays there. When the truck leaves, this value decays exponentially at each subsequent discrete step, reflecting the diminishing relevance of past presence:

$$E_{Z_i}(t + 1) = \lambda \cdot E_{Z_i}(t) \quad (5.5)$$

where $\lambda = 0.9968$ is the eligibility decay factor. Thus, the eligibility trace remains at 1 during continuous presence and begins exponential decay only after departure. This approach naturally prioritizes zones not recently visited. With the given decay rate, the eligibility trace drops to approximately 0.1 after roughly 359.5 minutes (~ 6 hours), indicating the duration needed

for the trace of a visit to become negligible.

- **Bike charge shortage score:** A measure of the fraction of bikes with battery levels below a specified threshold (e.g., 20% of full capacity). Let B_{Z_i} be the set of bikes in zone Z_i , let $c_b(t)$ denote the current battery level of bike $b \in B_{Z_i}$ and $C_{\max}^b$ its full charge capacity. The score $BCSC(Z_i)$ is given by

$$BCSC(Z_i) = \frac{\sum_{b \in B_{Z_i}} \mathbb{1} \left(\frac{c_b(t)}{C_{\max}^b} \leq 0.2 \right)}{\max(1, |B_{Z_i}|)}, \quad (5.6)$$

which represents the proportion of bikes in need of recharging or battery replacement to remain operable.

- **Truck position:** A binary indicator that is set to 1 if the truck is in zone Z_i and 0 otherwise, thereby tracking the truck's location within the network.

The truck-related information helps the agent localize itself in both space and time while also tracking its current load. To achieve this, the truck's internal state includes two one-hot encodings representing the current **day** and **hour**, allowing the agent to determine its temporal context. Additionally, a scalar value representing the **truck's load** is included, normalized by dividing the number of bikes carried by the truck's maximum capacity. The truck's state also incorporates an encoding of its **last action** and an encoding of the **current zone ID** to facilitate spatial localization. Finally, the truck's state includes **system bike saturation**, a binary indicator set to 1 if the system has reached the maximum number of droppable bikes while the truck is empty, meaning no further bikes can be placed in the network at that time. Otherwise, it is set to 0. This ensures that the agent can recognize when redistribution actions are constrained by system capacity.

ACTION SPACE

The agent interacts with the environment by selecting actions from a predefined set, each representing a real-world maneuver that the truck can execute. These actions allow the truck to navigate the operational area and manipulate bike distribution. Given the objective of maintaining a high degree of realism, each action requires time to complete, reflecting real-world constraints. The duration of each action is mapped to discrete steps in the MDP, ensuring that the

5.2. MDP

system progresses in structured time increments. The available actions can be classified into two main categories: movement actions, which govern the truck's navigation across the city, and operational actions, which allow it to interact with the BSS.

Movement Actions - The truck moves within the subgraph $\mathcal{G}_T$, transitioning between adjacent control zones. Each movement action requires a duration that corresponds to the real-world travel time between the truck's current node and the center node of the target zone. If a movement is not possible the action defaults to a STAY action.

- **STAY**: A null action in which the truck remains in its current location. Since no movement occurs, this action takes exactly one step.
- **UP**: Moves the truck from its current zone Z_i to the center node of the zone directly above, if such a zone exists. The time required is based on the shortest path distance between the two center nodes in the original city graph.
- **DOWN**: Moves the truck to the center node of the zone directly below, following the same distance-based time calculation.
- **LEFT**: Moves the truck to the center node of the zone to the left, if possible.
- **RIGHT**: Moves the truck to the center node of the zone to the right, if possible.

If an adjacent zone is unavailable (for example, if the truck is located in a zone at the border of the operational area), the corresponding movement action is censored, that is, removed from consideration. The optimal action is then selected from the remaining available actions.

Operational Actions - Beyond movement, the agent can interact with the BSS by picking up, dropping off, or recharging bikes. While the pickup and drop-off operations are assumed to be instantaneous, the duration of these actions is determined by the necessary truck movement. If an operation is not possible the action defaults to a STAY action.

- **PICK UP BIKE**: The truck selects a bike for pickup based on a weighted metric that prioritizes bikes with lower battery levels that are closer. Each bike b is assigned a weight:

$$w_b = \frac{1}{d_b + \epsilon} \cdot \left(1 - \frac{c_b}{C_{\max}^b}\right)$$

where d_b is the bike's distance from the truck, c_b is its current battery level, $C_{\max}^b$ is its full charge capacity, and ϵ is a small constant to avoid division by zero. The truck picks up the highest-priority bike, and the action duration is determined solely by the truck's movement.

- **DROP BIKE:** The truck deposits a bike at its current location. Since the drop-off itself is instantaneous, this action only consumes time due to preceding movement.
- **RECHARGE BIKE:** A composite action that consists of *PICK UP BIKE* and *DROP BIKE*, prioritizing bikes in urgent need of recharging. The same weighted selection function is applied, with a greater emphasis on bikes with critically low charge. The action's duration corresponds to the truck's movement required to complete both pickup and drop-off.

When performing an operational action, if the truck reaches its capacity (either full or empty), it automatically travels to the depot to either drop all bikes (if full) or pick up additional bikes (if empty). This additional trip increases the total action duration due to the extra travel involved. Furthermore, if the maximum number of bikes available in the system is reached and the agent does not have any bikes in its load, the *DROP BIKE* action becomes equivalent to the *STAY* action, meaning the agent takes no action but still receives the reward associated with the *STAY* action.

This formulation ensures that each action adheres to real-world execution constraints, enabling the agent to make informed decisions that balance operational efficiency with practical feasibility. By incorporating realistic action durations and movement limitations, the model encourages strategies that optimize rebalancing efforts while accounting for the inherent delays in physical operations. This structured approach not only enhances the realism of the learned policies but also ensures that the agent's behavior remains adaptable to the dynamic nature of the BSS, where timely and well-planned interventions are crucial for maintaining service reliability.

TRANSITION DYNAMICS

In the MDP framework, the transition dynamics describe how the system evolves from one state to another based on the agent's actions and the underlying stochastic processes governing the environment. As introduced in Chapter 2 (see

5.2. MDP

Eq. (2.5)), the transition function is formally defined by

$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

where S_t represents the system state at time step t , A_t is the action taken, and S_{t+1} is the resulting state. In principle, a well-defined transition function would allow an explicit computation of probabilities associated with each possible next state. However, due to the inherent complexity and stochastic nature of the *dockless* BSS, deriving an analytical expression for $\mathcal{P}_{ss'}^a$ is infeasible, as state transitions are driven by multiple interdependent and uncertain factors such as user demand, bike availability, vehicle rebalancing.

The difficulty in specifying an explicit transition function arises from the nature of the system itself, where bike requests and arrivals follow stochastic processes that are influenced by past and present supply distributions. User behavior introduces an additional layer of unpredictability, as real-world decisions often deviate from expected patterns, leading to fluctuating demand and non-stationary dynamics. Furthermore, the truck's rebalancing actions affect the system state by modifying bike availability in certain zones, but the impact of such interventions is neither immediate nor deterministic, as users continue to interact with the system in ways that cannot be fully predicted. Given this complexity, the transition probability is not explicitly parameterized but instead emerges naturally from the execution of a data-driven event-based simulator, which models trip requests, bike movements, and agent decisions in a way that reflects real-world system evolution.

Since the transition function is not explicitly available, conventional dynamic programming methods such as value iteration or policy iteration become impractical, as they require knowledge of all transition probabilities to perform explicit policy evaluation and improvement. Instead, the agent must rely on model-free RL, where policies are learned solely from sampled transitions. This approach is particularly well-suited to environments where dynamics are too complex to be captured in closed form, as it allows the agent to approximate optimal decision-making by interacting with the environment and refining its policy iteratively. The fact that transitions are not explicitly defined but rather generated through the simulator reinforces the necessity of DRL methods, which can approximate value functions and optimize policies without requiring an explicit model of the environment.

By structuring the transition function implicitly within the simulation, the agent experiences state changes as they would occur in a real deployment, leading to policies that are better aligned with the actual operational constraints of the system. This not only ensures that learned strategies are applicable in realistic conditions but also enables the model to capture emergent behaviors that arise from the stochastic nature of demand and supply dynamics. Unlike rule-based or handcrafted heuristics that may fail to generalize across varying conditions, learning from experience allows the agent to develop adaptive strategies that are robust to the inherent uncertainty in the system. Therefore, while the transition probability formally exists within the MDP formulation, it is neither explicitly computed nor estimated through parametric models. Instead, it is implicitly embedded within the stochastic execution of the simulator, making RL the most viable approach for optimizing rebalancing decisions in this highly dynamic environment.

REWARD FUNCTION

Designing a reward function for this RL setting poses significant challenges, as it must accurately represent complex operational factors, such as unmet demand and inefficient movements, while effectively managing edge cases. Various reward structures were evaluated, and although the piecewise approach presented here may not represent the perfect or theoretically correct formulation, it empirically demonstrated better performance than several alternative reward functions tested, effectively guiding the truck towards beneficial rebalancing actions.

The reward comprises multiple terms aimed at incentivizing actions that support high-demand (critical) zones and penalizing behaviors that stall the truck or negatively impact system efficiency. As previously detailed, each zone is characterized by a **risk score**, reflecting its urgency, and an **eligibility** trace indicating recent visitation frequency; these metrics are used directly here.

Stay Penalty and Position Penalty.

- *Stay Penalty*: Selecting a STAY action in a zone with a positive risk score incurs a penalty of -1.0 , discouraging idling in critical areas without intervention.
- *Position Penalty*: A mild penalty of -0.2 is applied if the agent remains in a non-critical zone (risk score ≤ 0) whose eligibility exceeds 0.7 , discouraging unnecessary lingering in recently visited locations.

Drop / Pick-Up Terms.

- *Drop Bonus*: Successfully dropping a bike in a zone with a positive risk score yields a reward of +1.0, reinforcing actions that mitigate imminent shortages.
- *Invalid Drop Penalty*: If a DROP action is attempted without available bikes (due to the truck's load being empty and the system's maximum capacity reached), a penalty of -1.0 is imposed.
- *Pick-Up Penalty/Bonus*: Picking up a bike from a critical zone results in a modest penalty of -0.1 , discouraging depletion of already under-supplied areas. Conversely, picking up from a non-critical zone yields a small bonus of $+0.1$, promoting strategic redistribution.

Movement Penalties (Two-Step Loops).

- *Directional Loop Penalty*: Executing a move action (*UP*, *DOWN*, *LEFT*, *RIGHT*) that directly reverses the previous movement triggers a penalty of -0.1 to prevent inefficient oscillations.
- *Load/Unload Loop Penalty*: Consecutive *PICK UP BIKE* or *DROP BIKE* actions in the same zone, effectively undoing a previous load/unload operation, incur a penalty of -0.25 .

Battery Charging Adjustments. Charging a nearly full bike results in a negative reward of -0.1 if done in a high-risk zone, and -0.3 otherwise, discouraging unnecessary actions. Conversely, charging a nearly depleted bike in a high-risk area provides a small positive bonus of $+0.5$, incentivizing beneficial charging activities.

Base Reward. A consistent $+1.0$ reward offset is included for all actions, providing a baseline positive reinforcement when other terms are neutral. This strategy has empirically shown to stabilize training by maintaining agent activity and promoting continued exploration.

The final reward, formed by aggregating these components, is computed as a scalar value at each timestep. While some reward elements may seem heuristic, extensive experimentation confirmed this configuration as the most effective method for fostering sustained, demand-aware rebalancing performance within this environment.

5.3 DOUBLE DQN WITH GRAPH NEURAL NETWORKS

The core of the agent’s decision-making process is the DQN, which computes Q-values to derive the optimal policy. In this implementation, a DDQN architecture is used. While DDQN was introduced in Chapter 2, this section focuses on the implementation of both the *Online* and *Target* networks, as well as the training process, covering the action selection strategy, a revised multi-step update for Q-values, the loss function, and the replay buffer.

DDQN IMPLEMENTING GRAPH ATTENTION NETWORKS

To effectively represent the environment, the DDQN integrates a Graph Neural Network (GNN) alongside a standard Feed Forward Neural Network (FFNN). Relying solely on a FFNN for feature extraction from a complex urban graph, such as that of Cambridge, would require an extensive number of neurons and layers, ultimately leading to inefficiencies and the loss of crucial spatial and relational information. The inclusion of a GNN allows the agent to focus on extracting meaningful structural and contextual features, enabling a more effective interpretation of the environment. This, in turn, enhances the agent’s ability to anticipate the future evolution of the BSS and respond dynamically to varying conditions.

To achieve this, the model leverages Graph Attention Networks (GATs), a class of GNNs that utilize attention mechanisms for adaptive feature aggregation. Introduced by Veličković *et al.* in 2018 [40], GATs improve upon traditional Graph Convolutional Networks (GCNs), Kipf *et al.* (2016) [18], by assigning different levels of importance to each neighboring node rather than treating them all equally. In standard GCNs, node features are typically updated by computing a uniform aggregation of neighboring node features, which may fail to account for the differing levels of influence exerted by various neighbors. GATs, on the other hand, employ a self-attention mechanism to compute attention coefficients, determining the relevance of each neighboring node’s features in the update process. These coefficients are normalized using a SoftMax function, ensuring that the final aggregation is weighted according to the relative importance of each neighbor.

This attention-based approach provides two major advantages. First, it enables the model to selectively emphasize more relevant information, improving

5.3. DOUBLE DQN WITH GRAPH NEURAL NETWORKS

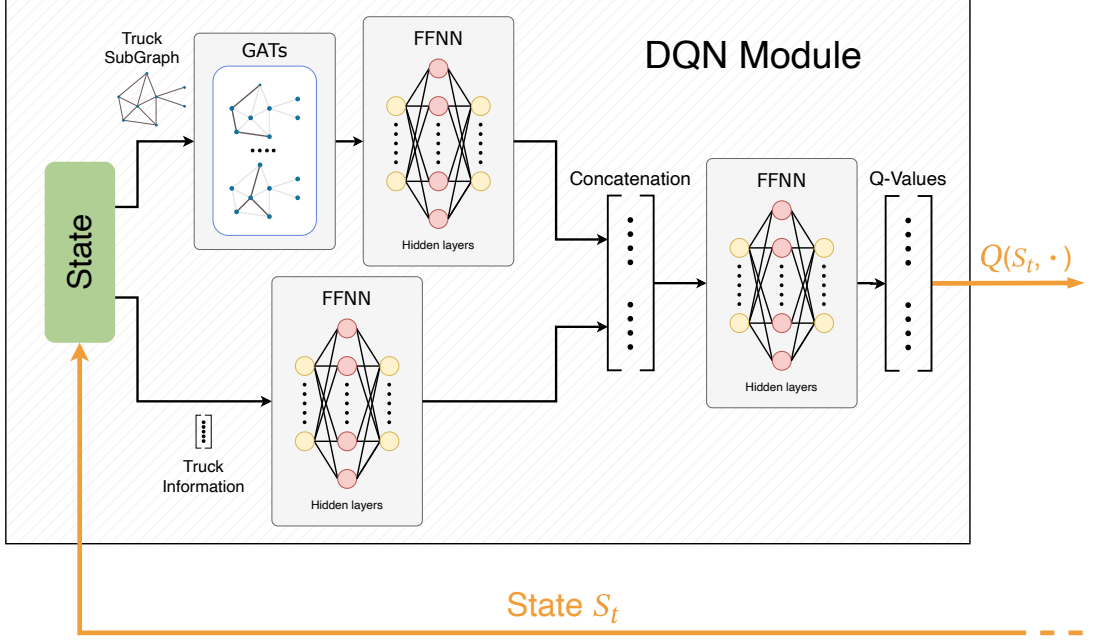


Figure 5.4: Illustration of the DQN Module with Graph Neural Networks. The agent leverages GATs to process subgraph information and extracts state representations through FFNNs.

the expressiveness of the learned representations. Second, it introduces a level of interpretability, as the attention coefficients reveal the influence of different nodes in shaping the system’s dynamics. In the context of this project, GATs are particularly beneficial not only for identifying the role of each node in the evolution of the BSS but also for capturing the impact of interactions between neighboring nodes. This allows the agent to develop a more comprehensive understanding of how local changes propagate through the network, ultimately leading to more informed and adaptive decision-making.

Specifically, the updated architecture consists of three stacked GAT layers with Rectified Linear Unit (ReLU) activation functions between them. The first GAT layer accepts an input with a dimension of 4 (number of features in each node) and produces an output of dimension 256 by employing four attention heads, each generating 64-dimensional features that are concatenated. The second GAT layer maintains the same structure, taking the 256-dimensional output of the first layer and again producing a 256-dimensional output through four concatenated attention heads. A third GAT layer further refines these features by reducing the output dimension to 256 through two attention heads, each producing 128-dimensional concatenated features. This deeper GAT stack

enhances the network’s capacity to capture complex interactions and spatial dynamics within the subgraph along which the truck operates.

Following the GAT layers, the network utilizes a global attention pooling layer, which aggregates node-level information into a graph-level embedding of dimension 256. This pooled embedding undergoes further processing through a FFNN composed of four layers with dimensions progressively reduced from 256 to 64, each interleaved with ReLU activation functions.

In parallel, the model employs a separate FFNN to process the agent’s state information. This FFNN accepts a 67-dimensional input vector encoding contextual information unrelated to the graph structure, such as the day, time of day, truck position, load, recent action history, and the maximum number of bikes present in the system. It comprises four layers, progressively reducing dimensionality from 256 to 64, with each layer separated by ReLU activations.

The outputs from the graph embedding pathway and the agent state pathway, each 64-dimensional, are concatenated into a single 128-dimensional representation. This combined embedding is then fed into a final FFNN composed of three layers, progressively reducing from 256 to 128 dimensions and culminating in an output layer whose dimension matches the number of available actions. The resulting vector provides the Q-values associated with each action, guiding the decision-making process of the agent. Figure 5.4 illustrates this enhanced DDQN module with Graph Neural Networks.

TRAINING AND EXPERIENCE REPLAY

As mentioned in Chapter 2, training a DQN requires a mechanism known as the replay buffer, where the agent’s experiences from interactions with the environment are stored. These experiences are then randomly sampled from the buffer and fed into the *Online* and *Target* networks to break temporal correlations between samples.

In this project, the replay buffer has been modified to support n-step returns by not only storing the classical experience tuple (s, a, r, s') , where s includes both the truck subgraph and the truck features, but also recording the number of steps n taken by the agent to complete the action. Additionally, instead of a single reward scalar r , a vector of rewards $(r_0, r_1, \dots, r_n)$ is stored, representing the cumulative rewards obtained along the n steps following the action. This additional information is crucial, as it modifies the target value computation in

5.3. DOUBLE DQN WITH GRAPH NEURAL NETWORKS

the Q-learning update. Instead of relying on a single-step reward and bootstrapping immediately from s' , the n -step return aggregates rewards over multiple steps before bootstrapping from the Q-value of the final state reached after n steps. The update rule, adapted from the classical DDQN equation in Eq. (2.13), incorporates this as follows:

$$Q_o(s, a) \leftarrow (1 - \alpha)Q_o(s, a) + \alpha \left[\sum_{i=0}^n \gamma^i r_i + \gamma^{n+1} Q_t(s', \arg \max_{a'} Q_o(s', a')) \right] \quad (5.7)$$

Here, the sum $\sum_{i=0}^n \gamma^i r_i$ represents the cumulative discounted reward obtained across the n steps. The term $\gamma^{n+1} Q_t(s', \arg \max_{a'} Q_o(s', a'))$ accounts for the bootstrapped Q-value of the next state s' after n steps, ensuring a smooth transition between short-term rewards and long-term value estimation. By incorporating this mechanism, the agent effectively extends its temporal credit assignment, treating the n steps like an eligibility trace that spreads the impact of rewards over a longer horizon. This helps in reducing variance in updates and improving stability, particularly in environments where meaningful rewards may be delayed over multiple steps.

$$\text{SmoothL1}(x) = \begin{cases} \frac{1}{2}x^2, & \text{if } |x| < 1 \\ |x| - \frac{1}{2}, & \text{otherwise} \end{cases} \quad (5.8)$$

This loss function has two key properties. When the error is small ($|x| < 1$), it behaves like Mean Squared Error (MSE), ensuring smoothness and differentiability around zero. When the error is larger, it transitions to a linear behavior like Mean Absolute Error (MAE), reducing sensitivity to outliers and improving training robustness. These characteristics help stabilize learning by preventing extreme errors from dominating updates.

Smooth L1 Loss can be seen as a simplified version of Huber Loss, where the transition point between the quadratic and linear regions is fixed at 1, whereas Huber Loss introduces a tunable parameter δ that allows this threshold to be adjusted. The general form of Huber Loss is:

$$\text{Huber}(x) = \begin{cases} \frac{1}{2}x^2, & \text{if } |x| \leq \delta \\ \delta(|x| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \quad (5.9)$$

By adjusting δ , the sensitivity to outliers can be controlled, making it more

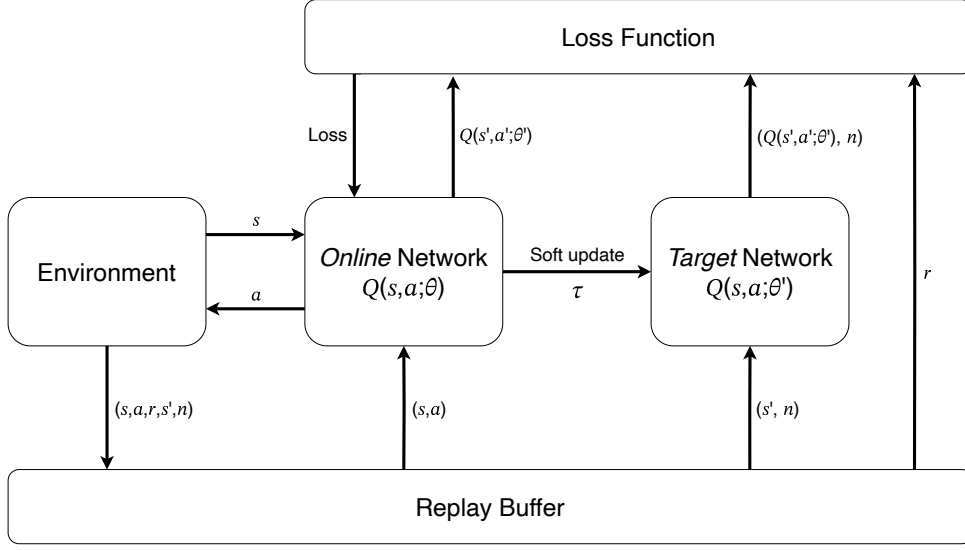


Figure 5.5: Training workflow of the DDQN. The agent interacts with the environment, storing experiences (s, a, r, s', n) in the replay buffer. The online network $Q(s, a; \theta)$ is updated using a loss function computed from target values generated by the target network $Q(s', a'; \theta')$. A soft update with parameter τ is applied to periodically update the target network, ensuring training stability.

flexible. However, for simplicity and robustness, Smooth L1 Loss was chosen in this project, as it does not require tuning any hyperparameters while still providing stability and resistance to outliers. The overall training process and dynamics so far described are illustrated in Figure 5.5.

Then, the training process employs Stochastic Gradient Descent (SGD) as the optimization algorithm. While adaptive optimizers such as Adam and RMSProp dynamically adjust the learning rate for each parameter and often accelerate convergence [15], they can introduce instability in RL settings where the data distribution is non-stationary. Empirical testing showed that SGD provided more stable training dynamics, reducing variance in updates compared to Adam, RMSProp, or SGD with momentum [15], which tended to introduce overly aggressive updates and erratic Q-value fluctuations. In this implementation, stability is prioritized over faster convergence, as consistent learning dynamics are crucial for RL algorithms to generalize effectively.

The final key aspect of the training process is managing the exploration-exploitation trade-off, which is handled using an ϵ -Greedy Policy. In this approach, with probability $1-\epsilon$, the agent selects the optimal action, ensuring exploitation of its current knowledge. With probability ϵ , it instead chooses a

5.3. DOUBLE DQN WITH GRAPH NEURAL NETWORKS

random action excluding the optimal one, promoting exploration.

At the start of training, ε is initialized to its maximum value, $\varepsilon_{\max}$, ensuring broad exploration across the state space. As training progresses, ε gradually decays according to an exponential schedule, reducing the frequency of random actions and allowing the agent to focus more on exploiting its learned policy. The decay function is defined as:

$$\varepsilon = \varepsilon_{\min} + (\varepsilon_{\max} - \varepsilon_{\min}) \cdot e^{-\varepsilon_{\text{decay}} t} \quad (5.10)$$

where t represents the training step, $\varepsilon_{\text{decay}}$ controls the decay rate, and $\varepsilon_{\min}$ defines the lower bound, preventing the agent from fully eliminating exploration. This scheduling strategy ensures a smooth transition from early-stage exploration to a more exploitation-focused policy as the agent refines its decision-making capabilities.

6

Simulations & Results

With the simulation framework established, this chapter focuses on its application for training and evaluating the proposed RL model. The first section outlines the integration of real-world data and the experimental setup, specifying the values and parameters used during training. To ensure a rigorous assessment, the performance metrics used for evaluation are also introduced. This is followed by a discussion of state-of-the-art approaches, providing context for the adopted methodology. Finally, the results are presented and analyzed, offering insights into the model’s behavior, strengths, and limitations.

6.1 SIMULATION ENVIRONMENT AND SETUP

Given the inherent complexity and highly dynamic nature of the problem, coupled with substantial stochastic variability in bike demand patterns, the experimental analysis presented in this chapter focuses on a preliminary case study conducted on a representative sub-area of the original graph, covering approximately 2.3 km² around MIT, as illustrated in Figure 6.1. Although earlier chapters discussed the urban network of Cambridge in its entirety, narrowing the scope was essential to rigorously evaluate and refine the dynamic rebalancing strategies. This targeted scope enables an in-depth investigation into the performance and behavior of the proposed RL framework under controlled yet realistic conditions, laying a robust foundation for potential scalability to the broader urban area in subsequent developments.

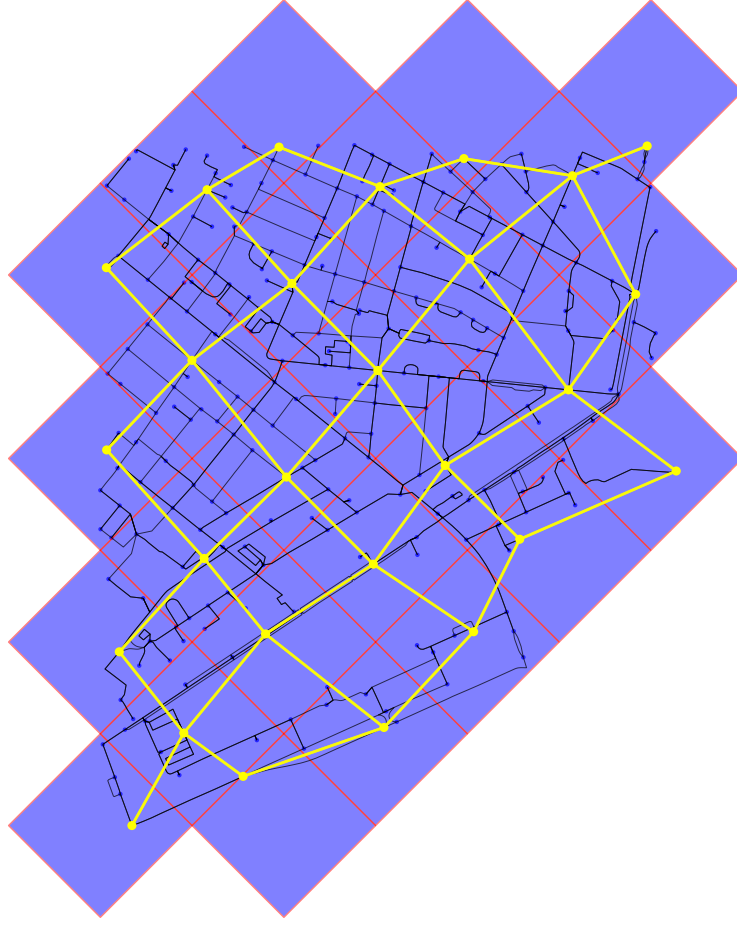


Figure 6.1: Graph representation of the selected sub-area ($\sim 2.3 \text{ km}^2$) around MIT, used for the preliminary analysis of dynamic bike rebalancing.

PRECOMPUTATION AND INTEGRATION OF REAL-WORLD DATA

In order to run the simulator described in Chapter 5, real-world data are required to ensure a realistic representation of trip dynamics. These data are sourced from the BlueBikes BSS, which operates in Boston, MA [2]. Specifically, data from September and October 2022 have been selected, as they correspond to the months with the highest usage of the service. However, since BlueBikes is a *docked* BSS, while the present study focuses on dynamic rebalancing in a *dockless* system, a transformation process is necessary to adapt the data. This section details the steps taken to extract, filter, and process the trip data to construct a demand model suitable for a *dockless* system.

The original dataset consists of trip records, each containing key information such as:

- Trip **start time** and **end time** in ISO 8601 format;
- Trip **duration** in seconds.
- **Departure station ID** and **arrival station ID**;
- **Geographical coordinates** of the stations in decimal degrees format;
- **Departure station name** and **arrival station name**;

An example of the dataset is shown in Table 6.1, illustrating the format in which the data are originally recorded.

Trip Information			
#	tripduration	starttime	stoptime
1	236	2022-09-04 14:11:19.6120	2022-09-04 14:15:15.7870
2	1585	2022-09-19 14:18:44.8190	2022-09-19 14:45:09.8350

Start Station Details				
#	start station id	start station name	start latitude	start longitude
1	74	Harvard Square at Mass Ave/Dunster	42.373	-71.119
2	67	MIT at Mass Ave / Amherst St	42.358	-71.093

End Station Details				
#	end station id	end station name	end latitude	end longitude
1	97	Harvard University River Houses at DeWolfe St / Cowperthwaite St	42.369	-71.117
2	20	Aquarium T Stop - 200 Atlantic Ave	42.360	-71.051

Table 6.1: Example of two trip records from the BlueBikes System Data [2], including trip duration, start and stop times, as well as start and end station details.

6.1. SIMULATION ENVIRONMENT AND SETUP

Since the dataset includes trips from across the Boston metropolitan area and nearby cities, the data are filtered to focus exclusively on Cambridge, MA. Stations within the selected sub-area are considered internal nodes, while all others are considered part of an external region. Trips are then processed by retaining only those that have either their departure or arrival station within the selected area, discarding all others. For trips involving an external region, the corresponding external station is replaced with a virtual node, assigned the ID 10000, which serves as a collective representation of all external locations.

To account for temporal variations in demand, the dataset is further segmented by day of the week (Monday to Sunday) and divided into discrete three-hour time slots, labeled from 0 to 7. The first time slot (slot 0) of each day begins at 1 AM, a choice made to better capture early-morning demand fluctuations and the rush-hour dynamics that follow. This segmentation ensures that the simulator can accurately model changes in trip activity throughout the day, reflecting observed patterns in the BlueBikes system.

Using the remaining trips, a square matrix is constructed, where node IDs serve as indices for both rows and columns, with the additional node 10000 included. For each pair of nodes (n, m) , the departure rate is computed by summing all trips from n to m and normalizing by the number of days in the month, converted into seconds. This ensures the matrix accurately captures trip flows while consolidating external movements into a single representative node. An example of the resulting matrix Λ is given below:

$$\Lambda = \begin{bmatrix} \Lambda_{1,1} & \cdots & \Lambda_{1,m} & \cdots & \Lambda_{1,10000} \\ \vdots & & \vdots & & \vdots \\ \Lambda_{n,1} & \cdots & \Lambda_{n,m} & \cdots & \Lambda_{n,10000} \\ \vdots & & \vdots & & \vdots \\ \Lambda_{10000,1} & \cdots & \Lambda_{10000,m} & \cdots & \Lambda_{10000,10000} \end{bmatrix} \quad (6.1)$$

The rate matrix Λ provides a comprehensive view of trip flows within the *docked* system. However, because BlueBikes operates with fixed stations, most entries in Λ are zero, as trips are limited to departures and arrivals at docking locations. To simulate a *dockless* system, where users can freely start and end trips across the network, an interpolation process is applied. This generates a new interpolated rate matrix, denoted as I , which estimates trip rates between previously unconnected nodes while preserving the structure of observed travel

patterns.

The interpolation follows two main phases. It begins by estimating missing arrival rates for nodes with known departures. Each row in the matrix corresponds to a departure node n , with columns representing the rates of trips to different destination nodes. If a node n has at least one nonzero departure rate, it means trips originate from there. However, for destinations m where the departure rate $\Lambda_{n,m}$ is initially zero, an estimation is performed by considering nearby destinations y (within 500m) that already receive trips from n . The missing rate $I_{n,m}$ is then computed as an inverse-distance weighted average of the known rates $\Lambda_{n,y}$:

$$I_{n,m} = \frac{\sum_{y \in \mathcal{Y}} \frac{\Lambda_{n,y}}{\delta_{n,y}}}{\sum_{y \in \mathcal{Y}} \frac{1}{\delta_{n,y}}} \quad (6.2)$$

where $\mathcal{Y}$ represents the set of nodes within a 500m radius of n that have nonzero departure rates.

The second phase addresses nodes with no recorded departures, meaning their entire row in the matrix consists of zeros. In the *docked* system, these nodes do not serve as origins, but in a *dockless* system, trips can start from anywhere. To estimate the missing departure rates, the interpolation process considers nearby nodes with known departures and reconstructs the entire row based on their departure patterns. This follows a similar inverse-distance weighting scheme as in Eq. (6.2), but applied to entire rows rather than individual destinations. Specifically, for a node n corresponding to a zero row Λ_n , the interpolated row I_n is given by:

$$I_n = \sum_{Y \in \mathbf{Y}} \sum_{y \in Y} \frac{\Lambda_{n,y}}{\delta_{n,y}} \cdot \frac{1}{\sum_{y \in Y} \frac{1}{\delta_{n,y}}} \quad (6.3)$$

where $\mathbf{Y}$ is the set of all the non zero rows of Λ corresponding to nodes within a 500m neighborhood. This ensures that nodes with no recorded departures inherit reasonable departure distributions from nearby active nodes.

Figure 6.2 visually represents this process. The large yellow dots indicate nodes with nonzero departure rates in the original *docked* system, while the yellow circles denote the radius within which these rates are interpolated to estimate missing destination rates. The blue circles correspond to nodes that initially had no recorded departures; for these, the entire departure distribution is reconstructed by aggregating data from nearby active nodes, again using a 500m radius. Black lines represent the distance between node centers involved

6.1. SIMULATION ENVIRONMENT AND SETUP

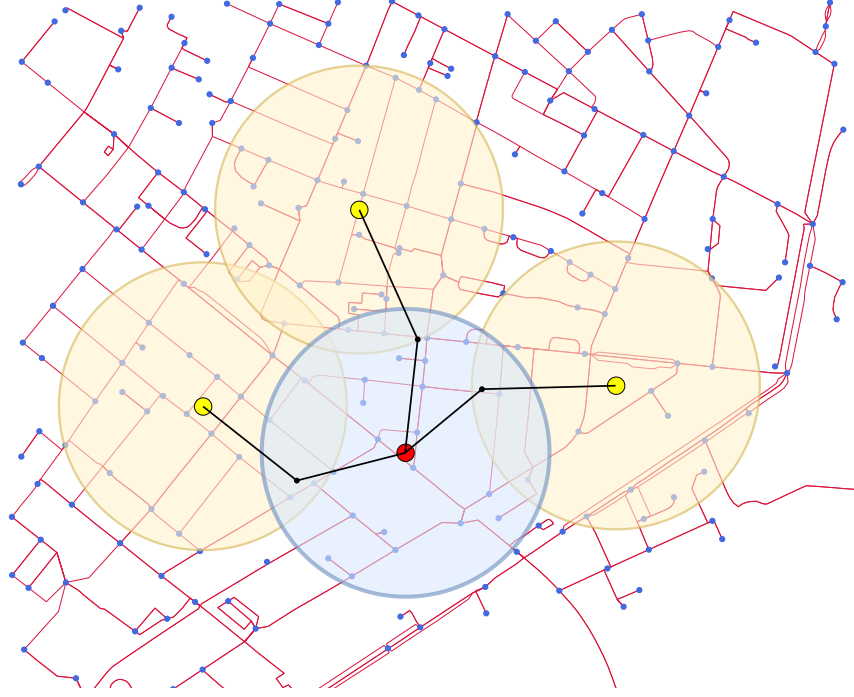


Figure 6.2: Visual summary of the two-step interpolation process using 500m-radius neighborhoods.

in the interpolation, helping ensure a continuous and spatially coherent distribution of trip rates in the resulting *dockless* system.

To maintain consistency with the original dataset, the total departure rate of the pre-interpolation matrix, as defined in Eq. (6.1), is computed. This value is then used to rescale the interpolated rate matrix I , ensuring that the overall demand remains unchanged. Additionally, this global departure rate serves as the foundation for the Poisson process governing trip departures in the simulation environment, as detailed in Chapter 5. Once interpolation is complete, a PMF is constructed by normalizing the total departure rates to sum to 1. This PMF defines the likelihood that a departure from one node is directed to another, providing the basis for trip generation in the *dockless* simulation.

Through this transformation, real-world data from a *docked* BSS are adapted for a *dockless* scenario while preserving the original travel dynamics. This approach maintains realistic movement patterns while enabling the flexibility required for dynamic rebalancing in the *dockless* system. All the resulting rate matrices are precomputed and stored as separate data files, organized by day of the week and time slot, to optimize the simulator's computational efficiency

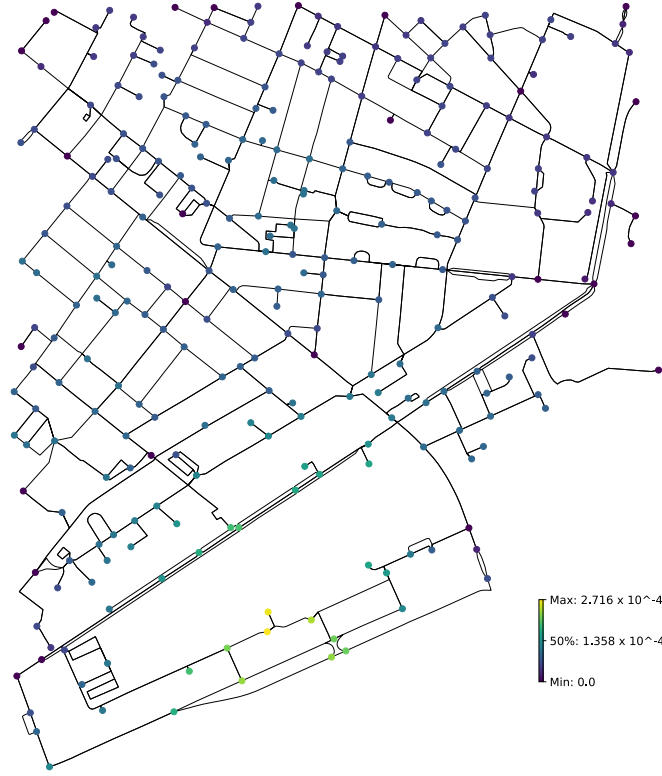


Figure 6.3: Interpolated trip demand graph, representing estimated trip rates for the *dockless* system, based on data from Monday, 07:00–10:00 AM. The colors indicate the intensity of the demand, with higher values corresponding to areas experiencing greater trip activity.

and ensure consistent retrieval of demand patterns during the simulation. An example of the interpolated trip demand graph for Monday, 07:00–10:00 AM, is shown in Figure 6.3.

Beyond the demand patterns of the BSS, additional precomputed data are required specifically for the truck simulation to accurately reflect real-world conditions. In particular, mean travel velocity and average energy consumption for an electric vehicle are extracted from the TomTom *Boston Traffic Report* [38]. As illustrated in Figure 6.4, these data are structured into matrices where rows correspond to days of the week and columns represent hourly intervals. At each simulation step, the mean velocity and energy consumption values for the current day and time are retrieved from these matrices and used to estimate travel times between nodes in the graph. This integration ensures that the truck simulation accurately reflects realistic traffic dynamics and vehicle performance,

6.1. SIMULATION ENVIRONMENT AND SETUP

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
12:00 AM	43 km/h	48 km/h	47 km/h	46 km/h	45 km/h	43 km/h	42 km/h
	45 km/h	49 km/h	49 km/h	47 km/h	47 km/h	45 km/h	44 km/h
02:00 AM	46 km/h	50 km/h	49 km/h	48 km/h	48 km/h	47 km/h	45 km/h
	51 km/h	54 km/h	54 km/h	53 km/h	53 km/h	52 km/h	51 km/h
04:00 AM	55 km/h	58 km/h	58 km/h	58 km/h	57 km/h	57 km/h	56 km/h
	57 km/h	55 km/h	55 km/h	55 km/h	54 km/h	55 km/h	57 km/h
06:00 AM	55 km/h	43 km/h	41 km/h	41 km/h	41 km/h	43 km/h	55 km/h
	54 km/h	34 km/h	32 km/h	32 km/h	32 km/h	36 km/h	53 km/h
08:00 AM	52 km/h	32 km/h	28 km/h	28 km/h	28 km/h	35 km/h	49 km/h
	49 km/h	36 km/h	31 km/h	31 km/h	31 km/h	38 km/h	45 km/h
10:00 AM	45 km/h	40 km/h	35 km/h	35 km/h	34 km/h	38 km/h	41 km/h
	41 km/h	40 km/h	37 km/h	37 km/h	36 km/h	37 km/h	38 km/h
12:00 PM	38 km/h	40 km/h	37 km/h	37 km/h	36 km/h	36 km/h	36 km/h
	37 km/h	39 km/h	37 km/h	36 km/h	36 km/h	34 km/h	35 km/h
02:00 PM	36 km/h	35 km/h	33 km/h	32 km/h	31 km/h	30 km/h	34 km/h
	36 km/h	33 km/h	29 km/h	28 km/h	27 km/h	27 km/h	33 km/h
04:00 PM	36 km/h	32 km/h	27 km/h	26 km/h	25 km/h	28 km/h	33 km/h
	36 km/h	31 km/h	24 km/h	24 km/h	24 km/h	29 km/h	33 km/h
06:00 PM	37 km/h	35 km/h	28 km/h	27 km/h	27 km/h	31 km/h	33 km/h
	39 km/h	40 km/h	36 km/h	34 km/h	33 km/h	34 km/h	35 km/h
08:00 PM	42 km/h	42 km/h	40 km/h	39 km/h	38 km/h	38 km/h	39 km/h
	44 km/h	43 km/h	42 km/h	41 km/h	40 km/h	39 km/h	40 km/h
10:00 PM	44 km/h	43 km/h	41 km/h	40 km/h	39 km/h	38 km/h	40 km/h
	45 km/h	45 km/h	42 km/h	42 km/h	40 km/h	39 km/h	41 km/h

Figure 6.4: Sample from TomTom’s velocity matrix for the Boston, MA area [38], showing average travel speeds by hour and day of the week.

further enhancing the fidelity of the *dockless* system model. The resulting matrices are stored as separate data files for efficient retrieval during simulation.

In the end, to ensure efficient runtime computations, several key datasets are precomputed before training begins. One of the most fundamental elements is the distance matrix L , which stores the shortest path distances between all node pairs. This matrix is generated using Dijkstra’s algorithm [10], allowing the simulation to retrieve shortest paths efficiently. Alongside it, global departure rates derived from the rate matrices are precomputed, enabling quick access to demand information. A node dictionary is also built to store the neighboring nodes for each node in the graph, improving neighborhood lookups. Additionally, the truck grid, which partitions the subgraph $\mathcal{G}_T$ into zones, is precomputed, storing both the nodes within each zone and the adjacent zones for seamless traversal. These datasets are stored separately in files to minimize computational overhead and ensure rapid access during simulation.

SETUP PROCESS AND TRAINING CONFIGURATION

With these precomputed datasets in place, the environment is initialized to streamline computations and optimize performance. To further enhance efficiency, event generation within the simulator is parallelized. While the environment processes a set of events corresponding to two consecutive time slots, the next set of events is precomputed in the background, ensuring they are readily available when needed. This approach reduces delays and allows the

Category	Hyperparameter	Value
Optimization	Learning Rate	10^{-4}
	Batch Size	64
Training Setup	Replay Buffer Capacity	10^5
	Episodes	200
	Time slots per episode	56
Exploration	Initial $\epsilon_{\max}$	1.0
	Final $\epsilon_{\min}$	0.01
	Decay Rate ϵ_{decay}	3.43×10^{-4}
Discounting	Discount Factor γ	0.95
Target Update	Target Network Soft Update Factor τ	0.05

Table 6.2: Hyperparameters used for training the RL model.

simulation to run smoothly without unnecessary computational bottlenecks.

The training process begins with the initialization of the environment, the replay buffer, and the agent. All hyperparameters governing this process are detailed in Table 6.2. At the start of each episode, the environment is reset to Monday at 1 AM, ensuring a consistent starting point for training. This initialization strategy ensures that each episode begins under comparable conditions, allowing the agent to learn from a structured and reproducible setting. Additionally, with 56 time slots per episode, each training episode spans one full week. The exploration decay parameter ϵ_{decay} is determined so that the exploration probability ϵ reaches 0.1 at 60% of the total training process, which is computed as:

$$\epsilon_{\text{decay}} = \frac{\log(10)}{\text{Episodes} \times \text{Time slots per episode} \times 0.6} \quad (6.4)$$

Empirical trials revealed that using a purely greedy policy (i.e., $\epsilon=0$) risked causing the agent to become polarized around a small set of actions, due to numerical issues. Maintaining a minimal level of exploration preserves a mechanism to deviate from favored actions when needed, preventing premature convergence on suboptimal policies. Thus, the exploration rate decreases steadily but remains above 0.01, balancing effective exploitation with essential exploration throughout training. All other hyperparameters were selected through a trial-and-error approach, without adhering to a specific optimization principle.

6.1. SIMULATION ENVIRONMENT AND SETUP

The full training process was conducted on a dedicated hardware server equipped with an Intel Core i3-7100 processor, 32 GB of RAM, and an NVIDIA GeForce RTX 3070 GPU with 8 GB of VRAM. The total training time ran for 200 episodes and took approximately 60 hours.

PERFORMANCE METRICS FOR EVALUATION

During training, several performance metrics were monitored to evaluate whether the model was learning effectively and to analyze its behavior over time. Key metrics included the total number of **failures** per time slot, **mean rewards** per time slot, and the **distribution of rewards** across different actions at each step. Additionally, **Q-values** and the **number of deployed bikes** per time slot were tracked to ensure that the agent’s policy was evolving correctly. To complement these time-series plots, histograms were generated to visualize the **frequency of action** selections within each time slot, as well as the **mean value of each action** in the final time slot of an episode.

Beyond numerical metrics, visualizations of the truck’s subgraph were used to provide an intuitive understanding of the model’s decision-making process. These images applied a color map to indicate, for each zone, the risk score, the number of bikes placed, and the total number of visits. This spatial representation helped in identifying emerging patterns in how the agent was distributing bikes and prioritizing different areas over time.

All these metrics and visualizations were continuously accessible online through a web server built using Flask and Plotly, which efficiently handled real-time data visualization without hindering training performance. This interactive system significantly accelerated the analysis of ongoing training, enabling immediate detection and resolution of potential issues without waiting for training completion or imposing unnecessary computational overhead.

These performance metrics were essential for diagnosing the stability and effectiveness of learning. By continuously monitoring them, it was possible to detect signs of divergence, instability, or inefficient policy updates. If the learning process showed erratic fluctuations in rewards, Q-values, or action distributions, adjustments to hyperparameters or training strategies could be made. Conversely, a smooth learning curve with stable improvements in performance confirmed that the model was converging toward an optimal policy.

6.2 STATE-OF-THE-ART APPROACHES

To evaluate the effectiveness of the proposed dynamic rebalancing strategy, it is essential to establish a robust benchmark. The benchmark adopted in this thesis is based on the static rebalancing approach presented by Mahony *et al.* in their study [29]. Their method calculates the net flow for each zone within the truck operational graph, defined as the expected difference between bike arrivals and departures over the period from the current moment to the next scheduled rebalancing event.

The static method proposed by Mahony schedules rebalancing events throughout the day, with the initial rebalancing always set at 3 AM. The number of subsequent daily rebalancing operations is determined based on the specific comparison scenario, varying from infrequent events to hourly rebalancing. At each rebalancing event, bikes are instantaneously redistributed across the city, adjusting the availability in each zone to match anticipated net flows, while simultaneously recharging all repositioned bikes to 100% battery capacity. After completing the instantaneous redistribution, the environment operates freely until the next scheduled event, during which user demands are tracked, and unmet requests are recorded as service failures.

To establish a fair benchmark for comparison with the proposed dynamic rebalancing model, two simulations were conducted:

1. A **static rebalancing simulation**, following Mahony's original framework, where the system is rebalanced only twice a day (at 3 AM and 3 PM).
2. A **dynamic rebalancing simulation**, where Mahony's approach is adapted to perform rebalancing every hour. This ensures a comparison against a fully dynamic approach while maintaining the same fundamental logic of adjusting availability based on net flow.

The results of these simulations are illustrated in Figure 6.5, where subfigure (a) shows the failures related to the static rebalancing approach, and subfigure (b) presents the failures for the dynamic rebalancing approach.

While failure rate provides a direct measure of service reliability, it does not account for the feasibility of rebalancing actions, since Mahony's model assumes instantaneous redistribution of bikes, which is unrealistic in practical operations. Therefore, an additional metric is introduced: rebalancing time, which represents the actual time a truck would require to execute rebalancing

6.2. STATE-OF-THE-ART APPROACHES

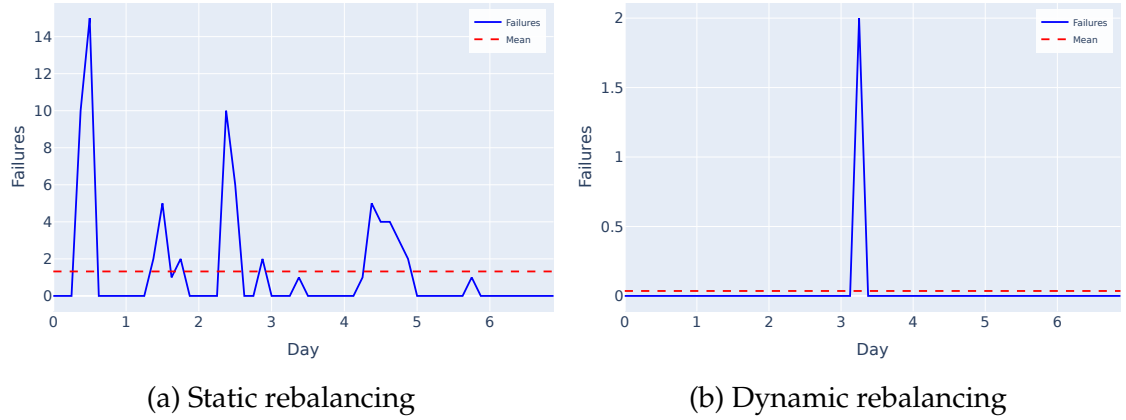


Figure 6.5: Benchmark comparison of failures in static and dynamic rebalancing simulations (Monday to Sunday).

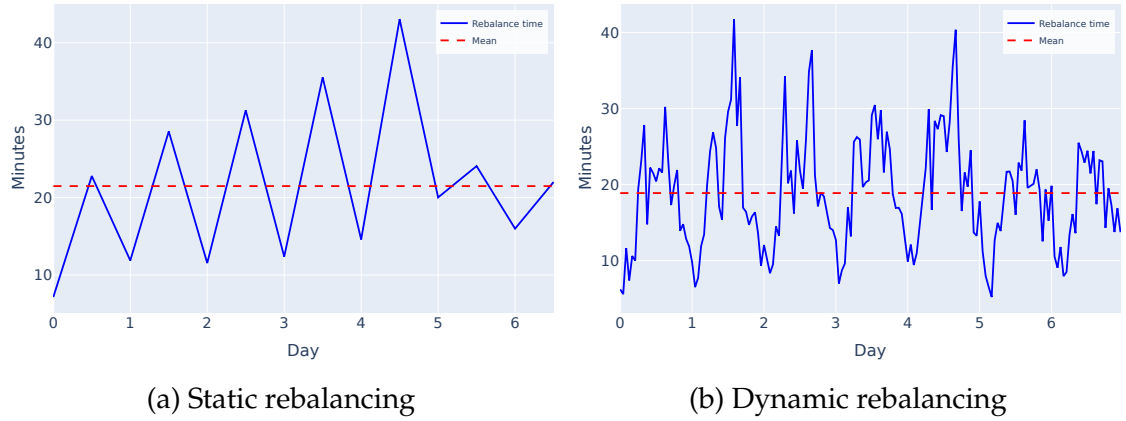


Figure 6.6: Benchmark comparison of rebalance time in static and dynamic rebalancing simulations (Monday to Sunday).

under real-world constraints.

To simulate realistic rebalancing times, a modified Traveling Salesman Problem (TSP) approach is employed over the truck's subgraph $\mathcal{G}_T$. This approach dynamically accounts for practical operational constraints such as limited truck capacity and the need for sequential visits to each zone based on bike availability. Initially, the method generates a route that covers all surplus and deficit zones. As the truck follows this route, it dynamically manages its load, picking up bikes from zones with surplus and delivering them to zones experiencing deficits. If the truck encounters a deficit node without sufficient bikes onboard to satisfy the demand, it temporarily skips this node. Once the truck accumulates enough bikes from subsequent surplus zones to meet the demand at the previously skipped nodes, it recalculates an optimized route specifically designed to

efficiently revisit these nodes.

The results of these rebalancing time simulations are presented in Figure 6.6, which compares the time required for the truck to complete the rebalancing process under static and dynamic strategies.

By incorporating a realistic time-based evaluation of rebalancing, this approach provides a more comprehensive comparison between static and dynamic strategies, capturing not only their impact on system failures but also their feasibility in real-world operations.

6.3 RESULTS AND ANALYSIS

This section presents a detailed analysis of the results obtained from a complete training cycle of the proposed RL model. To facilitate clear understanding, the following training methodology is briefly revisited: the model was trained over 200 episodes, each episode corresponding to a simulated week beginning on Monday at 01:00 AM and concluding the following Monday at 00:59 AM. Each simulated day was divided into 8 three-hour time slots, yielding a total of 56 time slots per episode and 11,200 time slots across the entire training period.

The results are structured to first provide a broader overview of the training process and outcomes, followed by a more focused presentation of validation results. This two-tiered approach ensures both a comprehensive understanding of general performance trends and detailed insights into specific aspects of the model's behavior. To further contextualize these findings, the analysis compares the total number of failures per time slot in a scenario with no rebalancing actions, offering a critical baseline that highlights the effectiveness of the dynamic rebalancing strategy. These observations, together with insights from previous sections, deliver a thorough and precise assessment of the model's capabilities and limitations.

From both an RL perspective and an operational standpoint, the overall training process demonstrates clear improvements. As shown in Figure 6.7, the reward per time slot generally increases over the 200 episodes, indicating that the agent is progressively refining its strategy to manage the BSS. Notably, the cumulative mean (red solid line) in each figure serves as a smoothed indicator of long-term trends, reducing the visual impact of occasional fluctuations.

In addition, the gradual decrease in exploration over successive training

6.3. RESULTS AND ANALYSIS

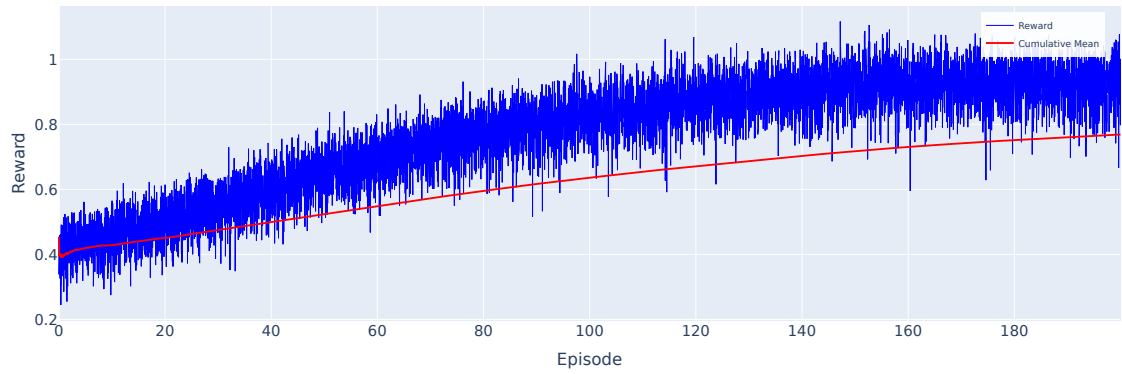


Figure 6.7: Evolution of the reward per time slot over training episodes. Each point corresponds to the time slot average reward, while the red line indicates its cumulative mean. The steadily increasing trend and reduced fluctuation range suggest that the learning process is gradually converging toward a higher overall reward.

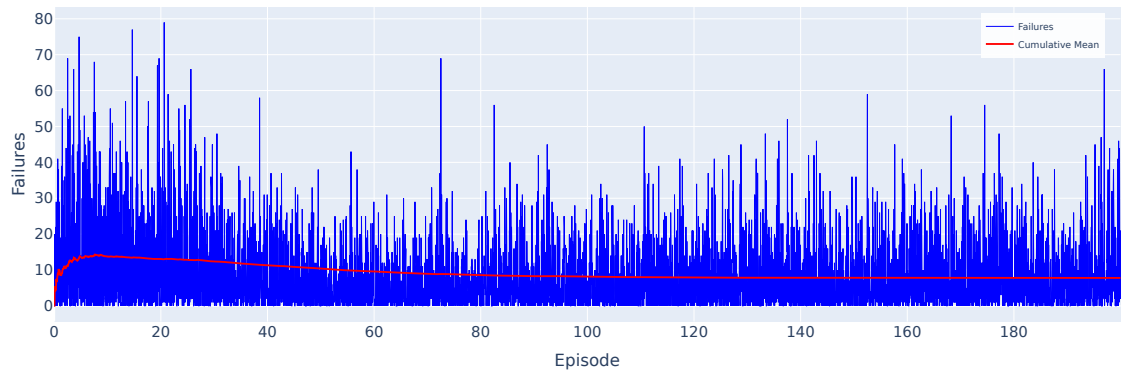


Figure 6.8: Number of service failures per time slot across training episodes, with a cumulative mean shown by the red line. The declining trajectory indicates that the agent increasingly learns to meet demand, thus reducing failures as training progresses.

episodes contributes to these improvements. As the agent transitions from more exploratory actions to primarily exploitative ones, the average reward increases and the number of failures decreases. This shift toward exploitation reflects the agent's enhanced understanding of the environment and its improved decision-making, ultimately aiding convergence toward a stable and effective policy.

Figure 6.9 further reinforces the evidence of convergence by showing the evolution of Q-values throughout training. In the early episodes, Q-values fluctuate significantly, reflecting high uncertainty and active exploration. As training progresses and exploration diminishes, the mean Q-values stabilize and exhibit less noise, indicating that the agent is consistently selecting high-

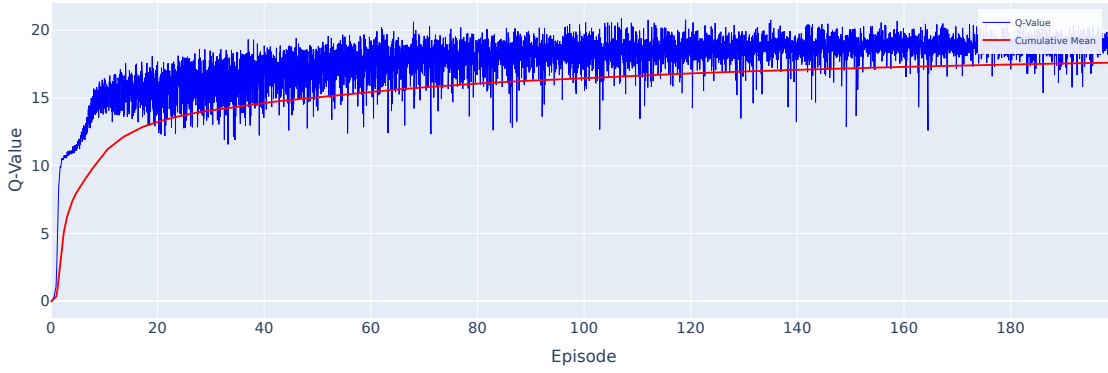


Figure 6.9: Mean Q-value per time slot, tracked across time slots and averaged cumulatively (red line). The stabilization and smoother profile at later episodes imply that the agent’s policy converges, reflecting consistent decision-making in selecting optimal rebalancing actions.

value actions. This trend aligns with the observed increase in rewards and decrease in failures, pointing to a more confident and reliable policy.

In such complex RL problems, convergence to a global optimum is not always guaranteed. External factors like stochastic demand, partial observability, and the large action space can cause the agent’s performance to fluctuate. Therefore, at the end of each training episode, a validation run was performed under identical conditions using an ϵ -greedy policy with $\epsilon = 0.01$. This small yet nonzero exploration rate, chosen empirically after multiple trials, measures performance in near-greedy conditions while avoiding numerical issues. If the number of failures in this validation run was lower than in all prior validations, the corresponding Double DQN state and its associated metrics were saved. This “best model” strategy is a common practice in RL research, ensuring that a temporarily discovered highly effective policy can still be recovered if subsequent performance drifts or oscillates.

To complement the broader perspective, attention now turns to a more focused view by examining the best validation episode recorded during training. Figures 6.7, 6.8, and 6.9 already illustrate how the agent successfully lowers failures and increases rewards. Below, the specific episode that outperformed all others is highlighted, and its dynamics are discussed in detail.

Before introducing the validation findings, Figure 6.10 highlights the scenario in which the system evolves freely with no rebalancing actions. The red line indicates the cumulative mean over the simulation, which settles at approximately 19 failures per time slot. This baseline quantifies the intrinsic mismatch

6.3. RESULTS AND ANALYSIS

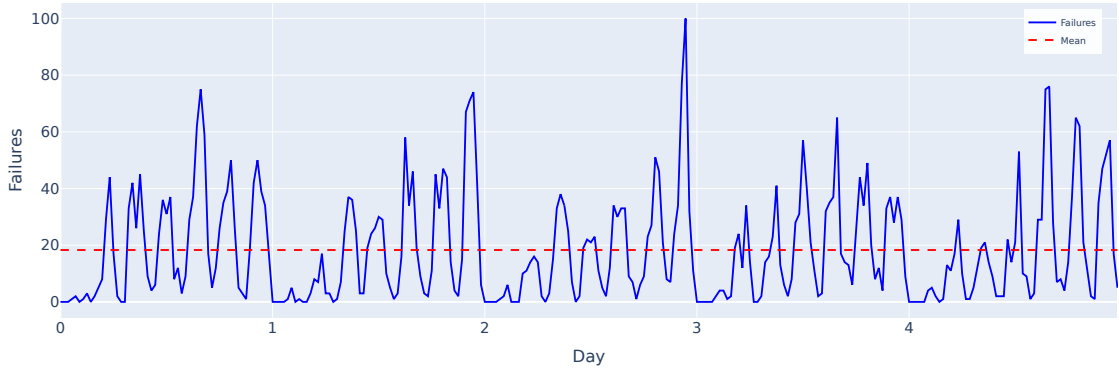


Figure 6.10: Failures in the baseline scenario without rebalancing. The red line depicts the cumulative mean, which stabilizes at about 19 failures per time slot.

between demand and supply in the absence of rebalancing, providing a crucial benchmark against which subsequent performance can be evaluated.

The best validation episode was recorded at iteration 185 and is shown in Figures 6.11 and 6.12. It achieved a mean of 3.58 failures per time slot, corresponding to an approximately 81% reduction from the no-rebalancing baseline of 19 failures per time slot, and an average deployment of 224 bikes. For context, the static rebalancing strategy (Figure 6.5a) described earlier yielded a mean of 1.32 failures per time slot, whereas a more frequent dynamic approach (Figure 6.5b) reduced it to about 0.03. Overall, the validation results demonstrate a substantial improvement in meeting user demand when compared to the system’s free evolution.

Additional insights emerge from Figure 6.13, which shows a predominance of *DROP BIKE* and *PICK UP BIKE* actions. Although this strategy effectively

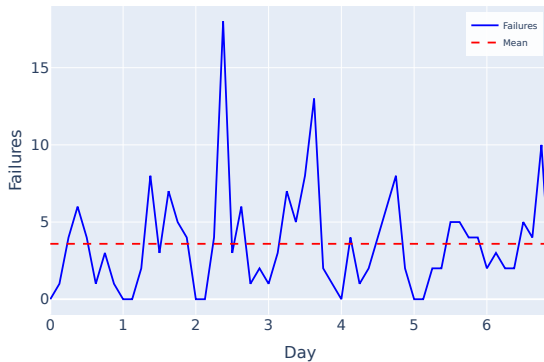


Figure 6.11: Failures over the validation days. The red line marks the time slot mean.

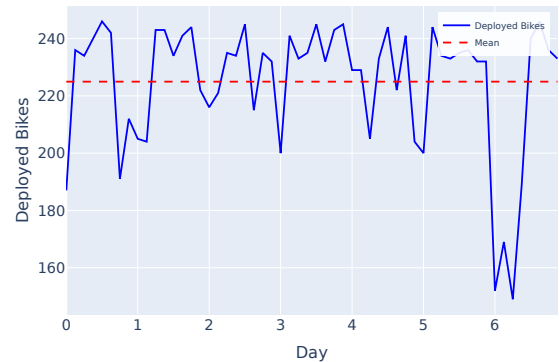


Figure 6.12: Number of deployed bikes per day. The red line shows the time slot mean.

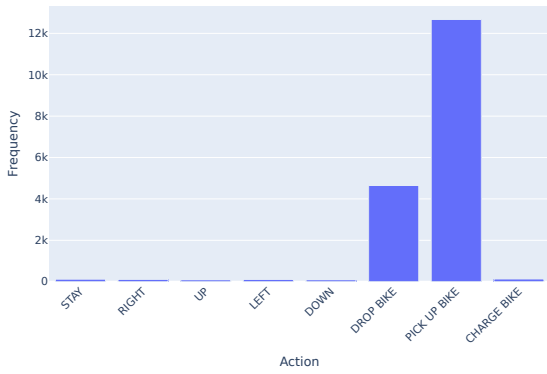


Figure 6.13: Frequency of each action in the validation episode.

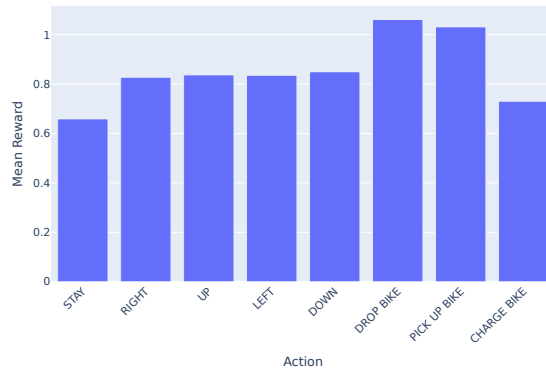


Figure 6.14: Mean reward associated with each action.

reduces failures, it reveals a tendency toward repetitive cycling around those two actions. Figure 6.14 confirms that these actions yield relatively high mean rewards, illustrating how the reward function can inadvertently promote polarized behavior if corner cases are not accounted for. Despite this limitation, the overall drop in failures underscores the effectiveness of the dynamic rebalancing strategy and highlights how careful reward design can significantly shape agent behavior.

By further breaking down Figure 6.11 across each of the eight three-hour time slots in a day, several distinct demand and failure patterns emerge:

- **Late Night & Early Morning (1 AM–4 AM, 4 AM–7 AM):** Demand remains low, and unmet requests are generally minimal. However, poor end-of-day positioning can cause occasional spikes when transitioning to the 7 AM–10 AM rush.
- **Morning Peak (7 AM–10 AM, 10 AM–1 PM):** Commuting hours drive a significant surge in demand, often resulting in shortages unless proactive rebalancing offsets any bike deficits. Although the policy reduces failures overall, some time slots still experience occasional peaks.
- **Afternoon Peak (1 PM–4 PM, 4 PM–7 PM):** Another surge typically appears mid to late day, often linked to lunch breaks or post-work activities. Shortfalls can persist if morning rebalancing was incomplete, reinforcing the need for timely redistribution.
- **Evening & Night (7 PM–10 PM, 10 PM–1 AM):** Demand tapers off compared to daytime peaks, and failures generally decrease. Nonetheless, left-over imbalances from the afternoon may lead to sporadic unmet requests

6.3. RESULTS AND ANALYSIS

before overnight operations begin.

Moreover, the analysis of agent performance reveals that most failures now predominantly occur on Wednesday and Thursday, aligning closely with patterns identified in the velocity matrix related to traffic data (Figure 6.4) from the TomTom *Boston Traffic Report* [38]. This alignment suggests a correlation between increased congestion and the agent's difficulties in effectively matching bike supply to user demand during peak traffic periods. Increased traffic congestion might also incentivize more commuters to opt for bikes, consequently exacerbating rebalancing challenges for the truck as it navigates congested roadways.

A final perspective on the validation episode can be obtained by examining

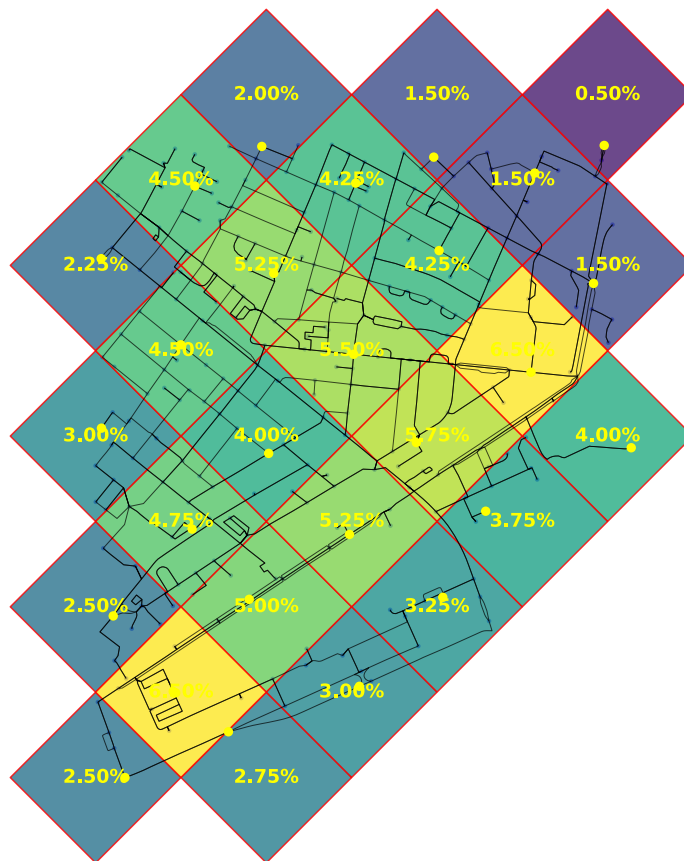


Figure 6.15: Heat map of the rebalancing area's zones, where each color indicates the number of visits made by the rebalancing truck. Higher intensity suggests more frequent visits.

the spatial distribution of key metrics across the rebalancing area, as shown in Figures 6.15, 6.16, and 6.17. The heat map of visits in Figure 6.15 indicates that the rebalancing truck does not remain in a single location but travels across multiple zones, suggesting that the agent is consistently seeking to redistribute bikes where needed. Notably, one of the most visited zones in the bottom regions, which accounts for 6.5% of the total visits, corresponds to a zone with a positive risk score (i.e., a potential shortfall of bikes). This correlation demonstrates how the agent actively addresses demand by moving bikes from nearby zones with a surplus (which also exhibit high visit frequencies) toward areas flagged as riskier. Overall, this pattern reinforces the previously observed avoidance of single-point stagnation.

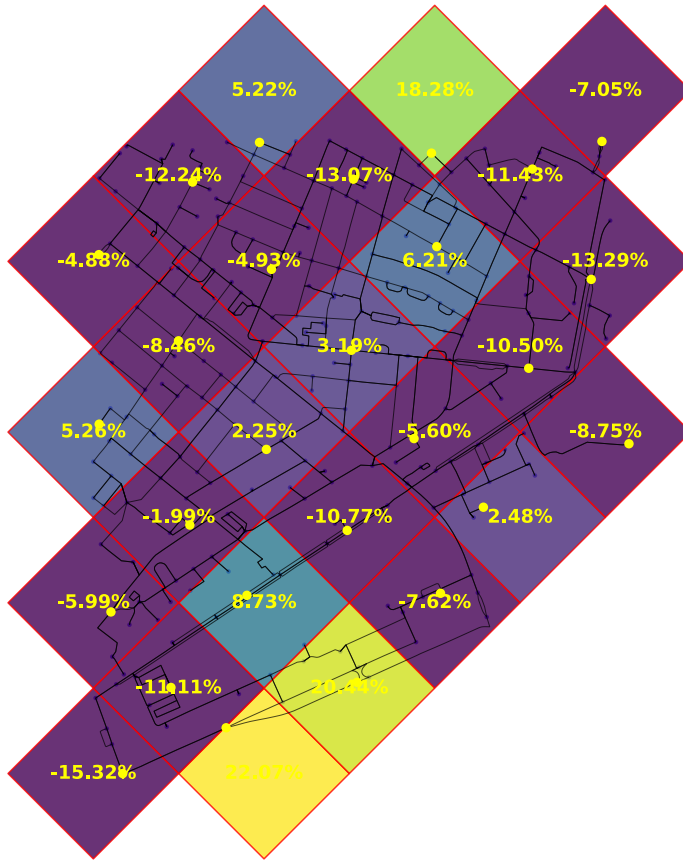


Figure 6.16: Average risk score of each zone over one validation episode. Negative scores represent a surplus of bikes relative to demand, whereas positive scores indicate a potential shortage.

6.3. RESULTS AND ANALYSIS

Figure 6.16 illustrates the mean risk score for each zone. Negative values signify a surplus of bikes relative to projected demand, while positive values indicate a shortfall that could lead to unmet requests. Although the majority of zones appear to have negative risk scores, a few remain positively skewed, implying that the agent sometimes leaves certain zones understocked despite regular visits.

Finally, Figure 6.17 shows the number of bikes deployed in each zone by the end of the episode. In many areas, the distribution appears relatively balanced, reflecting effective rebalancing. However, some zones with a large surplus (e.g., in the top-left and top-right regions of the map) remain underexploited by the agent, suggesting that there is still an opportunity to improve pickup decisions. Furthermore, certain areas, such as the bottom-central zone with an average of

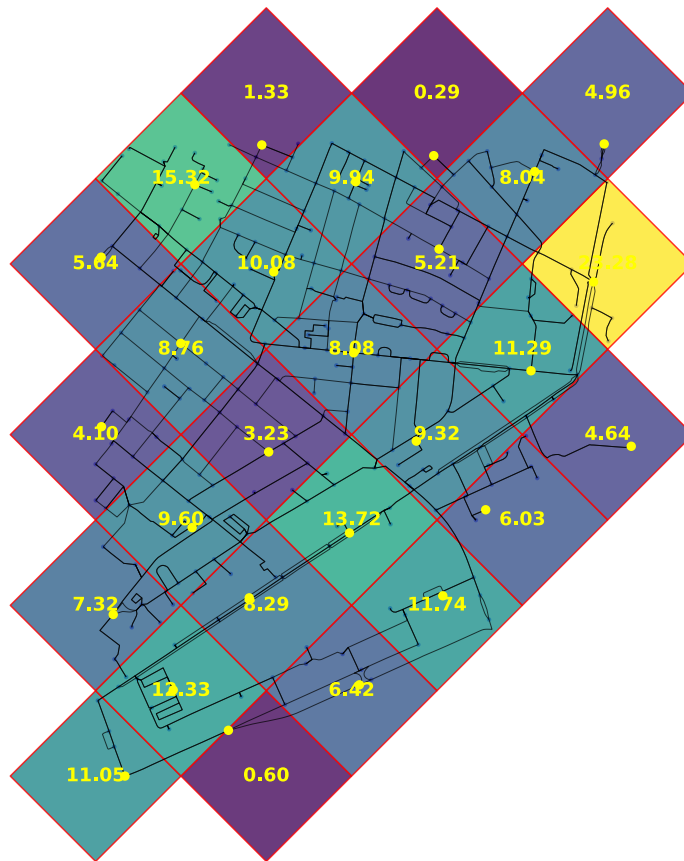


Figure 6.17: Distribution of bikes across zones at the end of the validation episode. Higher intensity indicates a greater number of bikes in that zone.

only 0.6 bikes per time slot, demonstrate high risk scores, indicating insufficient attention by the agent. This imbalance further highlights that the agent is not adequately addressing some critical areas.

The heat-maps offer direct insight into how well the agent responds to local imbalances in bike availability. Zones with persistent surpluses, such as the top-left region, indicate potential for deeper adjustments to the reward function or decision-making process, particularly in how pickups are valued relative to drops. Similarly, areas with repeated shortfalls highlight the need for better planned rebalancing routes and time-sensitive strategies. Identifying these trouble spots allows for more precise adjustments, making the system more efficient and equitable for all users.

Finally, the persistent failures observed in certain areas may stem from the agent's limited engagement with zones located at the corners of the operational area. These peripheral zones are visited and exploited less frequently, which contributes to suboptimal decisions and elevated failure rates. This behavior is likely influenced by movement restrictions near the boundaries, where some actions are censored, as detailed in the MDP Section of Chapter 5. Altogether, these observations suggest that the agent may benefit from a higher degree of exploration during training, potentially uncovering more effective strategies and improving performance in underutilized regions.



Conclusions and Future Work

This thesis presented a fully dynamic rebalancing framework for *dockless* BSSs based on DRL. Unlike most existing methods, which rely on coarse simplifications or focus primarily on static or periodic rebalancing, the proposed approach leverages an event-driven simulator to accurately capture fine-grained dynamics. These dynamics include user arrivals, stochastic trip durations, battery consumption, and detailed traffic conditions. The following sections summarize the thesis’s main contributions, highlight key insights, discuss limitations, and suggest directions for future research and practical implementation.

7.1 CONCLUSIONS

The primary contribution of this thesis is the development of a fully dynamic simulator that more faithfully mirrors the real-world operation of *dockless* BSSs compared to traditional static or discrete-time models. Incorporating fully dynamic events rather than averaging or clustering demands significantly increases modeling complexity. Managing real-time arrivals, traffic effects, and battery dynamics poses substantial algorithmic challenges; however, this level of detail proved essential for realistically assessing rebalancing strategies under actual operating conditions.

Within this dynamic environment, a DDQN agent was trained to make real-time rebalancing decisions, including where, when, and how many bikes to load or unload via a dedicated service truck. Although the proposed approach did

7.2. FUTURE WORK

not surpass certain state-of-the-art methods designed for simplified scenarios, it achieved three key outcomes:

1. It demonstrated, as a proof of concept, that effective rebalancing actions could be learned in a genuinely fully dynamic setting, reducing service failures and dynamically adapting to demand fluctuations.
2. It provided a scalable framework easily extendable to larger urban areas, such as the entirety of Cambridge, by adjusting local data inputs.
3. It identified significant challenges in reward engineering and neural network tuning, emphasizing the difficulty in capturing the multifaceted objectives of a *dockless* BSS within a unified learning pipeline.

Moreover, the intricate task of designing an appropriate reward function necessitates careful balancing of competing priorities. Similarly, tailoring neural network architectures capable of effectively addressing these diverse goals demands substantial computational resources and methodical experimentation. Despite these challenges, the approach clearly demonstrates its capability by substantially reducing imbalances compared to static or naive baseline methods, thus validating both its practicality and innovation.

Finally, while the proposed method can be compared with existing state-of-the-art approaches, it is crucial to recognize that many such comparisons involve simplified or stylized assumptions. Direct performance comparisons, therefore, inherently present limitations. Nevertheless, the fully dynamic and event-driven framework developed in this thesis offers valuable insights and represents an important step toward more realistic and practically applicable rebalancing strategies, capturing operational nuances that simpler or coarsely discretized models cannot.

7.2 FUTURE WORK

This research naturally opens several promising avenues for exploration, starting with enhancements to reward design. While the current study utilizes a piecewise reward structure, future work might investigate simpler yet economically-grounded reward functions explicitly integrating travel costs and rebalancing expenses. Refinements in state representation, such as incorporating more sophisticated spatiotemporal features or advanced interpolation techniques, could better capture the inherently *dockless* dynamics of demand.

Further improvements in network architecture could leverage cutting-edge deep learning approaches capable of effectively modeling complex, city-wide dependencies. Enhanced hyperparameter tuning and advanced exploration methods may improve learning stability and performance, narrowing the gaps currently observed with simplified methods. Additionally, deploying more robust computational resources would support increased computational demands and speed up training and evaluation cycles, particularly as simulations expand to larger urban scales.

A crucial immediate step involves extending the simulator to the entire city of Cambridge or other similarly sized urban areas. Scaling up the simulation will enable thorough evaluation under realistic urban operational scenarios, explicitly capturing complexities like loading/unloading times, accurate battery management, and operational cost constraints. Moreover, modeling more detailed demand patterns would necessitate accounting for additional real-world variables such as weather conditions, seasonal usage fluctuations, and variations across months.

Looking further ahead, introducing fairness metrics could ensure equitable bike distribution across different neighborhoods or demographic groups. Enhancing preprocessing techniques, including geographically informed interpolation or sophisticated demand forecasting methods, could further improve simulation accuracy. Ultimately, integrating Multi-Agent Reinforcement Learning (MARL) to coordinate multiple service trucks or autonomous vehicles represents an ambitious yet promising goal. MARL could significantly enhance scalability and system-wide performance through decentralized, parallelized decision-making, although it introduces complex challenges related to agent coordination and communication.

Advancing along these directions will deepen simulation-based insights and significantly improve real-world operational capabilities of BSSs. As cities increasingly adopt bike sharing to support sustainable and efficient urban mobility, robust and truly dynamic rebalancing frameworks will be essential in delivering cost-effective, reliable, and user-friendly transportation solutions.

Appendix

Symbol	Units	Meaning
$\mathcal{G}$	–	Graph representing the BSS.
$\mathcal{V}$	–	Set of nodes (stations) in the graph.
$\mathcal{E}$	–	Set of directed edges representing bike movements.
N	–	Total number of nodes in the network.
$S_n(t)$	–	Number of available bikes at node n at time t .
$\mathbf{S}(t)$	–	Vector representing system-wide bike availability at time t .
$S_n^{\max}$	–	Maximum capacity of bikes at station n .
w_{nm}	–	Weight of edge (n, m) , encoding distance, time, or cost.
L	–	Distance matrix representing distances between stations.
ℓ_{nm}	Km	Distance between nodes n and m .
τ	–	Index for time slots in a day.
$\mathcal{B}$	–	Set of all bikes in the system.
$p_b(t)$	–	Position of bike b at time t .
$c_b(t)$	Km	Battery charge of bike b at time t .
$C_{\max}^b$	Km	Maximum battery capacity of bike b .
v	$\frac{\text{Km}}{\text{h}}$	Mean travel velocity.
μ_v, σ_v^2	$\frac{\text{Km}}{\text{h}}, \left(\frac{\text{Km}}{\text{h}}\right)^2$	Mean and variance of truncated Gaussian velocity distribution.
$v_{\min}, v_{\max}$	$\frac{\text{Km}}{\text{h}}$	Lower and upper bounds for velocity.
$p_{\text{truck}}(t)$	–	Truck's position at time t .
$K_{\text{truck}}(t)$	–	Number of bikes carried by the truck at time t .
$K_{\max}$	–	Maximum capacity of the truck.
$c_{\text{truck}}(t)$	Km	Battery charge of the truck at time t .
$\delta_{\text{load}}(t)$	–	Number of bikes loaded onto the truck at time t .
$\delta_{\text{unload}}(t)$	–	Number of bikes unloaded from the truck at time t .
$\mathcal{S}$	–	Set of all possible bike distributions among stations.
$\mathcal{T}$	–	Set of all possible bike transitions between stations.
$S_n^a(t)$	–	Number of active bikes at station n at time t .
$S_n^d(t)$	–	Number of depleted bikes at station n at time t .
$S_{\text{ext}}(t)$	–	Number of bikes outside the system.
$T_{(n,m)}(t)$	–	Number of bikes traveling from station n to station m .
$\lambda_n(\tau)$	–	Total departure rate from station n in time slot τ .
$\lambda_n^{\text{int}}(\tau)$	–	Internal departure rate from station n .
$\lambda_n^{\text{ext}}(\tau)$	–	External departure rate from station n .

Table 1: Key symbols, units, and meanings used in Chapter 4 (Part 1).

Symbol	Units	Meaning
$p_{n \rightarrow m}(\tau)$	–	Probability of a trip from n to m in time slot τ .
$\mu_n(\tau)$	–	Total arrival rate at station n in time slot τ .
$\gamma_{n \rightarrow m}$	–	Rate of travel completion from station n to station m .
$\mathbf{Q}(\tau)$	–	Infinitesimal generator matrix of the CTMC.
$p(i, j; \tau)$	–	Transition probability in the embedded Markov chain.
$P(\tau)$	–	Transition matrix of the embedded Markov chain.

Table 2: Key symbols, units, and meanings used in Chapter 4 (Part 2).

Acknowledgments

This study was carried out within the MOST (the Italian National Center for Sustainable Mobility) and received funding from NextGenerationEU (Italian PNRR – CN00000023 - D.D. 1033 17/06/2022 - CUP C93C22002750006).

References

- [1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. “Finite-time analysis of the multiarmed bandit problem”. In: *Machine learning* 47 (2002), pp. 235–256.
- [2] Bluebikes. *System Data*. URL: <https://bluebikes.com/system-data>.
- [3] Bruno P Bruck et al. “The static bike sharing rebalancing problem with forbidden temporary operations”. In: *Transportation science* 53.3 (2019), pp. 882–896.
- [4] Matteo Cederle et al. “A Fairness-Oriented Reinforcement Learning Approach for the Operation and Control of Shared Micromobility Services”. In: *arXiv preprint arXiv:2403.15780* (2024). URL: <https://arxiv.org/abs/2403.15780>.
- [5] Robert Cervero. *Transport infrastructure and the environment: Sustainable mobility and urbanism*. IURD, Institute of Urban and Regional Development, University of California, 2013.
- [6] Daniel Chemla, Frédéric Meunier, and Roberto Wolfler Calvo. “Bike sharing systems: Solving the static rebalancing problem”. In: *Discrete Optimization* 10.2 (2013), pp. 120–146.
- [7] Federico Chiariotti et al. “A bike-sharing optimization framework combining dynamic rebalancing and user incentives”. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 14.3 (2020), pp. 1–30.
- [8] Federico Chiariotti et al. “A dynamic approach to rebalancing bike-sharing systems”. In: *Sensors* 18.2 (2018), p. 512.
- [9] Claudio Contardo, Catherine Morency, and Louis-Martin Rousseau. *Balancing a dynamic public bike-sharing system*. Vol. 4. Cirrelet Montreal, Canada, 2012.

REFERENCES

- [10] Edsger W Dijkstra. “A note on two problems in connexion with graphs”. In: *Edsger Wybe Dijkstra: his life, work, and legacy*. 2022, pp. 287–290.
- [11] Marco Fabris, Riccardo Ceccato, and Andrea Zanella. “Efficient Sensors Selection for Traffic Flow Monitoring: An Overview of Model-Based Techniques Leveraging Network Observability”. In: *Sensors* 25.5 (2025). issn: 1424-8220.
- [12] Elliot Fishman. *Bike share*. Routledge, 2019.
- [13] Robert G. Gallager. *Discrete Stochastic Processes*. Boston, MA: Springer, 1996. ISBN: 978-0-7923-9583-6.
- [14] Supriyo Ghosh et al. “Dynamic repositioning to reduce lost demand in bike sharing systems”. In: *Journal of Artificial Intelligence Research* 58 (2017), pp. 387–430.
- [15] Ian Goodfellow et al. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016.
- [16] Frank Harary. *Graph theory (on Demand Printing of 02787)*. CRC Press, 2018.
- [17] Il Post. *L’unica grande capitale europea senza bike sharing*. 2018. URL: <https://www.ilpost.it/2018/10/28/roma-bike-sharing-free-floating/>.
- [18] Thomas N Kipf and Max Welling. “Semi-supervised classification with graph convolutional networks”. In: *arXiv preprint* (2016). URL: <https://arxiv.org/abs/1609.02907>.
- [19] Yuxi Li. “Deep Reinforcement Learning: An Overview”. In: *arXiv preprint* (2017). URL: <https://arxiv.org/abs/1701.07274>.
- [20] Jiaqi Liang et al. “A Reinforcement Learning Approach for Dynamic Rebalancing in Bike-Sharing System”. In: *arXiv preprint* (2024). URL: <https://arxiv.org/abs/2402.03589>.
- [21] Junming Liu et al. “Rebalancing bike sharing systems: A multi-source data smart optimization”. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 1005–1014.
- [22] Russell Meddin. *The Bike-Sharing World Map*. Accessed: 2025-01-29. 2025. URL: <https://bikesharingworldmap.com/>.
- [23] Francisco S Melo. “Convergence of Q-learning: A simple proof”. In: *Institute Of Systems and Robotics, Tech. Rep* (2001), pp. 1–4.
- [24] Peter Midgley. “The role of smart bike-sharing systems in urban mobility”. In: *Journeys* 2.1 (2009), pp. 23–31.

- [25] Volodymyr Mnih. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).
- [26] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.
- [27] Natalie Mueller et al. "Changing the urban design of cities for health: The superblock model". In: *Environment international* 134 (2020), p. 105132.
- [28] Mark J Nieuwenhuijsen and Haneen Khreis. "Car free cities: Pathway to healthy urban living". In: *Environment international* 94 (2016), pp. 251–262.
- [29] Eoin O'Mahony and David Shmoys. "Data analysis and optimization for (citi) bike sharing". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 29. 1. 2015.
- [30] World Health Organization. *World Health Day 2010: Urbanization and Health*. 2010. URL: <https://www.intrahealth.org/vital/world-health-day-2010-urbanization-and-health>.
- [31] Baoran Pan, Lixin Tian, and Yingdong Pei. "The Novel Application of Deep Reinforcement to Solve the Rebalancing Problem of Bicycle Sharing Systems with Spatiotemporal Features". In: *Applied Sciences* 13.17 (2023), p. 9872.
- [32] Ling Pan et al. "A deep reinforcement learning framework for rebalancing dockless bike sharing systems". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 1393–1400.
- [33] John Pucher, Ralph Buehler, and Mark Seinen. "Bicycling renaissance in North America? An update and re-appraisal of cycling trends and policies". In: *Transportation research part A: policy and practice* 45.6 (2011), pp. 451–475.
- [34] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
- [35] Tal Raviv, Michal Tzur, and Iris A Forma. "Static repositioning in a bike-sharing system: models and solution approaches". In: *EURO Journal on Transportation and Logistics* 2.3 (2013), pp. 187–229.
- [36] Jasper Schuijbroek, Robert C Hampshire, and W-J Van Hoes. "Inventory rebalancing and vehicle routing in bike sharing systems". In: *European Journal of Operational Research* 257.3 (2017), pp. 992–1004.

REFERENCES

- [37] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [38] TomTom. *Boston, MA Traffic Report | TomTom Traffic Index*. 2025. URL: <https://www.tomtom.com/traffic-index/boston-ma-traffic/>.
- [39] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with double q-learning”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.
- [40] Petar Veličković et al. “Graph attention networks”. In: *arXiv preprint* (2017). URL: <https://arxiv.org/abs/1710.10903>.
- [41] Zhuoli Yin, Zhaoyu Kou, and Hua Cai. “A deep reinforcement learning model for large-scale dynamic bike share rebalancing with spatial-temporal context”. In: *The 12th International Workshop on Urban Computing*. 2023.
- [42] Sheng Zhang, Guanhua Xiang, and Zhongxiang Huang. “Bike-Sharing Static Rebalancing by Considering the Collection of Bicycles in Need of Repair”. In: *Journal of Advanced Transportation* 2018.1 (2018), p. 8086378.