REINFORCEMENT LEARNING APPROACHES ON ELEVATOR GROUP CONTROL PROBLEM

by
EMRE YAVAŞ

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
October 2024

ABSTRACT

REINFORCEMENT LEARNING APPROACHES ON ELEVATOR GROUP CONTROL PROBLEM

EMRE YAVAŞ

MECHATRONICS ENGINEERING M.Sc. THESIS, OCTOBER 2024

Thesis Supervisor: Asst Prof. Dr. Sinan Yıldırım

Thesis CoSupervisor: Assoc. Prof. Dr. Ahmet Onat

Keywords: Deep Reinforcement Learning, Deep Q Network, Elevator Group Control, Group Traffic Control

Elevators play a crucial role in the functionality of skyscrapers. As the number of skyscrapers increases, so does the demand for efficient elevator systems. Traditional single lift traffic control systems are inadequate for high-rise buildings, leading to the adoption of multiple lift traffic control systems. These systems feature multiple shafts, each containing one or more elevator cars. Effective elevator group control (EGC) strategies are essential to ensure elevators work together to minimize passenger waiting times.

This thesis explores the application of reinforcement learning (RL) to enhance the performance of elevator group control systems (EGCS). Given the NP-hard nature of scheduling elevator cars, where no known optimal solution exists, a learning-based approach offers a promising alternative to heuristic methods. This thesis proposes two deep Q-network (DQN) techniques that enable the EGCS to dynamically assign passenger calls to the most suitable elevator car, aiming to reduce passenger waiting times and improve overall system efficiency.

ÖZET

ASANSÖR GRUP KONTROL PROBLEMI ÜZERINE PEKIŞTIRMELI ÖĞRENME YAKLAŞIMLARI

EMRE YAVAŞ

MEKATRONİK MÜHENDİSLİĞİ YÜKSEK LİSANS TEZİ, EKİM 2024

Tez Danışmanı: Dr. Öğr Üyesi Sinan Yıldırım

Tez Eşdanışmanı: Doç. Dr. Ahmet Onat

Anahtar Kelimeler: Derin Pekiştirmeli Öğrenme, Derin Q Ağı, Asansör Grup Kontrolü, Grup Trafik Kontrolü

Asansörler, gökdelenlerin işlevselliğinde kritik bir rol oynamaktadır. Gökdelen sayısının artmasıyla birlikte, verimli asansör sistemlerine olan ihtiyaç da artmaktadır. Geleneksel tek asansörlü trafik kontrol sistemleri, yüksek binalar için yetersiz kalmakta ve bu durum, çoklu asansör trafik kontrol sistemlerinin gerekliliğini göstermektedir. Bu sistemler, her birinde bir veya daha fazla asansör kabini bulunan birden fazla şaft içerir. Etkili asansör grup kontrolü stratejileri, asansörlerin yolcu bekleme sürelerini en aza indirmek için birlikte çalışmasını sağlamak açısından faydalıdır.

Bu tez, asansör grup kontrol sistemlerinin performansını artırmak için pekiştirmeli öğrenme metodlarının uygulanmasını incelemektedir. Asansör kabinlerinin zamanlaması, bilinen optimal bir çözümün olmadığı, NP-Zor bir problem olduğu için, öğrenme temelli bir yaklaşım, sezgisel yöntemlere alternatif olarak umut verici bir çözüm sunmaktadır. Bu çalışmada, yolcu çağrılarını en uygun asansör kabinine dinamik olarak atayarak yolcu bekleme sürelerini azaltmayı ve genel sistem verimliliğini artırmayı amaçlayan iki derin Q-ağı (DQN) tekniği önerilmektedir.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my co-supervisor, Assoc. Prof. Dr. Ahmet Onat, for his invaluable encouragement, thoughtful guidance, and insightful feedback throughout the completion of this thesis. His kindness, patience, and unwavering support have made a profound impact on my academic journey. I am also deeply grateful to Assoc. Prof. Dr. Onat for providing continuous financial support during my master's studies.

I extend my heartfelt thanks to my friends, each of whom has played a significant role in this journey. In particular, I am immensely thankful to Berker for his valuable insights and emotional support during moments of self-doubt. My deepest gratitude also goes to Ali Enver for his steadfast support and assistance throughout this process. Additionally, I am sincerely appreciative of my friends Mahmut, Recep, Onur, and Banu. Despite the physical distance, their presence in my life has always been a source of encouragement and strength.

I would also like to acknowledge my friends from campus and the lab, including Osman, Alperen, Burak, Harun, Arif, Ahmed, Feyza, Işıl and Zeynep for their companionship and the enjoyable moments we shared during this journey. Their conversations, humor, and shared activities provided a welcome break and helped me maintain a balanced perspective. I am grateful for their presence and the memories we created together.

Finally, I would like to express my heartfelt gratitude to my parents, Hüseyin and Esra, and my brother, Evren, for their unwavering love and support throughout this journey. I am truly fortunate to have a family who is always there for me whenever I need them. Their belief in me has been a constant source of strength and motivation, and I am deeply grateful for their presence in my life.

Dedicated to the field

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATONS

DQN Deep Q Network . . 2, 9, 10, 11, 12, 13, 14, 15, 22, 29, 31, 32, 33, 34, 35, 36, 37, 40, 41, 43, 44

EGC Elevator Group Control 2, 4, 15, 43

EGCS Elevator Group Control System . . . . . iv, 4, 5, 16, 17, 29

FS Figure of Suitability 5

ReLU Rectified Linear Unit 33

RL Reinforcement Learning 2, 7, 8, 9, 15, 26, 29, 35, 40, 41, 43, 44

1. INTRODUCTION

1.1 Motivation

Elevators are essential in buildings to satisfy the vertical transportation requirements of occupants. Particularly in high-rise buildings, passengers expect elevators to move between floors within a reasonable time-frame. With the increasing number of people in these buildings, the demand for elevator services has surged. A single elevator is insufficient to handle the vertical commuting needs in high-rise structures, necessitating the installation of multiple elevators and a coordinated control mechanism. This control mechanism, known as the Elevator Group Control System (EGCS), is responsible for efficiently managing all elevator requests by assigning elevators to floor calls while optimizing various criteria, such as reducing energy consumption, minimizing total travel time, handling peak-hour traffic efficiently Crites & Barto (1998) Wei et al. (2020).

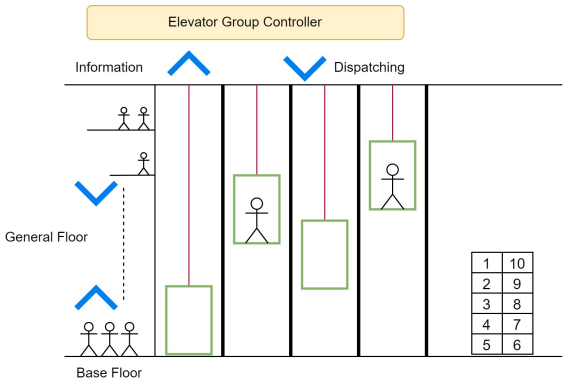

Figure 1.1 A typical elevator group control system

Reinforcement learning (RL) is a sub-field of machine learning in which an agent interacts with an environment by performing actions. After the action, environment

is affected and the agent observes new state of the environment and receives an immediate reward based on how well its action aligns with the objective of the problem. This cycle occurs sequentially. Since an RL agent must make a series of decisions in a changing environment, RL algorithms are well suited for application on dynamical systems, which require continuous adaptation to changing conditions. Sutton & Barto (2018). The general framework of RL is illustrated in Figure 1.2

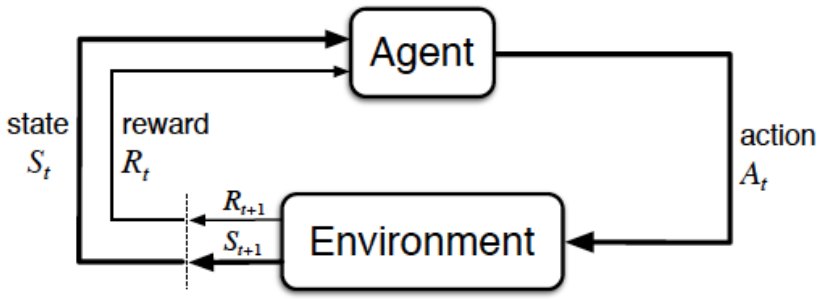

Figure 1.2 The agent–environment interaction

The elevator group control (EGC) problem can be modeled as a sequential decision-making process. Elevator systems operate within high-dimensional continuous state spaces and function in continuous time as discrete-event dynamic systems. These systems often have states that are not fully observable and are non-stationary due to fluctuating passenger arrival rates Crites & Barto (1998). Given the dynamic nature of elevator systems, which involve changing passenger demands and unpredictable traffic patterns, RL algorithms present a promising approach by continuously learning from incoming elevator requests to optimize the EGC problem Wei et al. (2020).

This thesis presents a new approach to elevator group control. We propose a deep learning based group controller which is not constrained by conventional empirical approaches that are based on experiences of elevator designers. We support our ideas using a realistic simulation of an elevator group control problem, encompassing vertical traffic, passengers, and building dynamics. Two reinforcement learning approaches, Deep Q-Network (DQN) and Double Deep Q-Network (Double DQN), are applied to the environment using two distinct state space representations. A nearest-car algorithm serves as a baseline for performance comparison.

The first agent is a DQN model employing a convolutional neural network. It utilizes an elevator state matrix and passenger vectors, which are detailed in subsequent sections. This agent performs slightly better than the baseline under light traffic but struggles in other scenarios.

The second agent is a Double DQN model with embedding and fully connected layers. It is evaluated under three distinct traffic patterns, each with varying traffic

intensities. This agent outperforms the nearest-car algorithm in two patterns and performs comparably, though slightly worse, in the third pattern across all traffic intensities.

1.2 Outline

The outline of the thesis is presented in this section.

Chapter 1: Introduction

This chapter provides an introduction to the work and the thesis. It sets the context for the study and outlines the objectives and scope of the research.

Chapter 2: Literature Survey and Background

This chapter presents a comprehensive literature survey on the elevator group control problem. It reviews a heuristic solution to the problem and explores various reinforcement learning approaches.

Chapter 3: Reinforcement Learning Fundamentals

This chapter delves into the fundamentals of reinforcement learning, with a specific focus on the deep Q-network (DQN) method used in this study.

Chapter 4: Simulation Environment

This chapter details the simulation environment developed for this research. It describes the design and implementation of the simulation used to test the reinforcement learning and nearest-car algorithms.

Chapter 5: Experimental Results

This chapter presents the results of experiments conducted using both reinforcement learning and a heuristic approach (nearest car). It shows the performance and effectiveness of these methods.

Chapter 6: Future Work and Conclusion

This chapter discusses potential future work and concludes the thesis. It summarizes the key findings and contributions of the research.

2. Literature Survey and Background

In this chapter, the EGC problem is described, and several control strategies addressing the problem will be briefly outlined. The nearest-car strategy will be discussed in more detail, as it serves as the baseline for the results.

2.1 Elevator Group Control Problem

A single elevator cannot always meet the vertical transportation demands of passengers. Consequently, multiple elevators are often installed side by side. However, if these elevators operate independently with separate call buttons, they may function sub-optimally. Passengers might call multiple elevators but use only one, or fail to select the elevator that minimizes travel time. Therefore, an EGC strategy is essential for optimizing the overall performance of the elevators Barney & Al-Sharif (2015).

An EGCS includes hall call buttons that are located on each floor for incoming passengers, car call buttons inside every elevator cabin, and a centralized group controller. The EGCS is responsible for managing several elevators within a building to transport passengers efficiently. Its primary task is to allocate the most appropriate car for each landing call.

The optimization of EGCS is considered as complex task. Given the uncertainty and combinatorial nature of elevator group scheduling, purely mathematical methods have limited effectiveness in improving the performance of the vertical transportation system Li (2010).

Various approaches to the EGC problem are discussed in the literature. These include rule-based and heuristic methods, which are categorized as classical control techniques, as well as Artificial Intelligence-based approaches such as fuzzy

logic control, genetic algorithms Li & He (2023) and multi-agent systems Crites & Barto (1998). However, these are not the only approaches to the problem, as other techniques, including optimization methods and hybrid strategies, have also been explored in the literature.

2.1.1 Nearest Car Algorithm

The nearest-car strategy is a widely used approach in EGCS, primarily due to its simplicity and effectiveness in reducing passenger waiting times in low-traffic scenarios. his method assigns an elevator to a landing call based on rules that take into account the direction of travel and the floor locations of both the passenger and the elevators. A key component of this approach is the calculation of a Figure of Suitability (FS) value for each elevator. This value is determined using predefined criteria, such as the distance between the elevator and the passenger's floor, the current direction of the elevator relative to the call, and the state of the elevator (e.g., idle, moving, or full). The elevator with the highest FS score is dispatched to handle the request, ensuring a reasonably efficient allocation in some scenarios. The simplicity of the FS calculation also makes the nearest-car strategy computationally lightweight, which is advantageous in real-time systems with limited processing power. However, the nearest-car strategy offers no guarantee of optimality and is often sub-optimal in high-traffic conditions. Despite its simplicity, it serves as a useful baseline for more advanced optimization techniques in EGCS Barney & Al-Sharif (2015).

Nevertheless, the nearest-car strategy serves as a useful baseline for evaluating more sophisticated optimization techniques in EGCS, such as heuristic-based methods, artificial intelligence, or predictive models. Its role in hybrid systems, where it is combined with other algorithms to address its shortcomings, also highlights its versatility and continued relevance in elevator group control Barney & Al-Sharif (2015).

The nearest-car algorithm operates based on a set of rules to determine which elevator is most suitable to respond to a landing call. These rules are used to calculate the FS for each elevator in the system. While the exact implementation may vary slightly depending on the systems, the algorithm generally follows these steps:

Algorithm 1 Nearest-Car Strategy for Elevator Assignment

Require: List of elevators (with state, current floor, direction), landing call (floor, direction)

Ensure: Assigned elevator with the highest FS

$N \leftarrow$ total number of floors in the building
$max\_FS \leftarrow -\infty$
$assigned\_elevator \leftarrow None$
$d \leftarrow 0$
for each elevator e in elevators **do**
    $d \leftarrow |e.current\_floor - landing\_call.floor|$ ▷ Calculate distance to the landing call
    if $e.state =$ MOVING **and** $e.direction = landing\_call.direction$ **then**
        $FS \leftarrow (N+2) - d$ ▷ Rule 1: Moving towards the call in the same direction
    else if $e.state =$ MOVING **and** $e.direction \neq landing\_call.direction$ **then**
        $FS \leftarrow (N+1) - d$ ▷ Rule 2: Moving towards the call in the opposite direction
    else if $e.state =$ IDLE **then**
        $FS \leftarrow (N+1) - d$ ▷ Rule 3: Elevator is idle
    else if $e.state =$ MOVING_AWAY **then**
        $FS \leftarrow 1$ ▷ Rule 4: Moving away from the call
    else
        $FS \leftarrow 0$ ▷ Default FS for undefined states
    end if
    if $FS > max\_FS$ **then**
        $max\_FS \leftarrow FS$
        $assigned\_elevator \leftarrow e$
    end if
end for
return $assigned\_elevator$

3. Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning in which an agent learns to take optimal actions by continuously interacting with a dynamic environment to achieve a given objective. The key components of RL are an *agent*, an *environment*, *actions* and *rewards*. At each time step, the agent observes the environment to some extent and selects an action. The environment responds by transitioning to a new state and providing the agent with a scalar reward, which reflects how agent's action aligns with agent's goal in that step. This feedback loop iteratively occurs until the agent achieves the goal or encounters failure.

RL is different from supervised and unsupervised learning in terms of; environment interaction, feedback mechanism, data availability and learning objectives. Firstly, RL deals with dynamic environments, where the agent's actions can affect not only the immediate state of the environment but also the future states of the environment. Secondly, the agent receives a scalar numerical reward signal rather than labeled examples. This reward may be immediate or delayed, and it does not serve as a direct label, but instead as an evaluative feedback signal. Thirdly, in RL problems there are no explicit data, but the agent collects its own data by sensing the current and next states of the environment, receiving the rewards and the actions it takes. Finally, instead of simply optimizing a single-step performance measure, RL aims to maximize the cumulative reward, often referred to as the *return* or *cumulative gain*, over the entire interaction horizon.

To formalize this objective, the concept of *policy* (π) is introduced, which serves as a mapping from states s to probabilities of selecting each possible action a. At each time step t, the agent observes a state s_t, takes an action a_t drawn from $\pi(a_t \mid s_t)$, and receives a reward r_{t+1}. The process continues until a terminal state or indefinitely if the task does not terminate. The cumulative gain $J(\pi)$, also known as the expected return under policy π, is defined as the expected discounted sum of future rewards:

$$J(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \right], \tag{3.1}$$

where $\gamma \in [0,1]$ is the discount factor that determines the importance of future rewards relative to immediate ones.

To evaluate how good it is to be in a certain state (or to take a certain action in that state), *value functions* are introduced. The *state-value function* $V_\pi(s)$ under a policy π is defined as the expected return when starting from state s and following π thereafter:

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_0 = s \right]. \tag{3.2}$$

Similarly, the *action-value function* $Q_\pi(s,a)$ is the expected return when starting from state s, taking action a, and following π afterwards:

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_0 = s, a_0 = a \right]. \tag{3.3}$$

The Bellman equations provide a recursive relationship for these value functions. For the state-value function under policy π:

$$V_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V_\pi(s') \right], \tag{3.4}$$

where $p(s', r \mid s, a)$ is the probability of transitioning to state s' and receiving reward r given the current state s and action a.

For the action-value function, the Bellman equation is:

$$Q_\pi(s,a) = \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma \sum_{a'} \pi(a' \mid s') Q_\pi(s', a') \right]. \tag{3.5}$$

These equations form the theoretical backbone of Reinforcement Learning. By repeatedly applying these relationships, either directly in tabular form for small problems or via function approximation (e.g., neural networks) in larger, more complex environments, an RL agent can estimate value functions and improve its policy to maximize the cumulative gain J.

In summary, RL focuses on maximizing long-term performance as measured by the expected return J. Through interaction with the environment, the agent updates its value functions and refines its policy. The Bellman equations and their related formulations guide this iterative improvement process, ultimately steering the agent towards policies that yield higher returns over the long run.

3.1 Deep Q Network

Q-learning is a foundational RL algorithm designed to solve Markov Decision Processes (MDPs). Q-learning is an off-policy RL algorithm that learns an action-value function $Q(s, a)$, which estimates the expected cumulative reward for taking action a in state s and following an optimal policy thereafter Watkins & Dayan (1992). The Bellman equation (3.5) forms the basis for Q-learning and update step for action value function is shown in equation 3.6.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right) \tag{3.6}$$

In traditional Q-learning, a table of Q-values is maintained for each state-action pair as illustrated in Figure 3.1. However, this approach becomes impractical for large or continuous state spaces. To overcome this limitation, Deep Q Network (DQN) is introduced by Mnih (2013). DQN utilizes deep neural networks (also called deep Q networks) to approximate the Q-value function (see Figure 3.2). The input to the network is the state s and the output is a vector of Q-values for all possible actions Mnih (2013). This architecture allows DQN to generalize over large state spaces and effectively capture complex features of the environments.

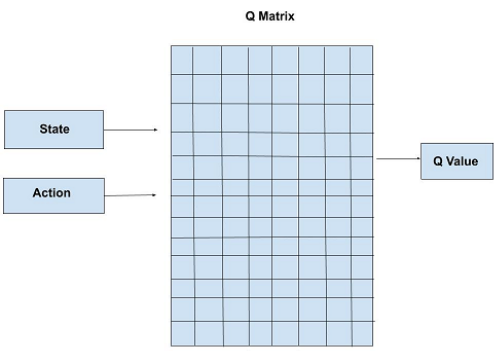

Figure 3.1 Q-learning table

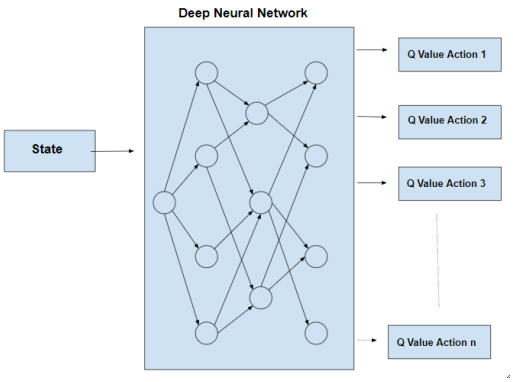

Figure 3.2 Deep Q Network Mapping

The DQN algorithm uses the Bellman equation to iteratively update Q-values. However, instead of storing Q-values explicitly, the network minimizes the temporal difference error, which is the difference between the predicted Q-value and the target Q-value:

$$\text{Loss} = (y_t - Q(s_t, a_t; \theta))^2 \tag{3.7}$$

where the target Q-value y_t is computed as:

$$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) \tag{3.8}$$

DQN relies on the Bellman Equation (3.5) to iteratively update the Q-values. The Bellman equation expresses the Q-value for a state-action pair as the sum of the immediate reward and the discounted maximum expected future reward. The Bellman update used in DQN is given by:

$$Q(s_t, a_t; \theta) \leftarrow Q(s_t, a_t; \theta) + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) - Q(s_t, a_t; \theta) \right]. \tag{3.9}$$

The training process in DQN is based on minimizing the temporal difference (TD) error, which quantifies the difference between the predicted Q-value and the target Q-value. The target Q-value is computed using the immediate reward from the current action and the discounted maximum Q-value of the next state. To stabilize training and improve sample efficiency, DQN employs experience replay. Instead of

updating the network parameters sequentially after every action, experience replay stores transitions (s_t, a_t, r_t, s_{t+1}) in a replay memory. During training, mini-batches of transitions are randomly sampled from this memory to break the temporal correlations between consecutive experiences, which helps to mitigate instability and divergence in the learning process Mnih (2013).

Additionally, DQN adopts the epsilon-greedy exploration strategy to balance exploration and exploitation. During training, the agent selects actions with a probability of ϵ for exploration (choosing a random action) and with a probability of $1-\epsilon$ for exploitation (choosing the action with the highest Q-value) as shown in Algorithm 2. The value of ϵ typically starts high to encourage exploration in the early stages of training and gradually decays over time to focus more on exploitation as the agent learns a more accurate Q-function Mnih (2013).

Algorithm 2 Epsilon-Greedy Action Selection

```
Sample a random number p from a uniform distribution U(0,1)
if p < ε then
    Select a random action a from the action space A        ▷ Exploration
else
    Select action a = arg max_a' Q(s,a')                    ▷ Exploitation
end if
Execute action a
```

While the vanilla DQN introduces several key innovations, it still faces challenges with stability during training. One such issue arises from the moving target problem, where the Q-network is used to both estimate the current Q-values and compute the target Q-values. This dual use can lead to oscillations or divergence during training.

To address this issue, Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski & others (2015) introduces the idea of target network. DQN incorporates a target network as an additional stability mechanism. The target network is a separate copy of the Q-network that is updated less frequently. Instead of directly using the Q-network to compute the target Q-values, the target network is used. Specifically, the target Q-value for a given transition (s_t, a_t, r_t, s_{t+1}) is computed as:

$$y_t = r_t + \gamma \max_{a_{t+1}} Q_{target}(s_{t+1}, a_{t+1}; \theta^-) \tag{3.10}$$

where $Q_{target}(s_{t+1}, a_{t+1}; \theta^-$ represents the target network's estimate of the Q-value, parameterized by θ^-. The parameters θ^- of the target network are updated by

periodically copying the parameters θ from the Q-network as shown in Algorithm(3). This decoupling of the target generation and policy evaluation processes reduces the risk of feedback loops and contributes to more stable convergence during training Mnih et al. (2015).

Algorithm 3 Deep Q-Network (DQN) Algorithm

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for each episode **do**
    Initialize starting state s_0
    for each step in the episode **do**
        With probability ϵ, select a random action a_t
        Otherwise, select $a_t = \arg\max_a Q(s_t, a; \theta)$
        Execute action a_t and observe reward r_t and next state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in replay memory $\mathcal{D}$
        Sample a random mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on loss $L = (y_j - Q(s_j, a_j; \theta))^2$
        Every C steps, update target network $\hat{Q}$ by setting $\theta^- = \theta$
    end for
end for

DQN has been successfully applied to a variety of complex tasks, such as playing Atari video games directly from pixel input, where it outperformed human-level performance in many cases Mnih et al. (2015). This demonstrates its potential for solving high-dimensional decision-making problems in both discrete and continuous environments.

Building on this foundation, the Poljak update method, introduced by Polyak (1964) to accelerate the convergence of iterative methods, can also be applied to the DQN framework to enhance stability during training. This approach updates the target network parameters θ^- using a weighted combination of the current policy network parameters θ and the existing target network parameters θ^-. The update rule is defined as:

$$\theta^- = \tau\theta + (1-\tau)\theta^-$$

where:

- θ represents the parameters of the current policy network,

- θ^- represents the parameters of the target network,
- τ $(0 < \tau \leq 1)$ is the weighting factor controlling the degree of interpolation between θ and θ^-.

When $\tau = 1$, this update rule reduces to directly copying the policy network parameters to the target network, which is the standard approach in DQN. Smaller values of τ result in more gradual updates, improving stability and reducing the risk of divergence. This makes the Poljak update particularly useful in scenarios where consistency and stability are critical."

3.1.1 Double Deep Q Network

Even though the DQN algorithm achieves an impressive success in solving complex tasks, it suffers from a known limitation called overestimation bias. This occurs when the algorithm tends to overestimate the value of certain state-action pairs due to the use of the same network for both selecting and evaluating actions. Over time, these overestimations can lead to sub-optimal policies, particularly in environments with noisy or stochastic rewards van Hasselt, Guez & Silver (2016).

To overcome this limitation, Double Deep Q Network (Double DQN) was introduced. The key difference between DQN and Double DQN lies in how the target Q-value is calculated. Instead of using a single Q-network to both select and evaluate the action, Double DQN decouples these roles. It uses the Q-network to select the best action, but it uses the target network to evaluate the value of that action. This modification reduces the likelihood of overestimation and leads to more stable learning.

The update rule for Double DQN is given by:

$$y_j = r_j + \gamma Q(s_{j+1}, \arg\max_{a'} Q(s_{j+1}, a'; \theta); \theta^-) \tag{3.11}$$

In this equation, the action selected $(\arg\max_{a'} Q(s_{j+1}, a'; \theta))$ is based on the current Q-network, but the evaluation of the selected action uses the target network with parameters θ^-. This decoupling helps in reducing the bias and improving the accuracy of the Q-value estimates.

Double DQN has been shown to outperform standard DQN in many environments, especially those where overestimation is problematic. By mitigating the overestimation bias, it achieves more stable learning and improves overall policy performance van Hasselt et al. (2016).

$$Q(s_t, a_t; \theta) \leftarrow Q(s_t, a_t; \theta) + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) - Q(s_t, a_t; \theta) \right] \tag{3.12}$$

4. Elevator Group Control Problem Simulation Environment

An event-based simulation environment with a graphical user interface (see Fig 4.1) for the EGC problem is implemented to closely mimic real-world conditions in high-rise buildings. The simulation is designed to realistically replicate elevator dynamics, passenger behavior, and vertical traffic patterns, making it a robust testing platform for the DQN and nearest-car algorithms. The simulation accounts for events such as passengers arrivals, boarding and leaving from elevators, elevator door operations (moving and closing) and moving between floors, with the time consumed for each event carefully considered to ensure realistic modeling. Python is selected as the programming language for the simulator due to its widespread use in RL environments and algorithms. It contains widely used environments and data structures with OpenAI Gym Brockman, Cheung, Pettersson, Schneider, Schulman, Tang & Zaremba (2016) toolkit. The implementation leverages data structures from the OpenAI Brockman et al. (2016) for the environment and employs PyTorch Paszke, Gross, Massa, Lerer, Bradbury, Chanan, Killeen, Lin, Gimelshein, Antiga, Desmaison, Kopf, Yang, DeVito, Raison, Tejani, Chilamkurthy, Steiner, Fang, Bai & Chintala (2019) as the foundational framework for the neural network. The system's primary objective is defined as minimizing the average travel time, ensuring that the results are not only practical but also reflect realistic operational constraints and goals of high-rise building management.

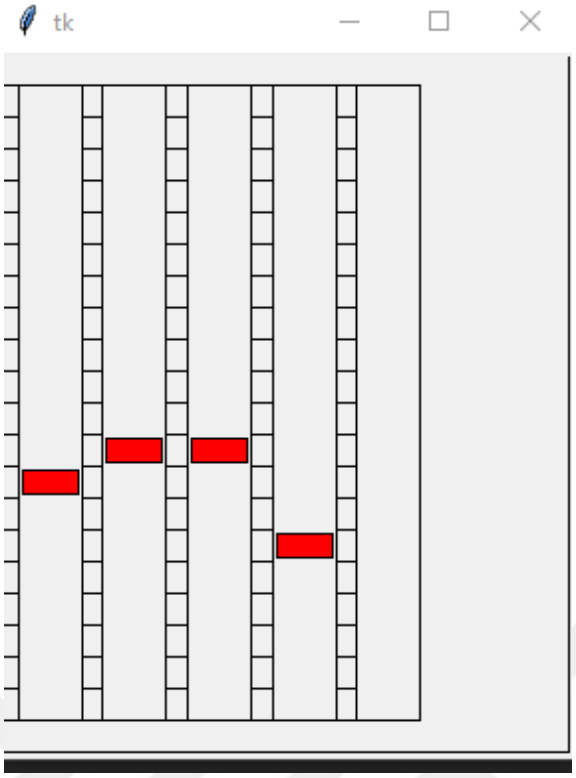

Figure 4.1 GUI from the simulator

The simulation incorporates an EGCS governed by "on-call" traffic control rules, meaning the system responds only to landing calls when they are initiated. Elevator movements are based on the collective control strategy, specifically the full collective (directional collective) approach described by Barney & Al-Sharif (2015). This strategy operates as follows:

Elevators stop to serve both car calls and landing calls in the elevator direction of travel, in floor sequence. When no further calls are registered in the elevator direction ahead of the elevator, the elevator moves to the furthest landing call in the opposite direction if any, reverses its direction of travel and answers the calls in the new direction.

These rules impose the following constraints on the simulation:

- Rule 1: Car calls take precedence over landing calls. This rule guarantees that passengers already in the elevator reach their destinations regardless of hall call demands.
- Rule 2: An elevator does not change its direction of travel if there is a passenger in the elevator. This rule ensures that an elevator serves all the assigned passengers in that direction before reversing its direction.

Although the primary goal is to replicate the real-life systems, certain assumptions

are made in the simulation environment.

- Each passenger can only use the elevator assigned to them.
- Passengers may only exit the elevator at their designated destination floors.
- Elevators do not stop between floors.
- Elevator operations, such as opening and closing doors, cannot be interrupted.
- Passenger actions, such as boarding and exiting, cannot be interrupted.
- Passengers must wait for all exiting passengers to leave the elevator before boarding.

This section outlines the various aspects of the simulation, including the constraints and assumptions of the EGCS, and provides details on passengers, traffic, the building and elevators, as well as state space, action space, and reward function of the environment.

The simulation environment is structured with several classes, including those for elevators, passengers, traffic, and the simulator itself. It requires parameters such as the number of floors in the building, the number of installed elevator shafts, the simulation duration for each episode, the capacity of each elevator, and traffic-related parameters like passenger frequency and traffic patterns.

4.1 Traffic

The Traffic class is responsible for generating passengers in the system. Traffic class generates an arrival time, arrival floor, and destination floor for each passenger. To randomly generate these values, the class requires a user-provided traffic rate and traffic pattern (up-peak, down-peak, or uniform, as detailed below). Additionally, a random seed is used to ensure reproducibility of results while maintaining randomness.

The time interval between passengers arriving at the building typically follows an exponential distribution Barney & Al-Sharif (2015). The formula for the exponential distribution function is as follows:

$$f(t;\lambda) = \begin{cases} \lambda e^{-\lambda t} & t \geq 0, \\ 0 & t < 0, \end{cases} \tag{4.1}$$

where t is the time between passenger arrivals, λ is the rate parameter and defines the intensity of traffic. As λ increases, the traffic gets crowded.

In real-world scenarios, four primary traffic patterns are commonly observed by Barney & Al-Sharif (2015):

- 1-Up-peak Traffic: his pattern involves passengers primarily boarding elevators from the ground floor and traveling upward. It is typically observed in office buildings during morning hours.
- 2-Down-peak Traffic: This pattern refers to passengers boarding elevators from upper floors and traveling to the ground floor. It commonly occurs in the evenings in office buildings.
- 3-Mid-day Traffic: This pattern reflects the movement of passengers either boarding elevators from the ground floor or traveling to the ground floor during mid-day hours
- 4-Random Inter-floor Traffic: This pattern occurs when there is no discernible trend in elevator call requests, with calls distributed randomly between floors.

The simulation focuses on 3 main traffic patterns, excluding mid-day traffic.

- Up-peak Traffic: In this scenario, 90% of passengers enter the building at the main (ground) floor and travel to upper floors, with destinations distributed uniformly. The remaining 10% travel between randomly selected floors.
- Down-peak Traffic: Here, 90% of passengers originate from uniformly distributed higher floors and exit the building via the ground floor using hall call buttons. The remaining 10% travel between randomly selected floors.
- Random Inter-floor Traffic: All passengers have uniformly random arrival and destination floors

4.2 Passengers

Passengers arrive at the building at predefined times according to the traffic class and wait in the building until the assigned elevator arrives at their floor with a matching direction. This period is referred to as the passenger's waiting time. After boarding, the passenger enters the assigned cabin and begins traveling. Once the cabin reaches the passenger's destination floor, the passenger exits. The time from boarding to exiting is known as the service time. Both boarding and leaving operations take 0.8 seconds.

4.3 Elevator Cabins

At the start of the simulation, all elevator cabins are positioned at the ground floor in an idle state with no passengers on board. Each cabin maintains its own set of boarding calls (with directions) and car call memory. Elevators travel at a fixed velocity, taking 2.3 seconds to pass one floor, 3.4 seconds to close the doors, and 2.4 seconds to open the doors. Regarding their doors, elevators have two primary states: open or closed. Within the simulation, five distinct states are defined for an elevator cabin:

- Open for Leaving State: The doors of the elevator are open, and passengers are exiting the cabin at their destination floor.
- Open for Boarding State: The doors of the elevator are open, and passengers are entering the cabin to be transported to their desired floors.
- Closed: The doors of the elevator cabin are closed, indicating that all necessary operations (boarding and departing) at the current floor have been completed.
- Idle: There are no hall or car calls registered for the elevator. It remains at its current floor until a new hall or car call is received.
- Running: The elevator is in motion, traveling to a destination floor. This is the only state when the elevator is between floors.

Throughout the simulation, each cabin transitions between these states according to the rules outlined in Barney & Al-Sharif (2015) and discussed earlier in this section. A state transition diagram illustrating these changes is presented in Figure 4.2. The explanation of the state transition diagram is as follows.

- **I**dle:

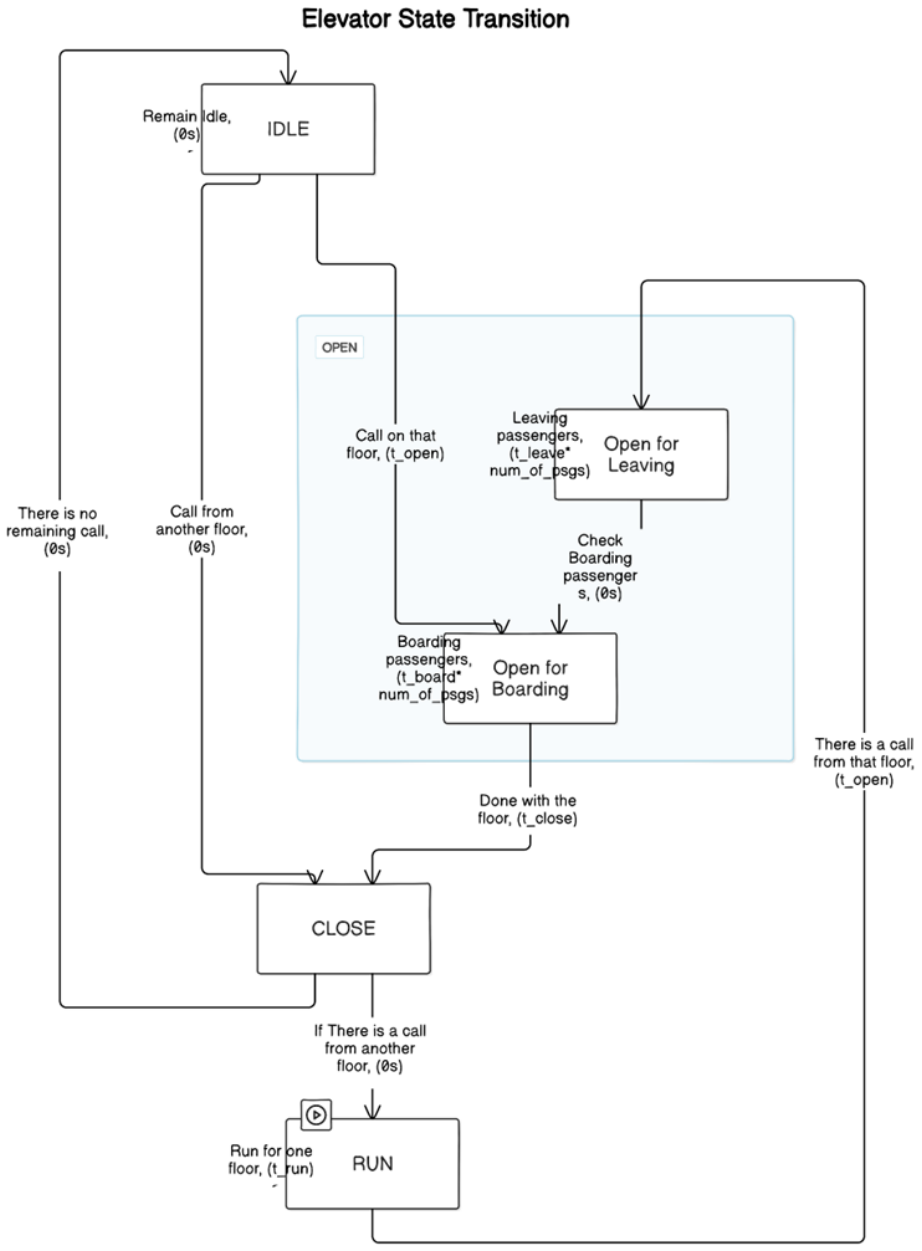

Figure 4.2 State Transition Diagram of the Elevator Cabins

  - The elevator remains idle if there are no remaining calls.
  - If a call is received from another floor, the elevator sets its direction accordingly: "up" for calls from higher floors or "down" for calls from lower floors. It exits the idle state and transitions to the Close state immediately (0 seconds).
  - If a call originates from the current floor, the elevator transitions to the Open for Boarding state and aligns its direction with the passenger's direction in 2.4 seconds.

- **O**pen for Leaving State:
  - Passengers inside the elevator who have reached their destination floors exit, with each passenger taking 0.8 seconds to leave.The elevator remains in this state until all necessary passengers have exited.
  - It then transitions to the Open for Boarding state.

- **O**pen for Boarding State:
  - Passengers at the current floor board the elevator if it is assigned to them, their direction aligns with the elevator's direction, and there is sufficient capacity. Each boarding operation takes 0.8 seconds and repeats until no such passengers remain.
  - The elevator then transitions to the Close state in 3.4 seconds.

- **C**lose:
  - If there are no landing or car calls, the elevator stays idle until a passenger is assigned to it (0 seconds).
  - If there is a call from another floor, the elevator transitions to the Running state immediately (0 seconds).

- **R**unning:
  - At each floor, the elevator checks for car calls or landing calls. If a call is present, the elevator stops and transitions to the Open for Leaving State in 2.4 seconds.
  - If there is no call on the current floor, the elevator continues running, spending 2.3 seconds to travel between floors.

4.4 State Representation (Neural Network Input)

Various approaches can be employed to represent the system to the agent or neural network. In the simulation, the state space captures the status of the elevator system, the building, and incoming passengers at each time step. Key elements of this state space include elevator positions, elevator directions, and details of incoming passengers. It may also include time-related values for passengers both in the building and inside elevators. Different representations have been presented for the Deep Q Network (DQN) algorithm.

4.4.1 Car Call and Hall Call matrices

The state of the environment is represented using two matrices: the car-call matrix and the hall-call matrix. Consider a building with N elevators and T floors.(see figure 4.3 for N=4 and T=10)

- 1- Car-Call Matrix: The car-call matrix captures the state of the elevators and the passengers currently inside them. It is a $T \times N$ matrix where each column represents information about corresponding elevator shaft. One example is a specific floor has been selected as a destination by passengers inside a particular elevator. A value of 1 indicates that the elevator has been requested to stop at the corresponding floor, while a value of 0 indicates no request. This matrix provides crucial information for tracking the internal operations of each elevator and planning future actions.
- 2-Hall-Call Matrix: The hall-call matrix represents the state of passengers waiting in the hallways on each floor. It is a $T \times 2$ matrix, where each row corresponds to a floor, and the two columns represent the "Up" and "Down" hall calls, respectively. A value of 1 in the "Up" column indicates that passengers on that floor are waiting to travel upwards, while a value of 1 in the "Down" column signifies passengers waiting to travel downwards. This matrix enables the system to understand the distribution of waiting passengers and prioritize elevator dispatch accordingly.

Together, these matrices provide a comprehensive representation of the environment, facilitating effective decision-making by reinforcement learning algorithms Wei et al.

(2020).

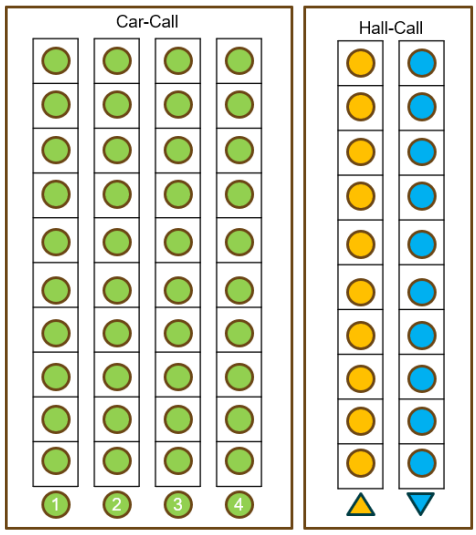

Figure 4.3 Hall Call and Car Call matrices (adapted from Wei et al. (2020))

4.4.2 Elevator State Matrix with Passenger Vectors

The structure of this representation is similar to the Car Call and Hall Call matrices, as explained in Section 4.4.1. However, it emphasizes newly arriving passengers rather than previously assigned passengers. The input comprises one matrix containing elevator-related information and two vectors representing the newly arriving passenger. The Elevator state vector (ESV) includes following components:

- Position: A one-hot encoded vector representing the current position of the elevator.
- Direction: A one-hot encoded vector indicating the elevator's current direction (e.g., up, down, or idle).
- Load: The current load of the elevator, normalized with respect to its maximum capacity.

These components are concatenated to form a single vector. The Elevator State Vectors are then assembled into the rows of an Elevator State Matrix (ESM). For each new passenger, a Passenger Vector (PV) is created, which consists of the following:

- Arrival Floor: A one-hot encoded vector representing the floor where the passenger is waiting.

- Destination Floor: A one-hot encoded vector representing the passenger's intended destination.

The Elevator State Matrix (ESM) and Passenger Vector (PV) together form the input to the Deep Q-Network (DQN). To illustrate this representation, Figure 4.4 provides an example of a building layout, highlighting the spatial relationship between elevators and passengers. Passengers on the third and fourth floors are already assigned, while a newly arriving passenger on the first floor intends to travel to the third floor. The corresponding Elevator State Matrix and Passenger Vector, shown in Figure 4.4, demonstrate how the input data is structured for the Deep Q-Network. These figures together clarify how the model encodes and processes elevator and passenger dynamics, with an emphasis on newly arriving passengers.

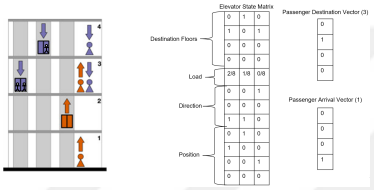

Figure 4.4 An example of a building with 3 shafts and 4 waiting passenger with elevator state matrices and passenger vectors.

4.4.3 Embedded Vector State

One of the challenges with most neural network architectures is that they require a fixed-size input, which necessitates creating and training a new agent whenever the number of floors or elevator shafts changes LeCun, Bengio & Hinton (2015). This limitation hinders scalability and flexibility, particularly in dynamic environments where the building's structure may vary. To address this, an embedding layer can be employed, allowing the model to handle a variable number of floors and shafts by mapping them into continuous, lower-dimensional vector spaces Goodfellow (2016); Mikolov (2013).

Additionally, embeddings resolve issues associated with sparse inputs. Representing shaft and floor numbers as a matrix often results in sparsity, particularly in larger buildings. Embedding these values into dense vectors compresses the information into a more compact and informative representation, reducing sparsity and enhancing the network's efficiency in learning relationships between floors and shafts Hinton & Salakhutdinov (2006). This approach improves the model's generalization across

different environments and mitigates computational challenges posed by sparse input data Sutskever (2014).

The encoding algorithm is used for embedding and converts the state of each elevator into a unique integer representation based on its position, direction, and state. These individual encodings are then combined into a state vector referred to as the Encoded Elevator State. This transformation ensures that the status of each elevator is represented consistently and efficiently. The encoding strategy is outlined in Algorithm (4) and detailed below.

Algorithm 4 Encoding Algorithm for Elevator State

```
Initialize Encoded Elevator State as an empty list
for each elevator in elevators do
    position ← current floor of elevator
    top ← number of floors in the building
    if elevator's direction = None then
        Append top ×0 + position to Encoded Elevator State
    else if elevator's direction = Up then
        Append top ×1 + position to Encoded Elevator State
    else if elevator's direction = Down then
        Append top ×2 + position to Encoded Elevator State
    end if
end for
return Encoded Elevator State
```

The encoding scheme ensures that the state of each elevator is uniquely represented based on its direction and position. For example:

- Idle: Position only .
- Moving Up: Offset by number of floors in building plus position.
- Moving Down: Offset by twice the number of floors plus position.

For instance, consider an elevator in a 10-floor building located on the 8th floor and moving upward. Using the encoding algorithm, this state is represented by the integer 18. This example demonstrates how the algorithm captures and encodes elevator states into a compact and meaningful format.

Similar encoding algorithm also applied to the new coming passenger. The algorithm numerically encodes the states of each new passenger by combining their arrival floor and intended travel direction (up or down). The encoding creates a unique integer

for each passenger state by leveraging the building's total number of floors (`top`) as a multiplier. Passengers traveling upwards are assigned values in the range $[0, top-1]$, while those traveling downwards are assigned values in the range $[top, 2 \times top-1]$. This ensures a distinct representation for each possible passenger state, enabling seamless integration into the decision-making process of the RL algorithm.

Algorithm 5 Encoding Algorithm for Passenger State

```
position ← arrival floor of the new passenger
top ← total number of floors in the building
if passenger's direction = Up then
    Encoded Passenger State ← 0 × top + position
else if passenger's direction = Down then
    Encoded Passenger State ← 1 × top + position
end if
return Encoded Passenger State
```

For example, consider a passenger in a 20-floor building waiting on the 7th floor and registering a down-call. Using the encoding algorithm, this state is represented by the integer 27. This example illustrates how the algorithm uniquely encodes passenger states based on their arrival floor and direction, ensuring consistency and efficiency.

4.5 Reward Function

Similar to state representation, a variety of reward functions can be designed to evaluate actions. Some examples are listed below:

- Cumulative Waiting and Service Time: Measures the total waiting and service time experienced by all passengers in the halls or elevators.
- Max Waiting and Service Time: Evaluates the performance based on the maximum waiting or service time encountered by any passenger, emphasizing fairness and avoiding extreme delays.
- Squared Waiting and Service Time: Penalizes longer delays more heavily by squaring the waiting and service times, encouraging the model to minimize extreme cases.

4.5.1 Cumulative Waiting and Service Time

The reward function is calculated by summing the total waiting time of passengers in the building and the total service time of passengers in the elevators, both weighted by a negative scalar factor. Additionally, it includes positive rewards for the number of passengers who board or leave the elevators. The reward function is defined as:

$$R = -\left(\sum_{i \in \mathcal{H}} w_i + \sum_{j \in \mathcal{C}} s_j\right) + n_{on}\alpha + n_{off}\beta$$

where:

- $\mathcal{H}$ is the set of all passengers in the hall.
- $\mathcal{C}$ is the set of all passengers in the elevator cars.
- w_i represents the waiting time of passenger i in the hall.
- s_j represents the service time of passenger j in the elevator cars.
- n_{on} is the number of passengers who board the elevators.
- n_{off} is the number of passengers who leave the elevators.
- α and β are constants that reward passenger boarding and leaving, respectively.

4.5.2 Max waiting/service time

The reward function is defined as the negative of the maximum waiting or service time among all passengers, either in the building or in the elevators.The reward function is given by:

$$R = -\left(\max_{i \in \mathcal{H}} w_i + \max_{j \in \mathcal{C}} s_j\right)$$

where:

- $\mathcal{H}$ is the set of all passengers in the hall.
- $\mathcal{C}$ is the set of all passengers in the elevator cars.

- w_i represents the waiting time of passenger i in the hall.
- s_j represents the service time of passenger j in the elevator cars.

4.5.3 Squared waiting time

The reward function is calculated by summing the squared waiting times of passengers in the building and the squared service times of passengers in the elevators, both weighted by a negative scalar factor.

$$R = -\left(\sum_{i \in \mathcal{H}} w_i^2 + \sum_{j \in \mathcal{C}} s_j^2\right) + n_{on}\alpha + n_{off}\beta$$

where:

- $\mathcal{H}$ is the set of all passengers in the hall.
- $\mathcal{C}$ is the set of all passengers in the elevator cars.
- w_i represents the waiting time of passenger i in the hall.
- s_j represents the service time of passenger j in the elevator cars.
- n_{on} is the number of passengers who board the elevators.
- n_{off} is the number of passengers who leave the elevators.
- α and β are constants that reward passenger boarding and leaving, respectively.

5. Simulation and Results

In this chapter, the performance of the proposed reinforcement learning RL approaches for the EGCS is evaluated. The primary objective of the experiments is to assess the effectiveness of Double DQN, DQN, and nearest-car algorithms in optimizing elevator scheduling with respect to minimizing average travel times of the passengers. The experiments are performed in a simulated high-rise building environment, detailed in Chapter 4, with varying numbers of floors, elevator shafts, passenger traffic patterns, and different hyper-parameters for DQN and Double DQN. For each building and traffic setting (i.e., traffic rate and pattern), an agent is trained from scratch and evaluated exclusively in that setting. This ensures that the training process is tailored to the specific conditions of each scenario, providing a fair and accurate assessment of the agent's performance.

The DQN and Double DQN agents in this simulation are designed to learn optimal elevator dispatching policies through iterative interactions with the environment. Both approaches share several common characteristics, outlined below:

- Discount factor (γ): 0.99
- Simulation time: 2000 seconds
- Bias toward the ground floor in Downpeak and Uppeak traffic patterns: 0.9
- Replay memory capacity: 1.000.000
- Total number of steps for learning: 300000
- Target network update frequency: every 10000 steps
- Policy network update frequency: every 20 steps
- Batch size for network updates: 64
- Minimum number of samples before the learning starts: 100000
- Learning Rate: 1×10^{-4} , decaying by one-third after each third of the training

- Optimizer: Adam optimizer from PyTorch
- Epsilon (ϵ) in the epsilon-greedy algorithm: A linear decay function that starts at 1 at the beginning of training, reduces to 0.1 after 20% of the training, and remains fixed at 0.1 thereafter.
- Reward Function: As elaborated in the preceding section 4.5 on reward design, this specific function utilizes the cumulative waiting and service times 4.5.1 with n_{on} and n_{off} set to 0. The resulting value is scaled by a factor of 10^{-4} and serves as feedback to the neural networks.
- Target Network Update: The Poljak update method is not applied (until Table 5.5), and the target network is updated by directly copying the policy network parameters ($\tau = 1$).

Two main approaches are applied to the problem:

- 1- Deep Q Network with Elevator State Vector approach
- 2- Double Deep Q Network with Embedded Vector State approach

In all result tables, the rate refers to the number of passengers arriving per minute, and the output represents the average travel time of passengers in seconds. This convention is used consistently throughout the analysis.

5.1 DQN with Elevator State Matrix and Passenger Vectors

The neural network architecture employed is a multi-input model consisting of convolutional, max-pooling, and fully connected layers. For the elevator component of the state, the input layer receives the current environment state in matrix form (including elevator positions and other relevant elevator features). This input is processed through one convolutional layer and one max-pooling layer, both with Rectified Linear Unit (ReLU) activation functions, followed by a flattened and fully connected layer as seen in Figure 5.1.

In contrast, the passenger component of the state is processed through a separate structure consisting of two fully connected layers. After processing both the elevator and passenger states independently, the outputs from these two streams are concatenated, forming a single representation. Finally, this combined output

is passed through an output layer, which represents the Q-values for each possible action.

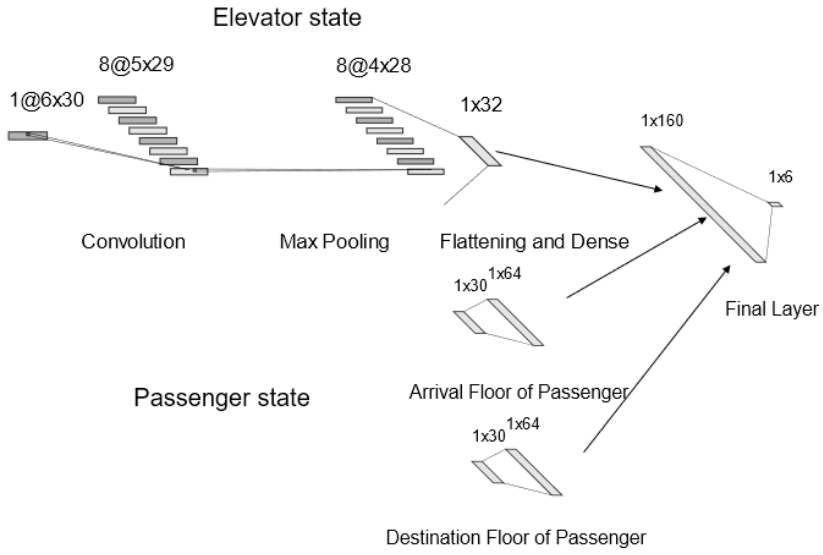

Figure 5.1 Multi Input Convolutional Neural Network Architecture example for 30 floors 6 shaft setting

The initial approach employed a Deep Q-Network (DQN) with Elevator State Matrix and Passenger Vectors as state representation. The results are seen in Tables 5.1 and 5.2. However, the results in Table 5.1 obtained from this approach were suboptimal, with only a few cases performing better than nearest-car baseline. Several factors might have contributed to this under-performance:

- Exploration Stagnation: The DQN may have become stuck in suboptimal policies, limiting its ability to explore the state-action space effectively.
- Hyperparameter Sensitivity: Inadequate tuning of hyperparameters (e.g., learning rate, discount factor, exploration strategy) may have restricted learning efficiency.
- State Representation Limitations: While the state representation was designed to capture relevant features of the elevator control problem, the results obtained using this approach were suboptimal. The performance of the DQN did not meet expectations, suggesting that the chosen representation might inadequately capture the dynamics of the system or that the learning algorithm struggles to generalize effectively in this scenario. These findings highlight the need for further investigation and potential refinement of the state representation or model architecture.
- Reward Design: Designing a reward function for environments without a fixed

time step is particularly challenging, as it requires more intricate adjustments to ensure the reward signal effectively guides the agent toward the desired objectives.

Table 5.1 Performance Comparison Between the Nearest-Car Algorithm and DQN Agent for a building with 20 floors and 4 elevator shafts

Traffic	Rate	Nearest Car	DQN	DQN/Nearest Car (%)
Uniform	1	33.4	**30.2**	90.41
	10	**76.3**	130.6	171.16
	30	**168.2**	220.7	131.21

It is expected that this approach would achieve improved performance in more complex settings. Therefore, more tests are performed for a building with 30 floors and 6 shafts. The results are seen in Table 5.2. It can be observed that the performance was similar even though the task is more complicated.

Table 5.2 Performance Comparison Between the Nearest-Car Algorithm and DQN Agent for a building with 30 floors and 6 elevator shafts

Traffic	Rate	Nearest Car	DQN	DQN/Nearest Car (%)
Uniform	1	51.3	**45.2**	88.10
	10	**85.7**	147.8	172.46
	30	**162.1**	210.5	129.85

The limitations observed with the initial approach may be attributed to the inherent stability challenges of the DQN algorithm and the chosen state representation. These issues likely hinder the model's ability to effectively learn and generalize across varying traffic scenarios, leading to suboptimal performance. Recognizing these shortcomings, an alternative method is proposed to improve both the learning process and overall results. To address these issues, the next subsection introduces a Double DQN method coupled with a revised state space representation.

5.2 Embedded Vector State with Double DQN

Building upon the limitations observed in the initial DQN approach, the second method leverages a Double DQN framework to mitigate the risk of stagnation in

suboptimal policies. Additionally, a state representation based on embeddings as described in 4.4.3 was employed to provide a richer and more nuanced representation of the environment.

The Double DQN model utilizes a fully connected neural network architecture consisting of two hidden layers with 128 and 64 neurons, respectively, each employing the ReLU activation function. The embedding dictionary maps categorical state features into dense vector representations, with the size of each vector set to half the number of floors in the building. These embeddings represent the states of each elevator and passenger, and are concatenated before being passed to the hidden layers. This approach effectively captures underlying relationships in the data, enhancing the expressiveness and representational capacity of the state representation.

The results of this approach, shown in this section, demonstrate significant improvements over the baseline heuristic and the initial DQN model.

The transition to Double DQN with a state representation based on embeddings yielded significantly better results compared to the initial DQN approach. This improvement can be attributed to several factors: The use of Double DQN, which reduces overestimation bias and improves learning stability, and the embedding-based state representation, which captures the environment's complexity more effectively.

Table 5.3 shows a baseline for comparison of the performance of the Double DQN approach against the heuristic baseline (Nearest Car method) across a single traffic pattern. The results indicate that the Double DQN outperforms the baseline in two out of three scenarios. [1]

Table 5.3 Baseline Performance Comparison between Car Algorithm and the Double DQN Agent for a building with 10 floors and 4 elevator shafts

Traffic	Rate	Nearest Car	DDQN	DDQN/Nearest Car (%)
Uniform	7.5	**33.27**	45.55	136.89
Uppeak	7.5	60.94	**42.51**	69.73
Downpeak	7.5	46.85	**44.57**	95.15

To further analyze the performance, experiments were conducted using different traffic rates. While the baseline results were limited to a single traffic rate, this analysis incorporates a broader range of rates as shown in Figure 5.2.

[1] In the tables and figures below, Double DQN is referred to as DDQN for brevity.

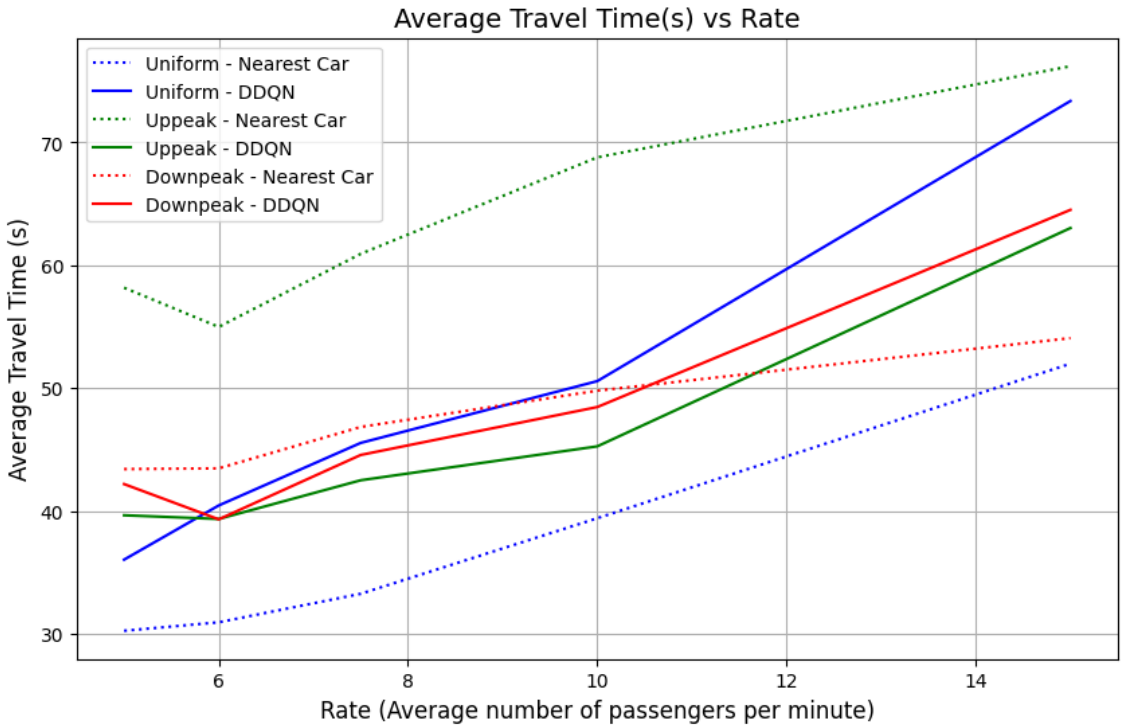

Figure 5.2 DDQN results with changing traffic rates

The findings reveal strong performance by the Double DQN approach in up-peak traffic across all examined traffic rates. For down-peak traffic, the Double DQN performs well up to a rate of approximately 10 passengers per minute, after which its performance begins to decline as traffic rates increase. This decline could potentially be mitigated through further hyperparameter tuning.

In contrast, for the uniform traffic pattern, the Double DQN approach does not perform as well as the nearest-car heuristic. The nearest-car algorithm assigns the same value to an idle elevator and an elevator moving toward a passenger, even if the passenger's direction is opposite to the elevator's current movement. This behavior results in a distribution of idle elevators throughout the building, which is advantageous for uniform traffic scenarios. However, this pattern is not effectively captured by the Double DQN algorithm, limiting its performance in such cases.

For practical applications, up-peak and down-peak traffic patterns require a more strategic approach, as these periods typically correspond to the highest elevator usage in high-rise buildings—such as morning arrivals for work and evening departures. These scenarios are particularly critical, and our approach demonstrates superior performance by effectively managing elevator positions to handle the increased demand efficiently.

Another experimental setting was conducted to replicate the results. As shown in

Table 5.4 a taller and more crowded building is chosen. This scenario involved a 20-floor building equipped with four elevator shafts, providing a more challenging environment to evaluate the performance of the approach.

Table 5.4 Performance Comparison Between Nearest Car Algorithm and the Double DQN Agent for a building with 20 floors and 4 elevator shafts

Traffic	Rate	Nearest Car	DDQN	DDQN/Nearest Car (%)
Uniform	10	**78.61**	104.42	132.84
	15	**114.28**	156.0	136.49
Uppeak	10	106.16	**100.25**	94.45
	15	**115.17**	119.48	103.74
Downpeak	10	**79.85**	95.43	119.51
	15	**103.47**	126.35	122.11

Given the taller building, only crowded traffic scenarios were examined. The results, presented in Figure 5.2, are consistent with the previous experiments. However, excluding the up-peak traffic pattern, it was observed that as traffic becomes more crowded, the performance of the Double DQN approach deteriorates further. This decline is likely attributed to the data-driven nature of RL, where crowded traffic introduces a wider range of scenarios, increasing the complexity of the problem. Addressing this issue would require more training time, additional samples, or extensive exploration (trial-and-error) processes to capture and learn from these diverse scenarios effectively.

To address the observed instability and improve performance, particularly in the uniform traffic scenario, Poljak update method as described in the Section 3.1 is used as an alternative to directly copying the policy network to the target network. This modification aimed to enhance learning stability and provide more consistent updates to the target network, which could potentially yield better results in complex traffic patterns such as uniform traffic.

Table 5.5 Performance Comparison Between Nearest Car Algorithm and the Double DQN Agent ($\tau = 1$), and the Double DQN Agent ($\tau = 0.1$) for a building with 10 floors and 4 elevator shafts

Traffic	Rate	Nearest Car	DDQN (tau=1)	DDQN (tau=0.1)
Uniform	5	**30.26**	36.05	32.41
	7.5	**33.27**	45.55	36.14
	15	**52.0**	73.38	88.9
Uppeak	5	58.18	39.66	**35.76**
	7.5	60.94	42.51	**39.31**
	15	76.21	63.04	**60.12**
Downpeak	5	43.42	42.2	**39.36**
	7.5	46.85	44.57	**43.9**
	15	**54.08**	64.52	62.28

The implementation of the Poljak update method resulted in improved performance compared to the previous configuration. However, the Double DQN approach with this update still did not outperform the nearest-car heuristic in the uniform traffic scenario. This outcome suggests that while the Poljak update enhances stability and consistency in learning, further examinations are necessary to outperform nearest-car algorithm.

To address this, the total number of time steps was significantly increased to 15 million, allowing the agent to explore and learn more effectively from the environment. This adjustment aimed to provide the model with additional opportunities to refine its policy and improve performance in challenging traffic scenarios.

Table 5.6 Performance Comparison Between the Nearest-Car Algorithm and the Double DQN Agent ($\tau = 0.05$) for a 10-Floor Building with Four Elevator Shafts

Traffic	Rate	Nearest Car	DDQN (tau=0.05)	DDQN/Nearest Car (%)
Uniform	3.75	**25.43**	26.95	105.97
	5	30.26	**29.49**	97.45
	7.5	**33.27**	36.72	109.02
Uppeak	3.75	54.2	**34.84**	64.28
	5	58.18	**37.40**	64.28
	7.5	60.94	**46.4**	76.14
Downpeak	3.75	41.00	**35.93**	87.63
	5	43.42	**37.49**	86.34
	7.5	46.85	**44.65**	95.30

With the increased number of time steps, the performance of the Double DQN agent improved significantly. It outperformed the nearest-car heuristic in almost every scenario. In the rare cases where it did not surpass the nearest-car algorithm, its performance was only slightly worse (e.g., an average travel time ratio of 1 vs. 1.05). These results demonstrate that extending the training duration allowed the agent to better adapt to the environment, achieving a level of performance that closely aligns with or exceeds the baseline in most cases.

To further illustrate the learning process, Figures 5.3, 5.4, 5.5, and 5.6 depict the agent's reward and average travel time over episodes, respectively. These graphs highlight the convergence behavior of the agent during training. The reward graph demonstrates the improvement in the agent's decision-making as it learns to optimize elevator dispatching. Similarly, the travel time graph shows a decreasing trend, indicating the agent's ability to minimize passenger travel times as training progresses. Together, these figures provide additional insight into the effectiveness and stability of the Double DQN approach over extended training periods.

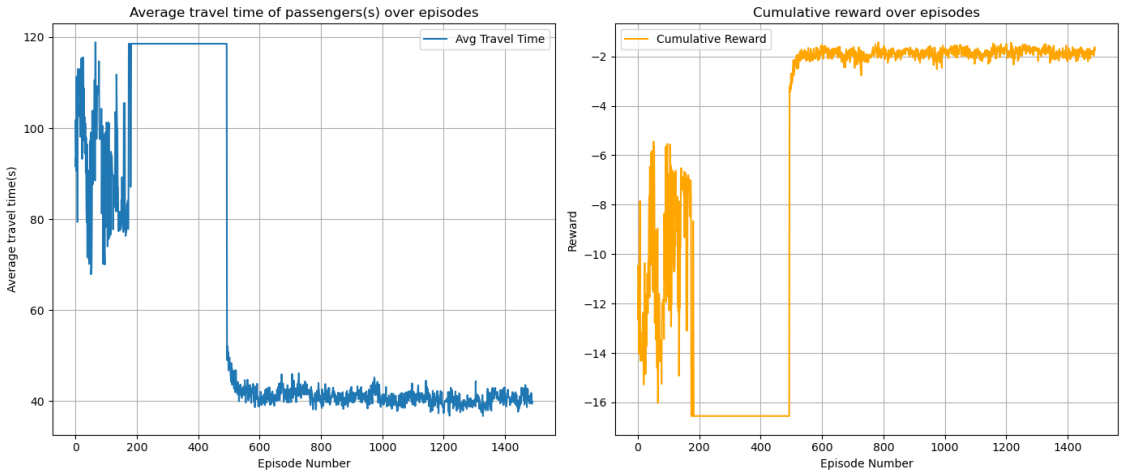

Figure 5.3 Average Travel Time and Reward vs. Episodes for Uniform Traffic Pattern (Rate = 7.5 Passengers/Minute)

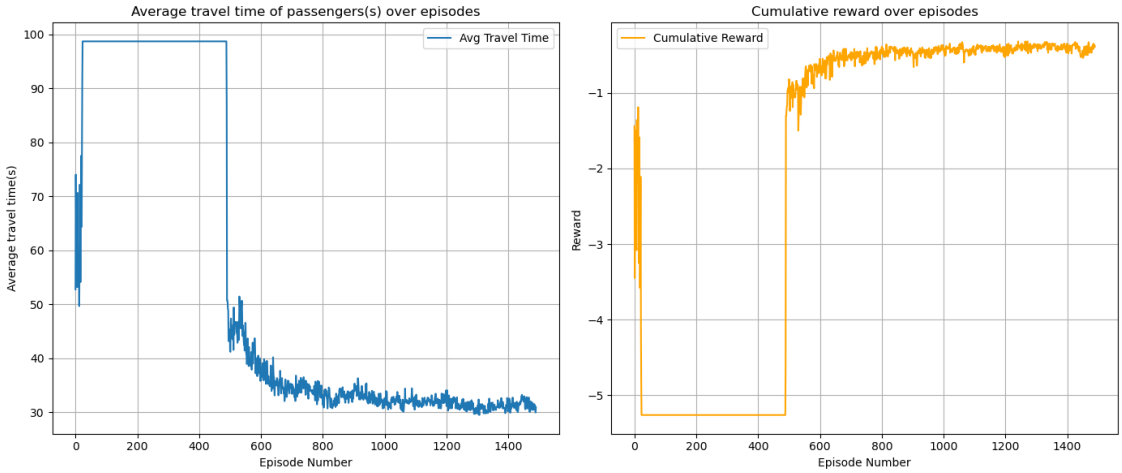

Figure 5.4 Average Travel Time and Reward vs. Episodes for Uniform Traffic Pattern (Rate = 5 Passengers/Minute)

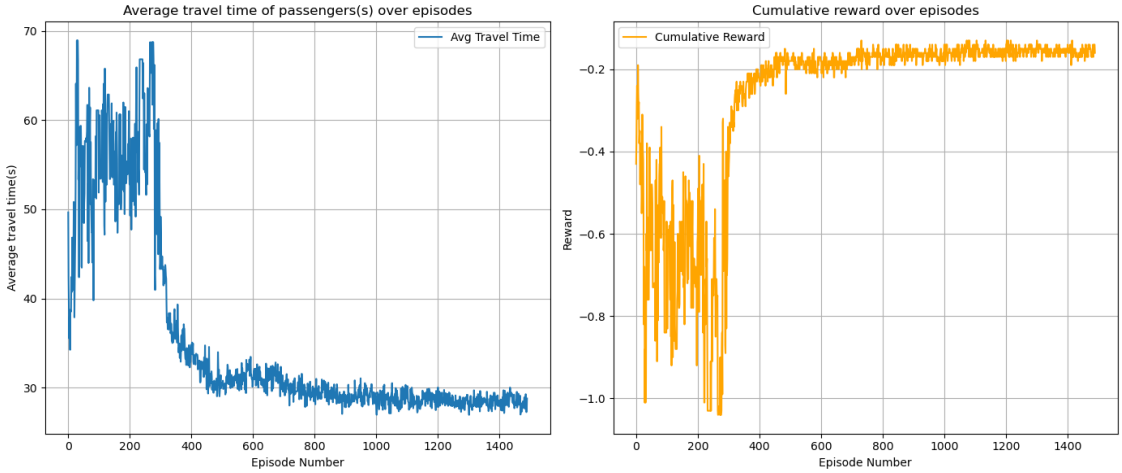

Figure 5.5 Average Travel Time and Reward vs. Episodes for Uniform Traffic Pattern (Rate = 3.75 Passengers/Minute)

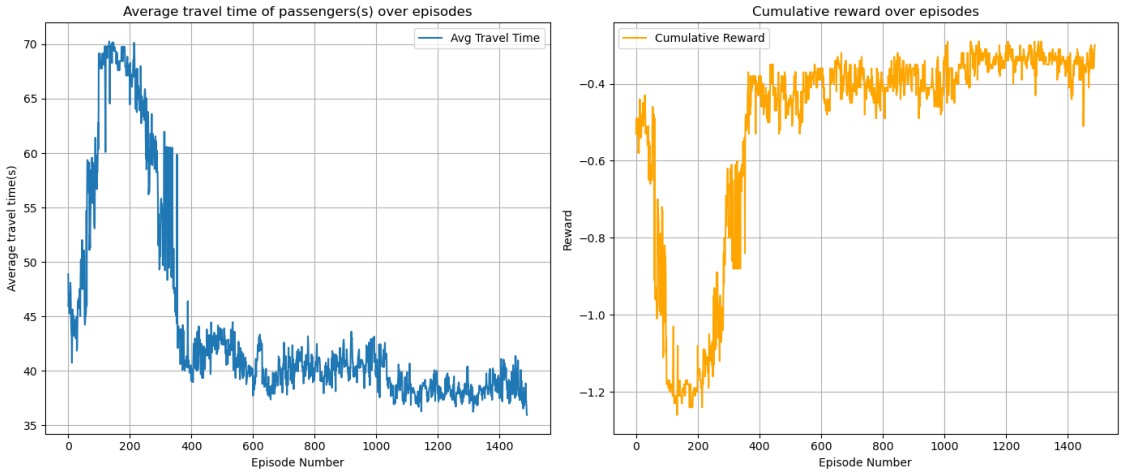

Figure 5.6 Average Travel Time and Reward vs. Episodes for Uppeak Traffic Pattern (Rate = 7.5 Passengers/Minute)

It is noteworthy that the Figures5.3, 5.4, 5.5, and 5.6 exhibit symmetry with respect to the x-axis. This symmetry highlights the direct relationship between the convergence of average travel time and the cumulative reward during training. The alignment of these metrics indicates that the reward function is well-designed and aligns effectively with the goal of minimizing average travel time. This behavior demonstrates the reinforcement learning agent's ability to optimize its policy over episodes while maintaining consistency in performance.

While each agent was trained from scratch and evaluated exclusively for a specific building and traffic setting, further analysis revealed that an agent trained on one scenario also performed reasonably well in other scenarios. For example, as shown in Table 5.7 the agent trained for a traffic rate of 5 passengers per minute with a uniform traffic pattern demonstrated competitive performance when applied to other traffic settings. In some cases, it outperforms the agent specifically trained for that setting. Although the performance was not always better in all cases, it remained competitive, indicating that the agent's learned strategy is adaptable to varying traffic patterns and building configurations.

This observation highlights the potential of reinforcement learning approaches to develop policies that are not only tailored to specific scenarios but also robust enough to handle diverse conditions, further underscoring the versatility of the proposed methods.

Table 5.7 Performance Comparison Between the Nearest-Car Algorithm and a Selected Trained Double DQN Agent ($\tau = 0.05$) for a 10-Floor Building with Four Elevator Shafts

Traffic	Rate	Nearest Car	DDQN (tau=0.05)	DDQN/Nearest Car (%)
Uniform	3.75	**25.43**	27.66	108.76
	5	30.26	**29.49**	97.45
	7.5	**33.27**	36.76	110.48
Uppeak	3.75	54.22	**35.94**	66.28
	5	58.18	**34.67**	59.59
	7.5	60.94	**40.63**	66.67
Downpeak	3.75	41.00	**37.76**	92.09
	5	43.42	**38.19**	87.95
	7.5	46.85	**36.76**	78.46

In summary, the results demonstrate that the proposed reinforcement learning approach, particularly the Double DQN with the Poljak update and extended training time, significantly improves elevator scheduling performance. By increasing the total time steps to 15 million, the agent was able to consistently outperform the nearest-car heuristic in nearly all scenarios, with only minor deviations in a few cases. These findings highlight the potential of reinforcement learning algorithms to handle complex elevator group control problems more effectively than traditional heuristics.

The improvements observed in this study underscore the importance of fine-tuning hyperparameters, such as training duration, to fully leverage the capabilities of reinforcement learning. However, while the current results are promising, further exploration of alternative update methods, state space representations, reward functions, and traffic scenarios could provide additional insights and lead to even greater advancements in performance.

5.2.1 Boosting Double DQN Using Nearest Car Algorithm

This section introduces a hybrid approach called "Boosting Double DQN Using the Nearest-Car Algorithm" to enhance the performance and convergence of the reinforcement learning agent. Unlike purely RL-based methods, this approach leverages the nearest-car heuristic during the exploration phase of the Double DQN algorithm.

In this hybrid strategy, the nearest-car algorithm is incorporated to guide the agent's exploration by providing heuristic-based action suggestions. This integration ensures that the agent can explore high-quality action spaces more effectively, especially in the early stages of training, where random exploration may lead to suboptimal or inefficient learning trajectories. By combining the strengths of heuristic decision-making with reinforcement learning, the method aims to strike a balance between leveraging domain knowledge and allowing the agent to learn from experience.

While this approach aims to improve performance and stability, it is important to note that it is not a purely RL-based method, as it relies partially on predefined rules from the nearest-car algorithm. This distinction is crucial for understanding the contributions and limitations of the hybrid approach in comparison to purely data-driven methods.

Table 5.8 Performance Comparison Between the Nearest-Car Algorithm and the Hybrid Approach for a 10-Floor Building with Four Elevator Shafts

Traffic	Rate	Nearest Car	DDQN	DDQN/Nearest Car (%)
Uniform	7.5	**33.27**	35.16	105.68
	5	**30.26**	30.29	100.09
Uppeak	7.5	60.94	**36.01**	59.09
	5	58.18	**34.19**	58.76
Downpeak	7.5	**46.85**	48.5	103.52
	5	43.42	**35.73**	82.28

The results of the hybrid approach can be, "Boosting Double DQN Using the Nearest-Car Algorithm," highlight both its potential and its limitations. In this implementation, the exploration process relied entirely on the nearest-car heuristic, which constrained the agent's ability to fully explore the action space independently. Consequently, the performance of this method did not show significant improvements compared to the standalone Double DQN approach.

Table 5.8 summarizes the performance metrics of this hybrid method across various traffic patterns and intensities. The results indicate that while the hybrid approach achieves comparable outcomes in some scenarios, it does not consistently outperform either the nearest-car heuristic.

These results suggest that a balanced exploration strategy—combining heuristic guidance from the nearest-car algorithm with standard exploration techniques—might yield better outcomes. This hybrid method was introduced primarily to present an alternative approach to integrating domain knowledge into reinforce-

ment learning frameworks, highlighting the importance of balanced exploration in achieving effective learning and performance.

Future investigations could explore a more balanced exploration strategy, where the nearest-car heuristic and traditional exploration methods are used in tandem, allowing the agent to benefit from both structured guidance and autonomous learning.

6. Conclusion and Future Work

6.1 Conclusion

This thesis presented an in-depth study on the application of reinforcement learning (RL) to the Elevator Group Control EGC problem. Traditional control methods such as rule-based or heuristic algorithms, while simple and effective in low-traffic scenarios, struggle to scale efficiently in dynamic environments with high passenger demand. By utilizing advanced RL techniques, particularly Deep Q-Networks DQN and Double DQN, it is demonstrated that an agent can be applied to elevator dispatching policies through interaction with the environment, adapting to changing traffic patterns in real-time.

A novel approach incorporating embeddings into the Double DQN framework was introduced to address the limitations of state representation and improve learning efficiency. This enhanced method not only stabilized training but also delivered promising results, outperforming the nearest-car heuristic in most scenarios. Even in cases where it did not outperform the heuristic, the performance was comparable, indicating the potential of RL to handle the complexity of multi-elevator group control effectively.

The RL-based methods were computationally intensive, and the time required for training the models was a limiting factor in real-world deployment. However, the potential of RL to handle the complexity of multi-elevator group control in dynamic environments was demonstrated, suggesting that with further refinements, RL approaches could play a key role in the future of intelligent building management systems.

6.2 Future Work

Several potential directions for future work can be pursued to build on the research presented in this thesis. One promising avenue is the incorporation of real-time learning mechanisms, allowing the RL agent to continue learning and adapting after deployment by utilizing real-world passenger data to refine its policies continuously. This would enable the system to respond effectively to seasonal variations, daily traffic patterns, and unforeseen events like maintenance or power outages.

Further improvements in training efficiency could be realized by exploring alternative RL algorithms, such as Proximal Policy Optimization (PPO) Schulman, Wolski, Dhariwal, Radford & Klimov (2017) or Actor-Critic methodsKonda & Tsitsiklis (1999), which may offer faster convergence and lower computational requirements compared to DQN. Additionally, integrating metaheuristic approaches or hybrid models that combine traditional optimization techniques with RL could enhance system performance in highly dynamic environments.

Similar approaches can be applied to Multi-Car Elevator systems Liew, Lim, Tan & Tan (2015) to further reduce average travel time by utilizing multiple elevators within a single shaft. Additionally, certain constraints could be relaxed to assess the applicability of RL in related problem domains.

BIBLIOGRAPHY

Barney, G. & Al-Sharif, L. (2015). *Elevator traffic handbook: theory and practice*. Routledge.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. arxiv. *arXiv preprint arXiv:1606.01540*, *10*.

Crites, R. H. & Barto, A. G. (1998). Elevator group control using multiple reinforcement learning agents. *Machine learning*, *33*, 235–262.

Goodfellow, I. (2016). Deep learning.

Hinton, G. E. & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *science*, *313*(5786), 504–507.

Konda, V. & Tsitsiklis, J. (1999). Actor-critic algorithms. *Advances in neural information processing systems*, *12*.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444.

Li, Z. (2010). Pso-based real-time scheduling for elevator group supervisory control system. *Intelligent Automation & Soft Computing*, *16*(1), 111–121.

Li, Z. & He, K. (2023). A review of research on the development of elevator group control technology. *Engineering Advances*.

Liew, Y. C., Lim, C. S., Tan, M. L. P., & Tan, C. W. (2015). A review of multi-car elevator system. *Jurnal Teknologi*, *73*(6).

Mikolov, T. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, *3781*.

Mnih, V. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529–533.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., & Garnett, R. (Eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Polyak, B. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, *4*(5), 1–17.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Sutskever, I. (2014). Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*.

Sutton, R. S. & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, *30*(1).

Watkins, C. J. & Dayan, P. (1992). Q-learning. *Machine learning*, *8*, 279–292.

Wei, Q., Wang, L., Liu, Y., & Polycarpou, M. M. (2020). Optimal elevator group control via deep asynchronous actor–critic learning. *IEEE transactions on neural networks and learning systems*, *31*(12), 5245–5256.