UNIVERSITY OF TWENTE.

Master Thesis

Optimising early design decisions for SBS/RS warehouses through a Deep Q-Learning model integrated with simulation software at Vanderlande

by

Jan van Eldik

Industrial Engineering and Management
Specialization Production and Logistics Management
Orientation Supply Chain and Transportation Management
Faculty of Behavioural, Management and Social Sciences

Examination committee
Dr. B. Alves Beirigo
Dr. L. Xie
University of Twente

External supervision
Job van der Loo
Vanderlande

2025

Abstract

This thesis investigates the development of a Deep Q-Learning model to support early design decisions for Shuttle-Based Storage and Retrieval systems for warehousing systems at Vanderlande. The model aims to automate the configuration process using simulation feedback on throughput and costs to create an agent that is able to determine the optimal design for an ADAPTO warehouse. A DQL agent was trained to interact with the simulation software, which serves as the environment of the agent. By proposing configurations and adjusting the policy based on the feedback, moving towards an optimal policy. The model did not yield a policy consistently providing optimal configurations and converging reliably. Nevertheless, the research provides key insights into the reward structure, exploration strategy and DQL variations. Mainly, the agents benefit from a simple reward structure that punishes states which do not reach the goal and rewards states that reach the goal based on cost-effectiveness. Extending the replay memory from uniform random sampling to Prioritised Experience Replay did not improve performance. While Double DQN resulted in equal performance, additional tuning of hyperparameters specifically for the Double DQN could improve performance. In conclusion, the current modelling approach could not generalise across a complex reward environment with a shifting throughput goal. Nevertheless, it provides a valuable foundation for future work, offering guidance on hyperparameter tuning and architectural extensions to improve convergence and scalability.

Management Summary

This management summary provides an overview of the research conducted in this MSc thesis, which focuses on the development of a Deep Q-Learning model to solve early design choices in Shuttle Based Storage and Retrieval System (SBS/RS) warehouses for Vanderlande, using simulation based performance evaluation.

Objective

The primary objective of this research is to develop a Deep Q-Learning model which can accurately determine early design choices in SBS/RS warehouses based on throughput performance of the warehouse and the affiliated costs. The model is developed for the ADAPTO system at Vanderlande.

Key Findings

1. **Current Operational Challenges:** Vanderlande's simulation department experiences high demand, often forcing engineers to decline simpler simulation requests. Automating such basic design tasks would significantly enhance operational efficiency.

2. **Available models:** An extensive literature review shows the lack of Machine Learning solutions within the early design choices of SBS/RS warehouses compared to the general trend in ML research and applications, a clear gap in the literature for research.

Proposed Solution

The research proposes a Deep Q-Learning model to determine the optimal configurations for the ADAPTO systems. The model consists of an agent which learns through interacting with the environment; the environment in this research is the simulation software. The agent proposes a configuration based on ADAPTO variables and receives feedback through the throughput of the configuration and the associated costs. Based on the feedback, the agent adjusts its policy to approach an optimal design. Though training the model is computationally intensive, the agent can quickly propose configurations once trained. Potentially generating viable solutions within an hour for new requests.

Results

Unfortunately, the model cannot yet deliver on the expected performance and cannot efficiently determine optimal configurations for the ADAPTO system. Nevertheless, this research has contributed in the progress towards such a model. The following conclusions were drawn from this research:

- **Reward Structure:** A simple and direct reward structure proved most effective. Configurations not meeting throughput targets were penalised, while successful configurations were rewarded based on cost-efficiency.
- $\epsilon - Greedy$ **strategy:** Uniform random sampling from past experiences yielded better results than prioritised sampling.
- **Double DQN:** While DDQN aims to reduce overestimation, initial experiments showed no substantial performance improvement without further tuning.

Challenges

The results show there is development for the creation of a model that can accurately determine the configurations of ADAPTO warehouses for every client. To advance towards a deployable tool, the following challenges must be adressed:

- **Run time:** The extensive run time of the simulations resulted in slow and limited experimentation. More extensive research is possible by speeding up this process, potentially through cloud-based computing or computing clusters.
- **Additional hyperparameter tuning:** An application of additional experimentation is longer runs for experiments to give the agent more time to learn from the environment. Additionally, more hyperparameter settings can be tested and cross-validation can take place.
- **More complex models:** Additional DQL variants—including further exploration of DDQN and potentially Dueling DQN or distributional approaches—should be evaluated for fit and effectiveness.

Conclusion

Overall, the dynamic nature of client-specific throughput requirements remains a key difficulty for the DQL model to identify an optimal policy for the ADAPTO system effectively. These findings form a valuable starting point for future development toward a robust, automated design tool for ADAPTO warehouse systems

Preface

Dear reader,

Writing this preface marks the end of my thesis and my study at the University of Twente. At the start of my bachelor I was unsure whether this study was the right fit for me, but after completing both the bachelor and master degree, I can wholeheartedly conclude that Industrial Engineering & Management has been the right fit for me. The past several months, I have had the privilege of conducting my research at Vanderlande, guiding me towards the start of my professional life.

I would first like to thank the simulation department at Vanderlande for making me feel so welcome in the department and enabling me to conduct my research there. I would like to especially thank Job van der Loo as my personal supervisor, who has always provided me with excellent feedback and expertise.

Next, I would like to thank Breno Alves Beirigo for the guidance during my thesis as my first supervisor from the University of Twente. The meetings shaped my research in the right way and always gave me great insights on how to continue.

Lastly, I would like to thank my friends and family for the continuous support everyone has provided me during this process. It has been a very welcome distraction and has reenergised me to continue the research.

Jan van Eldik
's Hertogenbosch
May 20, 2025

Contents

Acronyms

ALNS Adaptive Large Neighbourhood Search. 23, 25

AS/RS Automatic Storage and Retrieval System. 9, 13, 15, 17–19, 23, 24

CRISP-ML Cross-Industry Standard Process for Machine Learning. 4, 5, 31

DC Double Cycles. 9, 17, 18, 22

DDQN Double Deep Q-Network. iv, x, 47, 60, 61, 64, 66, 67

DP Dynamic Programming. 28

DQL Deep Q-learning. iv, 24, 32, 33, 36–40, 43, 47, 61, 63, 64, 66–68

DQN Deep Q Network. x, 29, 39, 46, 47, 60, 61, 64, 66, 67

FIFO First In, First Out. 24

GA Genetic Algorithm. 20, 23, 24

KPI Key Performance Indicator. xi, 3, 6, 9, 12, 16–18

ML Machine Learning. 4, 20, 22, 24–26, 29, 31, 36, 39, 63, 69

OELDSW Optimisation of Early Layout Design of SBS/RS Warehouses. 29

PER Prioritised Experience Replay. 47, 59

ReLU Rectified Linear Unit. 37, 38

RL Reinforcement Learning. 24–28, 35, 39, 44, 67

SBS/RS Shuttle Based Storage and Retrieval System. iii, vii, 1, 2, 6, 19–26, 29

SGD Stochastic Gradient Descent. 38

SKU Stock Keeping Unit. 9–11, 13, 14, 36

SPT Short Processing Time. 24

TSU Transport and Storage Unit. 9–11, 13–15, 17–21, 33, 35, 36

Glossary

α Variable that determines the range of the positive reward when reaching the goal. ii, ix, xi, 29, 44, 45, 53–55, 65, 66

α^* Optimal value for α identified through experimentation. xi, 44, 53

β Variable that determines the negative reward when not reaching the goal. ii, ix, xi, 44, 45, 53–55, 61, 65

β^* Optimal value for β identified through experimentation. 55

γ Discount factor used in the Bellman equation Equation (4.17) to discount future reward. ix, xi, 44, 45, 54, 55

ADAPTO The ADAPTO system is the SBS/RS product provided by Vanderlande. ix, xi, xv, 2–6, 9–13, 15–19, 25, 28, 29, 31–35, 40, 61, 63, 66, 68

List of Figures

List of Tables

Listings

List of Algorithms

Chapter 1

Introduction

This chapter provides an overview of the research context, introduces Vanderlande, and outlines the structure of this MSc thesis. Section 1.1 establishes the relevance of warehouses in modern supply chains and introduces Vanderlande, a market leader in process automation in the warehousing, parcel and airport sector. Next, Section 1.2 analyses the current process and reduces the main problem to a single core problem. Section 1.4 provides the main research question (RQ1) and the sub-research questions (RQ2- RQ7) that structure the research. Section 1.5 emphasizes this research's real-world application and importance. This research's clear scope and limitations are covered in Section 1.6, ensuring a well-defined research focus. Finally, the structure of the thesis is shown in Section 1.7, giving an overview of the chapters and their respective content, including the context, the literature study, model development and model performance.

1.1 Background and Context

In recent times, e-commerce has moved warehouses to focus on the distribution of goods rather than solely on storage. The storage system must be able to handle a large throughput of orders containing few products (Miguel et al., 2020). A higher throughput in a warehouse results in more storage and retrieval actions, thus more movement. A significant development in the warehousing sector to increase efficiency was the introduction of a Shuttle Based Storage and Retrieval System (SBS/RS). The picking of products accounts for an estimated 55% of the total warehousing costs. By optimising this process, high operational costs and unsatisfactory service are averted. An SBS/RS automates and optimises the picking process in warehouses (De Kosten and Roodbergen, 2007). The warehousing sector has since been optimising the SBS/RS towards an optimal system.

Vanderlande is a market-leading, global partner for logistic process automation in the warehousing, airports and parcel sectors. Vanderlande was established in 1949 and acquired by Toyota Industries Corporation (Vanderlande, 2025). Through the effective collaboration with sister companies like Bastian Solutions and Viastore, a globally reliable solution is presented to the target markets. The airport solution offered by Vanderlande is the market-leading baggage handling system along with related passenger solutions, capable of moving over 4 billion pieces of baggage around the world per year. The Vanderlande baggage handling system is active at more than 600 airports, including 12 of the world's largest airports. Vanderlande's process automation

solutions for the parcel market are installed for the world's leading parcel-handling companies, moving over 52 million parcels every day. The warehousing solution is the ADAPTO system, which is an SBS/RS, used by 9 out of the 15 largest global food retailers. ADAPTO allows a tailored SBS/RS to serve the customers' wishes in their throughput requirements.

1.2 Problem Statement

This research focuses on the ADAPTO system of Vanderlande Industries. The ADAPTO system is a highly adaptable Shuttle Based Storage and Retrieval System (SBS/RS) that Vanderlande produces as a part of the full warehousing solution. As part of the sales process of an ADAPTO system, the simulation engineering team at Vanderlande simulates different configurations of the ADAPTO system until a configuration is reached that satisfies the requirements of the customer while delivering a competitive bid. Due to the high demand for ADAPTO systems and other Vanderlande products, the simulation engineering department is very busy. It therefore has difficulty delivering on every simulation request from customers or other Vanderlande departments. The simulation demand exceeds the capacity of the department. The simulation of stand-alone ADAPTO models is a time-intensive task with many manual steps and waiting time. To identify the core problem of the capacity issue, a problem cluster has been created in Figure 1.1.

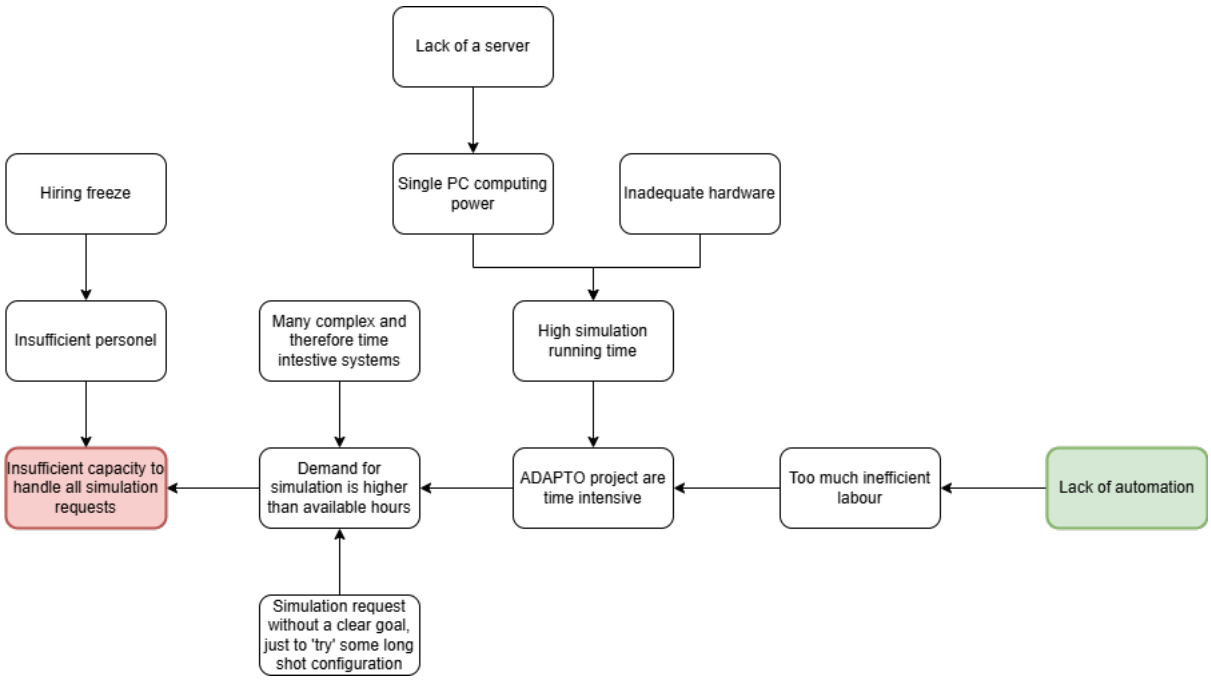


Figure 1.1: The problem cluster where the red box shows the main problem and the green box shows the core problem tackled in this research

The flowchart shows the main problem, shown in the red box, at the simulation engineering department and the core problem in the green box. The following sections cover each section of the problem cluster.

- The simulation department having insufficient personnel causes the department to be unable to handle every simulation request. An initial simple solution is to

increase the department's capacity by hiring more simulation engineers. Nevertheless, an expensive solution which only remedies the issue, while the problem can be permanently solved by handling the workload more efficiently.

Thus, the root of the problem is identified by analysing the department's workload. Three ways can be found to balance the requests for simulations and the capacity to perform these simulations.

- First, ADAPTO is a highly flexible system allowing numerous variations. An advantage for the customer, but an increased complexity for the simulation engineering department, leading to large and long-running projects. Reducing the complexity of systems can reduce the workload for the department. However, where this benefits this department, it is unacceptable for Vanderlande as it reduces the company's competitiveness.

- Secondly, not every simulation request from sales has a clear goal. Scenarios arise where requests originate from stabs in the dark with configurations that hardly result in useful configurations. Hence, better alignment between the sales and simulation engineering departments might benefit the workload. However, this is not the focus of this research.

- The third option to reduce the workload for the department is to reduce the time required per simulation request. The run time of a model is relatively high, especially for large systems. Decreasing this run time could be achieved by looking at cloud computing options.

The core problem faced by the systems simulation department is the excessive manual labour in designing ADAPTO systems. A detailed description of the simulation process is provided in Chapter 2, but a brief overview follows in this section to drill down to the core problem. The process starts with a simulation engineer configuring a first set of parameters for the ADAPTO system based on their experience and expertise. These parameters are the early design decisions for an ADAPTO and entail aspects like the number of aisles, number of levels, number of shuttles, etc. The set of parameters is the input for the simulation software, which returns a set of Key Performance Indicator (KPI) with throughput being the primary metric. Based on these KPI's, the parameters are adjusted based on general guidelines and the expertise mentioned above. This process is repeated until a configuration is reached that satisfies the demand of the customer. This approach is both time-consuming and labour-intensive, creating an opportunity to optimise the process and reduce the time spent on ADAPTO projects. The primary inefficiency arises from the repetitiveness of the process for a simulation engineer inputting ADAPTO configurations and waiting for the simulation results. The iterative process presents a clear opportunity for automation using an optimisation model. This allows simulation engineers to spend their time more efficiently and take on more projects. This research focuses on creating a model that can accurately manage the simulation process and will reduce the time required to simulate an ADAPTO model.

1.3 Problem Approach

The lack of automation significantly impacts the simulation engineering department at Vanderlande and requires a structured approach to developing an automation and optimisation model. The model must be able to grasp the link between the requirements from the customer and the capabilities of the ADAPTO system. An initial look at the literature shows the area of modelling that allows for this optimisation and automation issue. Heuristics, Machine Learning (ML) or a combination of both, is currently presented in the literature as a viable option for optimisation issues in the warehousing sector. An in-depth analysis of the available models and methodologies is provided in Chapter 3.

To ensure the effective development and implementation of the automation and optimization model, the CRISP-ML methodology, developed by MLOps, has been selected. Cross-Industry Standard Process for Machine Learning (CRISP-ML) offers a structured framework for guiding ML development, as proposed by Visengeriyeva et al. (2025). The first step in this methodology is business and data understanding. This step identifies the scope of the model, the success criteria and the data quality verification to ensure the project's feasibility. This step creates the framework in which the model development takes place. This step applies to both ML and Heuristics approaches, as setting a clear goal and understanding the system is crucial in developing the model. The literature is consulted within this framework to find which model type best suits the problem. When the framework is clear, data engineering is the next step in the methodology.

Correct and complete data is essential for a ML model and a heuristic (Gudivada et al., 2017; Han et al., 2014). This step for ML entails data selection, cleaning, feature engineering, and standardisation. The data quality must be high for the ML to uncover all relevant patterns and learn effectively. In the case of heuristics, although data quality is essential, domain expertise and established rules of thumb also play a critical role in model development. According to Han et al. (2014), clear and well-structured data are necessary for building an efficient heuristic model.

With a well-defined framework and prepared data, the model can be constructed. The selected model type from the literature is adapted to the specifics of the problem and tailored using the prepared data. For ML models, the engineering contains the model specialisation and model training tasks. As heuristics do not contain any training steps, this step selects the correct algorithm to use and implements the logic from the rules of thumb and expert opinion. After the model has been developed, it must be tested and evaluated. The business and data understanding has created both modelling and business goals, against which the model is tested. This can be done through test data sets and other validation techniques identified in the literature. The validation step shows the model functions as intended according to the model and business goals.

The fifth step of the CRISP-ML methodology is deployment, which is the final step included in this thesis. Vanderlande requires a practical solution to enhance its current simulation process, making a deployment guide essential for smooth model implementation and a guide on using the model. The last step of the CRISP-ML methodology is monitoring and maintenance. While the guide from deployment covers this step partially, monitoring and maintenance primarily fall under Vanderlande's responsibility and are therefore limited to a deployment guide and future recommendations.

1.4 Research Questions

To guide this research, the following research questions are formulated:

1.4.1 Main Research Question

RQ1: What optimization model can accurately predict optimal ADAPTO warehouse configurations during the early design phase for custom order patterns and varying throughput requirements?

1.4.2 Sub-Research Questions

Each chapter of this thesis corresponds to a sub-research question addressing specific aspects of the main research question. These sub-research questions are as follows:

1. **Chapter 2: System & Data description**
    - **RQ2:** How is the current design process of an ADAPTO system structured and integrated with the simulation software?
2. **Chapter 3: Literature**
    - **RQ3:** Which modelling technique is most suitable for automating and optimising the design of ADAPTO systems?
3. **Chapter 4: Solution approach**
    - **RQ4:** How should the optimisation model be configured to achieve optimal performance?
4. **Chapter 5: Experimental setup**
    - **RQ5:** What experiments must be performed to identify the optimal hyperparameter settings?
5. **Chapter 6: Experimental results**
    - **RQ6:** How does the proposed solution perform when applied to real-world cases from Vanderlande, and how does it compare to relevant benchmarks?
6. **Chapter 7: Conclusion, Discussion & Recommendations**

- **RQ7:** How can the proposed model be integrated into Vanderlande's infrastructure to ensure practical applicability and what aspects of the model can be improved?

1.5 Significance of the Study

The successful automation and optimisation of the design process of an ADAPTO warehouse at Vanderlande allows the simulation department to deliver optimised designs for more standard requests and therefore focus their expertise on more complex issues. This will enable the department and Vanderlande to maximise efficiency and provide an optimal design to all clients, enhancing customer satisfaction. This solidifies Vanderlande's position as a market leader in the warehousing sector. In addition, this research contributes to the broader field of applying modelling techniques in the process automation sector to add value to the supply chain.

1.6 Scope and Limitations

As the sales process of an ADAPTO system involves many departments, it's essential to define the scope of the research clearly. The automation and optimisation of the configurations of the ADAPTO system is performed at the simulation engineering department. The study focuses on the variables available for variation by the simulation engineering department. Chapter 2 gives a detailed description of all parameters, variables and KPI's available for the department and thus, the research is limited to these variables. In addition, due to the confidentiality of Vanderlande's simulation model, the model is treated as a black box model. A general description of the model's functioning is given in Chapter 2, while the exact functioning of the simulation model remains undisclosed. The research focuses on automating and optimising the configuration process of , the simulation model is merely a tool to obtain the KPI's of each configuration.

1.7 Thesis Structure

This thesis is structured as follows:

- **Chapter 2: System & Data description** - In this chapter, the current simulation process and environment are described in detail.
- **Chapter 3: Literature study** - This chapter provides an in-depth review of the existing literature related to optimisation methods for SBS/RS warehouses.
- **Chapter 4:Solution approach** - This chapter presents the model developed to address the problem and all relevant hyperparameters.
- **Chapter 5: Experimental Evaluation** - In this chapter, the performance of the model is discussed and compared to real-life cases from Vanderlande and compared to a benchmark obtained from literature

- **Chapter 6: Discussion and Implementation** - This chapter discusses the implications and provides practical implementation of the model into the systems of Vanderlande.
- **Chapter 7: Conclusion** - The final chapter summarizes the key findings, contributions, and future research directions.

1.8 Summary

This chapter introduces the context, problem statement, research objectives, and research questions of this MSc thesis. Optimising early design choices for warehouse layouts represents the challenge of optimising systems for individual customers while maintaining a standardised product. Subsequent chapters will delve into the details of this problem and the proposed solution.

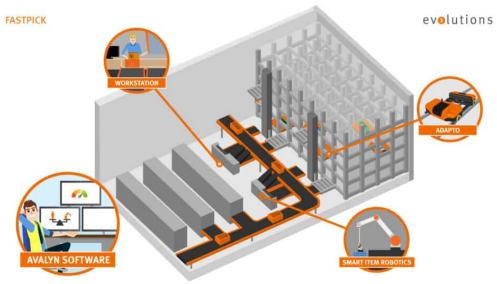
Chapter 2

System & Data Description

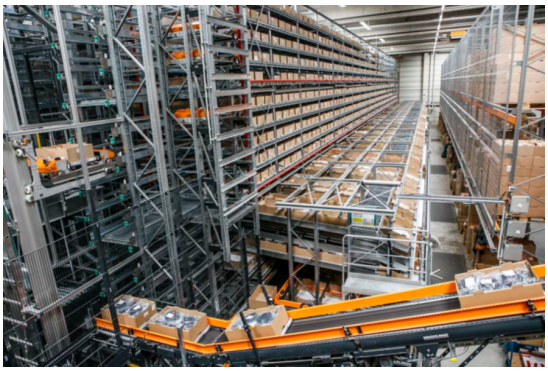
Vanderlande delivers various warehousing solutions for small warehouses and huge distribution centres. A clear overview of the current system is required to optimise the design process of a warehousing system. This section gives an in-depth description of the ADAPTO system and the design process of the ADAPTO system. First, an overview of the design process is given, followed by a detailed description of the ADAPTO system, including the parameters, variables and KPI's. These sections answer the first research question:

2.1 ADAPTO System

The ADAPTO system is the Automatic Storage and Retrieval System (AS/RS) of Vanderlande, adaptable for usage by both small warehouses and huge distribution centres. Vanderlande delivers the ADAPTO system, which is tailored to the customer's requirements. The ADAPTO system has a basic functionality with adaptable features, such as the number of lifts, the type of lift, the number of shuttles, etc. The ADAPTO system uses Transport and Storage Unit (TSU) to store Stock Keeping Unit (SKU) in the system. A TSU is a tub that contains an SKU; this allows the ADAPTO system to handle all SKU's the same, improving the efficiency of the system. A single task of storing a TSU and retrieving a TSU is called a Double Cycles (DC). But first, the basic functionality is described in this section and the adaptable features are discussed in Section 2.2. Figure 2.1a shows an overview of a warehouse containing an ADAPTO system. The figure shows the shuttles on each level and the lifts transporting the TSU's to the next section. Figure 2.1b shows an implementation of an ADAPTO system where the bottom shows an input or output belt, the left shows a lift and a shuttle.



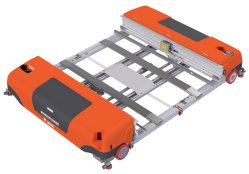
(a) General overview of a warehouse with an ADAPTO system.



(b) Real-life example of an ADAPTO system.

Figure 2.1: Example of ADAPTO system
source: Vanderlande

The ADAPTO system functions fully autonomously and can integrate seamlessly with any Warehouse Management Systems present. The system utilises shuttles to store and retrieve the TSUs. A shuttle is a robotic transporter that moves through the warehouses to store and retrieve TSU's. The shuttles are battery-powered and charge when interacting with the lift. An additional charging tower is required when aisles are very long and only charging when interacting with the lift is insufficient. However, this is not often needed.
The shuttle has two variations, it is either equipped with two belts to move the TSU in and out of the racks. Or the shuttle is equipped with two telescopic hooks to retrieve or store a TSU, called the piranha. The two shuttle variations are shown in Figure 2.2a and Figure 2.2b. The two belts are faster but limited to single deep systems, whereas the Piranha is equipped for double deep storage systems. Double deep storage systems allow 2 SKUs to be stored in 1 location, improving storage capacity but reducing throughput capacities. These shuttles can move in multiple aisles on a level due to a cross-aisle presence. However, the shuttles are level-captive as they cannot move vertically to different levels.



(a) The twinbelt shuttle

(b) The piranha shuttle

Figure 2.2: The two shuttle types present in ADAPTO systems
source: Vanderlande

The vertical transportation of TSU 's is done by lifts at the end of the aisles. The

lifts are always a combination of an inbound and an outbound lift. Multiple configurations are available for the shuttles, lifts, and cross-aisles. The different variations of the ADAPTO system are discussed in the following sections of this chapter. A large advantage of the ADAPTO system is its modular design and, therefore, huge scalability of the system. The process of storing and retrieving the TSU's is given in the flowchart in Figure 2.3. The process starts when a SKU arrives in the ADAPTO system in a TSU. The routing towards the storage location is calculated, after which the TSU is transported to the correct level and then moved to the correct rack. The TSU awaits in the ADAPTO until a retrieval request is received. Again, the shuttle and lift movement is calculated, after which the product leaves the ADAPTO system. This figure does not go in depth into the routing and storage algorithms as this is not within the scope of the research.

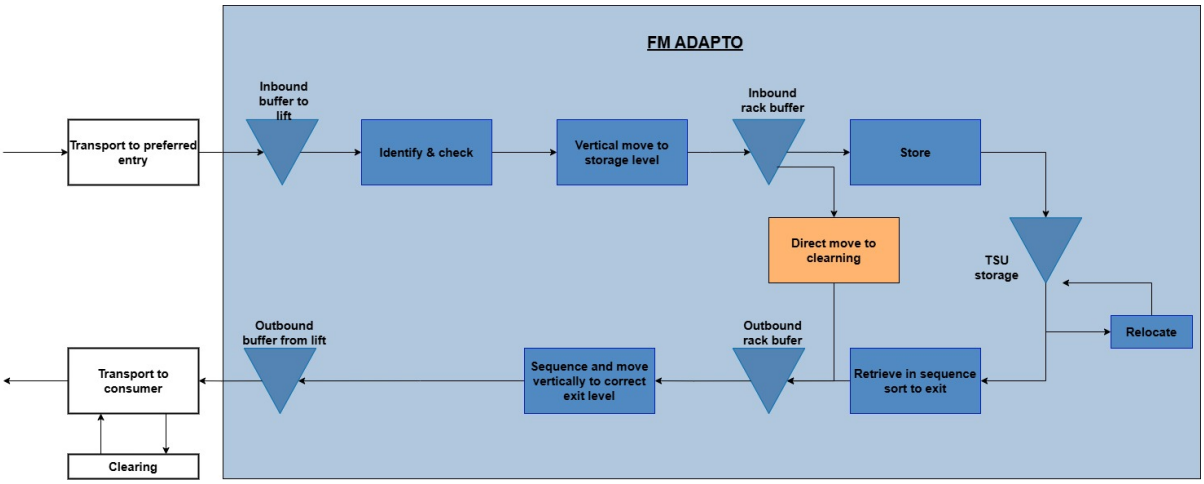


Figure 2.3: Flowchart of the storage and retrieval operations in an ADAPTO warehouse.
source:Vanderlande

2.2 ADAPTO System Design

The steps taken in the design process of an ADAPTO warehouse must be clear to understand the requirements for the model presented in this research. Figure 2.4 shows an overview of the design process within a flowchart on a very high level. The process starts with the customer wanting an automated warehouse and considers Vanderlande's ADAPTO as one of the options. The process enters the system's simulation department whenever the sales engineers determine the requirements and obtain the required data, such as sizing and order profiles. In some cases, initial designs have already been made and especially the warehouse sizing has already been explored. Thus, the system simulation team starts with a basis from which a full design of the warehousing system is created.

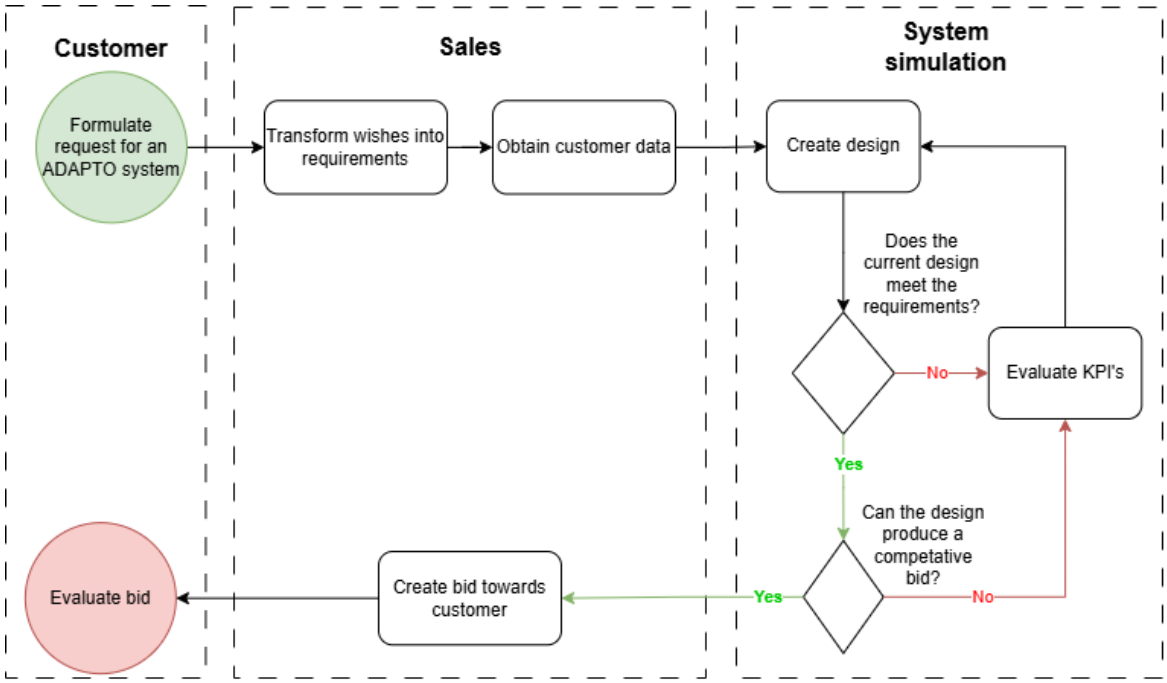


Figure 2.4: Flowchart of the design process of an ADAPTO system. The process starts at the green circle and moves through the different departments until it arrives at the stopping point in the red circle.

Vanderlande has a simulation tool that allows for the team to simulate the performance of the warehouse while adapting the design of the warehouse. The parameters are discussed in more detail in Section 2.3. Through a process of adapting these parameters based on the results of the simulation, a configuration is reached where the throughput is achieved, which was set in the requirements. The other KPI's from the simulation model are used to grasp where the most additional throughput can be achieved. For instance, a very high lift utilisation shows that additional throughput can be obtained by adding more lifts. These KPI's and their impact are discussed in detail in Section 2.5. Together with guidelines based on the KPI's, the simulation engineers use their expertise to determine these configurations. The process stops when the throughput has been reached and the simulation engineer and the sales engineer are satisfied with the result and are confident that the configuration leads to a competitive bid to the customer. Nevertheless, there are too many options to try every configuration. Thus, it is never sure whether the obtained configuration is optimal.

2.3 ADAPTO System Parameters

To deliver a tailored solution to the warehousing problem, specific parameters must be known to the systems simulation. This section will note these parameters and explain what they entail and what variations are available. An overview of the parameters can be found in Table 2.1.

2.3.1 Facility Specifications

The first aspects that must be clear are the specifications of the customer's facility where the AS/RS will be installed. This determines the maximum surface area of the AS/RS and the maximum height of the racks. However, the warehouse does not solely consist of AS/RS but is always combined with other warehousing activities such as palletisers or picking stations. There are two possible scenarios with regard to the space available. The facility in which the AS/RS must be built can either be a brownfield, where a warehouse already exists or a greenfield, which is a piece of land where the building is not yet built. The second option gives more freedom in the size of the AS/RS. However, in most cases the sales department has already determined the available space for the AS/RS.

2.3.2 Order Profile

The order patterns of the customer are essential to the AS/RS specifications. These parameters generate the orders within the simulation system. An accurate set of parameters significantly impacts the validity of the simulation results. Multiple aspects of the order pattern serve as input for the ADAPTO model: order size, the sequencing and the average number of TSU retrievals per order.

Order Size

The order size is the number of different TSU's per order, so customers like web shops mostly have a low number of TSU's per order, as the order consists of just one or two products. On the other hand, customers like supermarkets have a larger number of SKU per order.

Picking Seqeuence

Linked to the number of TSU's per order is the picking sequence of these TSU's. There are three options for the picking sequence. The strict policy ensures that each order is completed before a new order is picked and that the TSU's in each order are picked in a specific order. The relaxed sequences ensure that each order is completed before the next is picked, but the sequence of TSU's inside an order does not matter, also known as the pick order. Lastly is the unsequenced policy, where the TSU's of different orders are picked without sequence. The only important aspect is that the TSU's picked belong to orders that end up on the same pallet, also known as batch picking.

Velocity Classes

The order pattern is not only linked to how large orders are and in what order they must be retrieved, but also how the velocity classes are distributed. There are three velocity classes into which all TSU's are divided. This division is made according to the Activity Based Classification (ABC) as described by Fu and Gao (2024). This paper describes how the products can be divided into three classes based on the velocity of the products. The model requires the percentage of products and orders they represent. The three velocity classes each have their own section in the warehouse to improve throughput. Figure 2.5 shows locations of the TSU's in the different classes, where

the bottom shows the cross aisles and the vertical rectangles represent the aisles. The fast movers are situated at the beginning of each aisle, the medium speed movers are situated in the middle and the slow movers are at the back of each aisle.

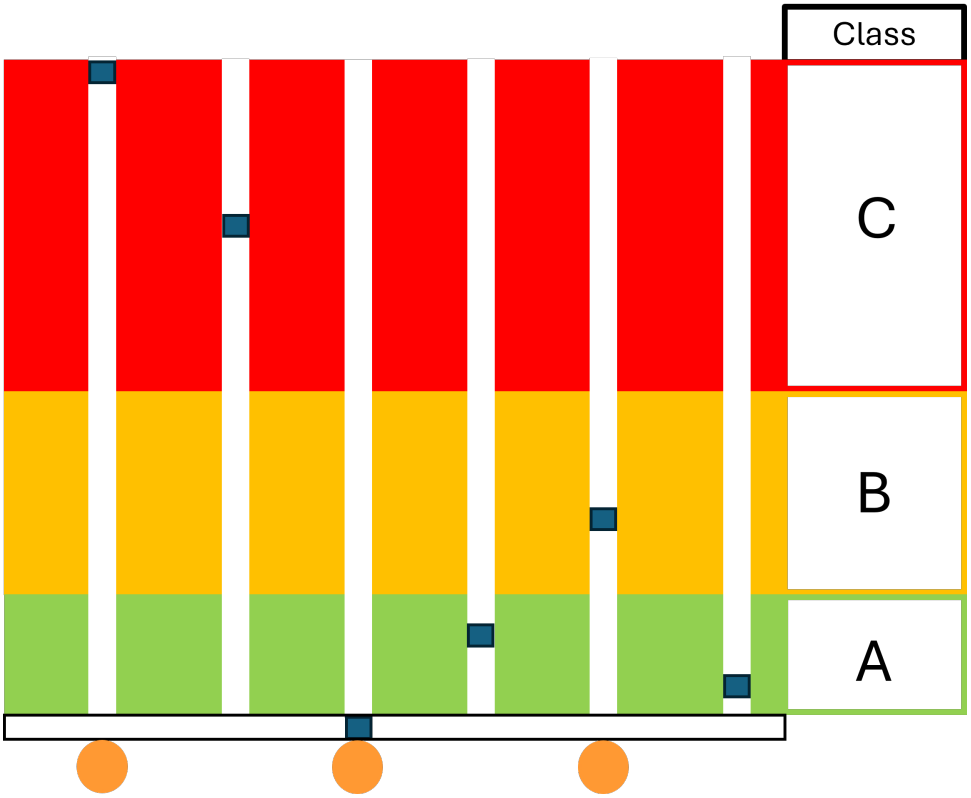


Figure 2.5: Top view of the different storage locations of the different velocity classes in the warehouse. The orange circles represent the lifts that serve as I/O points. *source:Vanderlande*

Number of Locations

The final essential aspect of the SKUs is the number of slots in the warehouse occupied by a SKU. A SKU can have one or multiple locations in a warehouse. A shuttle will move to the closest available location where the SKU is stored. The more locations a SKU has, the more retrieval options are present, resulting in faster storage or retrieval. The simulation software requires a distribution of the number of locations per TSU. The distribution is adaptable to reach the desired throughput, where more locations per TSU result in a higher throughput. However, more locations per TSU result in higher requirements for the size of the warehouse.

2.3.3 Smart Lift Allocation

Smart lift allocation determines whether a TSU must travel to a specific picking station and use a specified lift. It is important to note that every TSU travels to the same lift within an order. The situation mentioned is for a specific TSU that must always be delivered to a specific lift. Whenever this is not required, a shuttle can deliver the TSU to a lift which delivers the TSU to a picking station the fastest. Thus, disabling

smart lift allocation results in a decrease of flexibility of the shuttles and therefore less throughput. However, this option is hardly ever required by the customer and cases where smart lift allocation is required are often special cases.

Parameter	reference	Unit
Velocity classes	Section 2.3.2	distribution
Nr of TSU locations	Section 2.3.2	distribution
TSU 's per group	Section 2.3.2	distribution
Order sequence	Section 2.3.2	
Smart lift allocation	Section 2.3.3	Yes or No

Table 2.1: Overview of the parameters

2.4 ADAPTO Variables

The following section describes the variables that can be changed in the warehouse design in this research. The extent to which these variables can be altered and the impact the alteration makes is discussed in this section. An overview of all the variables mentioned in this section can be found in Table 2.2.

2.4.1 Topology

The first set of variables regards the basic layout of the warehouse. The number of aisles, the number of levels per aisle and the number of storage locations per level are determined. The values of these parameters are limited by either the minimum and maximum size of the ADAPTO system or the maximum size of the client's location. In addition, the depth of the location is determined, which can either be single deep or double deep. Single deep warehouses focus on high throughput, whereas double deep systems concentrate on storage.

2.4.2 Lifts

The lifts are an essential part of the AS/RS as the shuttles can solely move horizontally throughout the system. Multiple variables determine the efficiency of these lifts. First is the height at which the lifts are connected to the conveyor belts that further transport the TSU's in the warehouse or where new TSU's arrive for storage. The lift type can be a single or a double platform lift. The lifts can both deliver the TSU's to the same conveyor and therefore have to interact with each other to ensure efficient lift usage. When the single conveyor point limits the system's throughput, the system allows for a vertical merge. The vertical merge gives each platform in the lift a conveyor to deliver or retrieve TSU's too. These conveyors are then merged at a later point on the conveyor. While the vertical merge is an option in theory, the practice shows that a vertical merge is hardly ever proposed to customers. Experts at Vanderlande have therefore advised to exclude this option from the optimisation. Figure 2.6 schematically illustrates this vertical merge.
Each lift allows for buffer positions at each level of the warehouse; the system allows for two or three buffer positions. The last variable related to the lifts is the number of

lifts and their location. The lifts are situated behind the cross aisle at the start of an aisle. Hence, the maximum number of lifts is equal to the number of aisles. Standard practice is to alternate lifts over the aisle, so the number of lifts is equal to half the number of aisles.

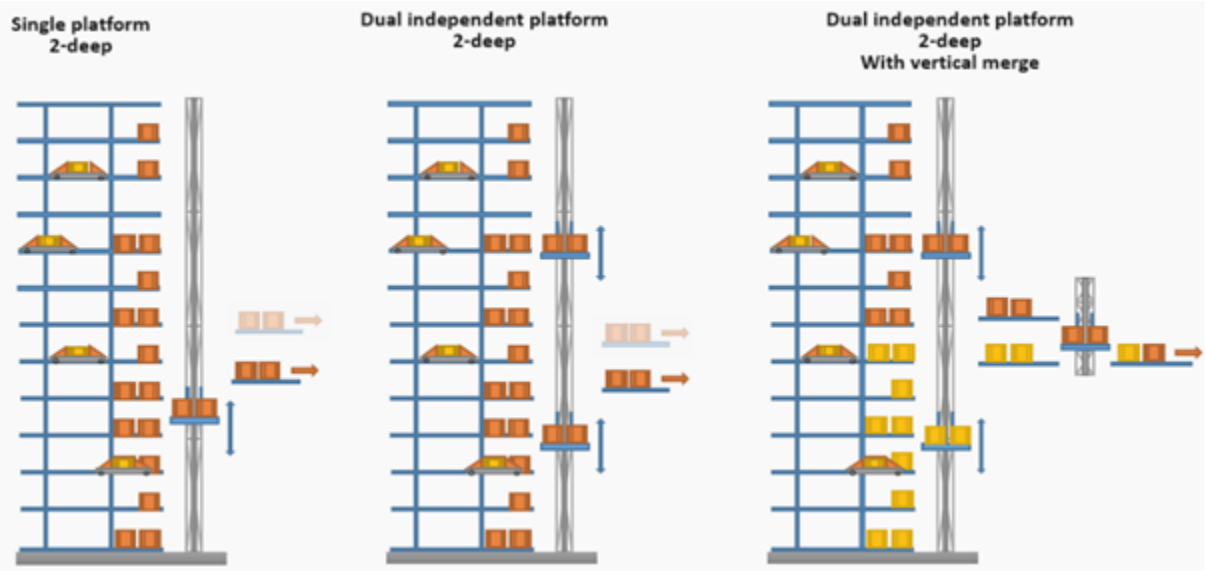


Figure 2.6: Overview of the different lift options available.
source:Vanderlande

Variable	reference	Unit
Nr of Aisles	Section 2.4.1	Integer
Levels	Section 2.4.1	Integer
X-positions	Section 2.4.1	integer
Location depth	Section 2.4.1	1 or 2
Nr of lifts	Section 2.4.2	integer
Type of lift	Section 2.4.2	single or double
Vertical merge	Section 2.4.2	Yes or no
In-rack buffer position	Section 2.4.2	2 or 3

Table 2.2: Overview of the variables available in the ADAPTO system

2.5 KPI's

The following section will highlight the various KPI's that are presented by the ADAPTO simulation software that show how the configuration of parameters performs. The KPI's are essential in the design process as the KPI's show where bottlenecks or inefficiencies are within the configured system and, therefore which settings could potentially improve the design. The bottleneck analysis is performed after each run to evaluate the performance of each part of the system. Understanding the impact of the KPI's is vital to understanding the design process. This section shows the most essential KPI's and how they impact the model. An overview of all the KPI's can be found in Table 2.3.

2.5.1 Throughput

The throughput of the system is the most important KPI for the design of the AS/RS and it is measured in the number of DC s per hour. The throughput is the combination of the number of stores and retrievals by the system. As the system aims to combine a drop off of a TSU at the lift with a storage of a TSU, the number of stores and retrievals is balanced in the throughput. The customer determines a minimum throughput, which must be met with the system design.

2.5.2 System Performance

There are seven KPI's related to the performance of the system, which are shortly discussed regarding their meaning and usage:

Average Shuttle Cycle: This KPI measures the number of double cycles per hour per shuttle. A low number shows there are too many shuttles in the system and they are often idle. However, increasing the shuttles doesn't necessarily increase the throughput. At some point, more shuttles are in each other's way and drop throughput. Simulating multiple numbers of shuttles shows the sweet spot for the shuttles. The maximum number of shuttles is determined by the number of lifts as there cannot be more shuttles than lifts due to the charging requirements of the system. The number of aisles also represents a restriction as there cannot be more shuttles per level than the number of aisles minus 1. The last shuttle constraint is Wi-Fi. Due to Wi-Fi limitations, the maximum number of the whole system is 250 shuttles.

Average Cross-Aisle Throughput: The Average Cross-Aisle throughput shows the utilisation of the cross aisle by the shuttles. This indicates whether the system is halted by the cross-aisle usage. A low average cross-aisle usage indicates the number of shuttles per level might need to be decreased as the shuttles hinder each other on the cross-aisle. More lifts might be required to shorten cross-aisle movements. This KPI is measured in the number of double cycles per hour.

Lift Utilisation: This KPI shows the fraction of time the lift is moving or interacting with a TSU in a buffer. However, this KPI is discussed in more detail in the lift cycle KPI.

Relocates: The final KPI of this section shows the percentage of TSU's which are outbound and are relocated in the warehouse. Within a double deep ADAPTO system, to reach a TSU located behind another TSU, the first TSU must be relocated to reach the second TSU. This KPI is measured as the percentage of the total number of outbounds.

Shuttle Cycle: The Shuttle Cycle section provides insight into the actions of the shuttles. The KPI is split into eight different sections that each show how much time was spent on that action. The various activities are: idle, cross-aisles, aisles, P&D at rack-buffer, wait for rack-buffer, wait for aisle, wait for cross-aisle and extra charge time. The waiting times especially show the bottlenecks in the system.

Lift Cycle: Similar to the Shuttle Cycle, the lifts' activities are also registered but in three categories: Busy, Interaction, and Idle. Where interaction is the time spent interacting with the buffers to place or retrieve a TSU, the other two speak for themselves. This KPI shows whether to use another lift type as the lift is too busy and a better option must be selected or the chosen option is too good and an easier and cheaper option.

Average Lift Throughput: The number of TSU's that pass through each lift per hour. This KPI shows the average number of TSU 's move through the lift. The simulation team does not use this KPI as the picking stations are not their concern.

Operator Efficiency: The operator efficiency shows the percentage of time the operator is working versus idle. This variable is similar to the Average Lift Throughput. Again, this is not considered in the simulation aspect of the design of the AS/RS, as this falls outside the scope of the simulation study.

KPI	Unit
Throughput	DC/h
Average Shuttle Cycle	DC/h
Average Cross-Aisle Throughput	DC/h
Lift Utilisation	%
Relocates	%
Shuttle Cycle	s
Lift Cycle	s
Average Lift Throughput	DC/h
Operator Utilisation	%

Table 2.3: Overview of the KPI's of the system performance

2.6 Conclusion

This section gives an overview of the current design process of an ADAPTO system. It describes a simulation engineer's process when the customer requirements are in. In addition, the simulation software for the ADAPTO system is crucial in this process. For an overview of all the parameters, variables and KPI's, Table 2.1, Table 2.2 and Table 2.3 can be consulted. This chapter gives an overview of the different design options for the ADAPTO system and shows how to evaluate the performance of the system using KPI's.

Chapter 3

Literature Study

The identification of the problem revealed the lack of an automated optimisation model for the ADAPTO system. To identify a suitable model to fill this gap, a literature study is conducted on the current developments on SBS/RS to identify potential optimisation strategies. The goal is to explore and map the optimisation models currently applied in SBS/RS systems, providing a foundation for developing an optimization model for this research. Finally, the most relevant optimisation models for the scope of this research are studied in more detail.

3.1 Definitions

Before moving to the in-depth analysis of the different warehousing systems, it is vital to clearly define both an AS/RS and an SBS/RS. An Automatic Storage and Retrieval System (AS/RS) is a major category of material-handling equipment that automatically stores and retrieves goods without manual labor (Roshan et al., 2018; Rajkovic et al., 2019). An AS/RS is widely used in manufacturing facilities, distribution centres and warehouses (Li et al., 2022). A SBS/RS is a type of AS/RS where the transportation of the TSU's is done by shuttles at a level and a lift at the I/O point does the vertical transportation. This difference in movement is the main distinction between an AS/RS and a SBS/RS (Lehrer et al., 2017). An important distinction to be made within the SBS/RS is the ability for shuttles to move between the different levels. If shuttles are confined to a single level, the system is considered tier-captive (Eder, 2020). If shuttles move between different levels, the system is considered tier-to-tier. The same goes for aisles; the system is considered aisle-captive if shuttles are confined to a single aisle.

3.2 Warehouse Layout Optimisation

The selection of the ideal warehouse layout is a special optimisation process with a huge or infinite number of layout alternatives. Therefore, it is not a typical mathematical optimisation, as evaluating all possible alternatives is impossible (Kovács, 2021). The design of the ADAPTO system matches these criteria for the warehouse layout design. Furthermore, an important characteristic of a warehouse layout design is the inability to define certain relationships within mathematical formulas. The throughput calculations provided by the simulation software for the ADAPTO systems cannot be defined with a constraint of a Mixed Integer Problem (MIP). Kovács (2021) presents

a step-by-step process that is widely used in warehouse layout optimisation. The nine steps guide the design process from setting goals and initial data collection to creating viable designs and optimising these towards a final and optimal design. Optimisation is performed using a model consisting of three different phases. The first phase creates the initial design of the warehouse that adheres to the limitations and constraints. The second step is to make the set of alternatives finite by eliminating configurations through heuristics. The last step is to evaluate the design in the finite set based on the objective function to select the optimal design. Diaz et al. (2024) takes a different approach by treating the warehouse layout optimisation as a bin packing problem. The optimisation is performed with a Variable Neighbourhood search using hill climb operators. Lyu et al. (2021) proposes the combination of a mathematical model and a dynamic analysis to solve a Systemic Layout Planning (SLP) to reduce the carbon emissions of a warehouse. Hu and Chuang (2023) combines a SLP with a Genetic Algorithm (GA) to solve the non-linear optimisation model. Mayadunne (2024) applies a two-step model to solve a warehouse layout problem; two different models for the two steps are solved sequentially. The first model generates an optimal configuration compared to the requirements for the warehouse and the second model fits the desired layout into the available space. The paper acknowledges the risk of the solution space becoming too large, resulting in the inapplicability of this model in this situation. The Warehouse Layout Optimisation is currently solved through heuristics and GA's, the application of ML in this optimisation problem has not yet been explored.

3.3 SBS/RS Configurations

The definition of a SBS/RS shows the variety possible within a SBS/RS system. In the last decade, SBS/RS's have become a preferable solution for the automated handling of TSU 's in the case of extremely high throughput demands. In addition, a SBS/RS delivers more flexibility and has a higher energy efficiency (Kosanic et al., 2018). This development calls for the optimisation of the design of the SBS/RS. Thus, before moving to the modelling behind the design optimisation, different innovations of the SBS/RS are reviewed in the literature.

The study conducted by Ekren (2017) shows that simulations can be used to make decisions in the design choices of a SBS/RS warehouse. The study considers several numbers of bays, tiers, arrival rates and rack designs to show the trade-offs in these variables via graphs. The study concludes that the approach increases the practitioners' operational efficiency and decreases costs. Nevertheless, due to the aisle-captive design, the SBS/RS used in the research models the warehouse as one aisle. Hoffman and Asada (2017) proposes a new configuration of the lifts for a SBS/RS. The current structure, which is solely vertical transport, can be switched to a design incorporating diagonal transport options for the lift. This does provide a larger challenge in collision avoidance. The new elevator path design provides highly adaptive scheduling capabilities.
Ekren et al. (2023) propose an alternative design to a tier-captive SBS/RS with a moveable lift. The lifts transport the shuttles through the warehouse with a smart anti-collision algorithm for the lifts. The research shows that for high throughput warehouses, the SBS/RS outperforms the proposed solution in terms of performance and

costs. However, for medium to low throughput warehouses, the SBS/RS can quickly become too expensive compared to the movable lifts, while both reach the desired throughput. Lerher et al. (2015) researched the impact of lift movements on the throughput of the SBS/RS. In these systems, the lifts are often the bottlenecks, and the velocity of lifting mechanisms is therefore important to optimise. An important relationship in a tier-captive SBS/RS is the change in lift performance and the actual change in the throughput. Eder (2023a) states that an increase in throughput due to different storage and retrieval policies requires an increase in lift capacity. However, the increase to 100% of lift capacity results in a 50% increase in throughput. An important aspect to note is that the shuttles are aisle-captive in this scenario. A scenario with non-aisle-captive shuttles does not strictly adhere to this relationship between lift capacity and throughput. Min and Lim (2023) proposed an alternate approach to the general rack structure with a bidirectional infinite-loop modular design. This design offers maximized storage capacity through enhanced space utilisation, faster processing speeds and improved land usage efficiency.
Ekren et al. (2022) studied and compared the performance of a flexible and non-flexible design of a SBS/RS, the performance is based on throughput, total investment costs and energy consumption per transaction. The flexible design is a tier-to-tier shuttle design, while the non-flexible design is tier-captive as well as aisle-captive. The study shows that the non-flexible design performs better in terms of energy consumption per transaction. However, the flexible design performs better in throughput and total costs. An unique SBS/RS design in the literature is proposed by Wu et al. (2020). Where the shuttles are aisle-bound, tier-captive with the presence of a cross aisle which is occupied by a single cross-aisle shuttle. So shuttles are not able to move between aisles, but the transport of TSU's between the aisles is done by the cross-aisle shuttle, allowing for more flexibility for TSU's than solely having an aisle-captive SBS/RS. The optimisation model proposed uses a complex queuing network to calculate the travel time of shuttles. The model applied for the optimisation in the design process of the warehouse is to first determine the number of aisles required by the system. After this, the other parameters, such as the number of columns, tiers, lifts, and workstations are determined. Finally, the configuration with the minimal facility costs is determined and selected. A comparison was drawn between the SBS/RS and a robotic fulfilment system. The comparison shows lower costs of the robotic fulfilment system when the throughput and capacity of the system is quite low. The SBS/RS becomes a superior system when the throughput and capacity increase.

3.3.1 Shuttle Configurations

In a warehouse, the speed of the shuttles plays an important role in the system's performance. In an optimisation based on the Pareto front, the conclusion is drawn that a shuttle which is too slow is harmful to efficiency. Whereas a shuttle which is too fast harms the energy consumption and CO_2 emissions. Hence, a trade-off must be made in the design process of the shuttles (Borovinsek et al., 2017). Zhao et al. (2020) gives a detailed approach on using a Semi-Open Queuing Network to coordinate the shuttle sub-system for a Tier-to-Tier system. This study focuses on the retrieval process as this is vital in the e-commerce business, where the key takeaway is that an increase in shuttles does not always lead to higher efficiency. Ha and Chae (2019) shows that the calculations regarding shuttle requirements differ depending on the type of system,

either tier-to-tier or tier-captive. Ha and Chae state that the travel time model supports the decision model to adjust the number of shuttles based on specific throughput capacity requirements. The papers show that faster shuttles lead to an increase in throughput up to a certain point. At some point, the increase in throughput will stagnate and even lead to a decrease, showing the importance of carefully examining all aspects of a system to maximise throughput. Another important trade-off in the adaptation of the shuttle speed is the travel time of a shuttle versus the energy consumption of the shuttle, where a decrease in travel time is achieved at the expense of an increase in energy consumption. Concerning the initial design of the SBS/RS, the study finds that an increase in the height of a rack leads to a rise in energy consumption for the completion of the same storage and retrieval requests (Yang et al., 2023).

3.3.2 Class-Based Storage

A paper by Kriehn et al. (2018) shows the importance of class-based storage. The paper proposes multiple methods for dividing products into classes and determining which locations in the warehouse the different classes should occupy. Through simulation, the various configurations of classes in warehouses increase the throughput by reducing the waiting time of the shuttles. In addition, the zoning leads to lower energy consumption, which is an essential theme in many papers in warehouse optimisation. Thus, the class-based zoning of goods in a warehouse is essential when optimising a warehouse. The combination of the class-based storage with the multi-deep storage in a warehouse. The correlation of these two concepts is studied by Eder (2022) and shows an improvement in the current warehousing systems is identified. The study also proposes how the proposed approach can be used in the design process of a SBS/RS to meet the desired requirements. The class-based storage based on the Pareto principle aids in the optimisation of the throughput in a SBS/RS.

3.4 Existing Optimisation Methods in a SBS/RS

The previous section has provided an overview of the different configurations of an SBS/RS available in literature. Next, various modelling techniques available within the design optimisation of an SBS/RS are identified. The research has identified three main modelling techniques: Queueing, Heuristics and ML models. The following section provides an overview of the relevant optimisation models within the scope of the research.

3.4.1 Queueing

The only mention of a SBS/RS which involves tier-captive shuttles but not aisle-captive by Eder (2023b). In the paper, a continuous-time open-queuing system with limited capacity is created to determine the performance measures and is validated using a Monte-Carlo simulation. In addition, the optimal geometric dimensions of the number of tiers and several slots to achieve the highest throughput were achieved through a parameter variation. The research has led to interesting improvements through design choices. An improvement of 5% is gained when the DC s are done in a single aisle instead of the storage and retrieval locations distributed randomly in a tier. The paper

shows the potential of design optimisation of a non-aisle-captive, tier-captive SBS/RS where simple parameters were optimised, leading to a significant performance increase. In the case of a double-deep, a queuing system has been created by Eder (2020) to aid in the design modelling of a tier-captive SBS/RS. A queuing system shows again the ability to calculate performance and use the performance to aid in early design decisions. The disadvantage of a queuing system is its specificity for a specific SBS/RS variation. The ability of a queueing model to calculate the throughput of a system is very efficient for a SBS/RS. However, it lacks a general model through which the SBS/RS can be optimised in the early design process without having to create a different model for each situation. A very time-consuming operation within an early design optimisation.

3.4.2 Heuristics

Yang and Ren (2023) states that the selection of the type of system is crucial in the pre-design stage of the system. Next to the throughput, the costs of a design should be taken into account in the objective of the system. They propose a custom heuristic solution algorithm that optimises the model in steps. First, the number of aisles is determined by maximizing the throughput for the available size and with a maximum number of lifts. Next, the throughput of shuttles and lifts is analysed and the minimum number is determined, which reaches the desired throughput. The downside of this approach is the sequence in which the optimisation is done. First, the number of aisles is determined and other adaptations are made only after this is set. When increasing the number of variables involved in the optimisation, the solution space is very limited by this approach.
Next to optimising the entire model, heuristics are also created to solve more specific issues in the warehousing optimisation. Chen et al. (2023) have taken the double lift configuration of Vanderlande and applied an Adaptive Large Neighbourhood Search (ALNS) to solve the retrieval request scheduling problem. Next to the research into the destroy operators, repair operators and the operator selection, the research also presents two practical conclusions. The optimisation of the retrieval process is vital for good system performance and the velocity profiles of both the shuttles and the lifts influence the warehousing operations considerably. This approach involves all variables at once, leading to a huge solution space and, therefore an increase in computational time. Nevertheless, by creating suitable destroy and repair operators and a smart operator selector, this approach can lead to near-optimal solutions. An advantage of this approach is the explainability and interpretability of the solution and the optimisation steps taken.

Two approaches often seen in literature are an Ant Colony Optimisation and a Genetic Algorithm (GA). Nia et al. (2017) applied these algorithms to a sequencing optimisation for an AS/RS with multiple racks and multiple rack locations while considering the emission of greenhouse gases. Their research showed that the GA outperformed the Ant Colony Optimisation in terms of total costs, performance costs and emission costs. Dong et al. (2019) proposes using a Bacterial Foraging Optimisation algorithm to solve the picking optimisation of an AS/RS with multiple carriers and irregular goods. Using multiple objective functions for this NP-hard problem, the other

algorithms found in literature by the authors, such as Simulated Annealing, GA and Ant Colony Algorithm, were outperformed. Another optimisation model is proposed by Sui et al. (2019) with a Genetic-based Particle Swarm Optimisation (GPSO). This algorithm optimises the storage/retrieval scheduling problem in a tier-to-tier SBS/RS. The model outperforms a GA and a basic Particle Swarm Optimisation (PSO), where the GPSO has a faster computing speed and higher optimisation efficiency. A GA has been used in multiple papers as a benchmark for the proposed optimisation method, where most methods outperformed a GA. He et al. (2024) proposes an improvement on the GA, a GA with priority selection, adaptive operators, and a decay factor (GA-DF). The adapted GA-DF shows improved performance over traditional GA's, Simulated Annealing algorithms and Improved Genetic Algorithms (IOSA), outperforming these algorithms up to 50%. This shows the power of a GA when carefully constructed to the scenario.

Another optimisation method is proposed by Di (2024), where a Tabu Search algorithm is proposed in combination with a neural network to optimise storage sequencing. It must be noted that the model is not equipped to handle varying goods well. So, further research is required to adapt the model such that it is able to handle more complex situations. This research shows the ability to combine a heuristic like Tabu Search with a ML component.

3.4.3 Machine Learning

While ML has seen an enormous rise in mentions in research articles, as Figure A.1 in Chapter A shows the number of articles mentioning ML per year. However, the application of ML within the optimisation of a SBS/RS warehouse is very limited. In a paper by Ekren and Arslan (2022), a Reinforcement Learning (RL) method has been implemented in the picking order optimisation in a SBS/RS. A Q-learning approach has outperformed FIFO and Short Processing Time (SPT) scheduling rules and the authors suggest that the implementation of more complex RL methods such as deep Q learning could outperform both original scheduling rules. This research shows the potential of RL in the optimisation of a SBS/RS. A more advanced version of Q-learning is a variant of Deep Reinforcement Learning called Deep Q-learning (DQL). DQL is described by Arslan and Ekren (2022) to optimise the transaction selection policy. The DQL selection policy is tested against FIFO and SPT in a simulation model where the DQL is applied in real time. When a shuttle becomes available in the system, the next transaction is selected based on one of these policies if more than one transaction is present in the system. DQL consistently outperforms the FIFO and SPT selection policies when looking at the average cycle time of the model. When taking a broader approach and looking at ML within AS/RS in general, the papers are limited and no application within early design optimisation can be found in the literature. A data-driven approach for dwell point optimisation was proposed by Rizqi et al. (2024). Where a meta-heuristic is created with a ML aspect in the policy evaluation. Logistic regression, K-neighbours and decision tree models jointly optimise the hyperparameters of individual learners. A more advanced implementation of RL in the warehousing sector is a Q-learning algorithm to optimise dynamic routing planning for automated guided vehicles. The study by Zhang et al. (2024) shows that dynamic routing planning for multi-AGVs operating in a large-scale warehouse is possible with

the use of RL. Many of the optimisation methods used in the papers are heuristics. A possible disadvantage of a heuristic is the inability to handle large solution spaces. The heuristic best equipped to handle the large solution space due to its high efficiency is the ALNS proposed by Chen et al. (2023).

3.5 Model Selection

The literature highlights the diversity of SBS/RS's, with multiple design choices available during the design process, particularly concerning the shuttle movements and the lift configurations. Various modelling techniques have been applied to address both design and operational optimisation, such as order picking and shuttle routing. The three most commonly used model categories are queueing models, heuristic approaches, and ML-based methods. The literature indicates that queueing does not fit the scope as the models are too rigid to handle many different design options. Queueing models typically rely on strict assumptions regarding system behaviour and are less adaptable to dynamic environments like the ADAPTO system.

Heuristics and ML are both suitable for research and are thus examined more closely. The solution space of the optimisation problem consists of 43,568,504 different configurations of the model, emphasising the requirement of a model able to handle such a solution space. In addition, this number is calculated in Section 4.1.2 and only takes the limited number of variables into account from Chapter 2. Outside of the scope of this research are more variables that can also be taken into account after further development of the model. Given the complexity and size of the solution space, an Adaptive Large Neighbourhood Search (ALNS) heuristic is proposed as the most suitable due to its operator versatility. These operators allow domain knowledge from the experts at Vanderlande to be implemented within the model. Reinforcement learning is chosen as most suitable for ML as both supervised and unsupervised do not fit the project's scope. Q-learning and Deep Q-learning are implemented in warehousing optimisation in the literature and the characteristics of the models align with the goal of the research. Table 3.1 gives an overview of the optimisation methods found in the literature. Since the literature does not yet cover early design optimisation using deep Q-learning, the reviewed articles are evaluated based on five metrics.

The first criterion for evaluating a paper is the objective of the paper. The paper must focus on the early layout design of a warehouse to align with the research of this paper. Within a warehouse, there are numerous different options for a picking system. The ADAPTO system is a SBS/RS system, and a paper's methodology should align with the system optimised. In line with this is the third criterion, the non-aisle captivity of the system. The aisle captivity results in a single shuttle per aisle, as well as a lift for each aisle. This leads to a modular design with a single aisle as the module. This results in different trade-offs and options that impact the optimisation. The optimisation goal of an aisle captive system translates to optimising the throughput of a single aisle. When determining the size of the system for the throughput, only the number of aisles and the length of the aisles must be determined. This varies significantly from the ADAPTO system, where the system's interaction between shuttles and routing between aisles is crucial. Aisle captive system optimisations, therefore tackle a

different optimisation problem. The last critical aspect of this research is the data used to validate the optimisation model. This research uses real data from Vanderlande to validate the model, hereby showing the actual real-world application of the optimisation model. Validation through data is done in every paper, so the criterion is based on a traceable source of the data within the paper.

Table 3.1: Overview of the applicability of the papers named in the literature review.

Paper	Early layout design optimisation	SBS\RS	Non-aisle-captive	Use of ML	Use of real-world data
Diaz et al. (2024)	✓		✓		
Lyu et al. (2021)	✓		✓		✓
Hu and Chuang (2023)	✓		✓		✓
Mayadunne (2024)	✓		✓		✓
Dong et al. (2019)	✓		✓		✓
Sui et al. (2019)		✓			
Eder (2020)		✓			
He et al. (2024)		✓	✓		
Ekren and Arslan (2022)		✓		✓	
Arslan and Ekren (2022)		✓		✓	
Chen et al. (2023)		✓			✓
Di (2024)					✓
Yang and Ren (2023)	✓	✓			✓
Proposed method	✓	✓	✓	✓	✓

Table 3.1 shows the novelty of this paper. To the best of our knowledge, this is the first paper which deals with the early layout design optimisation of a non-aisle captive SBS/RS system using a ML algorithm, which is validated using real-world data. The model is formally defined as Optimisation of Early Layout Design in SBS/RS Warehouses or OELDSW.

3.6 Optimisation Modelling

The previous section highlights the novelty of this research. This section dives into the ML models to give a clearer picture of what ML model is used and the functioning of the model. The papers by Ekren and Arslan (2022) and Arslan and Ekren (2022) give us Q-Learning and Deep Q-Learning, which are proven methods to apply within warehouse optimisation. These methods are part of the reinforcement learning methodology in ML, which is explored in the following section.

3.6.1 Reinforcement Learning

Reinforcement Learning is one of the three main directions in ML, next to supervised learning and unsupervised learning. RL consists of an autonomous agent that must make intuitive decisions and consequently learn from its actions. The key idea is to learn how the world works to maximize cumulative rewards over time through trial and error (Sutton and Barto, 2018). Figure 3.1 shows the basic interaction an agent has with its environment (Naeem et al., 2020). The four most important components that must be elaborated on are listed below.

State: A state represents a specific condition or configuration of the environment at a given time as perceived by the agent. The state sets the scene for the agent to make choices and select actions (Ghasemi et al., 2024).

Action: The actions are the set of possible moves or decisions by the agent based on the current state (Ghasemi et al., 2024).

Policy: The policy guides the behaviour of a learning agent by mapping the state into action. This can be a simple function, a lookup table or a complex computation. A policy can be both stochastic and deterministic (Ghasemi et al., 2024).

Reward: The rewards show the consequence of the action taken with respect to the objective of the RL model. Based on the objective, the reward can be both positive and negative. The reward helps update the policies according to the outcome of the action (Ghasemi et al., 2024).

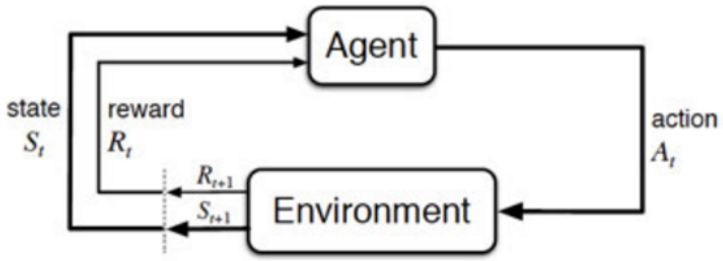


Figure 3.1: Basic interaction between a RL agent and the environment.
source:Naeem et al. (2020)

As stated, in each state, the agent must take an action, and the choice of action is based on the expected reward of taking an action at time step t, which is calculated through the expected reward function. After taking an action a, the expected reward function is updated based on the difference between the received reward and the expected reward, shown in the formula below (Ghasemi et al., 2024).

$$NewEstimate = Oldestimate + StepSize * [Target - OldEstimate]$$

The step size determines the extent to which new information overrides old information. How a RL model selects the action is crucial to the model's functioning. The model can have a greedy selection, where the maximal immediate reward is selected based on the current information. But to know all rewards, it requires all actions, which is problematic. A solution could be to select randomly from all actions with a small probability. This method, called ϵ-greedy, balances exploration and exploitation much more (Ghasemi et al., 2024). Exploitation in RL is exploiting the current knowledge and taking the action which results in the highest expected reward of the environment to maximise the cumulative reward. Exploration is improving the knowledge of the environment (Zangirolami and Borrotti, 2024). Due to the uncertainty associated with the estimation of action values, exploration is essential before exploitation. Where the exploration usually decreases over time (Ghasemi et al., 2024).
Value functions create a partial ordering over policies, allowing comparison and ranking based on expected cumulative rewards. The selection of a value function to estimate the return of an agent in a certain state or performing an action in a particular state depends on the agent's environment. Using a state-value function is beneficial

when there are many actions, as these methods reduce complexity. Action-value functions are used to evaluate and compare the potential for different actions when they take place in the same state. These functions are crucial if the goal is to determine the most appropriate action for each situation. More complex RL models use a combination of these types of functions (Ghasemi et al., 2024; Feng and Zhong, 2023).

3.6.2 Reinforcement Learning Methods

Sutton and Barto (2018) describes Q-Learning, Dynamic Programming, Monte Carlo methods, Actor Critic and Policy Gradient methods. Q-Learning is the most common RL technique with the main advantage of not requiring a model (Manju and Punithavalli, 2011). Furthermore, in the case of a complex state and action space, the model can be extended to a Deep Q-Learning model, which is capable of handling the larger state and action space (Jang et al., 2019). Dynamic Programming (DP) and Monte Carlo methods require a model of the environment, including transition probabilities. The ADAPTO simulation model is too complex to map these probabilities in this research, therefore, these modelling types are unsuitable. The Policy Gradient Method is known to face convergence issues within a large state space (Xiao, 2022). The state space for the simulation model is considered large, where 4.1.2 goes into the size of the state space more extensively. Hence, Policy Gradient methods are not a good fit for this research. Lastly, Actor-critic models do not require a model such as DP and Monte Carlo and it can also be extended to a deep reinforcement learning model (Abdalla et al., 2023). However, the approach benefits from a model environment with stochastic rewards and not deterministic. Based on these models, Q-Learning and Actor-Critic models are most suitable for this research. Due to the preference for stochastic actions of Actor-Critic models, where the simulation model has deterministic actions, Q-Learning is selected as most suitable for this research. This is based on the model-free approach, the ability for deep reinforcement learning and the ease of implementation.

3.6.3 Q-Learning

Q-Learning is a form of Reinforcement Learning algorithm which does not need a model of its environment (Manju and Punithavalli, 2011). The main goal of the Q-learning algorithm is to learn an optimal control policy from the data collected from the interaction between the agent and the environment. To derive the optimal control policy, Q-learning is based on a value iteration algorithm aiming to find the optimal state-action value function, also known as the Q-function (Perrusquía et al., 2024). Q-Learning can be classified into single-agent and multi-agent algorithms. Single-agent algorithms or basic Q-learning use an off-policy method to separate the acting and learning policies. As a result, even if the action selected in the next state was mediocre, the information was not included in the updating of the Q-function of the current state, and the dilemma is that of a wrong choice (Jang et al., 2019). The equation of Q-learning is the following:

$$Q_{\text{new}}(s_t, a_t) = Q(s_t, a_t) + \alpha * [r_{t+1} + \gamma * \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Where α is the learning rate of the model, the Q value $Q(s_t, a_t)$ of the action for the current state S is updated with the sum of the existing value $Q(s_t, a_t)$ and the equation which determines the best action in the current state. The Q-values are stored in a Q-table. Q-learning is continued by updating the Q-value for each state continuously using the formula above. This process is repeated so that the overall Q-value converges to a specific value, where the table of Q values can be used to solve a given problem (Zhong and Wang, 2025). Due to the required Q-table for the model, a considerable amount of storage memory is required. Hence, in a multi-agent environment with two or more agents, a large state-action space memory is needed, which causes issues and limits effective learning (Jang et al., 2019). In cases with multiple agents, Deep Q-learning provides an algorithm that is able to handle multiple agents.

3.6.4 Deep Q-Learning

A Deep Q Network (DQN) is an extension of Q learning, which is a typical deep reinforcement learning method (Ohnishi et al., 2019). Deep Q-learning combines basic Q-learning with Convolution Neural Networks (CNN), originally developed by Google Deep Mind. Q-learning employs an approximation function using a CNN when expressing the value function for every state becomes difficult (Jang et al., 2019). At each iteration of a DQN, a mini-batch of states, actions, rewards and next states are sampled from the replay memory as observations to train the Q-network. In addition, DQN uses another neural network named the target network to obtain an unbiased estimator of the mean-squared Bellman error used in training the Q-network. The target network is synchronised with the Q-network after each period of iterations to ensure coupling between the two networks (Fan et al., 2020). To avoid a low in an unusual direction due to a correlation between samples, a DQN collects many samples, which are stored in memory. However, using too much memory negatively impacts the learning speed of a DQN (Jang et al., 2019). In addition, the research by Fu et al. (2019). emphasises the advantages of a large neural network with regard to the learning stability of the model. Large neural networks also offer practical compensations for over-fitting and compensate for function approximation error. DQNs do have the drawback of requiring much more data for training compared to basic Q-learning due to the use of a neural network. If a DQN is selected as the optimal approach, the availability of the data must be ensured. Jang et al. (2019) shows the many applications of both Q-learning and deep Q-learning for operations research as well as an overview of additional applications of more in-depth Q-learning and DQN algorithms.

3.7 Summary

This section has given an overview of the current optimisation methods within early layout design optimisation and optimisation methods within SBS/RS warehouses. Table 3.1 shows the novelty of the research in both these areas. The optimisation area is therefore formally defined as Early Layout Design of SBS/RS Warehouses Optimisation or OELDSW, which is solved using ML and validated using real-world data. Lastly, the literature is consulted on applicable ML methods and Deep Q-learning is selected as most appropriate. The next chapter covers the development of the Deep Q-learning method and the integration with the ADAPTO simulation software.

Chapter 4

Solution Approach

This chapter covers the third phase of the CRISP-ML framework: the ML model engineering. This chapter combines the Deep Q-learning approach for the automation and optimisation of the simulation of the early design of the ADAPTO system. This model is split into the Q-learning structure through the state, agent and constraints. The second part of the chapter dives into the Deep Learning features of the model, going into the Neural Network architecture. Lastly, the pseudocode of the main algorithms is given. This chapter delivers the framework for the model such that different hyperparameter setups can be evaluated in the experimentation phase of the research.

4.1 General Structure

This section gives a general overview of the Q-Learning structure of the model, its input and output structure and its integration with the simulation software. Deep Q-learning combines Q-learning, a reinforcement learning technique, with deep neural networks to approximate the optimal action-value function. The Q-learning framework gives the structure to the model with the state, actions and rewards and describes the interaction with the environment. The neural network component approximates the Q-values used for decision-making. The architecture of the neural network and the setup for the initial and terminal states are described in Section 4.2.

Figure 4.1 illustrates the iterative learning process of the Deep Q-Network. In the figure, the agent's architecture consists of the Deep Q-Learning model and the Epsilon Greedy structure. The environment of the model consists of the simulation model of the ADAPTO system. The process starts with an initial configuration loaded into the simulation environment. The simulation provides the reward based on the current state and action. This experience is stored in replay memory, and the Q-value is updated using a loss function. The model then selects the next action using the ϵ-greedy policy, updates the simulation, and repeats this cycle until the stopping criteria are met. Section 4.1.1 describes the different states the agent can find itself in, where Section 4.1.2 goes into the actions it can take in each state. Section 4.1.3 presents the constraints that the environment is limited by, where Section 4.1.4 goes into the environment itself.

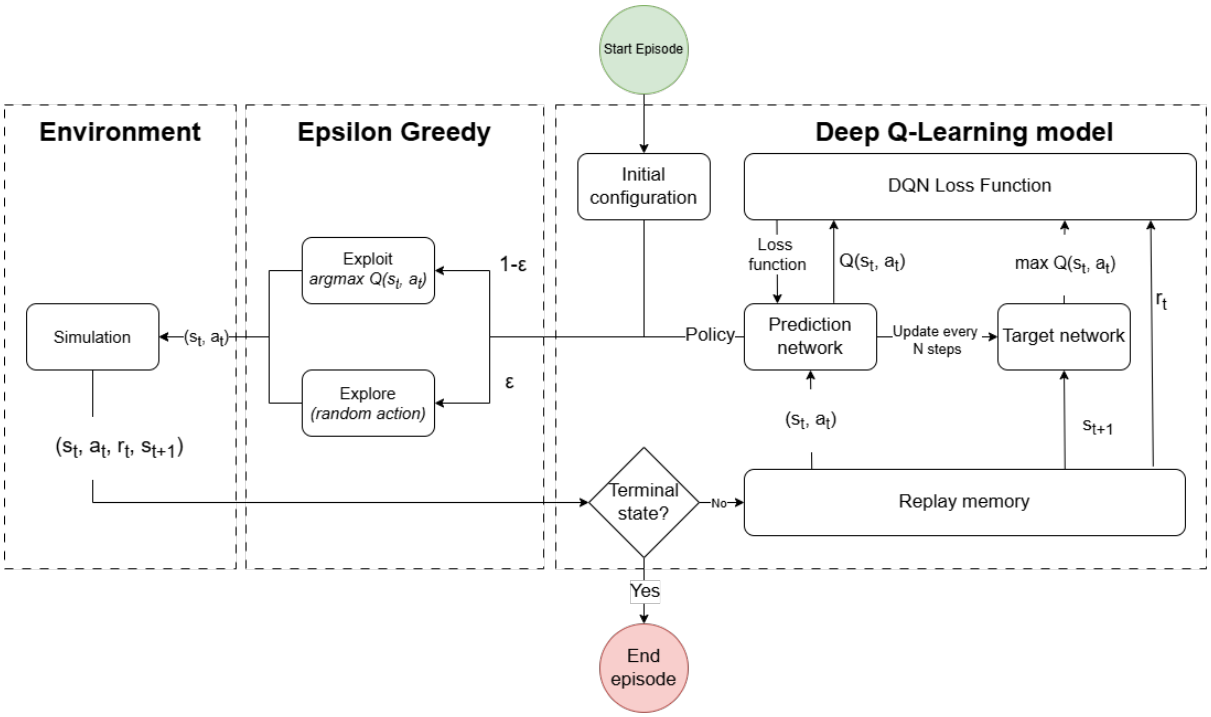


Figure 4.1: Flowchart showing the training process of the Deep Q-learning network. The environment consists of the ADAPTO simulation software and the agent consists of the Deep Q-Learning model and Epsilon Greedy structure.

4.1.1 State

The state of the DQL represents the agent's perception of the environment, providing the necessary information to decide on the following action. The agent uses this state representation to determine the best action toward achieving the optimal result. The state vector in this model consists of a set of variables from Section 2.4, parameters from Section 2.3, the throughput goal and the throughput of the current state. The model aims to find the optimal configuration of variables such that the configuration reaches the throughput goal of the model for the order profile parameters. Table 2.2 gives an overview of the data type and the minimum and maximum values of the variables.

Variable	Symbol	Variable Type	Minimum Value	Maximum Value
Number of levels	lv	$\mathbb{Z}$	17	41
Number of aisles	ais	$\mathbb{Z}$	4	25
Number of x positions	p	$\mathbb{Z}$	20	150
Depth	d	$\mathbb{Z}$	1	2
Lift type	lt	$\mathbb{Z}$	1	2
Number of rack buffers	r	$\mathbb{Z}$	2	3
Number of shuttles per level	sl	$\mathbb{Z}$	1	14
Number of lifts	ln	$\mathbb{Z}$	2	24

Table 4.1: Overview of the variables in the state, their symbol, variable type and minimum and maximum value.

The variables have a clear range in which they lie, easily limited by a minimum

and maximum. The parameters in the state that describe the order pattern of the customer are more complex. In order to train the model, the state requires values for the parameters that describe the system. To address this, each parameter has been discretised into a set of representative cases that capture the majority of real-world scenarios. These cases were defined based on historical data and insights from experienced simulation engineers. These cases serve as the input for the simulation model to create the environment in which the agent can interact to train its policy. It is important to note that a Deep Q-Learning algorithm does not require a training dataset as it learns through interaction with the environment. The model can effectively represent a wide range of sales scenarios by combining these parameter cases. The parameter *goal* represents the minimal throughput the customer requires from the ADAPTO system. An overview of the parameters is given in Table 4.2. The values for the velocity classes are placed in Table 4.3 so that both tables remain clear.

Parameter	Symbol	Variable type	Values	Numerical representation
Velocity classes	V	$\mathbb{Z}$	general, custom, single	{0,1,2}
TSU distribution	$TsuD$	$\mathbb{Z}$	normal, uniform, bathtub	{0,1,2}
TSU range	$TsuR$	$\mathbb{Z}$	{1,5,10}	
Sequencing	Seq	$\mathbb{Z}$	strict, relaxed, unsequenced	{0,1,2}
TSU group distribution	$TsuG$	$\mathbb{Z}$	uniform	
Smart lift allocation	sla	Binary	{0, 1}	
Goal	G	$\mathbb{Z}$	[500,6000]	
Current throughput	T	$\mathbb{Z}$	[0,9200]	

Table 4.2: Overview of the parameters and their symbols, variable type and values.

	General	Custom	Single
Locations	{20%, 30%, 50%}	{5%, 9%, 86%}	{100%, 0%, 0%}
Movement	{80%, 15%, 5%}	{41%, 45%, 14%}	{100%, 0%, 0%}

Table 4.3: Overview of the velocity classes

The definition of these variables and parameters leads to the following mathematical formulation of s_t, the state at time t:

$$s_t = \{lv_t,\ ais_t,\ p_t,\ d_t,\ lt_t,\ r_t,\ sl_t,\ ln_t,\ V_t,\ TsuD_t,\ TsuR_t,\ Seq_t,\ TsuG_t,\ sla_t,\ G_t,\ T_t\} \quad (4.1)$$

Each variable and parameter takes a value limited by the minimum and maximum value given in 4.1 and 4.2. By taking an action, the model varies one of the variables of the state to move towards the next state. The parameters are always determined at the start of an episode and remain constant for the entire episode. The interaction of the state with the neural network is described in 4.2.

4.1.2 Action Space

The action space in the DQL model defines the set of possible actions an agent can take within the environment, $a_t \subset A$. The action space A consists of the actions the agent can take in each state. Each variable in Table 4.1 can either be increased or decreased

with step size 1, except *ais*, where the step size is 5. The action space is equal to:

$$A = \{lv^-,\ lv^+,\ ais^-,\ ais^+,\ p^-,\ p^+,\ d^-,\ d^+,\ lt^-,\ lt^+,\ r^-,\ r^+,\ sl^-,\ sl^+,\ ln^-,\ ln^+\} \tag{4.2}$$

The model selects one of these actions every step, selecting the correct action more often as the model is trained. It is important to note that the action space is always limited by the constraints of Section 4.1.3. The agent selects one of these options at each step to either increase, denoted by $^+$, or decrease the variable, denoted by $^-$. The model can navigate through the solution space, which consists of 43,568,504 unique configurations. The neural network will estimate the best action to take based on the variables, parameters and the goal by predicting the Q value of each action. Each variable is limited by a range given in Table 4.1 where the configuration cannot exceed these limits. If the agent selects an action that leads to an invalid configuration by exceeding a limit, the agent will remain in the old state. The agent is punished for the wrong move by receiving a negative reward while the throughput remains constant. The reward function is given in Section 5.1 to elaborate on this structure.

4.1.3 Constraints

The actions described in Section 4.1.2 are limited by a set of constraints. These constraints limit the variables to a configuration that is possible in a real ADAPTO system. The first set of constraints, Equation (4.3) to Equation (4.10), limits the actions to the minimum and maximum value of the variable. The maximum value is determined by the maximum capacity of the system or specific customer requirements, such as space limitations.

$$lv^{\min} \leq lv_t \leq lv^{\max} \tag{4.3}$$
$$ais^{\min} \leq ais_t \leq ais^{\max} \tag{4.4}$$
$$p^{\min} \leq p_t \leq p^{\max} \tag{4.5}$$
$$d^{\min} \leq d_t \leq d^{\max} \tag{4.6}$$
$$lt^{\min} \leq lt_t \leq lt^{\max} \tag{4.7}$$
$$r^{\min} \leq r_t \leq r^{\max} \tag{4.8}$$
$$s^{\min} \leq sl_t \leq s^{\max} \tag{4.9}$$
$$ln^{\min} \leq ln_t \leq ln^{\max} \tag{4.10}$$

The minimum and maximum values are set as equal to the values in Table 4.1. However, if the size constraints are more limited due to the available size of the client, these maximum values can be limited to these custom maximum values.

$$ais_t \leq lv_t \tag{4.11}$$
$$lv_t * sl_t \leq 250 \tag{4.12}$$
$$sl_t \leq ln_t \tag{4.13}$$

$$ln_t < ais_t \tag{4.14}$$

Equation (4.11) ensures the number of levels is equal to or higher than the number of aisles. The ADAPTO is designed for optimal performance for high systems (more levels) compared to broad systems (more aisles). Equation (4.12) limits the total number of shuttles to be less than or equal to 250. This limit is set in Section 2.5.2 because of the Wi-Fi connection limitations. Equation (4.13) ensures the number of shuttles is not more than the number of aisles in a system. Due to the charging requirements of the shuttles, each shuttle requires at least one lift. Lastly, Equation (4.14) limits the number of lifts to 1 less than the number of aisles. The number of lifts not exceeding the number of aisles speaks for itself. However, as the ADAPTO system does not want an aisle captive system, the number of lifts can not equal the number of aisles.

4.1.4 Environment

This section covers the environment with which the agent interacts, the ADAPTO simulation model. 3.6.1 covers the basic functionality of the environment in RL, this section goes into the functioning of the simulation model. As previously stated, the exact functioning of the model cannot be shown due to confidentiality. Nevertheless, a global description of the most important aspects is given below.

Order Generation

The simulation model creates a digital twin of the ADAPTO model where the layout is based on the variables determined by the simulation engineer and the orders are based on the order pattern of the customer. Based on the parameters V, $TsuD$, $TsuR$, Seq and $TsuG$, the model generates orders for TSU's that are retrieved from the system and TSU's that are stored in the system. This allows Vanderlande to accurately simulate the performance of the ADAPTO configuration. While there is a random aspect in the order generation in the model, the seed used is constant, removing the randomness from the model. Therefore, the model results in an identical throughput if the same state is simulated multiple times. Thus, the results of the simulation model are deterministic. The model has been extensively tested and validated by Vanderlande, thus, using a single seed does not lead to a discrepancy between the simulation and a real ADAPTO system.

Runtime

The ADAPTO simulation model takes a heavy toll on the computational performance of the RL model. However, the model must also be able to run for a sufficient amount of time to produce accurate results. Therefore, the aim is to run the simulation for the minimum amount of time where the results are reliable. According to expert opinion at Vanderlande, this is 1 hour of simulation time. In real time, this results in a simulation time of 30 seconds to 1 minute, depending on the size of the simulation. The runtime of the simulation increases if the simulated system becomes larger. Every simulation requires the model to be built from scratch. Therefore, one change to the configuration requires a new model.

Impact of the Variables

The variables given in Table 4.1 each have their positive or negative impact on the throughput of the model. This section gives an overview of the impact the variables have on the throughput. The exact impact on throughput cannot be given as this differs per order pattern. A high-level overview of the general impact is given below:

- **Aisles, Levels & X positions:** These variables determine the size of the system. Increasing these variables without increasing the number of lifts or shuttles generally results in lower throughput as the same number of shuttles have to travel larger distances per order.
- **Depth:** Increasing the depth of a system from 1 to 2 results in a lower throughput. If an order requires the second TSU, the shuttle must first move the TSU in front, resulting in additional handling time.
- **Lift type:** Changing the lift type from a single platform lift to a double platform lift always increases the throughput.
- **Number of rack buffers:** By changing the number of buffers at each lift from two to three, the shuttles are more likely to deliver a TSU and move to the next storage or retrieval. Hence, always increases the throughput of the system.
- **Number of Lifts:** Increasing the number of lifts always results in a higher throughput as the shuttles have to travel less far to a lift and the waiting time on an available lift also decreases.
- **Number of shuttles:** Increasing the number of shuttles generally increases the throughput of the system as more orders can be handled simultaneously.

It must be noted that this is a very general description and the impact differs per order profile. For example, consider a system with a low number of aisles and a large number of x positions. If the order profile has many SKU's per order, adding shuttles can result in long waiting times when an aisle is occupied and even congestion of the system. Therefore, adding shuttles would decrease throughput. This shows that the impact of these variables has a general trend but is always influenced by the order profile of the customer.

4.2 Neural Network Architecture

A neural network predicts the Q-function; the neural network is built using the Keras library. Keras, a high-level API built on TensorFlow, simplifies developing and training deep learning models. Tensorflow is an end-to-end platform for ML (Tensorflow, 2024). Keras provides an approachable, highly-productive interface for solving ML problems, focusing on modern deep learning Tensorslow (2023). The Keras library is used in this research to create and train the neural network responsible for the prediction of Q-values. The following section details the architecture of the neural network, including the prediction & target networks, their layers, loss function, optimiser, and the implementation of experience replay in the DQL model.

4.2.1 Prediction & Target Network

The DQL model consists of two neural networks: the prediction and target networks. Mnih et al. (2015) states that reinforcement learning is known to be unstable or even to diverge. To address these instabilities, both a prediction and a target network are created. The prediction network is updated at every iteration, while the target network is only updated every 10 iterations, as suggested by Rafael1s (2020).

4.2.2 Layers of the Model

A layer in a neural network is a collection of neurons within an artificial neural network that simultaneously receive the same type of information (López et al., 2022). This section covers all the layers in the artificial neural network in the Deep Q-learning model.

Input Layer

The first layer of the neural network is the input layer; the input layer transfers data via synapses to the second layer of the model (Shan et al., 2018). The input layer contains the input data for the neural network and the input data describes the current state according to Wan and Hwang (2018). Thus, the neural network's input layer contains 14 neurons, the number of elements in s_t. The description of the elements in the state can be found in Section 4.1.1. The model continuously checks whether a configuration is viable and whether all variables are correct before they are input into the neural network. Therefore, any incorrect input into the input layer is impossible. The activation function of the input layer is the Rectified Linear Unit (ReLU). The ReLU is a simple function that is the identity function for positive input and zero for negative input, as shown in Equation (4.15) (Dubey et al., 2022).

$$ReLU(x) = max(0, x) = \begin{cases} x, & if\ x \geq 0 \\ 0, & if\ otherwise \end{cases} \tag{4.15}$$

Hidden Layer

A layer in a model is a hidden layer if it only interacts with other layers in the model and not with the 'outside world' (López et al., 2022). There is currently no standard method to determine the number of hidden layers the model requires and what size these layers should be. The main approach is through testing; nevertheless, there are general guidelines that aid in determining these parameters. During the evaluation of performance, underfitting indicates there are too few hidden layers, so the model cannot grasp the patterns in the data. According to Raut and Dani (2020), if the model overfits heavily, there are generally too many hidden layers and the number of layers must be reduced. For this model, three hidden layers have been selected. The relationship between the variables and parameters is not linear; more than one layer is required. The model is estimated to require deeper learning as it replaces a complex simulation model. Thus, three layers have been selected. According to Lawrence et al. (1996), a neural network benefits from a larger number of neurons in the hidden layers compared to a smaller number of neurons in the input layer, taking a number of neurons for the hidden layers larger than the number of input neurons. Berry and Linoff

(1997) states that the number of neurons in a hidden layer should not exceed twice the number of input neurons. Based on these two rules of thumb, the first and third layer have 24 neurons and the second layer has 32 neurons. The activation function is the ReLU function from Section 4.2.2.

Output Layer

The output layer is the layer that presents a pattern to the external environment (Karsoliya, 2012). The goal of the network is to estimate these Q-values so that the model can select the optimal action by choosing the one with the highest predicted Q-value. Hence, the output layer has neurons equal to the number of actions from Section 4.1.2. While the input and hidden layers of the network use the ReLU activation function to introduce non-linearity and prevent vanishing gradients, the output layer uses a linear activation function. This choice ensures that Q-values remain unbounded, allowing the network to learn accurate value approximations without imposing artificial constraints.

4.2.3 Loss Function

In Deep Q-learning, the loss function optimises the model's parameters by minimizing the error between the predicted and target Q-values (Terven et al., 2024). The goal is to adjust the network weights so that the estimated Q-value $Q(s, a, \theta_t)$ becomes more accurate over time. The difference between the estimated and actual value is calculated using the loss formula in Equation (4.16) from Tensorflow (2023). The formula uses the Mean Squared Error approach, a commonly used approach to determine the loss of the prediction, with the advantage that estimated values that lie further away from the actual value are punished more. The variable y_i is the Temporal Difference (TD), which is calculated by the current reward and the maximum reward given the state and action from the previous iteration, calculated in Equation (4.17). $Q(s, a, \theta_t)$ is the approximation of the Q function by the model. By minimising $L_t(\theta_t)$, the model will accurately predict the Q-values and converge to the true optimal Q-function $Q^*(s, a)$.

$$L_t(\theta_t) = E_{s,a,r,s' \sim p(.)}[(y_t - Q(s, a, \theta_t))^2] \tag{4.16}$$

Where

$$y_t = r + \gamma \max_{a'} Q(s', a', \theta_{t-1}) \tag{4.17}$$

4.2.4 Optimiser

The optimiser of a DQL model updates the network's weights to improve learning stability and convergence. Optimizers are algorithms that update the network's weights in response to the computed loss (KDnuggets, 2025). Keras (2025) provides several optimizers, with the most commonly used being Stochastic Gradient Descent (SGD), RMSprop, and Adam. SGD is the simplest approach, but can be slow in convergence. RMSprop improves on SGD by adapting the learning rate, making it more effective for non-stationary problems.
The Adam optimiser, introduced by Kingma and Ba (2019), combines the benefits of

momentum and adaptive learning rates, making it computationally efficient and well-suited for large-scale ML tasks. It is particularly effective for stochastic objective functions and non-convex optimisation problems, common in deep learning. The Adam optimiser is therefore selected as the optimiser used in the DQL model.

4.2.5 Initial & Terminal State

During the training phase, the model requires a starting point s_0 from which it moves towards the goal. To encourage broad exploration and avoid premature convergence to local optima, initial states are sampled uniformly within each variable's minimum and maximum ranges, as outlined in Table 4.1. Each starting state s_0 must satisfy two conditions: it must comply with all constraints of Section 4.1.3 and cannot already meet the throughput goal, as this would prevent meaningful learning.

A DQN requires a state considered terminal as an end point of an episode. A natural terminal state is a state where the configuration satisfies the throughput goal set at the start of the episode. Nevertheless, it is not guaranteed that a model finds a state where the throughput demand is satisfied, especially when the agent has not had many interactions with the environment to tune the Q-values. A cap on the number of iterations within one episode is required to ensure the model does not get stuck in sub-optimal states, thereby improving the learning efficiency and ensuring convergence. The cap must ensure the model is able to reach each state in the state space within a single episode. To allow the agent to reach any state it requires, the maximum number of actions required is 113. This is the number of actions the agent requires to move from the minimum of each variable to the maximum of each variable. This allows the agent to reach the state with maximal throughput each episode. However, the goal of the model is to configure the simple requests to relieve sales engineers such that they are able to focus on special cases. Simulation requests that involve throughput goals of more than 2/3 of the capacity of the system are always considered special cases and outside the scope of this research. This is represented in the range of the goal of the model in Table 4.2 as the goal cannot be more than 6000. Hereby excluding these cases from training. To ensure that the model does not attempt to optimize for throughput beyond the 2/3 system capacity, a conservative cap of 75 iterations per episode is set. This number is roughly two-thirds of the 113-step maximum and is intended to align with the typical throughput range for the system in this research. The agent tracks the number of iterations per episode and receives a True on the Terminal State if the number of iterations reaches 75.

4.2.6 Framework of the Method

The DQL developed in this research is focused on the implementation of a basic version of a DQL, which is implemented in a novel area, not the development of the most advanced version of a DQL. To be sure that a functioning model is used as the basis for this research. A framework for the Deep Q-learning part of the model is taken from Rafael1s (2020). The model was developed on a very common RL problem, the cart pole balancing problem provided by Foundation (2025). Taking the basic functioning from this already tested agent ensures that the basic functioning of the model is correct. This ensures the research is not limited by the agent's fundamental performance issues.

4.2.7 Training & Testing Data

This section describes the data used for the training process of the agent and the data used to evaluate the performance of the agent. An important characteristic of DQL is the training of the agent is done through interaction between the agent and the environment. The variables and parameters are limited by the ranges given in Table 4.1 and Table 4.2 and Section 4.1.3. These ranges and constraints are data used for training the agent. The ranges for the parameter of the order data are scenarios that summarise common cases that the agent must be able to handle. To evaluate the performance of the agent, the agent is tested against actual sales requests from customers of Vanderlande.

4.3 Pseudocode of the Main Algorithms

This section presents the pseudocode for the key algorithms driving the DQL model. As stated in Section 4.2.6, the basic structure is taken from Rafael1s (2020), adapted to be able to train on the ADAPTO environment. The full implementation can be found in Chapter B. First, Algorithm 1 is the main driver of the model, structuring the training process. In each episode, the model first generates a viable initial solution. Afterwards, the model enters a loop that runs until a configuration is found, which is a terminal state. Within the loop, the process of Figure 4.1 is followed.

Algorithm 1: The Deep Q-Learning optimisation model for the ADAPTO configurations using simulation tools

Data: Range of ADAPTO variables & parameters, actions $a_t \subset A$ & model hyperparameters
Result: Trained Deep Q-Network

Initialize Prediction Network, Target Network, ADAPTO simulation environment & t;
for $i \leftarrow 1$ **to** $NrEpisodes$ **do**
    InitialiseState();
    while *not TerminalState* **do**
        $a_t, s_{t+1} \leftarrow$ SelectAction(s_t, ϵ);
        $r_t, s_{t+1} \leftarrow$ CalculateReward(s_{t+1});
        TerminalState $\leftarrow$ IsTerminal(s_{t+1});
        StoreExperienceReplay$(s_t, a_t, r_t, s_{t+1}$, TerminalState);
        TrainNetwork();
        $s_t \leftarrow s_{t+1}$ #Update state;
        $t \leftarrow t + 1$ #Update Iteration Nr;
    $\epsilon \leftarrow \epsilon * \epsilon_{decay}$ #Decay exploration rate

The first step of the algorithm is to configure a starting state from which the agent must change to a state that meets the throughput goal. The initial state is checked whether it does not breach any constraints. The initial state is described in more detail in Section 4.2.5. The SelectAction method is explained in more detail in Algorithm 2. The CalculateReward method is responsible for running the ADAPTO simulation with

the new state and returning the reward of the new state, as well as the state that is updated with the throughput of that state. The next step is to check whether the state has reached a terminal state; if so, it is the last iteration in the episode. States which are considered terminal are states which have reached the throughput goal or if the while loop has reached 75 iterations. The 75 iterations ensure the while loop does not get stuck indefinitely. The next step of the algorithm is to store the current iteration in the replay buffer and train the neural network, the training process is shown in Algorithm 3. The loop ends by updating the state for the next iteration and updating the iteration number. If the episode ends, the exploration rate is updated based on the ϵ_{decay} scheme. Algorithm 2 is used for the action selection. First, a random number is sampled, which determines whether the agent will explore or exploit. This is done through the epsilon decay scheme, variations for this scheme are possible so the variation tested within this research are given in Section 5.2. Based on the choice, the model selects either a feasible random option or the action it expects to yield the highest reward. After the action selection, the new state is calculated based on the step size given in Table 4.2 after which the method returns the action selected and the updated state.

Algorithm 2: SelectAction method from Algorithm 1, the action selection of the agent using an ϵ-greedy approach

Data: Current state s_t, exploration rate ϵ
Result: Selected action a_t, next state s_{t+1}
Initialize Rnd = Random number in [0,1];
if $Rnd < \epsilon$ **then**
    $a_t \leftarrow$ SelectRandomAction(s_t);
else
    $a_t \leftarrow$ SelectBestAction(s_t);
s_{t+1} = GetNextConfiguration(s_t, a_t);
return a_t, s_{t+1}

The last algorithm that is explained is the training of the network in Algorithm 3. This algorithm ensures the policy of the agent is updated after each iteration, such that the variables responsible for the estimation of the Q-values converge to the optimal values. First, a batch of $B = 32$ samples is taken from the replay buffer. The Tensorflow environment takes the Q-values predicted by the prediction network for these samples and compares them to the actual values obtained from the reward function. The loss between the predicted Q-values and actual Q-values is taken using the Bellman equation in Equation (4.17). Based on the loss, the variables of the neural network are updated through back propagation. Every $N =$ steps, the target network is also updated using the weights of the prediction network.

Algorithm 3: Training of the neural network

Data: ReplayBuffer, UpdateInterval N, BatchSize B, TargetNetwork, PredictionNetwork, *StepCounter*
Result: Updated PredictionNetwork & TargetNetwork

Sample B experiences from ReplayBuffer;
Predict Q-values using PredictionNetwork;
Compute loss using Bellman Equation(Equation (4.17));
Update weights PredictionNetwork;
if $StepCounter \mod N == 0$ **then**
    Update weights TargetNetwork;
$StepCounter \leftarrow StepCounter + 1$;

Chapter 5

Experimental Setup

The previous chapter has given the basis of the Deep Q-Learning model by developing the agent and the interaction with the environment. As well as the design for the neural network used to predict the Q-values. This chapter goes into the variable aspects of the model, such as the reward function,ϵ decay strategy and variations on the DQL model. The hyperparameters tested to find the optimal setup for the agent are defined in this chapter.

5.1 Reward Function

The reward function is a fundamental component of a DQL and its importance cannot be overstated. The reward function directly impacts the agent's ability to understand the environment and guides the agent to an optimal policy Sutton and Barto (2018). The reward function must be defined to align with the desired behaviour and goals of the agent. This process is split into two primary areas, reward engineering and reward shaping Ibrahim et al. (2024).

Reward engineering involves the creation of the reward function itself. A reward function must provide informative feedback to the agent and incentivise the agent to show good behaviour Ibrahim et al. (2024). The first distinction in the reward function is whether to use a sparse or dense reward function. A dense reward structure has a state-to-state difference, where sparse rewards only provide rewards for a few select states (Memarian et al., 2021; Vasan et al., 2024). The agent's objective in this model is to identify a system configuration that satisfies a predefined throughput goal at the lowest possible cost. Therefore, throughput itself is not the reward, but the cost of the configuration is. Not every state reaches the goal and has a positive reward, so a dense reward structure is inappropriate. Hence, the model uses a sparse reward structure where states where the throughput goal is satisfied receive a positive reward.

However, except for thresholds that lie very close to the maximum throughput capacity of the system, most cases have multiple configurations that reach the throughput threshold. The goal of the model is to find the cheapest configuration that satisfy the throughput threshold, so a distinction must be made between these configurations based on the costs of the configuration. Such that the configurations that reach the

threshold for the lowest costs are rewarded more than more expensive configurations. Klar et al. (2021) has a similar reward structure within their facility layout planning and uses a similar reward structure. The key difference is the ratio between the negative and positive reward. This leads to the following initial reward function.

$$\text{reward} = \begin{cases} -\beta, & \text{if throughput} < \text{goal} \\ \alpha * (1 - \frac{costs - costs^{\min}}{costs^{\max} - costs^{\min}}), & \text{if throughput} \geq \text{goal} \end{cases} \tag{5.1}$$

Here, costs are normalised to a [0, 1] scale using $costs^{min}$ and $costs^{max}$, which are domain-specific values representing the minimal and maximal configuration costs. The reward function incentivises the agent to minimise configuration costs while still meeting the throughput constraint. The parameters α and β control the relative strength of positive and negative feedback: α scales the reward for achieving the goal efficiently, and β penalises configurations that fail to meet the throughput requirement. Different values for α and β are tested in the model to find the balance; the different variations are shown in Table 5.1. The values for α must be assigned a value such that the agent can identify the point at which the goal is reached by ensuring the action that leads to the best action is assigned the highest Q-value. The value for β must be configured so that the agent is incentivised to take the fewest number of actions to move towards the optimal configuration. The ratio between α and β is configured through trial and error, which is standard practice in RL, where the values for the variables are based on previous cases obtained from literature. The initial experiment runs with the values that are derived from Mnih et al. (2015), negative results receive a value of -1 and reaching the goal is rewarded with the value 1, so α=1 and β=-1. Furthermore, Klar et al. (2021) has also used a balance of -1 for β and a range of [0,1] for α. Nevertheless, the ratio has been identified through testing. Due to the sparsity of the reward, a higher reward for reaching the goal could be applicable, so the reward gained for reaching the goal impacts Q-values enough that the agents' actions move towards the goal. Therefore, experiments are performed with α =100 and α =1000.

Another method is proposed by Vasan et al. (2024), a contact reward formulation. This method rewards 1 if the target is met and 0 otherwise, translating to α=1 and β=0. This does require a discount factor, denoted by γ, such that a model is incentivised to take the shortest path towards the optimal configuration. Vasan et al. (2024) proposes a value of 0.99 and this is implemented in the model. This experiment is performed with the value for α that has resulted in the best policy.

Experiment	α	β	γ
1	1	-1	1
2	100	-1	1
3	10,000	-1	1
4	α^*	0	0.99

Table 5.1: The varying values for experimentation of α, β and γ to identify the optimal reward structure, where α^* is the value for alpha with the best policy. The full set of hyperparameters are presented in Table 5.3.

Theoretically, a general reward function should be able to deal with a sparse reward setting through sufficient training (Eschmann, 2021). Sutton and Barto (2018) states

that the reward signal is not the place to impart prior knowledge to the agent about achieving the goal. In practice, finding a solution in a sparse environment might be infeasible (Eschmann, 2021). Hence, tuning the reward function after inconclusive results from the base reward function is required. Reward shaping comes into play to fine-tune the reward function and enhance reward signals (Ibrahim et al., 2024). Laud (2004) has developed a theoretical foundation for tuning the reward function. The following section covers several possibilities for tuning the basic reward function. But crucially, the reward function must only be modified to guide the agents' behaviour without changing the optimal policy. Laud proposes the following formula.

$$R' = R + F \tag{5.2}$$

Where R is the base reward function from Equation (5.1) and F is a shaping function. The shaping function should assist the agent with prior knowledge to guide the agent towards the goal. Two cases of prior knowledge were evaluated, denoted as F_1 and F_2. The first shaping function F_1 focuses on throughput, as throughput directly reflects the progress towards the goal and is therefore an ideal candidate to shape the reward function. Guidance of the reward is required when the throughput goal is not met, and is not required when the throughput is sufficient. So the function F_1 replaces the negative reward β and will be scaled according to the optimal value for β through γ, leading to formula Equation (5.3):

$$\text{reward} = \begin{cases} \gamma * throughput, & \text{if throughput} < \text{goal} \\ \alpha * (1 - \frac{costs - costs^{\min}}{costs^{\max} - costs^{\min}}), & \text{if throughput} \geq \text{goal} \end{cases} \tag{5.3}$$

The second shaping function F_2 focuses on the boundaries of the solution space. Currently, the punishment for selecting an unfeasible action is an additional punishment β without moving closer to the goal, leading to a lower Q-value. However, this punishment is only small and the model could benefit from a higher punishment for an illegal move, a state where a variable falls outside the predetermined ranges of Table 4.1. Therefore, the shaping function adds a punishment of the value α such that an illegal action always receives a reward lower than any other action. This leads to Equation (5.4).

$$\text{reward} = \begin{cases} -\beta - \alpha, & \text{if the action is illegal} \\ -\beta, & \text{if throughput} < \text{goal} \\ \alpha * (1 - \frac{costs - costs^{\min}}{costs^{\max} - costs^{\min}}), & \text{if throughput} \geq \text{goal} \end{cases} \tag{5.4}$$

5.2 Epsilon Decay Scheme

Figure 4.1 shows the flowchart on the action selected using the ϵ-greedy approach from Sutton and Barto (2018). The ϵ-greedy balances the exploration and exploitation of the model. In the model, exploration is taking a random action to explore the action space and escape local optima. Exploitation always selects the action that has the highest Q-value prediction to converge towards an optimum. The model starts with a high degree of exploration as the agent has no sense of the environment yet. As the episodes progress, the model starts exploring less and exploiting more. This is done through a linear cooling scheme presented by Mnih et al. (2015) in Equation (5.5).

$$\epsilon = max(\epsilon * \epsilon_{decay}, \epsilon_{boundary}) \tag{5.5}$$

The value of ϵ is updated at the end of each episode with the variables ϵ where the initial value is $\epsilon_{start} = 1$ and ϵ_{decay}. This is done until the $\epsilon_{boundary} = 0.1$ is reached, as the model should not explore less than 10% of the time (Mnih et al., 2015). The value of ϵ_{decay} determines after how many episodes the $\epsilon_{boundary}$ is reached and therefore what percentage of time the model explores more than the minimum of 10%. Sutton and Barto (2018) propose using values of 0.9 and 0.99 as values for the ϵ_{decay}, but Sutton and Barto do state that more exploration is required with noisier rewards. Hence, a ϵ_{decay} of 0.999 is also tested. An overview can be found in Table 5.2.

Experiment	ϵ_{start}	ϵ_{decay}	$\epsilon_{boundary}$
6	1	0.9	0.1
7	1	0.99	0.1
8	1	0.999	0.1

Table 5.2: The varying values for ϵ_{start}, ϵ_{decay} *and* $\epsilon_{boundary}$ for the evaluation of different schemes for the ϵ-greedy approach. The full set of parameters for the experiments are shown in Table 5.3

.

5.3 Experience Replay

To improve learning stability and efficiency of the neural network, the experiences of the agent are stored in a replay buffer through a tuple $e_t = (s_t, a_t, r_t, s_{t+1}, TerminalState)$ at each time step. During the learning process, samples of experience are drawn from the buffer (Mnih et al., 2015). As reinforcement learning is a trial-and-error-based model, the learning is improved by storing past experiences and repeatedly presenting its past experience to the learning algorithm. Experiencing the past results in faster convergence of the model (Lin, 1992). The key advantage of drawing samples from the experience replay instead of learning from consecutive samples is that strong correlations between samples are decreased (Mnih et al., 2015). There are three important aspects of a replay buffer: the size of the buffer, the sample size and the sample selection method.

The size of the replay buffer partially determines the delay on important transitions of the model (Fedus et al., 2020). Mnih et al. (2015) proposes a replay buffer size of 1.000.000 samples on a total of 50.000.000 samples and this sizing is also used by Hessel et al. (2018) in the comparison of many DQN's. Initial testing shows the model only reaches 10,000 in the designated training time, so 1.000.000 is unrealistic. The ratio can be scaled proportionally, resulting in a replay buffer size of 200 experiences. If the buffer is full and a new sample is entered, the oldest sample is removed from the replay buffer.
Mnih et al. (2015) also proposes a size for the replay buffer; each iteration, 32 experiences are sampled from the replay buffer to train the neural network. A sample size of 32 is also used in the Deep Q-Learning code from OpenAI by Kaufmann and Pzhokhov (2019).

Lastly, the method for selecting 32 samples from the 200 stored samples in the replay buffer. A common approach is sampling uniformly from the experience replay, such that $(s_t, a_t, r_t, s_{t+1}, TerminalState) \sim U(D)$ from replay buffer D (Hessel et al., 2018; Mnih et al., 2015). According to Schaul et al. (2016), prioritising which transitions are replayed can make experience replay more efficient and effective than if all transitions are replayed uniformly. Schaul et al. (2016) proposed Prioritised Experience Replay (PER), where each sample receives a probability based on the expected learning impact on the model. The most promising experiences receive a higher probability. Both uniform sampling and prioritised sampling are evaluated in the DQN through experimentation.

5.4 Double DQN

A common problem in reinforcement learning, including in Q-Learning, is the systematic overestimation of action values (Thrun and Schwartz, 1994). To battle the overestimation of Q-Learning networks, Hasselt (2010) has proposed a double Q-Learning model. The model stores two Q-networks, Q^A and Q^B. At each iteration, one of the two Q-networks is used to select the next action by identifying the action with the highest Q-value, while the other network is used to evaluate the value of that selected action. The research of Hasselt (2010) shows increased performance of the double Q-Learning over the original Q-Learning model, as demonstrated in the experimental results. This concept was implemented into DQL by van Hasselt et al. (2015), leading to a Double Deep Q-Network (DDQN). The paper shows that overestimations of a DQN lead to poorer policies and shows the benefits of reducing overestimations. A DDQN does not require an additional prediction network because the target network functions as the second network. The model selects the action by taking the highest predicted Q-value from the prediction network. The episode's reward is calculated by inserting the Q-value predicted by the target network of the selected action and inserting the Q-value into Equation (5.6). The new Bellman equation still uses the temporal difference formula in Equation (4.16) to calculate the expected value.

$$y_t = r + \gamma * Q^{\text{target}}(s', \text{argmaxQ}^{\text{prediction}}(s', a, \theta_{t-1}), \theta_{t-1}) \tag{5.6}$$

5.5 Conclusion

This chapter has proposed a set of hyperparameter settings for the DQN. The values for the cooling scheme of the ϵ-greedy method and the size and sampling size of the replay buffer have been determined through the literature. The different reward functions, ϵ-decays, sampling methods and a different Bellman equation have been proposed. The optimal values for these hyperparameters are determined through experimentation in the next chapter. An overview of the various experiments and the corresponding hyperparameters is given in Table 5.3.

Experiment	α	β	Discount factor	Tuning Function F	$\epsilon - start$	ϵ_{decay}	$\epsilon - boundary$	Sampling method	Model Type
1	1	-1	1	-	1	0.99	0.1	Uniform	DQN
2	100	-1	1	-	1	0.99	0.1	Uniform	DQN
3	10,000	-1	1	-	1	0.99	0.1	Uniform	DQN
4	α^*	0	0.99	-	1	0.99	0.1	Uniform	DQN
5	α^*	[-1,0]	1	Throughput	1	0.99	0.1	Uniform	DQN
6	α^*	[-1,0]	1	Variable Limits	1	0.99	0.1	Uniform	DQN
7	α^*	β^*	1	-	1	0.9	0.1	Uniform	DQN
8	α^*	β^*	1	-	1	0.99	0.1	Uniform	DQN
9	α^*	β^*	1	-	1	0.999	0.1	Uniform	DQN
10	α^*	β^*	1	-	1	0.99	0.1	PER	DQN
11	α^*	β^*	1	-	1	0.99	0.1	Uniform	DDQN

Table 5.3: The different values of the hyperparameters used in the experiments conducted.

Chapter 6

Model Performance

In this chapter, the performance of the Deep Q model is tested and evaluated to find the optimal configurations of the model to accurately predict warehouse configurations. The goal is to identify optimal hyperparameter settings that enable the model to learn effective policies. Multiple models are trained with varying configurations, including different learning rates, reward scales, and exploration strategies. Each configuration is assessed based on key performance metrics such as the loss function behaviour and Q-value estimates. The results are presented in sections that outline the hyperparameters used, analyse model performance over time, and highlight observable patterns in the learning process.

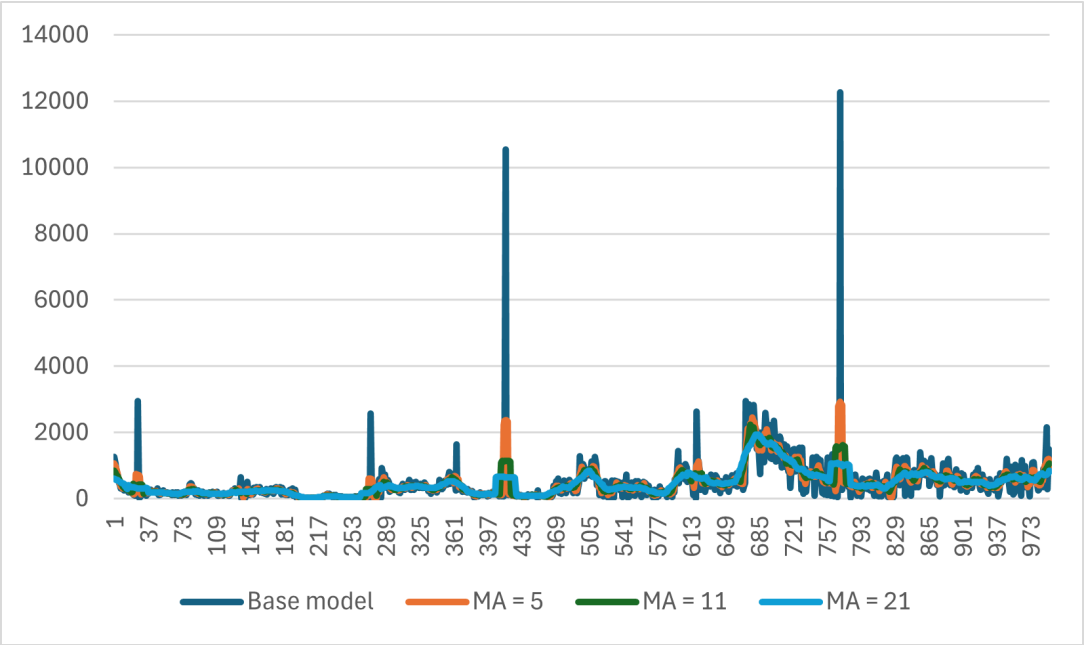
6.1 General Results

The first section of the results covers the performance of the base model. The base model serves as a baseline to compare to the performance of the models generated by the different hyperparameters, as described in Table 5.3. Through the base model's performance, the impact of the variation of hyperparameters becomes visible and the optimal hyperparameter setup for the most stable learning and best policies for the agent. The base model is evaluated based on two criteria: the progression of the loss function to gain insight into the training process of the agent and on previous sales projects of Vanderlande, and comparing the configuration from the agent to the configuration of the assigned simulation engineer. The configurations are evaluated based on the costs and the ability to meet the throughput. Based on the performance on multiple old sales projects, the agent could have outperformed, matched performance or underperformed compared to the sales engineer.

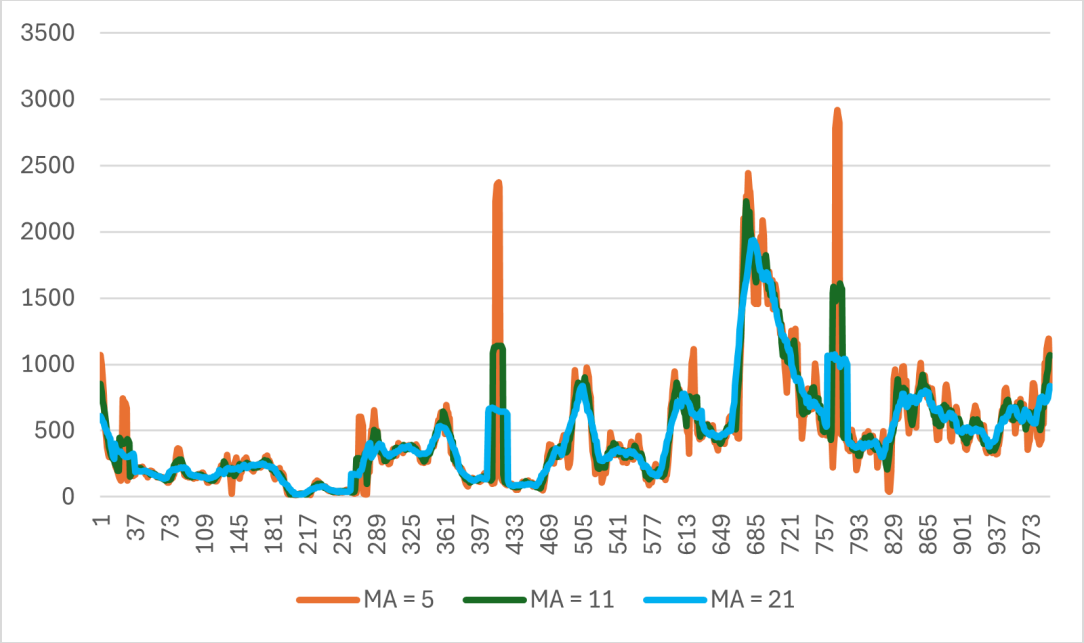
Unfortunately, the base model did not demonstrate satisfactory performance. In practical tests, the agent repeatedly selected actions that decreased warehouse attributes, often getting stuck in loops by continuously selecting a decrease and then an increase of the same variable. This leaves the model unable to get a configuration that satisfies the requirements and even a viable policy, which leads to configurations that satisfy the goal. Due to the inability of the agent to move the starting configuration to a configuration that meets the goal, it is difficult to know how far the agent is from a policy that does produce a result. It failed to recognise the necessity of meeting predefined throughput targets based on input parameters. It does not grasp the necessity of reaching a predefined throughput when presented with input data. There-

fore, the base model is not eligible for a comparison against previous sales projects of Vanderlande. Therefore, there is no final model for this research and leaving the conclusion that the current approach was not satisfactory to achieve a functioning model. Nevertheless, the benefit of a functioning model remains clear and thus, this section covers the different experiments conducted to aid further research in the development of a functioning model.

To better understand the model's training dynamics, this section compares the experiments based on the loss function. The data of the loss function is volatile, limiting the ability to spot trends in the data and reach conclusions. Hence, a moving average of the data is taken to smooth the curve. Figure 6.1a shows the different number of samples of the MA, together with a zoomed version without the original loss function, as this line severely stretches the graph and limits the ability to draw conclusions. Figure 6.1b shows the moving average with 5 samples, which shows quite volatile behaviour and large peaks. A moving average of 21 samples removes most of the peaks and parts of the trend. Hence, the middle value of 11 samples is selected as the optimal value for the moving average, as the volatility is smoothened and trends are clearly visible. Other results presented in this chapter are also portrayed through the moving average of 11 values to make a fair comparison.



(a) Impact of different ranges of the moving average compared to the original data, showing the iterations on the x-axis and loss on the y-axis.



(b) Different ranges of the moving average, excluding the base graph, showing the iterations on the x-axis and loss on the y-axis.

Figure 6.1: Comparison of the size of moving average ranges to select an optimal range to portray results, showing the iterations on the x-axis and loss on the y-axis.

Taking a closer look at the performance of the training of the agent in Figure 6.2, the model does not stabilize and improve in the learning process. The model starts with an initial peak, which is to be expected as the agent does not have any experience with the environment yet. After the initial decrease in loss, it hit the best loss in the training process after 200 training steps. From this point, the loss increases consistently and fluctuates constantly, both indicators that the training of the agent is suboptimal.

Stopping after only 200 iterations in the training process is not advised, as the agent has such little experience with the environment that a deeper understanding and an optimal policy is not yet possible. To prove this, experimentation was performed similar to a full training run and it ran into identical issues as the agents that completed a full training run. The rectangular peaks in the graph are the values influenced by the huge peaks of loss visible in Figure 6.1a. The following sections cover the different hyperparameter settings to potentially improve the performance of the agent.

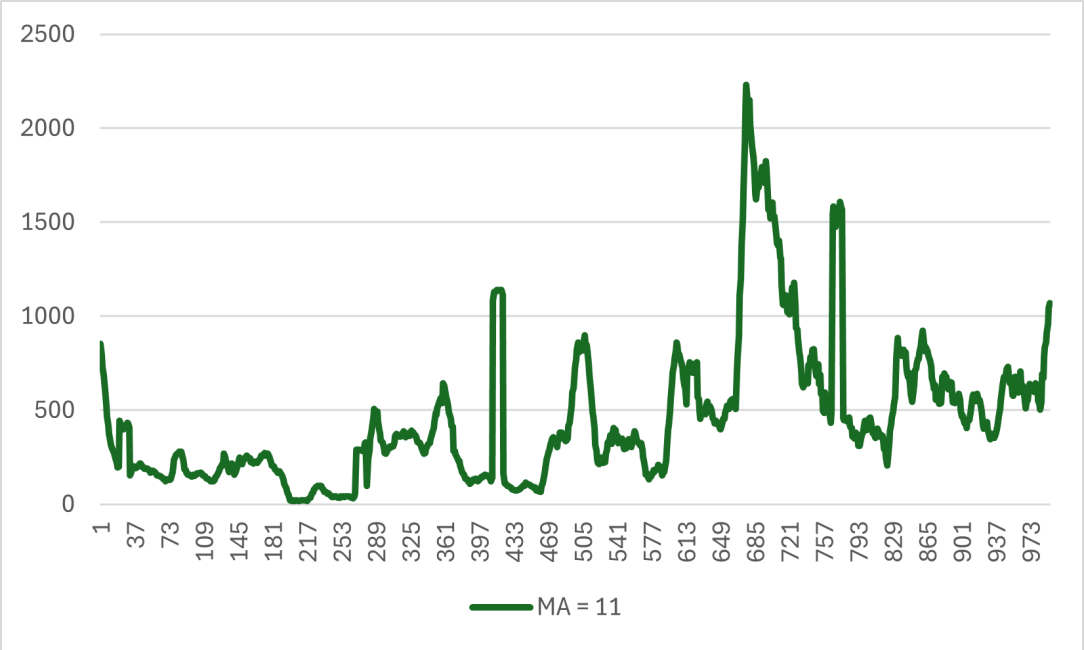


Figure 6.2: Moving average of the training loss of the agent during the training process, showing the iterations on the x-axis and loss on the y-axis.

6.1.1 Runtime

A hindering factor in the research was the run time of the model. The model has been run on a single NVIDIA RTX™ 3000 Ada Generation Laptop GPU on a HP ZBook Power 16 inch G11 Mobile Workstation PC. Due to the necessity of each iteration to run a simulation that required at least 1 hour of simulation time. Luckily, this is sped up to 40 seconds in real-time. Still, this severely limited the training length of the agent. As 1 episode can take up to 75 iterations, running a training loop of 75 iterations took 16.2 hours on average. Out of this time, 95% of the time was spent running the simulation to obtain the throughput for the new configuration. Therefore, the experiments performed are restricted by the time available for experimentation. The discussion reflects upon this fact and the limitations it has had on this research.

6.2 Reward Function

Section 5.1 covers multiple variations of the reward function found in the literature. This section covers each variation, where the best-performing reward function is utilised in the remaining experiments.

6.2.1 Experiments for α

The first variations of the reward functions have a set value of the negative incentive β=-1 and different variations of the positive reward when reaching the goal at α=1, α=100, α=10,000 to evaluate the height of the reward, such that the model can handle the sparse reward structure. Figure 6.3 shows the performance of the loss function during the training process. It becomes clear that the reward structure where α=10,000 performs extremely poorly. An important aspect to note is that when the rewards are increased, the Q-values also increase. So, a difference of 10 is much more impactful in the other scenarios than α=10,000. Nevertheless, the performance does not nearly equal the performance of the other values for α. This value can therefore be disregarded for further experimentation.

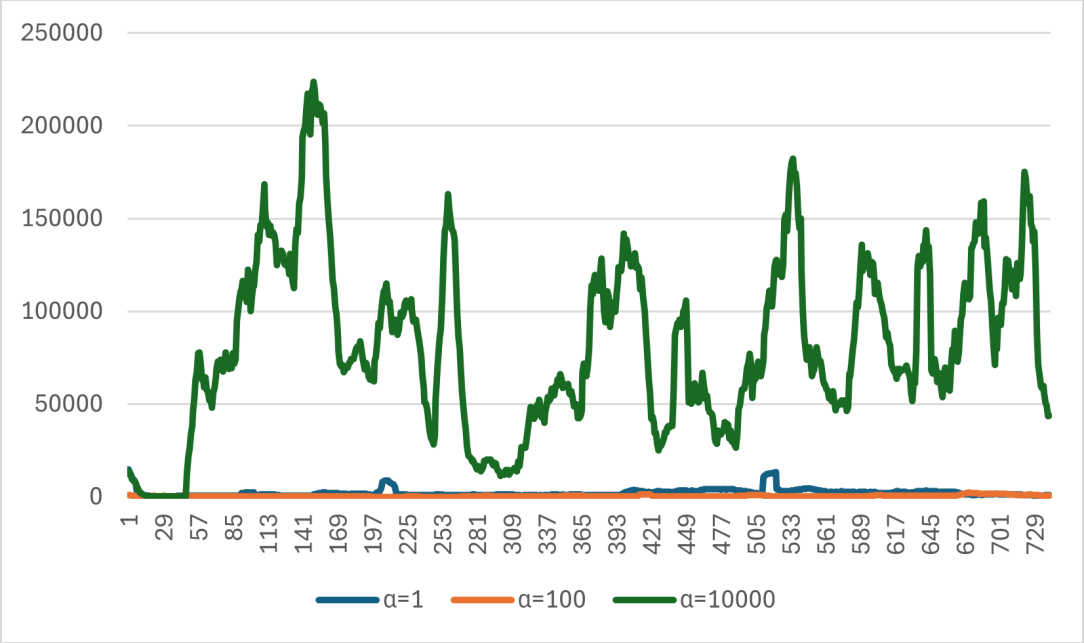


Figure 6.3: The moving average of the training loss for the different values of α in the reward function, showing the iterations on the x-axis and loss on the y-axis.

To better understand the performance of α=1 and α=100, a zoomed version of the graph is presented in Figure 6.4. This figure clearly shows the better performance of a value function with α=10 is more impactful. The evaluation of the loss for α=100 is continuously lower except for a few peaks and shows less volatile behaviour. This indicates a more stable and better learning process. Hence, for the remaining experiments, a value of α=100 is used. Therefore, the loss graph of α=100 is from here on used as the baseline for comparisons with different hyperparameters; this is therefore the value of α^*.

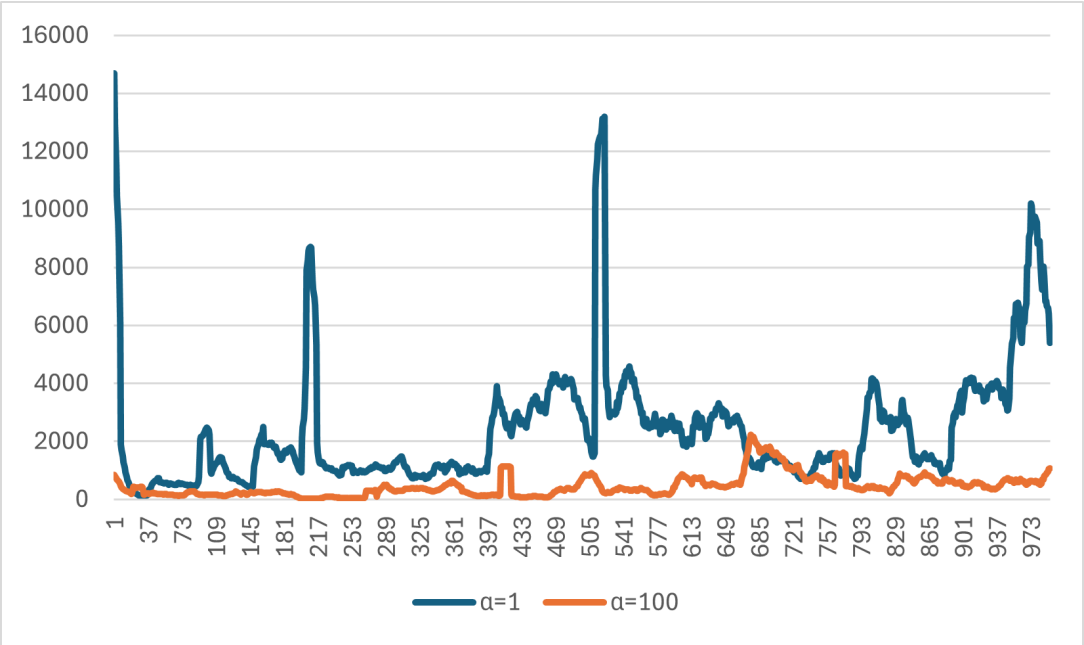


Figure 6.4: The moving average of the training loss of the two most promising values for α, showing the iterations on the x-axis and loss on the y-axis.

6.2.2 Experiments for β

The next experiment is setting the value of the negative reward for not reaching the goal β=0 instead of the -1 which was previously used. The -1 ensured that the agent takes the shortest path towards the optimal value. The discount factor (γ of the Bellman equation) is set to 0.99 to encourage this behaviour without the negative incentive.

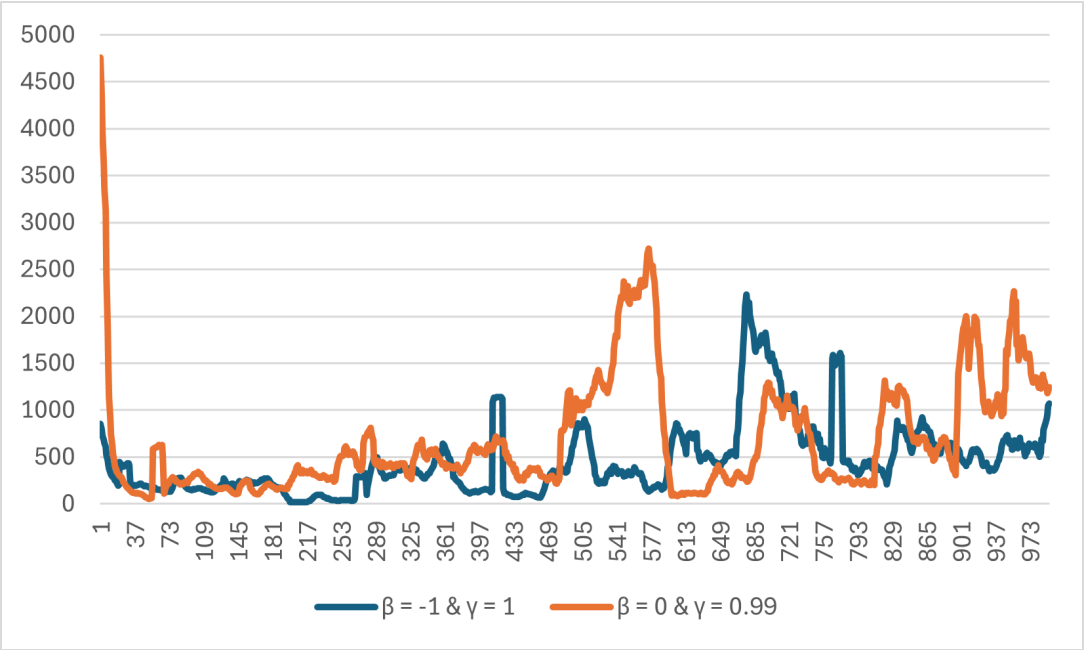


Figure 6.5: Comparison of the moving average of the training loss of base model with β=-1 and γ=1 with the training loss of the new reward function where β=0 and γ=0.99, showing the iterations on the x-axis and loss on the y-axis.

.

Figure 6.5 shows the performance of the new reward structure. The performance of the two reward structures do not differ significantly, but the β=-1 reward structure has a slight edge. The first 600 iterations, β=0 has equal or higher loss with a large peak at the end of this section. After which β=-1 also spikes, performing worse than β=0 up until 800 iterations, where the performance switches again. Overall, there is no clear argument for either to have the upper hand in performance. However, the agent with β=-1 has slightly better performance and lower peaks. Hence, the model with β=-1 and a discount factor of 1 is selected for further experimentation; this is therefore the value of β^*.

6.2.3 Reward Shaping

The final experiment involving the reward function is the addition of the reward shaping formula F_1 and F_2 from Section 5.1. The first shaping function F_1 gives an incentive to move towards the optimal value by giving a reward for an increase in throughput is added. Figure 6.6 shows the performance of the model with the intermediate reward function. The loss of the new model is massive compared to the base model. This shows that the current F function implementation is unsuitable for the agent. The base function shown in Figure 6.2 is no longer recognised as the difference is so vast, the base model appears as a flat line. The ratio between α and β determined in earlier experimentation does not translate to the ratio that determines the range of the intermediate reward. Additional experimentation is required to find a range for the intermediate reward such that it properly guides the agent's policy towards an optimal policy.

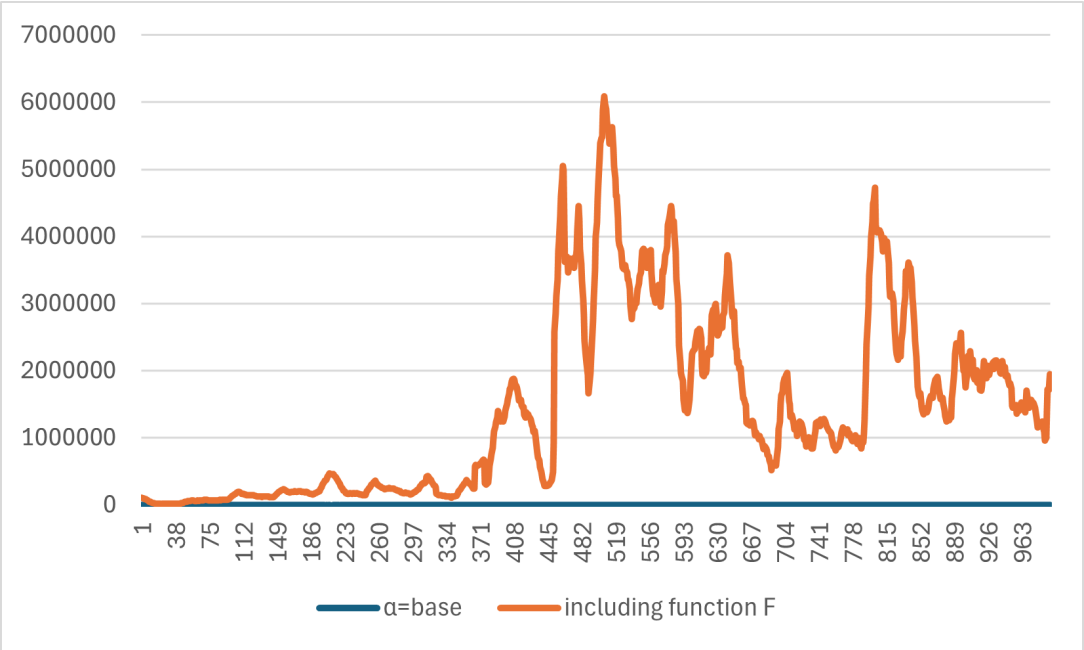


Figure 6.6: Comparison of the moving average of the training loss of the base model without an intermediate reward function F_1 and the model with an intermediate reward shaping function F_1, showing the iterations on the x-axis and loss on the y-axis.

The second reward shaping function F_2 punishes the agent more if the action taken leads to an illegal state, a state where a variables falls outside the predetermined ranges. Figure 6.7 shows the performance of the agent with the reward shaping function compared to the base agent, both lines are the moving average of 11 values. The figure shows that the loss of the two agents does not differ greatly. Nevertheless, the agent with the shaping function shows a higher loss than the base agent. Three different possibilities in the reward structure add additional complexity for the agent to grasp the prediction of Q-values. This could be an explanation for the higher loss during training. Therefore, no definite conclusion can be drawn regarding better training performance. As the base agent shows lower loss during training, the shaping function F_2 is not used in further experimentation. Nevertheless, the shaping function should not be disregarded as a viable extension of the model in the case of further experimentation.

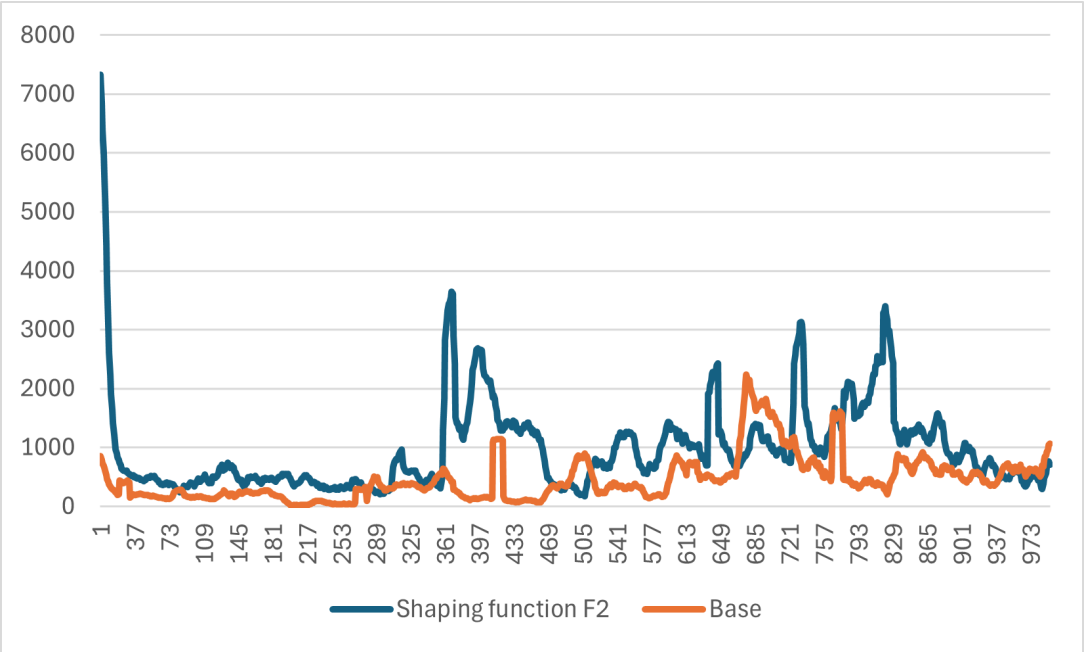


Figure 6.7: Comparison of the moving average of the training loss of the base model without an intermediate reward function F_2 and the model with an intermediate reward shaping function F_2, showing the iterations on the x-axis and loss on the y-axis.

6.3 ϵ-Greedy Strategies

The trade-off between the exploration of new options and the exploration of the most promising actions is determined by the ϵ-decay strategy. The ϵ_{decay} determines the rate at which the model moves from primarily exploring to primarily exploiting. To investigate the impact of different ϵ_{decay} rates on the learning stability of the agent, three ϵ_{decay} factors are evaluated, being 0.9, 0.99 and 0.999 as described in Section 5.2. Figure 6.8 presents the moving average of the loss for all three decay values. Figure 6.9 shows only the loss of 0.9 and 0.99 decay, as this offers a more precise comparison by excluding the more volatile 0.999 curve.

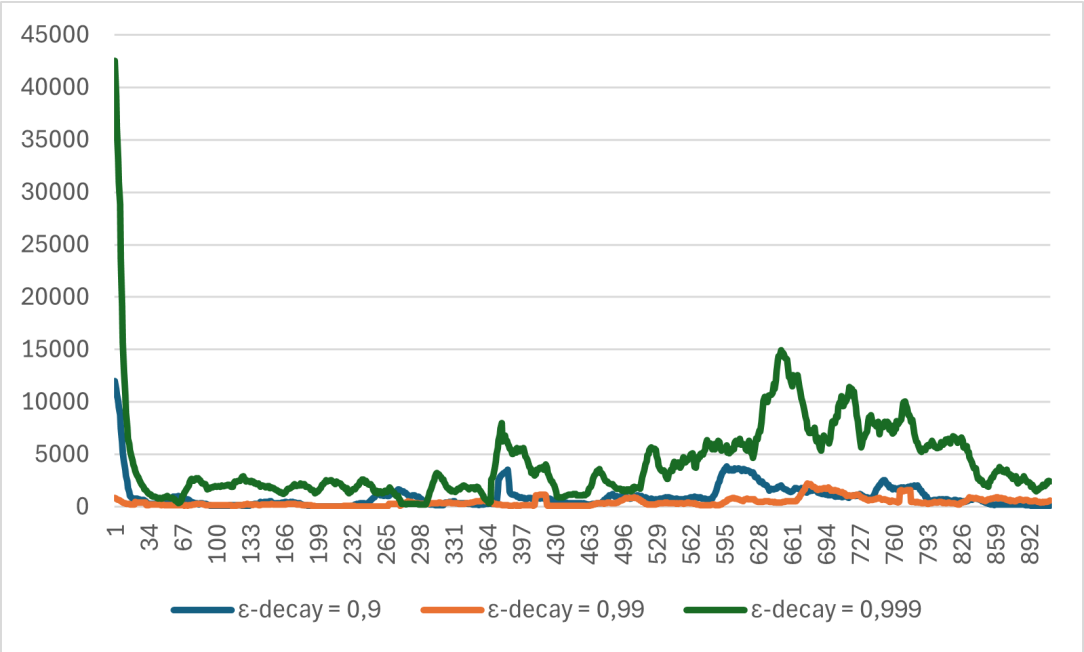


Figure 6.8: Comparison of the moving average of the training loss for the different values for ϵ_{decay}, showing the iterations on the x-axis and loss on the y-axis.

.

Figure 6.8 shows the loss graph of the $\epsilon_{decay} = 0.999$ shows notably unstable loss with multiple large spikes and a generally higher loss with higher variance during the full training run. The prolonged exploration and therefore delayed convergence cause learning instability. The curves of $\epsilon_{decay} = 0.9$ and $\epsilon_{decay} = 0.99$ are considerably smoother and result in lower loss. The figure is stretched due to extreme initial values and peaks for $\epsilon_{decay} = 0.999$. Hence, Figure 6.9 shows the same data, excluding $\epsilon_{decay} = 0.999$, to better analyse the behaviour of the more promising ϵ_{decay} values.

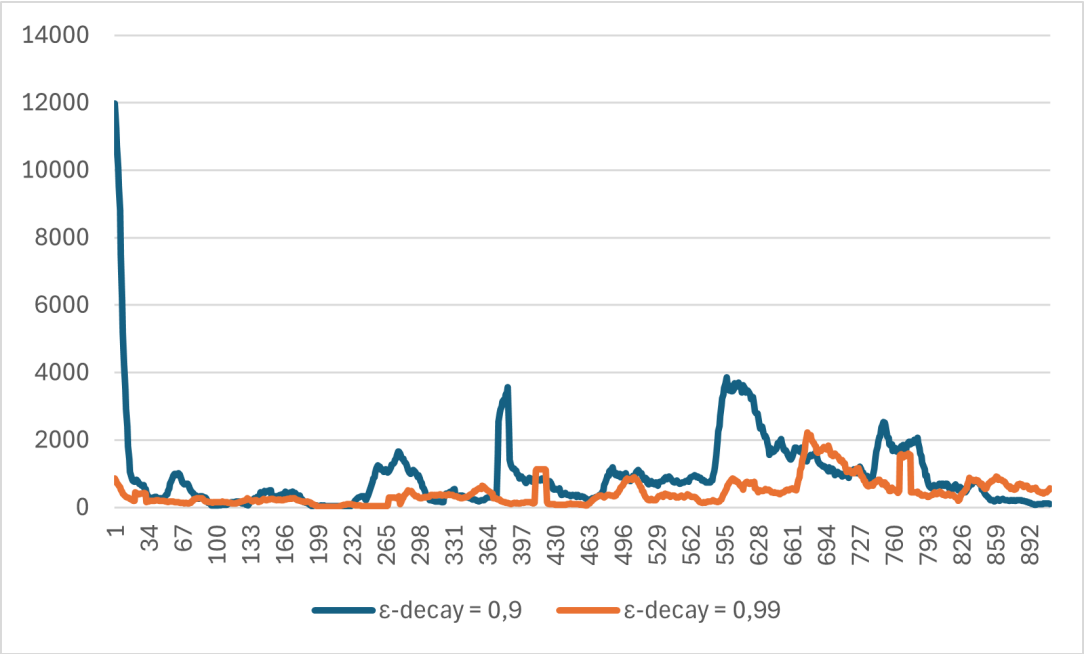


Figure 6.9: Zoomed version of the training loss comparison in Figure 6.8 comparing the two most promising values for ϵ_{decay}.

From Figure 6.9, it becomes evident that $\epsilon_{decay} = 0.99$ yields a more stable training process. Showing consistently lower loss values and a less volatile training process. Where $\epsilon_{decay} = 0.9$ shows more spikes in loss, especially between iterations 400-700. These results indicate the agents benefit from a ϵ_{decay} factor of 0.99 for the best training process compared to an overly aggressive strategy with $\epsilon_{decay} = 0.9$ or an overly conservative strategy with $\epsilon_{decay} = 0.999$.

6.4 Prioritised Experience Replay

The last hyperparameter evaluated in this research is the sampling method from the experience replay. The previous experiments used a uniform sampling method where each experience is equally likely to be sampled. The other method presented in Section 5.3 is Prioritised Experience Replay (PER), where the probability of selecting a sample is based on the learning potential of the sample. Figure 6.10 shows the performance compared to the base model. PER shows a considerably higher loss and volatile behaviour. In the first period of training, the model starts with a higher loss but appears to converge towards the policy of the base model. Nevertheless, after 500 iterations, the loss explodes and does not show signs of a stable policy, continuously spiking high losses. The model selects samples to calculate the loss based on samples from which the agent can learn a lot. By taking samples that have a high loss, the loss of following iterations is higher, but this should start converging towards a more optimal policy at some point. The graph does not show this, so in the current implementation of the model, PER does show improved performance.

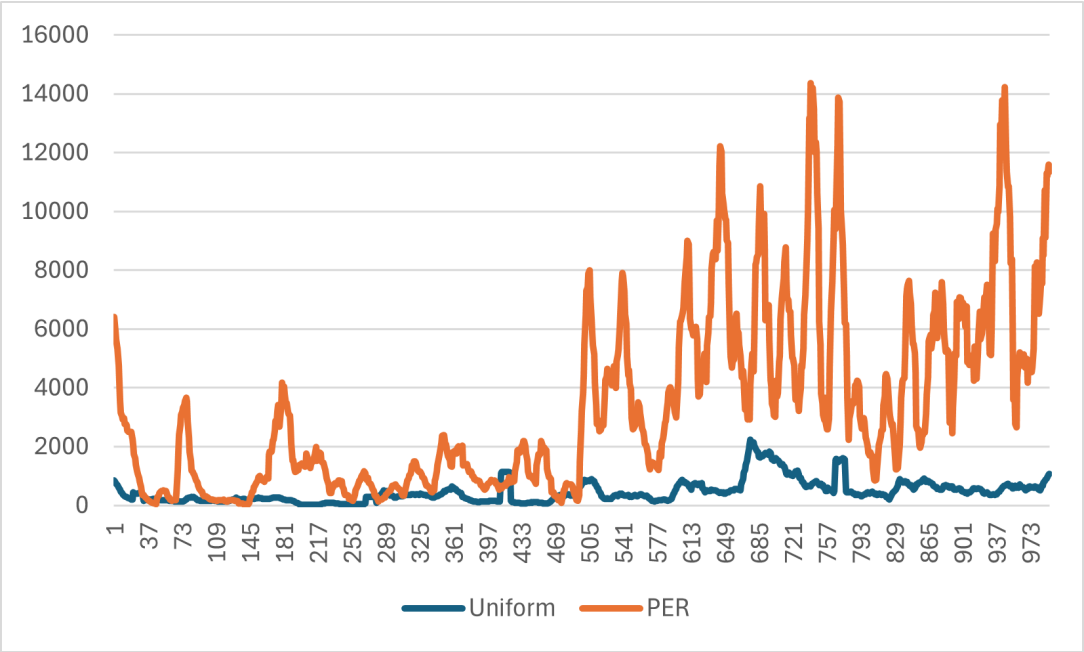


Figure 6.10: Comparison of the evaluation of the training loss between uniform samples from the replay buffer and Prioritised Experienced Replay (PER), showing the iterations on the x-axis and loss on the y-axis.

6.5 DDQN

The DDQN modifies the standard DQN by decoupling action selection and Q-value estimation using separate networks, thereby reducing overestimation bias. The training loss progression for both the DQN and DDQN is shown in Figure 6.11, where each curve represents a moving average to highlight general trends while smoothing local noise.

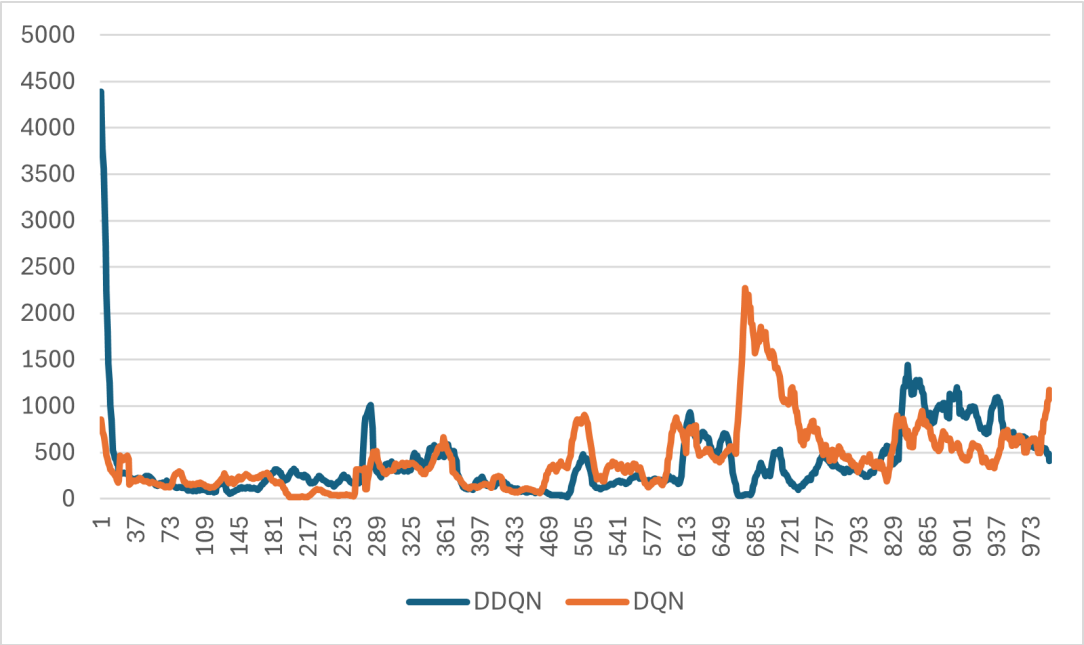


Figure 6.11: Comparison of the moving average of training loss between the base version of the DQN and a DDQN with identical hyperparameters, showing the iterations on the x-axis and loss on the y-axis.

Initially, the DDQN exhibits a notably higher loss than the DQN. However, this difference diminishes rapidly, and within approximately 10 iterations, the DDQN loss stabilises to a level comparable with the DQN. Beyond 450 iterations, both models start showing volatile spikes in the loss pattern. From this point, the DDQN shows a lower average loss than the DQN and has fewer spikes, suggesting relatively more stable training behaviour. Around 900 iterations, the DDQN briefly shows more loss than the DQN, but it dips below the DQN again at the end. While the DDQN shows modest improvements in loss stability over the DQN, the difference is not substantial enough to conclude a definitive performance advantage based on the loss metric.

6.6 Conclusion

This chapter evaluated various Deep Q-learning (DQL) configurations to identify the optimal hyperparameter settings, leading to an optimal policy of the agent. The agent did not reach a satisfactory policy to evaluate real sales cases of Vanderlande to test performance compared to experts. Nevertheless, the experimentation with the hyperparameters has led to meaningful insight into the hyperparameters used. A set of the best performing hyperparameters is shown in Table 6.1 that can be used for further research into the automation and optimisation of ADAPTO designs. The following chapter dives into the conclusions drawn from this research and the opportunities to move towards an optimal policy.

α	β	**Discount factor**	**Tuning Function F**	ϵ-start	ϵ_{decay}	ϵ-boundary	**Sampling method**	**Model Type**
100	-1	1	-	1	0.99	0.1	Uniform	DQN

Table 6.1: Table showing the optimal hyperparameters identified in this research.

Chapter 7

Conclusion, Discussion & Further research

7.1 Conclusion

This research is focused on developing an automation model for the design of warehousing solutions for Vanderlande. The order data from the customer serves as input for the simulation model of the ADAPTO system to determine a configuration of the ADAPTO system that satisfies the throughput requirements from the customer against minimal costs. A comprehensive literature review was performed to analyse the current best practices in warehouse optimisation tools, identifying a gap in solutions using ML for warehouse layout optimisation. A gap in the literature is identified to develop an optimisation tool capable of handling a diverse range of warehouse layouts compared to a tool that solely focuses on a single warehouse layout. In addition, this research has developed a Deep Q-Learning tool to optimise a diverse range of warehouse layouts to automate the design of ADAPTO systems.
Given the complexity of varying ADAPTO layouts and throughput requirements, DQL is selected as the most promising ML method, as literature shows the agent in a DQL is capable of handling diverse environments. An agent has been trained through interaction with the environment with different order patterns and throughput requirements, such that the agent has a good understanding of the ADAPTO configuration for different customer requirements.

Unfortunately, the model was unable to consistently generate an ADAPTO configuration that meets customer requirements and failed to converge to an optimal policy. However, significant research was conducted to optimise hyperparameters, and progress was made in identifying key factors that influence model performance. The optimal hyperparameter values evaluated in this research are shown in Table 6.1.

The results reveal that the reward function substantially impacted the agent's performance during training. The difference in loss between the best and suboptimal reward functions was significant, underscoring the importance of fine-tuning the reward function. Experimentation shows that the training performance increases when adding a negative incentive to states that do not satisfy the throughput goal, instead of giving them a value of 0 or an intermediate incentive in the form of throughput. Furthermore, as the model has a sparse reward structure, the height of the reward is very influential on the performance. A reward between 0 and 1 limits the ability

of the agent to find states that yield a positive reward, as it is disproportional to the negative incentive. A reward in the range [0, 10000] causes an imbalance, resulting in even higher loss. The best performance was achieved with a reward in the range [0, 100]. Additionally, the ϵ-decay strategy with $\epsilon_{decay} = 0.99$ outperformed other values of ϵ_{decay}, resulting in lower loss. Adding Priority Experience Replay and a DDQN did not lead to improved performance in training.
In conclusion, this research contributes to developing a machine learning-based tool for warehouse layout optimisation, though the model's suboptimal performance suggests that additional refinements are needed. The hyperparameter tuning insights and the challenges encountered provide valuable directions for future work, focusing on refining the reward function and exploring advanced strategies for achieving better convergence.

7.2 Discussion

The Deep Q-Learning model developed in this research did not reach a policy that was able to handle the optimization and automation problem proposed. The complexity of a non-functional Deep Reinforcement learning model is the black box characteristic of the Deep Learning aspect. The neural network responsible for the prediction of Q-values and therefore action selection is difficult to analyse. Identifying the exact reason why the model does not perform as expected is very complex due to its black box nature. The constant shift in the throughput level where the reward shifts from a negative to a positive reward in combination with the many different order patterns could be impossible for a standard DQL to predict accurately. This section dives into the aspects of the model that could have been developed differently, potentially resulting in improved agent performance.

Runtime

The performance of the Deep Q-Learning model was significantly impacted by the limited runtime. Due to the extensive time required for each simulation, the number of training episodes was constrained. One experiment of 50 episodes, resulting in 1100 iterations on average, took over 16 hours. In general, a DQN benefits from additional runtime, giving it more iterations to interact with the environment and improve its strategy. Take the research of Mnih et al. (2015) for example, the DQN was trained on 50.000.000 different frames. Far more than the agents in this research. Running the model for a longer period of time could improve the performance significantly. Unfortunately, the time was not there within this research and there is no definitive answer regarding this improvement. The loss of the training does show the trouble the DQN has with converging to minimal loss and there is no steady downward trend in the loss, which usually indicates performance increases with more training episodes. Nevertheless, it is a clear downside that the training time was limited for each experiment.

Variable Goal

A large concern during the development of the model is the variable goal of the model. A known obstacle for a neural network with a sparse reward structure is the difficulty

of the agent finding the goal, as a positive reward is only obtained upon reaching this goal. The number of iterations with a positive immediate reward is very low compared to a negative immediate reward. However, this model adds additional complexity by varying the goal for each episode. Resulting in a different point at which the immediate reward is positive. The neural network is fed the current throughput and the goal as input values and the hypothesis was that the agent uses this information to adapt its Q-values based on this input. The reality is that the model has not done this. A cause of this could be the limited training time of each model, and the agent has been trained too little to understand the importance of the goal input. However, it could also be the case that the modelling approach is not complex enough to handle this or the structure of the neural network should be different, such that more focus lies on the goal. All potential causes of the model not converging towards an optimal strategy. A potential improvement could be the addition of more KPI's that give feedback on performance.
Chapter 2 shows KPI's that are used by sales engineers to gain insight into which areas perform well and which areas have room for improvement. For example, a very high lift utilisation indicates more lifts are required, or a high waiting time for shuttles indicates a crowded setup, so some shuttles should be removed. These numbers only indicate performance and do not translate to a clear goal. Adding these variables to the neural network could give the agent more information. Another option is to add these variables into the intermediate reward function such that good performance is not only based on throughput but also on other performance indicators. The addition of the KPI's into the model might aid the agent in a better understanding of the current state and the goal, helping with the ability to find the dynamic goal. Furthermore, reducing the runtime would enlarge the number of experiments and the size of the experiments. So, running the simulations in the cloud or using other computationally powerful options reduces the simulation time of the model.

Additional Hyperparameter tuning

The experimentation of the model has tested many different values for the model's hyperparameters. However, there are two ways in which this process could have been improved. First, more values for the numeric variations can be tested. Currently, the experiments use several values extracted from literature that are scaled in such a way that they give a clear view of what range the optimal value should be in. However, with more experiments, an exact optimal value could have been identified. For example, the value of α has been set at 100 as it performed better than 1 and 100. However, a value of 80 or 150 could have resulted in even better performance. Especially for the intermediate reward function. This resulted in poor performance on the loss function. While more guidance in the performance should result in better results. So more experimentation on the ratio in ranges where the values for α and β lie, as well as a more advanced intermediate reward function.

Furthermore, more runs of the same experiments with different seed values would have given a better estimate of the performance, as it removes a bit of randomness from the runs. However, the run time limited this option for this research. If 5-fold experimentation was used, each run would have taken 80 hours. This was not an option with 10 different experiments, as this did not fit within the time frame available.

The second aspect was that each experiment was tested separately. The model performance was compared to the base model to independently evaluate each hyperparameter. However, hyperparameters also impact each other, so individual testing is sub-optimal. For example, the DDQN could have benefited from a higher value for α as it does not have the overestimation which was shown for the DQN. So testing each value of each hyperparameter against each other would have resulted in the optimal collaboration of hyperparameters for model performance. However, this would have resulted in 60 experiments compared to the current 10. And again, this is limited by the run time of the model. Overall, more in-depth research on the specific hyperparameter setup for this Deep Learning problem could have improved the performance of the model, but was limited by the large run time.

Increased Solution Space

Chapter 2 describes the ADAPTO system and the variables of the system which are used within this research. Furthermore, the scope of the research has focused on the simple cases for the simulation department and leaving the special cases to the sales engineers. Removing these constraints from the research will result in an even larger solution space as new variables are added and the ranges of the variables and parameters are increased. If this increased solution space were included in the research, the complexity of the problem at hand would increase even more. Since the current model is not able to accurately predict ADAPTO configuration, increasing the solution space will most likely not result in a functioning model. Therefore, the research should first focus on creating a DQL model which can handle the simple simulation cases with the limited set of variables. However, if the model has been configured in a way that is able to predict these configurations, the model should be able to handle a larger solution space. The current complexity most likely lies in consistently identifying the point at which the throughput goal is reached, which requires changes in the model. If a model is able to consistently identify this point, it could also do this for an increased solution space. Nevertheless, this will result in increased training time and if more KPI's are involved in the reward function of the model, KPI's related to the new variables must also be added. In conclusion, if a functioning model is presented with the current solution space, it should be possible to increase the solution space without drastic changes to the model.

7.3 Further Research

The adaptations of the model proposed in Section 7.2 can improve the performance of the model developed in this research. It is, however, not a guarantee that these improvements can result in a model that reaches the desired performance. It could be the case that the dynamic goal is too difficult for the model to handle, and a more complex model is required to solve this problem. This section proposes a couple of models that can result in improved performance and potentially an agent that can handle dynamic goals. The development of these models serves as further research.

DDQN

As stated in Section 5.4, the DDQN is a common approach to improve a DQN by limiting the overestimation that a DQN is prone to. However, only a simple adaptation of the DDQN was implemented. The results showed that no significant improvement was obtained by the implementation. Nevertheless, hyperparameters were not optimised and different variations of the DDQN could have improved performance. Wang et al. (2016) proposes a DDQN with a new duelling neural network architecture leading to dramatic improvements over existing methods in DQL. Hessel et al. (2017) evaluates many of the most common DQL methods and combines all the extensions to one model, a rainbow model. Clearly outperformed each individual model in the experimentation. Fortunato et al. (2019) proposes a new exploration strategy by introducing noise into the weights of the neural network. This concept is promising and explained in Section 7.3 in more detail.

NoisyNet

The $\epsilon - greedy$ exploration strategy is the most common strategy in deep reinforcement learning to determine the trade-off between exploration and exploitation. Fortunato et al. (2019) proposes a different approach called NoisyNet. NoisyNet introduces noise into the neural network's weights, encouraging more effective exploration of the environment. This allows the agent to explore more diverse strategies and reduces the risk of getting stuck in local optima. Since the agent was prone to continuously suggesting the same strategy. Fortunato et al. (2019) suggests the NoisyNet performs better in complex environments where $\epsilon - greedy$ might fall short. Hence, this modelling type might be suitable to explore in further research.

Supervised ML

A final suggestion for further research is to switch the Machine Learning approach from Reinforcement Learning to supervised learning. The most promising method to automate the design process is RL due to the agent forming an optimal policy. However, as the optimal policy was not achieved in this research, it could be achieved by switching to a completely different approach. Instead of taking an action at each iteration, a neural network could be trained to estimate a value for each variable directly. So the neural network could directly predict a number of aisles, a number of levels, etc. As the throughput is always known through the simulation software, supervised learning should be explored and not unsupervised learning. However, the other improvements suggested in the discussion and further research should first be explored, as the literature suggests this is the more suitable approach.

7.4 Recommendations

The discussion and further research have identified aspects of the model that can be improved such that a functioning model could be achieved. This section goes into the practical recommendations for Vanderlande and what steps could be taken such that Vanderlande benefits from this research. First, the implementation of a functioning automation and optimisation model for early design choices is still a development that

can greatly benefit Vanderlande by automating the design of less complex ADAPTO systems. To achieve a functioning model, the first step is to test versions of the more complex models suggested in Section 7.3 to identify whether they result in increased performance compared to the basic Deep Q-learning (DQL) model. In addition, the training time was a concern during this research, so the training of the models that are developed further must be increased to ensure the performance is not limited by the runtime. If these experiments are succesful, the best performing models should be evaluated further through hyperparameter optimisation. This is the next step Vanderlande should take and based on the results, further steps can be identified based on the performance. Potentially leading to a functioning DQL model that lowers the workload for the simulation department at Vanderlande, allowing the department to focus on complex simulation requests.

Appendix A

Nr mentions of Machine Learning on Sciencedirect

This figure shows the number of mentions of Machine Learning on Sciencedirect, to illustrate the rise in research in this area.

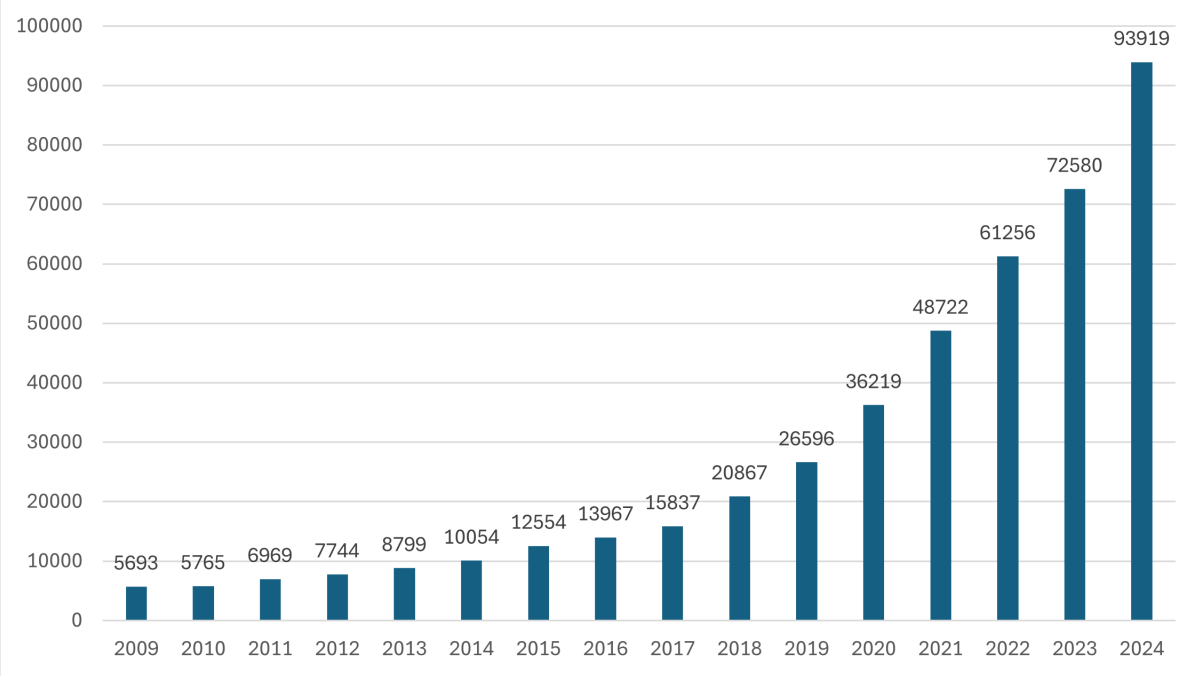


Figure A.1: Number of mentions of ML on ScienceDirect
source:ScienceDirect

Appendix B

Full code

This appendix contains the publishable parts of the code

B.1 Class DeepLearning

The following snippets of code compile the class DeepLearning together. Which is called by Listing B.11 and returns the trained model. The following sections describe each funtion of the model

B.1.1 Init

Listing B.1: Function which initialises all variables used throughout the DeepLearning Class

```
def __init__(self, gamma, epsilon, epsilonDecay, numberEpisodes,
   states, actions, MinMax, directory, velocity_classes,
   tsu_distribution, tsu_range,tsu_group_distribution,
   smart_lift_allocation,sequencing):
        #create an environment here with parameters & variables
        #parameters from input
        self.gamma = gamma
        self.epsilon = epsilon
        self.NumberEpisodes = numberEpisodes
        self.states = states
        self.nr_states = len(states)
        self.actions = actions
        self.nr_actions = len(actions)
        self.MinMax = MinMax
        self.epsilonDecay = epsilonDecay
        self.smartliftallocation = 0
        self.goal = states[14]
        self.throughput=1
        self.nrOfIterations=0


        #basic Deep Q-Learning parameters
        self.sizeReplayBuffer = 100
```

```
        self.sizeBatchReplayBuffer = 50
        self.updateTargetNetwork = 10
        self.counterUpdateTargetNetwork = 0
        self.replayBuffer = []
        self.predictionNetwork = self.createNetwork()
        self.targetNetwork = self.createNetwork()

        # Explicitly build the model to initialize weights
        self.predictionNetwork.build(input_shape=(None, self.
            nr_states))
        self.targetNetwork.build(input_shape=(None, self.
            nr_states))
        self.targetNetwork.set_weights(self.predictionNetwork.
            get_weights())
        self.actionsAppend = []
        self.EpsilonBoundary =0.1
        self.counter = 0
        self.sumRewardsEpisode = []

        #SETUP WRITING AND READING FILES
        # Parameters
        self.velocity_classes = velocity_classes
                                # normal, uniform
        self.tsu_distribution = tsu_distribution
                                  # normal, uniform, bathtub
        self.tsu_range = tsu_range
        self.tsu_group_distribution = tsu_group_distribution
                           # normal, uniform, negexp
        self.smart_lift_allocation = smart_lift_allocation
                             #false, true
        self.sequencing = sequencing

        self.starting_input_location = os.path.join(directory,"
            input_default")
        self.new_input_location = os.path.join(directory,"
            input_new")
        self.new_output_location = os.path.join(directory,"Output
            ")
        self.bat_file_path = -
        self.inbound_location = os.path.join(directory,"Output\
            InboundThroughput.txt")
        self.outbound_location = os.path.join(directory, "Output\
            OutboundThroughput.txt")
        self.write_files = write_input(self.new_input_location)
        self.read_output = read_output(self.new_output_location)
        self.CalculateCosts = calculateCosts()

        #Additional variables
        self.count = 0
        self.begin_counter =0
        self.startTime= time.time()
```

```
        self.timeSum = 0
        self.timeAvg = 0
        self.nrSimulations = 0
        self.losses = []  # Store loss values
        self.WrongConfiguration = pd.DataFrame(columns=[f'Column_
           {i+1}' for i in range(len(states))])
        self.indexEpisode=0
```

B.1.2 Call Java

Listing B.2: This function ensures the correct directories are set, writes the new state in the directory such that the Java model can call upon them. Then the Java model is called to run the simulation and finally the throughput is calculated of the simulation

```
def call_java(self,states):
        try:
            # update variables
            feasible = False
            count = 0
            # create maps & files from default
            while feasible == False:
                #setup files based on the state
                shutil.rmtree(self.new_input_location,
                   ignore_errors=True)
                shutil.rmtree(self.new_output_location,
                   ignore_errors=True)
                shutil.copytree(self.default_input_location, self
                   .new_input_location)
                os.makedirs(self.new_output_location, exist_ok=
                   False)

                # update files
                self.write_files.Call_writing(states, self.
                   smartliftallocation)

                #call java
                env = os.environ.copy()
                env["ADAPTO_INPUT_FOLDER"] = self.
                   new_input_location
                env["ADAPTO_OUTPUT_FOLDER"] = self.
                   new_output_location
                env["DISABLE_VISUALISATION"] = "true"

                #run java and store simulation time
                startSim = time.time()
                subprocess.run(self.bat_file_path,env = env)
                endSim = time.time()

                #calculate averages
```

```
                self.nrSimulations += 1
                self.timeSum += (endSim-startSim)
                self.timeAvg = self.timeSum/self.nrSimulations

                #read output
                read = read_output(self.inbound_location)
                output = 0.95*read.read_files(self.
                   inbound_location, self.outbound_location)
                if count == 10:
                    feasible = True
                elif self.throughput == 0:
                    states_df = pd.DataFrame([states], columns=
                       self.WrongConfiguration.columns)

                    # Use pd.concat to add the new row to the
                       DataFrame
                    self.WrongConfiguration = pd.concat([self.
                       WrongConfiguration, states_df],
                       ignore_index=True)

                    count +=1
                    states = self.new_states(states)
                else:
                    feasible = True
            states[15]=output
            print(f"The output is {output}")
            return output,states
        except:
            return 0,states
```

B.1.3 New States

Listing B.3: This section is responsible for configuring a new state when called upon

```
def new_states(self,states):
        #In case of an incorrect simulation, a new configuration
           is presented. No trouble shooting in the java file is
           possibl during training so we must provide a new
           configuration

        self.goal = random.randint(1500, 6000)
        states[14] = self.goal


        #reset throughput for a new run
        self.throughput = 1
        terminal_state = True
        while terminal_state:
            states[1] =np.random.randint(self.MinMax[1,0],self.
               MinMax[1,1]+1)
```

```
            states[0] =np.random.randint(max(self.MinMax[0,0],
               states[1]),self.MinMax[0,1]+1)
            states[2] =np.random.randint(self.MinMax[2,0],self.
               MinMax[2,1]+1)
            states[3] =np.random.randint(self.MinMax[3,0],self.
               MinMax[3,1]+1)
            states[4] =np.random.randint(self.MinMax[4,0],self.
               MinMax[4,1]+1)
            states[5] =np.random.randint(self.MinMax[5,0],self.
               MinMax[5,1]+1)
            states[7] =np.random.randint(self.MinMax[7,0],states
               [1])
            states[6] = min((math.floor(250/states[0])),(states
               [1]-1),(np.random.randint(self.MinMax[6,0],states
               [7]+1)))
            if self.is_possible(states) == True and self.
               is_terminal_state(states)==False:
                terminal_state = False
        return states
```

B.1.4 Costs & Reward

Listing B.4: These functions are called to calculate the costs of the configuration and thereby the reward as well as the calling the java function to obtain a throughput to see which reward function should be used

```
def calculate_costs(self,states):
        #costs are calculated in a seperate file, as this
           calculation contains sensitive data
        costs = self.CalculateCosts.calculate(states)
        return costs

    def get_reward(self,states):
        #obtain the throughput of the new state
        throughput,states = self.call_java(states)
        #obtain costs of new state
        costs = self.calculate_costs(states)
        #get the minimum and maximum costs, due to
           confidentiality this is not included in this file
        min_costs, max_costs = self.CalculateCosts.GetMinMax()

        #calculate reward
        if throughput >= self.goal:
            reward = 100 *(1- (costs -min_costs)/(max_costs-
               min_costs))
        else:
            reward = -1
        return reward,states
```

B.1.5 IsTerminal & IsPossible

Listing B.5: These two functions are called to check the state and action of the model. The IsTerminal function results whether a state is terminal and IsPossible check the state against the constraints

```
def is_terminal_state(self,states):

        #check whether the model has reached a terminal state
        if states[15] >= self.goal:
            #state is terminal if the goal is reached
            print("Terminal stage is reached through the goal,
               new configuration is prepared")
            return True

        if self.nrOfIterations == 75:
            #state is terminal if max number of iterations is
               reached
            return True
        else:
            #if no terminal state is reached, number of
               iterations is increased
            self.nrOfIterations +=1
            return False


    def is_possible(self,states):
        #The research states several constraints, this function
           serves as a double check that no constraints are
           breached
        if states[1] < states[6]: #number of shuttles cannot be
           more than the number of aisles
            return False
        elif states[0]*states[6]>250:   #total number of shuttles
            cannot exceed 250
            return False
        elif states[1]<=states[7]: #number of lifts cannot be
           more than the number of aisles
            return False
        elif states[7] <= states[6]: #number of shuttles cannot
           be more than the number of lifts (probably makes #1
           redundant but is there for certainty)
            return False
        else:
            return True
```

B.1.6 Get Starting Configuration

Listing B.6: This function is called to obtain a starting configuration for the model which does not violate any constraints or is already a terminal state

```
def get_starting_configuration(self, states):
        #First a new set of parameters is determined at the start
            of an episode
        states[8] =int(self.velocity_classes[np.random.randint(
           len(self.velocity_classes))])
        states[9] =int(self.tsu_distribution[np.random.randint(
           len(self.tsu_distribution))] )
        states[10]=int(self.tsu_range[np.random.randint(len(self.
           tsu_range))])
        states[13]=int(self.sequencing[np.random.randint(len(self
           .sequencing))])
        self.default_input_location = os.path.join(self.
           starting_input_location, f"default_{int(states[8])}_{
           int(states[9])}_{int(states[10])}_0_0_{int(states[13])
           }")

        #determine a goal, can adjust parameters
        self.goal = random.randint(500, 6000)
        self.nrOfIterations = 0
        states[14] = self.goal
        #change this to propper variables
        self.begin_counter+= 1
        print(f"Start of configuration {self.begin_counter} with
           the goal {self.goal} with an average simulation time
           of {self.timeAvg}")
       #store model every 10 iterations
        if (self.begin_counter) % 10 == 0:
            self.predictionNetwork.save(-)
            self.targetNetwork.save(-)
            print("Models are saved")
        terminal_state = True
        while terminal_state:
            #initial values for the variables is taken, according
                to the constraints
            states[1] =np.random.randint(self.MinMax[1,0],self.
               MinMax[1,1]+1)
            states[0] =np.random.randint(max(self.MinMax[0,0],
               states[1]),self.MinMax[0,1]+1)
            states[2] =np.random.randint(self.MinMax[2,0],self.
               MinMax[2,1]+1)
            states[3] =np.random.randint(self.MinMax[3,0],self.
               MinMax[3,1]+1)
            states[4] =np.random.randint(self.MinMax[4,0],self.
               MinMax[4,1]+1)
            states[5] =np.random.randint(self.MinMax[5,0],self.
               MinMax[5,1]+1)
            states[7] =np.random.randint(self.MinMax[7,0],states
               [1])
            states[6] = min((math.floor(250/states[0])),(states
               [1]-1),(np.random.randint(self.MinMax[6,0],states
```

```
                [7]+1)))


            self.throughput,states= self.call_java(states)
            states[15]=0
            #additional check is performed
            if self.is_possible(states) == True and self.
               is_terminal_state(states)==False:
                terminal_state = False
        return states[:]
```

B.1.7 Get Next Configuration

Listing B.7: This function is called when the agent has selected an action and the state must be updated according to the step size of the selected variable

```
def get_next_configuration(self,states, action_index):
        #change the correct state based on the action taken by
           the agent
        state_index = math.floor(action_index // 2)  # Determine
           which state to change
        #these are states with only 2 options,
        if state_index == 3 or state_index == 4 or state_index ==
            5:
            if action_index % 2 ==0:
                if states[state_index]==self.MinMax[state_index
                   ,0]:
                    states[state_index] += 1
                if states[state_index]==self.MinMax[state_index
                   ,1]:
                    states[state_index]-=1
        #The number of x positions are changed by 5
        elif state_index ==2:
            if action_index % 2 == 0 and states[state_index] >
               self.MinMax[state_index, 0]:
                states[state_index] -= 5
            elif action_index % 2 == 1 and states[state_index] <
               self.MinMax[state_index, 1]:
                states[state_index] += 5
        else:
            if action_index % 2 == 0 and states[state_index] >
               self.MinMax[state_index, 0]:
                states[state_index] -= 1
            elif action_index % 2 == 1 and states[state_index] <
               self.MinMax[state_index, 1]:
                states[state_index] += 1
        return states
```

B.1.8 Create Neural Network

Listing B.8: This function is responsible for the creation of the Neural Network used to predict QValues, it is only called at the start by the init and does not change throughout the run

```
def createNetwork(self):
        #the neural network is created through tensorflow
        model = Sequential([
            Dense(self.nr_states, activation='relu', input_shape
               =(self.nr_states,)),  # Input layer
            Dense(64, activation='relu'),  # More neurons for
               deeper learning
            Dense(128, activation='relu'),  # More neurons for
               deeper learning
            Dense(64, activation='relu'),
            Dense(self.nr_actions, activation='linear')  # Linear
                activation for Q-values
        ])
        model.compile(optimizer='adam', loss='mse')  # Mean
           Squared Error for Q-learning
        return model
```

B.1.9 Training

Listing B.9: This function is the main driver of the model and must be called after the init of the class to activate training. It runs through all the training episodes calling the other functions each time to obtain a trained model. After an action has been selected and a throughput and reward is calculated, the agent calculates the loss and updates the variables of the prediction network and target network accordingly. It also stores the loss and creates graphs for the loss function

```
def trainingEpisodes(self):
        #main driver of the model, which must be called in order
           to train the model

        for self.indexEpisode in range(self.NumberEpisodes):    #
           run over all episodes
            #initialise at start of each episodes
            rewardsEpisode=[]
            currentState= self.get_starting_configuration(self.
               states)
            terminalState = False
            while not terminalState:  #run an episode until
               terminal state is reached
                action, nextState = self.SelectAction(
                   currentState, self.indexEpisode)   #retrieve
                   action an new state s_t+1
                reward,nextStates = self.get_reward(nextState)
                   #calculate rewards of the new state
                terminalState = self.is_terminal_state(nextState)
                    #check if terminal state has been reached
                print(f"The reward of this episode is {reward}")
```

```
                rewardsEpisode.append(reward)   #update
                    experience replay
                self.replayBuffer.append((currentState, action,
                    reward,nextState,terminalState))
                if len(self.replayBuffer) > self.sizeReplayBuffer
                    :   #check if size of replay buffer is not
                    exceeded, if so, remove oldest value
                     self.replayBuffer.pop(0)

                if (len(self.replayBuffer)>self.
                    sizeBatchReplayBuffer):   #training is done if
                     a batch can be retrieved from the experience
                    replay
                     randomSampleBatch = random.sample(self.
                        replayBuffer, self.sizeBatchReplayBuffer)
                          #select sample based on uniform sampling
                     #initialise the arrays for the values of the
                        samples
                     currentStateBatch = np.zeros(shape=(self.
                        sizeBatchReplayBuffer,self.nr_states))
                     nextStateBatch = np.zeros(shape=(self.
                        sizeBatchReplayBuffer,self.nr_states))

                     for index, tupleS in enumerate(
                        randomSampleBatch):   #store the required
                        values of the samples in the arrays
                          currentStateBatch[index,:]=tupleS[0]
                          nextStateBatch[index,:]=tupleS[3]
                     #predict Q-values using the deep Q-model
                     QnextStateTargetNetwork = self.targetNetwork.
                        predict(nextStateBatch)
                     QcurrentStatePredictionNetwork = self.
                        predictionNetwork.predict(
                        currentStateBatch)

                     #store the networks based on the current
                        states
                     inputNetwork = currentStateBatch
                     outputNetwork = np.zeros(shape=(self.
                        sizeBatchReplayBuffer,self.nr_actions))
                     self.actionsAppend = []

                     #calculate reward with bellman equation
                     for index, (currentState,action,reward,
                        nextState,terminated) in enumerate(
                        randomSampleBatch):
                          if terminated:
                              y=reward
                          else:
                              y=reward+self.gamma*np.max(
                                 QnextStateTargetNetwork[index])
```

```
                        self.actionsAppend.append(action)
                        outputNetwork[index]=
                            QcurrentStatePredictionNetwork[index]
                        outputNetwork[index,action]=y

                    # Train model and get loss history
                    history = self.predictionNetwork.fit(
                        inputNetwork, outputNetwork, batch_size =
                        self.sizeBatchReplayBuffer, verbose=0,
                        epochs=5)

                    #Store the loss of the last iteration
                    loss = history.history['loss'][-1]
                    self.losses.append(loss)

                    """ Plot loss function over time """
                    plt.plot(self.losses)
                    plt.xlabel("Training Steps")
                    plt.ylabel("Loss")
                    plt.title(f"Loss Function over time at step {
                        self.counter}")
                    plt.grid(True)
                    plt.show()

                    #store loss
                    df = pd.DataFrame(self.losses)
                    df.to_csv(-)
                currentState=nextState
            self.sumRewardsEpisode.append(np.sum(rewardsEpisode))
            if self.epsilon > self.EpsilonBoundary:
                self.epsilon = self.epsilonDecay*self.epsilon
            else:
                self.epsilon = 0.1
        return self.timeSum
```

B.1.10 Select Action

Listing B.10: This function is the epsilon greedy aspects of the model and selects either a random action or the action which is expected to yield the best result

```
def SelectAction(self, state,index):
        #method to let the agent select an action based on the
            current state
        if index<1:
            #for the first iterations, take a random action as
                the model is not trained yet
            possible = False
            while possible == False:
                index=np.random.randint(self.nr_actions)
                NextState = self.get_next_configuration(state,
                    index)
```

```
                possible = self.is_possible(NextState)
            return index,NextState[:]
                    #epsilon greedy selection method
        randomNumber = np.random.random()
        #model explores the solution space by taking a random
           action
        if randomNumber < self.epsilon:
            possible = False
            while possible == False:
                index=np.random.randint(self.nr_actions)
                NextState = self.get_next_configuration(state,
                   index)
                possible = self.is_possible(NextState)
            return index,NextState[:]

        else:
            #model exploits the current knowledge of the
               environment and select the action which has the
               highest Q-value
            Qvalues = self.predictionNetwork.predict(state.
               reshape(1, self.nr_states))
            index=np.random.choice(np.where(Qvalues[0,:] ==np.max
               (Qvalues[0,:]))[0])
            NextState = self.get_next_configuration(state, index)
            return index,NextState[:]
```

B.2 Starting the Model

Listing B.11: This section is the part of the code where every module required is imported, the variables and parameter ranges are set and the class Deep Q Learning is called

```
import numpy as np
import random
import os
import shutil
import subprocess
import math
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential
from Write_input_files import write_input
from Read_output_files import read_output
import time
from costs import calculateCosts
import pandas as pd
import matplotlib.pyplot as plt
#Parameters which are not changed within the model
```

```
# Parameters
velocity_classes = [0,1,2]                          # normal,
   uniform
tsu_distribution = [0,1,2]                          # normal,
   uniform, bathtub
tsu_range = [1,5,10]


tsu_group_distribution = [0]                    # normal, uniform,
   negexp
smart_lift_allocation = [0,1]                     #false, true
sequencing = [0,1,2]                              #strict, relaxed,
    unsequenced

#Max_values of all variables for Q-tables
len_variables = [len(velocity_classes), len(tsu_distribution),len
   (tsu_range), len(tsu_group_distribution), len(
   smart_lift_allocation), len(sequencing)]

#values in this iteration
value_velocity_classes = 1
value_tsu_distribution = 1
value_tsu_range =1
value_tsu_group_distribution = 0
value_smart_lift_allocation =0
value_sequencing = 1

#storage of parameters for this run
states_parameters = [value_velocity_classes,
   value_tsu_distribution,value_tsu_range,
   value_tsu_group_distribution, value_smart_lift_allocation,
   value_sequencing]
#########################################################
#variables which can be changed within the model

#variables
min_warehouse_levels = 17
max_warehouse_levels = 41

min_warehouse_aisles = 4
max_warehouse_aisles = 25

min_warehouse_positions = 20
max_warehouse_positions = 150

min_warehouse_depth = 1
max_warehouse_depth = 2

min_lift_type = 1 #single platform
max_lift_type = 2 #double platform
```

```
min_in_rack_buffers = 2 #stands for 2 buffers
max_in_rack_buffers = 3 #stands for 3 buffers


min_nr_shuttles_per_level = 1
max_nr_shuttles_per_level = 30

min_nr_lifts = 2
max_nr_lifts = 30

#array creation
#add all min and max of the parameters
MinMax = np.array([
    [min_warehouse_levels,max_warehouse_levels],
    [min_warehouse_aisles, max_warehouse_aisles],
    [min_warehouse_positions, max_warehouse_positions],
    [min_warehouse_depth, max_warehouse_depth],
    [min_lift_type, max_lift_type ],
    [min_in_rack_buffers, max_in_rack_buffers],
    [min_nr_shuttles_per_level, max_nr_shuttles_per_level],
    [min_nr_lifts, max_nr_lifts]
    ])  # Adjusted to be within bounds (0 to max-1 for each axis)
states_variables = np.zeros((len(MinMax)))
#####################################################################
#parameters for Deep Q-learningl
# Parameters
gamma = 1
epsilon = 1
epsilon_decay = 0.99
numberEpisodes = 50
num_parallel_runs = 1 #number of cores I want to use
directory = -
#the different states of all variables, usually half of the
   number of actions
actions = [-1, 1] * len(states_variables)

goal = [0]
througput = [0]
states_temp = np.concatenate((states_variables, states_parameters
   ,goal))
states = np.concatenate((states_temp,througput))

#######################################



start_sim_time = time.time()
deep_q_learning = Deep_Q_Learning(gamma, epsilon, epsilon_decay,
   numberEpisodes, states, actions, MinMax,directory,
   velocity_classes, tsu_distribution, tsu_range,
   tsu_group_distribution, smart_lift_allocation,sequencing)
```

```
training_time = deep_q_learning.trainingEpisodes()


end_sim_time = time.time()
print(f"The total training time is {training_time} where the
    simulation time was {end_sim_time - start_sim_time}")
```

Bibliography

Abdalla, R., Hollstein, W., Carvajal, C., and Jaeger, P. (2023). Actor-critic reinforcement learning leads decision-making in energy systems optimization—steam injection optimization. *Neural Computing and Applications*, 35:16633–16647.

Arslan, B. and Ekren, B. (2022). Transaction selection policy in tier-to-tier SBSRS by using Deep Q-Learning. *International Journal of Production Research*, 61:21:7353–7366.

Berry, M. and Linoff, G. (1997). *Data mining techniques : for marketing, sales, and customer support*. Wiley.

Borovinsek, M., Ekren, B., Burinskiene, A., and Lerher, T. (2017). MULTI-OBJECTIVE OPTIMISATION MODEL OF SHUTTLE-BASED STORAGE AND RETRIEVAL SYSTEM. *TRANSPORT*, 32:120–137.

Chen, R., Yang, J., Yu, Y., and Guo, X. (2023). Retrieval request scheduling in a shuttle-based storage and retrieval system with two lifts. *Transportation Research Part E: Logistics and Transportation Review*.

De Kosten, R. and Roodbergen, K. J. (2007). Design and control of warehouse order picking: A literature review. *European journal of operational research*, 182.

Di, H. (2024). Design and research of automated warehouse simulation platform based on virtual visualization framework. *PeerJ Comput. Sci.*

Diaz, J., Rodriguez, H., Fajardo-Calderin, J., Angulo, I., and Onieva, E. (2024). A variable neighbourhood search for minimization of operation times through warehouse layout optimization. *Logic Journal of the IGPL*, 32(4).

Dong, B., Zhu, X., Yan, R., and Wang, Y. (2019). Development of Optimization Model and Algorithm for Storage and Retrieval in Automated Stereo Warehouses. *Journal Europeen des Systemes Automatises*, 52:17–22.

Dubey, S., Singh, S., and Chaudhuri, B. (2022). Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503:92–108.

Eder, M. (2020). An approach for a performance calculation of shuttle-based storage and retrieval systems with multiple-deep storage. *The International Journal of Advanced Manufacturing Technology*, 107:859–873.

Eder, M. (2022). An analytical approach for a performance calculation of shuttle-based storage and retrieval systems with multiple-deep and class-based storage. *Production & Manufacturing Research*, 10:321–336.

Eder, M. (2023a). An analytical approach of multiple-aisle shuttle-based storage and retrieval systems. *The International Journal of Advanced Manufacturing Technology*, 127:1585–1596.

Eder, M. (2023b). Analytical examination of shuttle-based storage and retrieval systems with multiple-capacity lifts. *The International Journal of Advanced Manufacturing Technology*, 133:5053–5064.

Ekren, B. (2017). Graph-based solution for performance evaluation of shuttle-based storage and retrieval system. *International Journal of Production Research*, 55:6516–6526.

Ekren, B. and Arslan, B. (2022). A reinforcement learning approach for transaction scheduling in a shuttle-based storage and retrieval system . *INTERNATIONAL TRANSACTIONS IN OPERATIONAL RESEARCH*, 31:274–295.

Ekren, B., Kaya, B., and Kucukyasar, M. (2022). Shuttle-Based Storage and Retrieval Systems Designs from Multi-Objective Perspectives: Total Investment Cost, Throughput Rate and Sustainability. *Sustainability (Switzerland)*, 15.

Ekren, B., Lerher, T., Kucukyasar, M., and Jerman, B. (2023). Cost and performance comparison of tier-captive SBS/RS with a novel AVS/RS/ML. *International Journal of Production Research*, 62:1648–1662.

Eschmann, J. (2021). Reward function design in reinforcement learning. *Reinforcement Learning Algorithms: Analysis and Applications*, 883:25–33.

Fan, J., Wang, Z., Xie, Y., and Yang, Z. (2020). A Theoretical Analysis of Deep Q-Learning. *Proceedings of Machine Learning Research*, 120:486–489.

Fedus, W., Ramachandran, P., Agarwal, R., Bengio, Y., Larochelle, H., Rowland, M., and Dabney, W. (2020). Revisiting fundamentals of experience replay. *arXiv*, 1.

Feng, G. and Zhong, H. (2023). Rethinking Model-based, Policy-based, and Value-based Reinforcement Learning via the Lens of Representation Complexity. *arXiv*.

Fortunato, M., Gheshlaghi, M., Piot, B., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, B., Pietquin, O., Blundell, C., and Legg, S. (2019). Noisy networks for exploration. *arXiv*, 3.

Foundation, F. (2025). Gymnasium documentation: Basic usage.

Fu, J., Kumar, A., Soh, M., and Levine, S. (2019). Diagnosing bottlenecks in deep q-learning algorithms. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2021–2030. PMLR.

Fu, M. and Gao, X. (2024). Application of EIQ-ABC Analysis in the Layout Planning of P E-Commerce Preposition Warehouse. *Advances in Transdisciplinary Engineering*, 48:519–528.

Ghasemi, M., Moosavi, A., Sorkhoh, I., Agarwal, A., Alzhouri, F., and Ebrahimi, D. (2024). An Introduction to Reinforcement Learning: Fundamental Concepts and Practical Applications. *arXiv*.

Gudivada, V., Apon, A., and Ding, J. (2017). Data quality considerations for big data and machine learning: Going beyond data cleaning and transformations. *International Journal on Advances in Software*, 10.

Ha, Y. and Chae, J. (2019). A decision model to determine the number of shuttles in a tier-to-tier SBS/RS. *International Journal of Production Research*, 57:963–984.

Han, M., Håkansson, J., and Rebreyend, P. (2014). How does data quality in a network affect heuristic solutions? *Working papers in transport, tourism, information technology and microdata analysis*. Retrieved from Högskolan Dalarna website.

Hasselt, H. (2010). Double q-learning. In Lafferty, J., Williams, C., Shawe-Taylor, J., Zemel, R., and Culotta, A., editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc.

He, P., Zhao, Z., Zhang, Y., and Fan, P. (2024). Optimization of Storage and Retrieval Strategies in Warehousing Based on Enhanced Genetic Algorithm. *IEEE Access*, 12:105703 – 105715.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. *arXiv*, 1.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. *Thirty-Second AAAI Conference on Artificial Intelligence*, 32(1).

Hoffman, R. and Asada, H. (2017). A Multi-Track Elevator System for E-Commerce Fulfillment Centers. *International Conference on Intelligent Robots and Systems*.

Hu, X. and Chuang, Y. (2023). E-commerce warehouse layout optimization: systematic layout planning using a genetic algorithm. *Electronic Commerce Research*, 23:97–114.

Ibrahim, S., Mostafa, M., Jnadi, A., and Osinenko, P. (2024). Comprehensive overview of reward engineering and shaping in advancing reinforcement learning applications. *arXiv*, 2.

Jang, B., Kim, M., Harerimana, G., and Kim, J. (2019). Q-Learning Algorithms: A Comprehensive Classification and Applications. *IEEE Access*, 7:1333653–133667.

Karsoliya, S. (2012). Approximating number of hidden layer neurons in multiple hidden layer bpnn architecture. *International Journal of Engineering Trends and Technology*, 3(6):714–717.

Kaufmann, T. and Pzhokhov (2019). baselines/baselines/deepq /deepq.py. Accessed: 2025-04-24.

KDnuggets (2025). Optimization algorithms in neural networks. Accessed: 2025-03-07.

Keras (2025). Optimizers. Accessed: 2025-03-07.

Kingma, D. and Ba, J. (2019). Adam: A method for stochastic optimization. *International Conference for Learning Representations*, 3(9):714–717.

Klar, M., Gatt, M., and Aurich, J. (2021). An implementation of a reinforcement learning based algorithm for factory layout planning. *Manufacturing letters*, 30:1–4.

Kosanic, N., Milojevic, G., and Zrnic, N. (2018). A Survey of literature on Shuttle Based Storage and Retrieval Systems. *FME Transactions*, 46.

Kovács, G. (2021). Special optimization process for warehouse layout design. *Vehicle and Automotive Engineering 3*, 3:194–206.

Kriehn, T., Scholz, F., Wehking, K., and Fittinghoff, M. (2018). Impact of Class-Based Storage, Sequencing of Retrieval Requests and Warehouse Reorganisation on Throughput of Shuttle-Based Storage and Retrieval Systems. *FME Transactions*, 46.

Laud, A. (2004). Theory and application of reward shaping in reinforcement learning. *Illinois Library*, 1:1–97.

Lawrence, S., Giles, C., and Tsoi, A. (1996). What size neural network gives optimal generalization? convergence properties of backpropagation.

Lehrer, T., Borovinsek, M., Ficko, M., and Palcic, I. (2017). Parametric study of throughput perfromance in SBS/RS based on simulation. *International Journal of Simulation Modelling*, 16:96–107.

Lerher, T., Ekren, Y., Sari, Z., and Rosi, B. (2015). Simulation analysis of shuttle based storage and retrieval systems. *International Journal of Simulation Modelling*, 14:48 – 59.

Li, M., Li, L., Zhang, C., Jiang, L. and. Liu, H., Lin, Z., and Wei, L. (2022). A Four-Way Shuttle Scheduling Method Based on Grey Wolf Algorithm. *China Automation Confress*.

Lin, L. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:193–321.

Lyu, Y., Ku, Y., and Ren, Q. (2021). Research on warehouse layout optimization under "double carbon" target. *"China Automation Congress"*.

López, O., López, A., and Crossa, J. (2022). Fundamentals of artificial neural networks and deep learning. *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, 1:379–425.

Manju, S. and Punithavalli, M. (2011). An Analysis of Q-Learning Algorithms with Strategies of Reward Function. *International Journal on Computer Science and Engineering*, 3:814–820.

Mayadunne, S. (2024). A multi-step mixed integer programming heuristic for warehouse layout optimization. *Supply chain analytics 8*.

Memarian, F., Goo, W., Lioutikov, R., Niekum, S., and Tocpu, U. (2021). Self-supervised online reward shaping in sparse-reward environments. *arXiv*, 3.

Miguel, J., Xiaodong, Q., and Xiuli, Z. (2020). E-commerce is reshaping the warehousing landscape – and it may impact disadvantaged communities. *INSTITUTE OF TRANSPORTATION STUDIES*.

Min, K. and Lim, D. (2023). Designing Automated LogisticsWarehouse Stackable Bidirectional Infinite-Loop Modules. *Appl. Sci.*, 13.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H. Kumaran, D., Wiestra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–533.

Naeem, M., Rizvi, S., and Coronato, A. (2020). A Gentle Introduction to Reinforcement Learning and its Application in Different Fields. *IEEE Access*, 8:209320 – 209344.

Nia, A., Haleh, H., and Saghaei, A. (2017). Dual command cycle dynamic sequencing method to consider GHG efficiency in unit-load multiple-rack automated storage and retrieval systems. *Computers & Industrial Engineering*.

Ohnishi, S., Uchibe, E., Yamaguchi, Y., Nakanishi, K., Yasui, Y., and Ishii, S. (2019). Constrained Deep Q-Learning Gradually Approaching Ordinary Q-Learning. *Frontiers in Neurorobotics*, 13.

Perrusquía, A., Zou, M., and Guo, W. (2024). Explainable data-driven Q-learning control for a class of discrete-time linear autonomous systems. *Information Sciences*, 682.

Rafael1s (2020). Deep-Reinforcement-Learning-Algorithms/Cartpole-Deep-Q-Learning - github repository. https://github.com/Rafael1s/Deep-Reinforcement-Learning-Algorithms/tree/master/Cartpole-Deep-Q-Learning. Accessed: 2025-04-30.

Rajkovic, M., Zrnic, N., Kosanic, N., Borovinsek, M., and Lerher, T. (2019). A MULTI-OBJECTIVE OPTIMIZATION MODEL FOR MINIMIZING INVESTMENT EXPENSES, CYCLE TIMES AND CO2 FOOTPRINT OF AN AUTOMATED STORAGE AND RETRIEVAL SYSTEMS. *TRANSPORT*, 34:275–286.

Raut, P. and Dani, A. (2020). Correlation between number of hidden layers and accuracy of artificial neural network. *Advanced Computing Technologies and Applications. Algorithms for Intelligent Systems*, 1:513–521.

Rizqi, Z., Chou, S., and Yu, T. (2024). Data-driven approach for dwell point positioning in automated storage and retrieval system: a metaheuristic-optimized ensemble learning. *Annals of Operations Research*.

Roshan, K., Shojaie, A., and Javadi, M. (2018). Advanced allocation policy in class-based storage to improve AS/RS efficiency toward green manufacturing. *International Journal of Environmental Science and Technology*.

Schaul, T., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay. *ICLR 2016*.

Shan, S., Kim, K., Kim, S., and Youn, Y. (2018). Artificial neural network: Understanding the basic concepts without mathematics. *Dementia and Neurocognitive Disorders*, 2(2):100–200.

Sui, Z., Duan, L., Hou, T., and Zhang, T. (2019). Modeling and Scheduling of tier-to-tier shuttle-based storage and retrieval systems. *Proceedings of the 38th Chinese Control Conference*.

Sutton, R. and Barto, A. (2018). *Reinforcement Learning: An Introduction*. MIT Press, Camebridge.

Tensorflow (2023). Introduction to rl and deep q networks. Accessed: 2025-03-06.

Tensorflow (2024). Tensorflow basics. Accessed: 2025-03-10.

Tensorslow (2023). Keras: The high-level api for tensorflow. Accessed: 2025-03-10.

Terven, J., Cordova-Esparza, D. M., Ramirez-Pedraza, A., Chavez-Urbiola, E. A., and Romero-Gonzalez, J. A. (2024). Loss functions and metrics in deep learning.

Thrun, S. and Schwartz, A. (1994). Issues in using function approximation for reinforcement learning. *Proceedings of the 1993 Connectionist Models Summer School*, 1.

van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1).

Vanderlande (2025). Vanderlande company profile. Accessed: 2025-02-12.

Vasan, G., Wang, Y., SHahriair, F., Bergstra, J. .and Jagersand, M., and Mahmood, A. (2024). Revisiting sparse rewards for goal-reaching reinforcement learning. *arXiv*, 2.

Visengeriyeva, L., Kammer, A., Bär, I., Kniesz, A., and Plöd, M. (2025). Crisp-ml(q). the ml lifecycle process. Accessed: 2025-02-12.

Wan, C. and Hwang, M. (2018). Value-based deep reinforcement learning foradaptive isolated intersection signal control. *IET Intelligent Transport Systems*, 12(9):1005–1010.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. *arXiv*, 3.

Wu, Y., Zhou, C., Ma, W., and Kong, X. (2020). Modelling and design for a shuttle-based storage and retrieval system. *International Journal of Production Research*, 58:4808–4828.

Xiao, L. (2022). On the convergence rates of policy gradient methods. *Journal of Machine Learning Research*, 23:1–36.

Yang, D. and Ren, R. (2023). A method of system selection for shuttle-based storage and retrieval system considering cost and performance. *Cluster Computing*.

Yang, P., Tao, P., Xu, P., and Gong, Y. (2023). Bi-objective operation optimization in multishuttle automated storage and retrieval systems to reduce travel time and energy consumption. *Engineering Optimization*.

Zangirolami, V. and Borrotti, M. (2024). Dealing with uncertainty: Balancing exploration and exploitation in deep recurrent reinforcement learning. *Knowledge-Based Systems*, 293.

Zhang, Z., Chen, J., and Zhao, W. (2024). Multi-AGV route planning in automated warehouse system based on shortest-time Q-learning algorithm. *Asian journal of control*, 26:683–702.

Zhao, X., Zhang, R., Zhang, N., Wang, Y., Jin, M., and Mou, S. (2020). Analysis of the Shuttle-Based Storage and Retrieval System. *IEEE Publications Database*, 8.

Zhong, Y. and Wang, Y. (2025). Cross-regional path planning based on improved Q-learning with dynamic exploration factor and heuristic reward value. *Expert Systems With Applications*, 260.