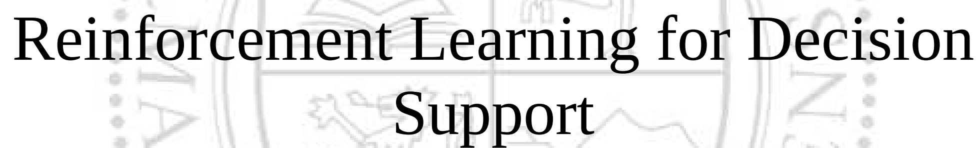
Chair of Information Technology

Master's Thesis



Reinforcement Learning for Decision Support

Martin Roth, BSc

May 2022

MONTANUNIVERSITÄT LEOBEN
www.unileoben.ac.at

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich die Richtlinien des Senats der Montanuniversität Leoben zu "Gute wissenschaftliche Praxis" gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 12.05.2022

Martin Roth

Unterschrift Verfasser/in
Martin Roth

Acknowledgement

First of all I want to thank Professor Dr. Peter Auer for his mentoring and supervision of this thesis and the countless hour-long discussions we had in the course of this master's thesis.

I would also like to thank my parents, who made this study possible for me in the first place.

Further I want to thank my girlfriend Lena for her psychological support during the creation to this thesis and her help on the work outside of the cosmos of this work during this process.

I want to thank my dear friend Nikolaus who helped adding the finishing touch to this thesis with his valuable remarks.

Additionally I want to thank Jürgen and Magdalena as my faithful companions throughout the whole studies be it preparations for exams, joint projects or bureaucratic affairs.

Finally I want to thank my colleagues at redPILOT, especially my now unfortunately ex colleague Stephan Spat, for their support.

Abstract

Personnel costs are an important performance indicator in warehouse operations as they directly influence the cost and profit. Therefore, to maximize the profit, good personnel planning is essential. In this Master's Thesis, we investigated the use of reinforcement learning using neural networks to support decision making for assigning work operators. At first, an introduction to reinforcement learning and neural networks is given. Then, based on a system analysis, a Markov Decision Process (MDP) is designed. Based on the MDP a simulation is proposed to provide the data necessary for reinforcement learning. Finally, the reinforcement learning approach to predict the expected profit for different decisions is described in the final step. The evaluation on different validation scenarios shows, that the proposed reinforcement learning approach achieves higher profits than a naive algorithm and should therefore be considered as valuable support in future warehouse operations.

Kurzfassung

Die Personalkosten sind ein wichtiger Leistungsindikator im Lagerbetrieb, da sie sich direkt auf Kosten und Gewinn auswirken. Um den Gewinn zu maximieren, ist daher eine gute Personalplanung unerlässlich. In dieser Masterarbeit haben wir den Einsatz von Reinforcement Learning unter Verwendung neuronaler Netze zur Unterstützung der Entscheidungsfindung bei der Zuweisung von Arbeitskräften untersucht. Zunächst wird eine Einführung in das Reinforcement Learning und neuronale Netze gegeben. Anschließend wird auf der Grundlage einer Systemanalyse ein Markov Decision Process (MDP) entworfen. Auf der Grundlage dieses MDP wird eine Simulation erstellt, um die für das Reinforcement Learning erforderlichen Daten zu generieren. Im letzten Schritt wird der Reinforcement-Learning-Ansatz beschrieben, um den erwarteten Gewinn für verschiedene Entscheidungen vorherzusagen. Die Auswertung verschiedener Validierungsszenarien zeigt, dass der entwickelte Reinforcement-Learning-Ansatz höhere Gewinne erzielt als ein naiver Algorithmus und daher als wertvolle Unterstützung für zukünftige Lageroperationen angesehen werden sollte.

Contents

Abstract **III**

Kurzfassung **IV**

List of Figures **VIII**

List of Tables **X**

List of Algorithms **XI**

1 Introduction **1**
1.1 Decision Support 1
1.2 Analysis of Warehouse Operations 1
1.2.1 System Analysis 2
1.2.2 Warehouse Staff 2
1.3 Goal of this Thesis 3

2 Machine Learning **5**
2.1 Reinforcement Learning 5
2.1.1 Overview 5
2.1.2 Markov Decision Processes 7
2.1.3 Preparations for Learning 9
2.1.4 Q-Learning 10
2.1.5 Nondeterministic Case 12
2.1.6 Function Approximation 13
2.2 Neural Networks 13
2.2.1 Single-layer Networks 14
2.2.1.1 Bias Unit 14
2.2.1.2 Perceptron Training Rule 16
2.2.1.3 Delta Rule 16
2.2.1.4 Activation Functions 19
2.2.1.5 Number Of Output Nodes 21
2.2.1.6 Loss Functions 21

2.2.2 Multi-layer neural networks . . . 22
2.2.2.1 Architectural Possibilities . . . 23
2.2.2.2 The Backpropagation Algorithm . . . 24
2.2.2.3 Optimization Techniques . . . 27
2.3 Deep Reinforcement Learning . . . 29
2.3.1 Target Network . . . 30

3 Implementation 31
3.1 Deep Q-Learning Approach . . . 31
3.1.1 Construction of the MDP . . . 31
3.1.2 Q-learning with function approximation by an MLP . . . 34
3.1.2.1 MLP . . . 34
3.1.2.2 Exploration-Exploitation-Scheduler . . . 37
3.1.2.3 Memory Replay . . . 37
3.1.2.4 Agent . . . 38
3.1.2.5 Training Loop . . . 40
3.2 Simulation . . . 40
3.2.1 Why Simulation is Needed . . . 40
3.2.2 Simulation Approach . . . 42
3.2.2.1 Initialization . . . 42
3.2.2.2 State and Action Management . . . 43
3.2.2.3 Reward Calculation . . . 44
3.2.2.4 Cloning and Resetting . . . 44
3.2.3 Order Generation . . . 44

4 Validation and Results 46
4.1 Upper Bound: Linear Program . . . 46
4.1.1 Definition of the Linear Program for Validation . . . 47
4.2 Lower Bound: Naive Algorithm . . . 48
4.3 Validation Scenarios . . . 49
4.3.1 General Remarks . . . 49
4.3.2 Validation Scenario 1: Target VS Main Network . . . 49
4.3.3 Results Scenario 1: Target VS Main Network . . . 51
4.3.3.1 Training . . . 51
4.3.3.2 Result . . . 54
4.3.4 Validation Scenario 2: Existing Network with different Operators . . . 55
4.3.5 Results Scenario 2: Existing Network with different Operators . . . 55
4.3.6 Validation Scenario 3: Retraining with Higher Workload . . . 58
4.3.7 Results Scenario 3: Retraining with Higher Workload . . . 58
4.3.7.1 Training . . . 58

4.3.7.2 Results 59
4.3.8 Validation Scenario 4: Retraining with Reduced Workload 61
4.3.9 Results Scenario 4: Retraining with Reduced Workload 61
4.3.9.1 Training 61
4.3.9.2 Results 62

5 Summary and Outlook **66**
5.1 Summary 66
5.2 Outlook 67

Bibliography **XII**

List of Figures

2.1 A simple perceptron 14
2.2 A perceptron with an additional bias unit 15
2.3 Linear separable (a) and not separable (b) data 15
2.4 Different activation functions 19
2.5 A neuron broken down into pre-activation and post-activation value 20
2.6 A multi layer network without (a) and with (b) additional bias neurons . . 23

3.1 UML diagram of the reinforcement learning algorithm using an MLP . . . 35
3.2 Linear decay of ε over time 38
3.3 Poisson Distribution with different values of λ 45

4.1 Scenario 1: Predicted cumulative reward for each time interval during training process using only the main network 51
4.2 Scenario 1: Predicted cumulative reward for each time interval during training process using the target network 52
4.3 Scenario 1: Predicted cumulative reward of the best performing network using only the main network 53
4.4 Scenario 1: Predicted cumulative reward of the best performing network using an additional target network 53
4.5 Scenario 1: Comparison of mean evaluation rewards during training 54
4.6 Scenario 1: Achieved rewards comparing main network and target network 55
4.7 Scenario 2a: Decreased operator working capacity to 90% on trained network 57
4.8 Scenario 2b: Increased operator working capacity to 150% on trained network 58
4.9 Scenario 3: Evaluation reward during training 59
4.10 Scenario 3: Evaluation reward during training 60
4.11 Scenario 3: Predicted cumulative reward of the best performing retrained network on 150% working capacity 60
4.12 Scenario 3: Increased operator working capacity to 150% on A retrained network 61
4.13 Scenario 4: Evaluation reward during training 63
4.14 Scenario 4: Predicted cumulative reward of the best performing retrained network on 75% working capacity 63
4.15 Scenario 4: Evaluation reward during training 64

4.16 Scenario 4: Decreased operator working capacity to 75% on A retrained network 65

List of Tables

3.1 Configured hyperparamters for the training loop 36

4.1 Scenario 1: Process step capacities 49
4.2 Scenario 1: Work capacity of the operators 50
4.3 Scenario 2a: Work capacity of operators 56
4.4 Scenario 2b: Work capacity of operators 56
4.5 Scenario 4: Working capacity of operators 62

List of Algorithms

2.1 Q-learning Algorithm 12
2.2 Gradient Descent 18
2.3 Gradient Descent 26

3.1 Train the neural network 39
3.2 The Training Loop 41
3.3 Generate Order 45

1 Introduction

1.1 Decision Support

In the field of logistics cost efficient processes are a must [1]. This is underlined by a survey from Cooke et.al where 71% of all respondents ranked "cost control/cost reduction" as top priority in the logistics industry [2]. One crucial cost factor in warehousing and logistics are personnel costs. These costs have a decisive influence on revenue as they account for a large share of total costs and continue to rise [3]. Personnel costs are directly linked to staffing levels. Therefore, accurate workforce planning is essential. At the same time, warehousing processes necessitate maximum flexibility as the daily demand fluctuates [4]. As we can see in the following chapter, usually the shift leader is commissioned with the task of assigning personnel to the workstations.
Complex situations, where a decision maker is confronted with decisions which are dependent on a big number of external variables, the need for external support arises. This is often provided by so called computer aided decision support systems. The usage of these support systems started in the mid-1960s and has evolved ever since. Complex situations, such as the personnel management decisions in modern warehouses, call for sophisticated intelligent decision support systems which can model non linear relations [5]. The goal of this master thesis is, to develop a system to support personnel planners in daily planning and decision making.
In order to provide a mathematical model, we first analysed an existing warehouse operation. We interviewed warehouse managers in order to identify the different roles existing within a warehouse and their tasks. The results of these analyses are presented in the following chapter. Based on these roles we consider the most promising, to receive decision support for planning and decision making. Based on the acquired information about this role reinforcement learning is chosen as promising approach. This master thesis sets out to investigate if a reinforcement learning policy aided by neural networks is suited to support decision making processes within a warehouse setting.

1.2 Analysis of Warehouse Operations

This thesis is conducted in cooperation with the company redPILOT which markets a software for warehouse operations optimization called redPILOT Operational Excellence

Solution. We will discuss general warehouse operations and how they are handled in redPILOT Operational Excellence Solution as we will use this models in our approach. Different key roles in warehouse operations are identified alongside their scope of work.

1.2.1 System Analysis

A so called customers system is a structure within redPILOT Operational Excellence Solution. The basis of a customers system is represented by the business unit. A business unit describes the site of the customer. It is covered by the so called process model within the application. The term operation is used to describe different KPI-driven operations of a given business unit. Within an operation, a so called system is the smallest possible working group. Resource planning happens on a system level. Each activity of a system is called process step. Each process step w has individual parameters which need to be considered when allocating operators o. First, each process step has a maximum $m_{w,max}$ as well as a minimum amount $m_{w,min}$ of operators that can be assigned to it. In this thesis we will consider only linear aligned process steps. Two process steps are linear aligned if the completed workload from one process step gets populated completely into the subsequent process step. Each process step is equipped with a buffer β that holds the goods to be processed.
The workload is presented to a system as orders x. For simplicity we consider each order to be of the amount one for this thesis. These orders arrive in a certain distribution over the course of shift and are directly filled into the buffer of the first process step of the system.
To process orders at the process steps operators are available for the system. Each operator has different skills at each process steps. These skills are defined as working capacity an operator can provide within a given time interval.

1.2.2 Warehouse Staff

In the following section, we will describe the key warehouse staff positions as well as their tasks. These informations were acquired performing interviews with selected warehouse executives.
Site Manager (SM): The SM has the overall responsibility for the warehouse and its administrations. A SM takes care of finances and makes strategic decisions. However, SMs are not involved in operational decision making processes.
Warehouse Manager (WM): The WM is responsible for all warehouse processes. A WMs field of decision does not cover any administrative decisions. The WM is also tasked with communicating with the purchasing department regarding delivery times. Most importantly, the WM sets up guidelines for daily planning. For example:

- Claim an additional shift

- Additional slot for weekends
- Communications with subsidiaries
- Fleet management
- General conditions for working time
- Operator pool
- Restrictions for workstations

Hall Manager (HM): The HM is responsible for basic personnel planning including the planning of vacations. The HM also recruits and dismisses operators. A HM is also involved in the scheduling of maintenance windows and coordinates the Shift Leaders.
Shift Leader (SL): The SL is responsible for the shift in consultation with the HM. The SL is tasked with coordinating workers' daily work hours including working overtime or leaving early. The SL is also tasked with rescheduling of operators within process steps. The SL makes situational decisions about maintenance events and is also tasked with managing workstations that suffer a breakdown. In general, SLs don't have any high level operator planning responsibility. Instead, they only manage the operator pool given to them.

1.3 Goal of this Thesis

The goal of this thesis is to train a reinforcement learning policy to support warehouse personnel assigning operators to process steps in order to minimize costs. We derive a Markov Decision Process (MDP) from the analysed system. The MDP features three essential components. First, it defines the different states the system can adopt. Secondly the MDP provides actions which cause a transition from one state to another. Lastly, a reward function is provided to give feedback to the reinforcement learning policy about the action taken. Looking at the existing data provided by real life system it emerged, that the available data is not sufficient for reinforcement learning. During real life operations only a certain way of controlling the system is represented in the data. This can lead to situations during training where an action is taken but the resulting transition is not present in the provided data. To overcome these limits during training, a simulation is introduced. The usage of a simulation to provide data for reinforcement learning is considered a common approach. The reinforcement learning policy will directly communicate with this simulation and receive feedback based on actions taken.
We continue by describing our proposed approach for reinforcement learning in detail. We also provide bounds to evaluate the results generated by our approach. For this we introduce an low level algorithm as well as a linear program to provide context to the

quality of the achieved results. The evaluation will take on different scenarios where we will test the overall performance of the reinforcement learning policy in different settings. We will as well test pre trained models with alternating sets of available operator sets. In the end we will provide a summary of the presented results as wells as potential further research topics derived from this thesis.

2 Machine Learning

2.1 Reinforcement Learning

2.1.1 Overview

This chapter will start by giving a general overview of reinforcement learning. It will then continue by giving detailed description of the Markov Decision Process as well as Q-Learning including the advanced cases of stochastic environments. This section about reinforcement learning is based on [6] and [7]. In reinforcement learning an agent learns to map situations to actions. Feedback is given via a reward, which needs to be maximized. In order to maximize the reward the learner must try different actions and discover which one yields the highest reward. If the agent takes an action, it may not affect only the immediate reward but also future rewards, as an action can influence all subsequent rewards. This results in the two most important features of reinforcement learning, the trial and error search and the delayed reward.
Reinforcement learning is different to supervised learning. In supervised learning, the learner is trained with a set of labelled examples which are provided by an external supervisor. Each example describes a situation together with the correct action the system should take in that situation (the label). When a system which was trained using supervised learning encounters a situation unknown to the system it extrapolates and generalizes its responses in order to act correctly in situations unknown to the system. In unsupervised learning the learner searches for structure within the set of unlabelled data. Reinforcement learning does not rely on labelled examples but instead maximizes the reward signal rather than finding hidden structure within the data. Therefore reinforcement learning is a third kind of machine learning paradigm besides supervised learning and unsupervised learning.

In reinforcement learning, the distribution of training examples is influenced by the action sequence that the agent selects. This leads to the question which strategy leads to the most effective learning. In RL, the learner can choose between exploring unknown states and exploiting states that have already been learned and are known to yield high rewards. This exploration and exploitation approach is not without its challenges. On one hand the reinforcement learning agent prefers actions, from which it knows from the past, that

they produce a high reward. On the other hand, to find these actions, it has to try actions never used before. This means that the agent has to exploit actions based on the current estimate of the value function, to obtain a high reward but also explore new actions for future selection. The agent needs to combine exploitation and exploration in order to learn the value function. On a stochastic task each action must be tried many times to gain a reliable estimate on its expected reward.

To better understand reinforcement learning, it is important to understand the following elements: The policy, a reward signal and the value function.

A policy defines the agent's behaviour at a given time. It defines the action to take in a certain state. A policy's form can range from a simple lookup table to a more complex computation process. The policy is the core of the reinforcement learning agent as it is solely responsible for the agent's behaviour.

The reward signal represents the goal of the reinforcement learning problem. At each time step t, after the agent has taken an action, the environment returns a single number called reward. The only goal of the learning agent is to maximize the cumulated reward over time. Thus, the reward signal indicates positive and negative events to the agent, and thereby can influence the policy. If the agent takes an action which leads to a low reward the policy might change to a different action with a higher reward in the future in order to achieve its goal of accumulating the highest reward possible.

The value function combines the immediate reward with the accumulated reward to be expected over time. Therefore, the value of a state defines how desirable a state is in the long term. It also takes the states and rewards which are likely to follow into account. It might occur that a state offers a relatively low immediate reward but it yields a high value as it is followed by states which yield high rewards. Naturally the opposite can be true. When making and evaluating actions it is therefore the values, not the rewards that need to be taken into account. We seek states which hold a high value rather than a high reward in order to achieve a high cumulated reward over time. However, determining values is much more difficult compared to rewards. The reward is given directly by the environment after taking an action. The value must be estimated and re-estimated from the observations of the agent over its entire lifetime. Therefore correctly estimating values is a cornerstone method in reinforcement learning.

The final element for reinforcement learning is the model of the environment. The model imitates the behaviour of the environment. The model might predict the next state based on the current state and the taken action. In the following section we define a model, the Markov Decision Process (MDP), often used in reinforcement learning and our derived model.

2.1.2 Markov Decision Processes

The Markov Decision Process (MDP) is considered a classical model for sequential decision making and is seen as a suitable model of the environment often used in reinforcement learning. Actions within an MDP do not just influence the immediate reward received, but also future rewards due to subsequent situations. MDPs also involve delayed rewards. Therefore the trade-off between immediate rewards and delayed ones needs to bed calculated.

In the MDP a learner and decision maker is called an agent. Every model consists of an agent interacting with an environment. An environment is everything outside the agent and is described by possible states S. The agent is capable of observing the sate of its environment, and knows a set of actions A which can be performed to alter the current state. The task of the agent is to learn a policy for choosing actions to reach a specific goal. As already described the goal can be be defined via a reward function. This function assigns a numerical value for each distinct action the agent might take from each distinct state. The reward function might be either known to the agent or only to the teacher who provides the reward for each action taken. However, the reward is an immediate pay-off after each action taken. Ideally, a control policy $\pi : S \rightarrow A$ would be able to choose actions maximizing the cumulative reward over time, independent of the initial state. In general it is desired to learn to control a sequential process. Cases where the sets A and S can be considered to be finite which is called a finite MDP. For simplicity we will continue describing a finite MDP.

In an MDP, the agent interacts with its environment in a sequence of discrete time steps $t = 0, 1, 2, \ldots$. At each time step t the agent observes an state $s_t \in S$ of the environment. With this knowledge the agent chooses an action $a_t \in A$, and receives the reward $r_t = r(s_t, a_t), r_t \in \mathbb{R}$ from the environment. Then the agent observes the subsequent state $s_{t+1} = \delta(s_t, a_t)$ at t. This results in a sequence:

$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, \ldots \tag{2.1}$$

between the agent and the environment. The functions $r(s_t, a_t)$ and $\delta(s_t, a_t)$ are part of the environment and not necessarily known to the agent. These functions only depend on the current state. Therefore the state must contain all past information about action-environment interaction that influence the future. If this is the case for a given state, then it is said to have the Markov property.

Moreover the functions $r(s_t, a_t)$ and $\delta(s_t, a_t)$ might be stochastic. In this case the functions can be viewed by first creating a probability distribution over s and a, and then drawing an outcome according to these distribution. A system is called a nondeterministic MDP if these probability distributions solely depend on s and a. The state-transition probabilities

are defined as follows:

$$p(s'|s,a) \doteq \Pr(s_t = s'|s_{t-1}, = s, a_{t-1} = a). \tag{2.2}$$

The expected reward can also be calculated for state-action pairs:

$$r(s,a) \doteq \mathbb{E}[r_t|s_{t-1} = s, a_{t-1} = a]. \tag{2.3}$$

Finally we can compute the expected reward for state-action-next-state triples:

$$r(s,a,s') \doteq \mathbb{E}[r_t|s_{t-1} = s, a_{t-1} = a, s_t = s']. \tag{2.4}$$

An advantage of MDPs is that they are abstract and flexible. Therefore they are applicable to many different problems in many different ways. For instance, the time steps must not necessarily refer to fixed intervals of real time. The actions can also range from low-level controls up to high-level decisions. They can be completely determined by low-level sensations, or they can be more high-level and abstract. A state could be made up by memory of past sensations or be entirely subjective. Equivalently, some actions can be totally mental or computational. Generally speaking, actions can be any decisions we want to learn how to perform. States can represent anything that can be known which might be useful to learn actions.

It is important to note that in general, the boundary between agent and environment is not the same as the physical boundary of for example a robot's body and the rest of the room. Most of the time the boundary in an MDP is drawn closer to the agent. Following the robot example, the sensors, motors, and the mechanical linkage would be considered part of the environment only the control unit would be considered the agent.

In this analogy, the rewards are most likely computed inside the robot. However, they are considered as external to the agent. In general, everything that cannot be changed by the agent arbitrarily is considered to be part of the environment. Further, we do not consider that everything of the environment is unknown to the agent.

Learning tasks vary in their level of difficulty. It is possible to face hard learning tasks although everything is known to the agent about how the environment works. The states and actions vary notably from task to task. Their representation can strongly affect the performance of the model. Currently these representations are considered more art than science.

2.1.3 Preparations for Learning

In 2.1.1 a reward was defined as a signal given to the agent by the environment after taking an action. The goal of the agent is to maximize the cumulative reward over a lifetime. In this section we will first define the term goal, continue by specifying rewards, the rewards sequence and discounting to segue to Q-Learning.
Defining goals in terms of rewards is not as limiting as it might seem at a first glance. For example the reward of a robot trying to find the exit of a maze could be set to -1 for each step which is not a step out of the maze and +1 for the step out of the maze. This can also be applied to competitive games like chess or checkers. There the reward is -1 in case of a loss, +1 in case of a win and 0 for all non terminal states or a draw. In all these examples the agent will always try to learn to maximize the reward. For the agent to reach a specific goal it needs to receive rewards in a way so that maximizing these rewards is congruent with that goal. A common pit fall in this approach is to include prior knowledge in the form of sub-goals to the agent. This can influence the agent in a way that it learns to achieve these sub-goals without achieving the overall goal. Recalling the chess example one might think it is a good idea to reward taking the opponents pieces. This might lead to a policy were only the sub-goal is achieved but the game is lost. Therefore the rewards signal should always be designed in a way that the agent knows what should be achieved in the end, but not how.
So far, the goal of the agent has always been defined as the maximization if the cumulative reward. The sequence of rewards received after t is defined as $r_t, r_{t+1}, r_{t+2}, \dots$. The expected sum of rewards g_t is the aspect of this sequence we want to maximize is and is considered a specific function of the reward sequence. Considering the case of a final time step T the return is simply the sum of the rewards:

$$g_t \doteq r_t + r_{t+1} + r_{t+2} + \cdots + r_T. \tag{2.5}$$

This is a valid approach in applications with terminal states which divides the interaction of the agent with the environment into subsections. These subsections are called episodes. Given the chess or checkers example, one episode would correspond to one game. Every episode ends with a terminal state which is followed by an independent starting state. These starting states are either constant (e.g.: chess, where all pieces always start from the same position) or taken from a distribution of starting sate (e.g.: robot in a position within a maze). Furthermore the terminal time T varies from episode to episode.
In many cases the agent-environment interaction is not broken into episodes. In these cases the formulation of g_t is problematic as $T = \infty$, and g_t could therefore also be infinite. Therefore we will use a slightly more complex definition of a return.
The concept of discounting future rewards fulfils our needs. Therefore we change the agents goal to maximize the sum of discounted rewards received over time. The agent

then chooses a_t to maximize the discounted return:

$$g_t \doteq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}. \tag{2.6}$$

The variable γ, with $0 \leqslant \gamma \leq 1$ is called discount rate. It determines the present value of rewards in the future. Thus a reward received after k time steps is only worth γ^k times what it would be worth if it was received immediately. Further the sum at (2.6) has a finite value as long as the rewards are bounded. In case $\gamma = 0$, the agent is only concerned about maximizing immediate rewards. The agent chooses a_t so as to maximize r_t. If each action taken only influences the immediate reward, not future ones as well, this agent can maximize (2.6) independently of maximizing each immediate reward. However maximizing the immediate reward might reduce access to future rewards and as a consequence reduce the return. The closer γ is to 1, the more value is put on future rewards and the agent becomes more farsighted.

2.1.4 Q-Learning

We continue by formalising the concepts of value functions and policies which were already introduced in 2.1.1. Nearly all reinforcement learning algorithms include estimating value functions. We roughly defined the value function to give information about how good it is to be in a certain state. "How good" in this case is defined to be the expected return given through the future rewards. These rewards depend, of course, on the actions taken by the agent in the future. Consider an agent following policy π at time t. Then, $\pi(a|s)$ defines the probability $a_t = a$ if $s_t = s$. The reinforcement learning method then defines how the agent has to change its policy based on the acquired knowledge from interacting with the environment.

We denote the value function of a state s under the policy π as $V^\pi(s)$. $V^\pi(s)$ returns the expected reward for an agent following π with s as its initial state. We define $V^\pi(s)$ for MDPs formally as:

$$V^\pi(s) = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.7}$$

The next step will be to find the optimal policy π^* to solve the reinforcement learning task. The different policies of finite MDPs can be compared by their associated value functions. For a policy π to be better than or at least equal to a policy π' its expected return must be greater than or equal to the one for π' for all states. This implies the existence of one or more policies with a higher expected return for all states than all other policies. This policy, called optimal policy, is denoted by π^* and is defined as:

$$\pi^* \doteq \arg\max_{\pi} V^\pi(s), \tag{2.8}$$

for all s. Starting from state s and following π^* afterwards results in the optimal state-value function $V^*(s)$ (for simplicity reasons we use V^* instead of V^{π^*}). This value function defines the highest discounted cumulative reward the agent can possibly achieve when starting from state s.
Now that the optimal policy π^* is defined, a method to learn this policy becomes necessary. The only training information available to the learner is the immediate rewards given by the environment. This makes the direct learning of the function $\pi : S \rightarrow S$ difficult, because this would call for training examples in the form of $\langle s, a \rangle$. Based on the sequence of returned rewards g_t it is more natural to learn a numerical evaluation function which is defined via actions and states. The function is then used to implement the optimal policy. The simplest function would be $V^*(s)$, because a state s_1 is preferred over a state s_2 if $V^*(s_1) > V^*(s_2)$. This leads to an optimal policy $\pi^*(s)$, were the agent selects action a, that maximizes the discounted future rewards, when in state s:

$$\pi^*(s) = \arg\max_a r(s,a) + \gamma V^*(\delta(s,a)) \tag{2.9}$$

However, the agent's policy does not rely on states but rather relies on actions. V^* can still be used to select between actions but that assumes perfect knowledge of the reward function $r(s,a)$ and the state transition function $\delta(s,a)$. This is not the case in many real-life problems. A more practical approach is to use the Q-function. Using V^* as well as $r(s,a)$ and $\delta(s,a)$ we can define the Q-function as:

$$Q(s,a) \doteq r(s,a) + \gamma V^*(\delta(s,a)). \tag{2.10}$$

$Q(s,a)$ matches exactly the maximization part of 2.9. Thus 2.9 needs to be rewritten:

$$\pi^*(s) = \arg\max_a Q(s,a). \tag{2.11}$$

This enables the agent to chose optimal actions while not knowing $r(s,a)$ and $\delta(s,a)$. The agent can choose an action a based only on the current status and without any any knowledge of the following states. All future rewards are summarised in the respective Q value. now that the Q value is defined, a strategy to learn Q is necessary. The learner needs to find a reliable estimate for any Q by continuously approximating it. Q and V^* are connected by:

$$V^*(s) = \max_{a'} Q(s,a') \tag{2.12}$$

and by 2.10 follows:

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a),a'). \tag{2.13}$$

$\hat{Q}$ is defined as the learners current hypothesis of the Q function. This basic algorithm features a table where the current $\hat{Q}(s, a)$ is stored for each state action value pair. After each experienced $\langle s, a, r, s' \rangle$ sequence the agent updates the respective $\hat{Q}(s, a)$ value in the table according to:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a') \tag{2.14}$$

Algorithm 2.1 Q-learning Algorithm

1: **init_q_table**(n)
2: $s \leftarrow$**observe_state**
3: **while** $i < n$ **do**
4: $\quad a \leftarrow$**select_action**
5: $\quad r \leftarrow$**take_action**
6: $\quad s' \leftarrow$**observe_state**
7: $\quad \hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$
8: $\quad s \leftarrow s'$
9: **end while**

Algorithm 2.1 shows how equation (2.14) works without the knowledge of $r(s, a)$ and $\delta(s, a)$ assumed in equation (2.13). The agent obtains all parameters needed by interacting with the environment. For a deterministic MDP utilizing algorithm 2.1 we can prove convergence of $\hat{Q}$ to Q.

2.1.5 Nondeterministic Case

In case of a stochastic MPD the functions $r(s, a)$ and $\delta(s, a)$ provide a sample according to an underlying probability distribution. These probability distributions only depend on s and a. When adapting algorithm 2.1 to handle stochastic MDPs, the first that needs to be changed is the learners objective. V^{π} is redefined as expected value of the discounted cumulative reward:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k}\right] \tag{2.15}$$

We continue adapting the Q function to use the expected value as well:

$$\begin{aligned} Q(s, a) &\doteq \mathbb{E}[r(s, a) + \gamma V^*(\delta(s, a))] \\ &= \mathbb{E}[r(s, a) + \gamma \mathbb{E}[V^*(\delta(s, a))]] \\ &= \mathbb{E}[r(s, a)] + \gamma \sum_{s'} \Pr(s'|s, a) V^*(s') \end{aligned} \tag{2.16}$$

The probability $\Pr(s'|s, a)$ defines probability that taking action a in state s results in state s'. Finally we can rewrite equation (2.13):

$$Q(s, a) = \mathbb{E}[r(s, a)] + \gamma \sum_{s'} \Pr(s'|s, a) \max_{a'} Q(s'a') \tag{2.17}$$

Now we need to update the training rule to fit equation (2.17). We can not reuse equation (2.14) because it would not converge. To overcome this we take the weighted decaying average of the current $\hat{Q}$. We denote $\hat{Q}_n$ as the agent's estimate at the nth iteration:

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \gamma \max_{a}' \hat{Q}_{n-1}(s',a')] \tag{2.18}$$

with:

$$\alpha_n = \frac{1}{\sqrt{1+v_n(s,a)}} \tag{2.19}$$

where $v_n(s,a)$ is the number of times, the currently updated $\langle s,a \rangle$ was visited at the nth iteration. This training rule updates $\hat{Q}$ more gradually than equation (2.14).

2.1.6 Function Approximation

The tabular method to store the Q values, which has been discussed until now has its limits. With an increasing size of the state space, certain problems arise. For large state spaces (e.g.: possible images of a digital camera) the tabular approach will not converge to a result with a reasonable amount of data and time. The main problem is not the memory needed for these large tables, but rather the time needed to fill them. We expect almost every new state to be one the algorithm has never seen before. For these problems we will not find an optimal policy or value function. Therefore we aim for a approximate solution instead. The algorithm needs to generalize previous visited states to make decision for similar ones encountered in the future. For this purpose we use function approximation to approximate the value function.
Let π be a known policy and V^π the value function to be approximated. We define the approximation $\hat{V}(s,\mathbf{w}) \approx V^\pi$ as the current approximate value of s using the weight vector $\mathbf{w} \in \mathbb{R}^d$. $\hat{V}$ can be defined as a linear function with $\mathbf{w}$ as feature weights or a function computed by a neural network with $\mathbf{w}$ as connecting weights in-between the layers. Changing a weight causes changes in the estimate for several states as there are generally fewer weights than states.

2.2 Neural Networks

To further proceed with deep learning we will now examine artificial neural networks as extensively written about by Mitchell T. [6] and Aggarwal C.[8]. Inspired by biological organisms, neural networks provide a method to learn the approximation of different target functions. Neural networks can be grouped into different types, e.g.: single-layer and multi-layer neural networks. Single-layer networks consist of an input directly mapped to an output, utilizing a generalized linear function. In a multi-layer network one or more

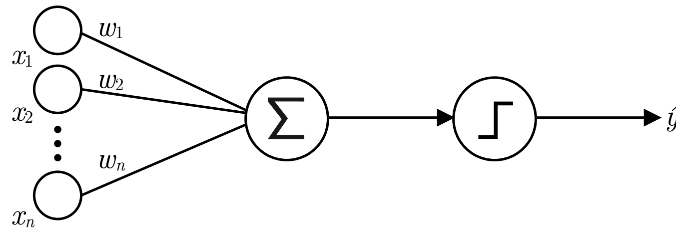


Figure 2.1: A simple perceptron

hidden layers separate the input from the output. These two types of neural networks, as well as their usage in reinforcement learning will be discussed in the following chapters.

2.2.1 Single-layer Networks

The most basic neural network is called perceptron. A perceptron consists of an input layer and a single output node. The single output node forms the only computation layer in the network. Hence, the perceptron is classified as a single-layer network. We consider the input as real valued n-dimensional vector $\mathbf{x} = [x_1, \ldots, x_n]$ and the possible output $y = \{-1, +1\}$. As seen in figure 2.1, the perceptron consists of n input nodes forming the input layer.
These nodes transmit the n feature variables to the output node, considering the respective weights $\mathbf{w} = [w_1, \ldots, w_n]$, represented as edge weights in the figure. Finally the output node delivers the prediction $\hat{y}$. Only the output node performs a calculation. This is done by using the linear function $\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{n} = w_i x_i$. Finally we use the sign to calculate the prediction $\hat{y}$:

$$\hat{y} = \operatorname{sgn}(\mathbf{w} \cdot \mathbf{x}) = \sum_{i=1}^{n} = w_i x_i. \tag{2.20}$$

In this case the sign function serves as a so called activation function. Now we can use the perceptron for binary classification. Consider a set of training data, each in the form $(\mathbf{x}, y)$, where $\mathbf{x}$ contains the input variables and y is the observed value. In training, the weights $\mathbf{w}$ of the perceptron are adjusted with respect to the error $E(\mathbf{x})$. Each time the error is not zero the weights are adjusted either into the positive or negative direction depending on the error gradient.

2.2.1.1 Bias Unit

If we think about an example where the feature values are mean centred but the mean of the class prediction is not zero, we need to add a bias to the perceptron. This bias

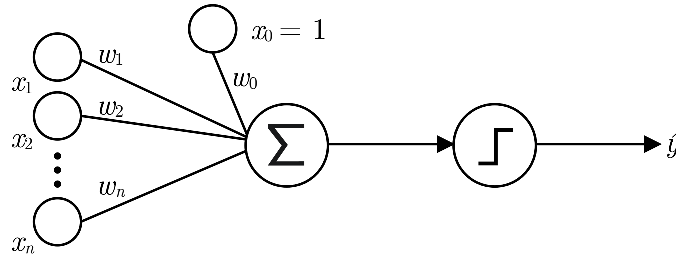


Figure 2.2: A perceptron with an additional bias unit

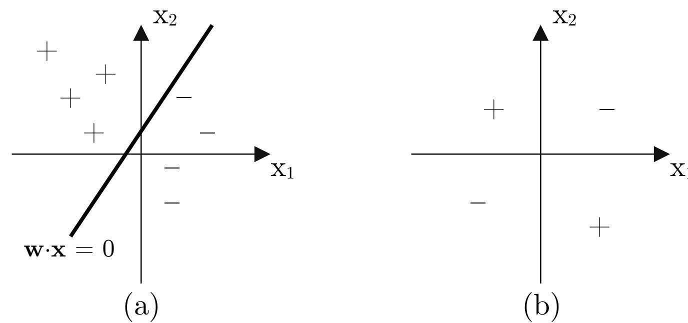


Figure 2.3: Linear separable (a) and not separable (b) data

shifts the activation function either to the left or right side. In other words, the bias is a threshold which the weight input combination needs to surpass in order for the linear function to output a +1. From an architectural perspective we add a bias neuron to the input layer of the perceptron. This bias neuron x_0 always emits $x_0 = 1$ to the output node. The actual bias is provided by the weight w_0 on the edge from the bias neuron to the output node, represented in figure 2.2.
Hence, we rewrite equation (2.20) to:

$$\hat{y} = \text{sgn}(\mathbf{w} \cdot \mathbf{x}) = \sum_{i=0}^{n} = w_i x_i. \tag{2.21}$$

From a representational point of view, the perceptron is a linear model. It represents a hyperplane defined by $\mathbf{w} \cdot \mathbf{x} = 0$. This linear hyperplane divides the set of training examples into two sets, as seen in figure 2.3. For instances on the one side the perceptron outputs +1. On the other side the output is -1. However, for some training sets, as seen in figure 2.3 (b), the data can not be separated by a hyperplane. A set which can be divided correctly by a hyperplane is called linearly separable.

2.2.1.2 Perceptron Training Rule

Now that we have introduced the architecture of the perceptron we continue with the learning of the weights in order to minimize the error $E(\mathbf{x})$. Although many different learning approaches for the perceptron exist, we will focus on two: the perceptron rule and the delta rule as they are the most common.
For the perceptron rule we start with an randomly initialized weight vector $\mathbf{w}$. We will discuss the details of setting initial weights later on. After the initialization we iterate through all training examples and apply the perceptron to each one. If a misclassification happens, the weights of the perceptron are changed. This procedure is continued until all available training examples are classified correctly. If weight change is necessary the perceptron rule is applied. The belonging weight w_i of the input x_i is altered the following way:

$$w_i \leftarrow w_i + \Delta w_i \tag{2.22}$$

with

$$\Delta w_i = \eta(y - \hat{y})x_i. \tag{2.23}$$

The change of weight, defined by the difference of the perceptron estimation $\hat{y}$ and the target output y of the training example, is influenced by the learning rate η. The learning rate indicates how strong the change of the weight is. Usually the initial learning rate decays as training progresses further. If the data is linearly separable, the perceptron rule converges in a finite number of iterations.

2.2.1.3 Delta Rule

If data is not linearly separable it may fail to converge. Therefore we introduce the delta rule. This training rule still converges to an best fit approximation if the data is not linearly separable. The space of possible hypotheses is searched through for weights which provide the best fit to training data using gradient descent. For this case we consider a single linear unit (i.e. a perceptron without threshold) providing the estimate $\hat{y}$ as follows:

$$\hat{y}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} \tag{2.24}$$

Further we define the training error of the prediction considering a training set $\mathcal{D}$:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{d \in \mathcal{D}} (y_d - \hat{y}_d)^2 \tag{2.25}$$

with y_d as target output of a training example d and $\hat{y}_d$ as the prediction of the linear unit for training example d. Within the resulting parabolic error surface of a linear unit one single global minimum exits. Starting from an initial weight vector, gradient descent alters this weights in each step. Each modification is performed towards the steepest descent along the error surface until the minimum is reached. The steepest descent is calculated by computing the partial derivative (i.e. the gradient) of E:

$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right] \tag{2.26}$$

In this definition the $\nabla E(\mathbf{w})$ indicates the direction of the steepest step away from the global minimum. Therefore the negative direction of $\nabla E(\mathbf{w})$ provides the desired decrease. With this knowledge we can express the training rule as:

$$\mathbf{w} = \mathbf{w} + \ \Delta\mathbf{w} \tag{2.27}$$

with

$$\Delta\mathbf{w} = -\eta \nabla E(\mathbf{w}). \tag{2.28}$$

The learning rate η defines the step size in a similar way as in equation (2.23). However the representation in equation (2.28) is not suitable for representing a single step in an algorithm. To achieve this we first express equation (2.28) in component form:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \tag{2.29}$$

as well as the weight update:

$$w_i = w_i + \Delta w_i. \tag{2.30}$$

Now we simply perform the derivative:

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in \mathcal{D}} (y_d - \hat{y}_d)^2 \\
&= \frac{1}{2} \sum_{d \in \mathcal{D}} \frac{\partial}{\partial w_i} (y_d - \hat{y}_d)^2 \\
&= \frac{1}{2} \sum_{d \in \mathcal{D}} 2(y_d - \hat{y}_d) \frac{\partial}{\partial w_i} (y_d - \hat{y}_d) \\
&= \sum_{d \in \mathcal{D}} (y_d - \hat{y}_d) \frac{\partial}{\partial w_i} (\mathbf{w} \cdot \mathbf{x}_d - \hat{y}_d) \\
\frac{\partial E}{\partial w_i} &= (y_d - \hat{y}_d)(-x_{id})
\end{aligned} \tag{2.31}$$

In this derivation x_{id} defines a single component of the vector $\mathbf{x}_d$ of an training example d. Now we can express the weight update in gradient descent as:

$$\Delta w_i = \eta \sum_{d \in \mathcal{D}} (y_d - \hat{y}_d) x_{id}. \tag{2.32}$$

Finally we can express the gradient descent rule as seen in algorithm 2.2. We start by initializing the weights to small random values. In the main loop we receive the prediction of the linear unit for each training example d and calculate Δw_i (c.f. equation (2.32)). Finally each w_i is updated according to equation (2.30). This algorithm finds the global

Algorithm 2.2 Gradient Descent

```
 1: procedure GRADIENT_DESCENT(D, η)          // D contains the training examples
    in the form ⟨x, y⟩
 2:     w ← [|x_0|]          // Initialize an array dependent on the input size
 3:     w.map{ |elem| elem ← rand(0,1) }          // Assign each element a random value
    within [0,1]
 4:     while !stopping_criteria do
 5:         Δw ← zeros(|x_0|)          // Initialize an array dependent on the input size
    with zeros
 6:         for all ⟨x, y⟩ ∈ D do
 7:             ŷ ← linear_unit(x)
 8:             for all w_i ∈ w do
 9:                 Δw_i ← Δw_i + η(y − ŷ)x_i
10:             end for
11:         end for
12:         for all w_i ∈ w do
13:             w_i ← w_i + Δw_i
14:         end for
15:     end while
16: end procedure
```

minimum sooner or later if a decent small learning rate η is selected. Strategies like reducing the learning rate as the training progresses will be discussed later on. However some prerequisites for the usage of gradient descent exist. The hypothesis space must contain continuously parametrized hypotheses and the error must be differentiable with respect to the hypothesis parameters. These limitations result in practical issues like the existence of more than one local minima. In this case it is not assured that gradient descent converges towards the global minimum. Furthermore, the convergence can be laggard. To overcome this issue a special variant called stochastic gradient descent exists. Instead of applying the weight update after iterating through all training examples, the weights are updated after each calculation of w_i. Regarding algorithm 2.2 we simply remove the for loop in lines 12 to 14 and replace the equation in line 9 with $w_i \leftarrow w_i + \eta(y - \hat{y})x_i$.

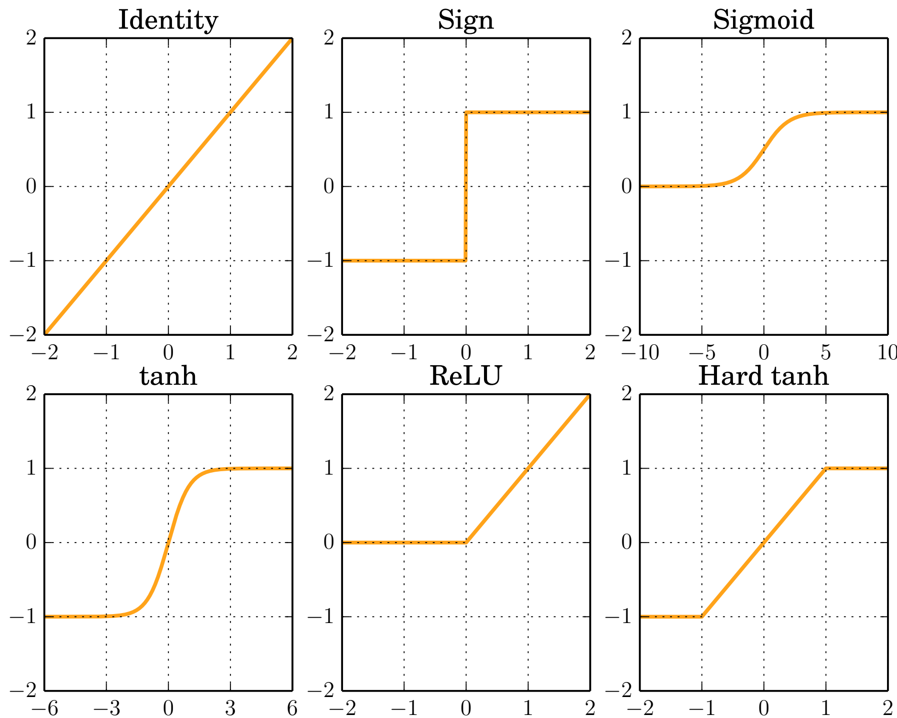


Figure 2.4: Different activation functions

For we can define the error function of each distinct training example d as:

$$E_d(\mathbf{w}) = \frac{1}{2}(y_d - \hat{y}_d)^2. \tag{2.33}$$

Then stochastic gradient descent updates the weights for each training example according to $E_d(\mathbf{w})$, whilst iterating through all training examples $d \in \mathcal{D}$. This method represents an approximation to the initial approach with the error function $E(\mathbf{w})$ and its respective gradient. Another advantage of stochastic gradient descent is the smaller number of computations performed per weight update. However, gradient descent may be used with a higher learning rate as it utilizes the true gradient. As mentioned earlier, due to the usage of different $E_d(\mathbf{w})$ instead of one $E(\mathbf{w})$ stochastic gradient descent is less vulnerable to converge to a local minimum.

2.2.1.4 Activation Functions

When introducing the perceptron, we assumed the classification of binary data. For this purpose the sign function was chosen as activation function as it fits this task. However, situations might arise where other kinds of data need to be classified. We show different kinds of activation functions in figure 2.4. We use these activation functions $\Phi(\cdot)$ to

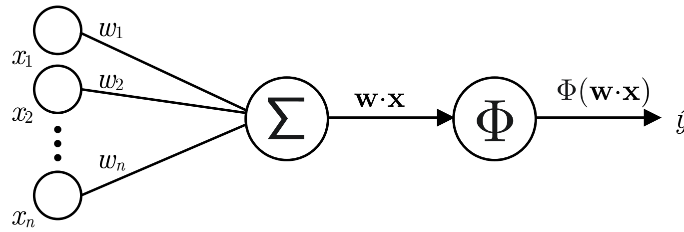


Figure 2.5: A neuron broken down into pre-activation and post-activation value

abstract the perceptron to an arbitrary neuron. We start with the general definition of the activation function:

$$\hat{y} = \Phi(\mathbf{w} \cdot \mathbf{x}) \tag{2.34}$$

The computation of a neuron can be broken into two different parts. First the so called pre-activation value is calculated by performing the dot product $\mathbf{w} \cdot \mathbf{x}$. Afterwards, activation function $\Phi(\cdot)$ is applied to the pre-activation value. This results is the output of the neuron, the so called post-activation value. This breakdown is shown in figure 2.5.

Continuing with the different activation functions, the simplest one is the identity function:

$$\Phi(v) = v \tag{2.35}$$

This activation function does not provide any non-linearity and is used for real valued target values. The sign function:

$$\Phi(v) = \text{sgn}(v) \tag{2.36}$$

was already introduced it in the beginning and it is suited for predicting binary target values. However, the sign function is not differentiable. The sigmoid activation function:

$$\Phi(v) = \frac{1}{1 + e^{-v}} \tag{2.37}$$

provides a probabilistic approach. It outputs a value in the interval [0,1]. The output is interpreted as the probability that the target value is 1. Another function, the tanh function, is similar in shape to the sigmoid function:

$$\Phi(v) = \tanh(v) = \frac{e^{2v} - 1}{e^{2v} + 1} \tag{2.38}$$

This function is re-scaled to an interval [-1, 1] and is consequently used if additionally

negative outputs are needed. Moreover it has beneficial features like a larger gradient, which results in faster training, and it is mean centred.
In the past the sigmoid and tanh activation functions were seen as sate of the art choices for applying non-linearity to a neuron. However, recently, functions which offer piecewise linear activation became more popular. The functions ReLU:

$$\Phi(v) = \max\{v, 0\} \tag{2.39}$$

and hard tanh:

$$\Phi(v) = \max\{\min\{v, 1\}, -1\} \tag{2.40}$$

provide a derivative of 1 in a certain interval. If units using these functions operate within this interval, the so called vanishing gradient problem occurs more rarely. The vanishing gradient problem will be described later on in detail. Further, ReLU provides a gradient which is faster to compute in respect to other activation functions.

2.2.1.5 Number Of Output Nodes

Next, we will cover deciding on the number of output nodes. This choice is tied to the problem scenario. For example when classifying k different classes we can use k outputs with a so called softmax activation function. Assuming the output nodes receive the input $\mathbf{v} = [v_1, v_2, \ldots, v_k]$ softmax is defined for the ith element:

$$\Phi(\mathbf{w})_i = \frac{e^{v_i}}{\sum_{j=1}^{k} e^{v_j}}. \tag{2.41}$$

Softmax calculates for all k classes how likely the input can be assigned to that class. Looking ahead to multi-layer networks, a common combination is a linear activation function in the last hidden layer, followed by a softmax activated output layer. Softmax has no associated weights as it only converts numbers into probabilities.

2.2.1.6 Loss Functions

On a final note we present some commonly used loss functions. The loss function should be chosen based on the desired outcome or goal. The simplest possibility is a squared loss of the form:

$$L = (y - \hat{y})^2 \tag{2.42}$$

Another common loss type is hinge loss. It can be used for binary as well as real-valued prediction. It is defined as:

$$L = \max\{0, 1 - y \cdot \hat{y}\} \tag{2.43}$$

For example, this loss is used with least-square regression to predict numeric outputs. We already introduced softmax as a probabilistic way to predict different classes or categories of targets. Nevertheless, one could also use the identity function to assign probabilities in case of a binary classification. In this case the output is a real valued number q. The probability assigned to +1 and -1 is consequently q and $1-q$. In this case the loss function of choice is called logistic regression:

$$L = \log(1 + e^{-y \cdot \hat{y}}) \tag{2.44}$$

However, other loss function and activation functions would provide a combination suitable for this learning task. In general there are several approaches to achieve the same results. Finally, we talk about using softmax for multi-way classification using a multi-class perceptron. We consider k classes $c(i) \in \{1, \dots, k\}$. Training data is given in the form of pairs $\langle \mathbf{x}_i, c(i) \rangle$, where $\mathbf{x}_i$ refers to the d dimensional input and $c(i)$ to the associated class of the ith training data pair. The perceptron predicts the class according to the maximal probability according to the output of the softmax activation function:

$$P(r|\mathbf{x}_i) = \frac{e^{\mathbf{w}_r \cdot \mathbf{x}_i}}{\sum_{j=1}^{k} e^{\mathbf{w}_j \cdot \mathbf{x}_i}} \tag{2.45}$$

where $P(r|\mathbf{x}_i)$ denotes the probability that $\mathbf{x}_i$ is of class r. Now the ith training set gets assigned the loss function L_i featuring cross-entropy. The cross-entropy refers to the negative logarithm of the probability for a label $c(i)$ given the input $\mathbf{x}_i$. Finally we recall $v_c(i) = \mathbf{w}_{c(i)} \cdot \mathbf{x}_i$ as the pre-activation value handed to the softmax node. Now we finally derive the corr-entropy loss:

$$\begin{aligned} L_i &= -\log\left[P(c(i)|\mathbf{x}_i)\right] \\ &= -\mathbf{w}_{c(i)} \cdot \mathbf{x}_i + \log\left[\sum_{j=1}^{k} e^{\mathbf{w}_j \cdot \mathbf{x}_i}\right] \\ &= v_{c(i)}) + \left[\sum_{j=1}^{k} e^{v_{c(i)}}\right] \end{aligned} \tag{2.46}$$

In the end the choice of loss function always depends on the specific task we want to learn.

2.2.2 Multi-layer neural networks

We continue with multi-layer networks. Recalling the architecture of the perceptron at the beginning of section 2.2.1, this unit only consisted of an input and an output layer, whereby only the output layer performs any computation. In this architecture the calculations are completely visible to the user. Multi-layer networks, in contrast, contain additional layer between the input and output layer. These layers are referred to as hidden

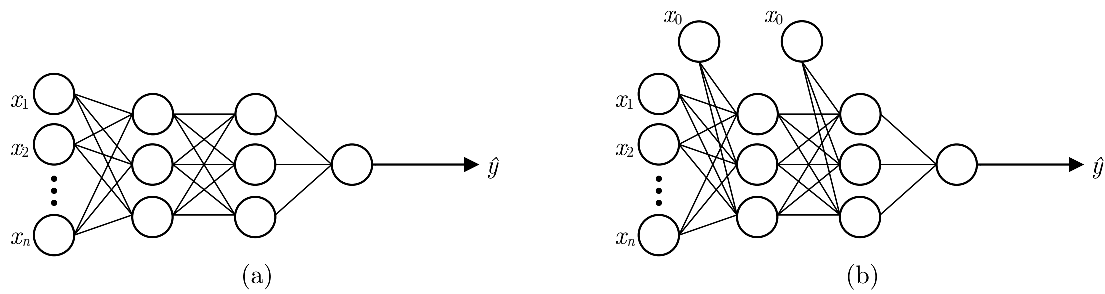


Figure 2.6: A multi layer network without (a) and with (b) additional bias neurons

layers. The calculations within these layers are not visible to the user. In some cases the input layer is not counted when describing multi-layer networks, as it does not take part in the computation. Starting from the input layer the inputs are fed into the respective subsequent layer in forward direction until the output layer is reached. Based on this sequence, these type of networks are also referred to as feed forward-networks. From an architecture point of view we need to define the number of hidden layers, the number of nodes within the layers and the activation functions of the nodes. In a standard feed-forward network all nodes of layer l are connected to the nodes of layer $l+1$. Finally the loss function to optimize must be defined. When predicting discrete values, a typical choice would be cross-entropy loss with softmax as activation function in the output layer. For predicting real values a good choice is squared loss combined with linear activation in the output layer.

2.2.2.1 Architectural Possibilities

Bias neurons can also be used in multi-layer networks. In figure 2.6 two 3-layer networks are depicted. The right one features bias neurons at each layer. Each layer has a dimensionality, set by the number of units it contains. The possible architectures for a multi-layer network are manifold. The individual architecture is defined by the goal we want to achieve with the network. For example, a binary like classification would call for one node in the output layer. However, for classifying numbers from 0 to 9 we would use ten output neurons, one for each possible digit. It is not necessary that two consecutive layers are fully connected. In some cases removing connections results in better predictions. Image recognition, for example, profits from these sparsely connected layers.

2.2.2.2 The Backpropagation Algorithm

Now as we established the architecture of multi-layer networks, we will now discuss how to train them. Training a multi-layer network comes with some challenges as the loss of each layer depends on the previous layers. Therefore the so called backpropagation algorithm is introduced. It computes the sum of local gradients along the paths through the network. We start with deriving the algorithm in detail and then provide the procedure in pseudocode form.
First we want the split the algorithm in two different parts, the first is the forward propagation of the input through the network and the second, the backpropagation of the error. In the forward part the input is given to the network which then computes its values according to the current weights in each layer until the predictions is emitted form the output layer. Then this output compared with the target given from the training example is fed into the loss function. In the backwards phase the derivative of the loss is computed based on the weights in the layers. The goal in this phase is to learn the gradient of the loss function of the respective weights. Finally the weights are updated backwards, starting from the output node based on the gradient.
We start with defining the loss function for our derivation. For simplicity we use a sigmoid activation function combined with squared loss of training example h:

$$L_h(\mathbf{w}) = \frac{1}{2}\sum_{l=1}^{k}(y_l - \hat{y}_l)^2 \tag{2.47}$$

The variable k denotes the number of possible classes y can be classified as. Therefore equation (2.47) is the squared error summed over all output nodes. Now we can express the gradient as $\frac{\partial L_h}{\partial w_{j,i}}$. The variable x_{ji} denotes the ith input of the jth unit in a layer and w_{ji} as the associated weights to this input. Further $v_j = \sum_{i=0}^{d} x_{ji}w_{ji}$ expresses the weighted sum of the input of unit j in a d dimensional layer. Now we rewrite the partial derivative as:

$$\begin{aligned} \frac{\partial L_h}{\partial w_{ji}} &= \frac{\partial L_h}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}} \\ &= \frac{\partial L_h}{\partial v_j} x_{ji} \end{aligned} \tag{2.48}$$

We continue to derive the training rule from $\frac{\partial L_h}{\partial v_j}$ for the case j is an output node and the case j is a node within a hidden layer. We start with the first case, that is j, an output node. Considering, that v_j influences the network through the output y_j of the node j. Consequently we can rewrite 2.48 to:

$$\frac{\partial L_h}{\partial v_j} = \frac{\partial L_h}{\partial \hat{y}_j}\frac{\partial \hat{y}_j}{\partial v_j} \tag{2.49}$$

First, we start with the left part:

$$\frac{\partial L_h}{\partial \hat{y}_j} = \frac{\partial}{\partial \hat{y}_j}\frac{1}{2}\sum_{l=1}^{k}(y_l - \hat{y}_l)^2 \tag{2.50}$$

We can set $k = j$ as all derivatives except this one will be zero. This results in:

$$\begin{aligned}\frac{\partial L_h}{\partial y_j} &= \frac{\partial}{\partial \hat{y}_j}\frac{1}{2}(y_j - \hat{y}_j)^2 \\ &= \frac{1}{2}2(y_j - \hat{y}_j)\frac{\partial(y_j - \hat{y}_j)}{\partial \hat{y}_j} \\ &= -(y_j - \hat{y}_j)\end{aligned} \tag{2.51}$$

Now we tackle the second part of equation (2.49). For this purpose we need the derivative of the sigmoid function already presented in section 2.2.1.4:

$$\frac{\mathrm{d}\Phi(v)}{\mathrm{d}v} = \Phi(v) \cdot (1 - \Phi(v)) \tag{2.52}$$

Therefore we express the right side as:

$$\begin{aligned}\frac{\partial \hat{y}_j}{\partial v_j} &= \frac{\partial \Phi(v_j)}{\partial v_j} \\ &= y_j(1 - y_j)\end{aligned} \tag{2.53}$$

If we now combine equation (2.51) and (2.51) we get:

$$\frac{\partial L_h}{\partial v_j} = -(y_j - \hat{y}_j)\hat{y}_j(1 - \hat{y}_j) \tag{2.54}$$

Finally we can express the stochastic gradient descent rule for output nodes as:

$$\Delta w_{ji} = -\eta\frac{\partial L_h}{\partial w_{ji}} = \eta(y_j - \hat{y}_j)\hat{y}_j(1 - \hat{y}_j)x_{ji} \tag{2.55}$$

Now we are left with the case, that the unit j lays within a hidden layer. In this case j influences all subsequent units in the network which include the output of j in their input combination. We denote this set of subsequent units as $S(j)$. Now we find that v_j can only influence the network via the units within $S(j)$. Further we define $\delta_i = -\frac{\partial L_h}{\partial v_i}$. This

leads to:

$$\begin{aligned}
\frac{\partial L_h}{\partial v_j} &= \sum_{l \in S(j)} \frac{\partial L_h}{\partial v_l} \frac{\partial v_l}{\partial v_j} \\
&= \sum_{l \in S(j)} -\delta_l \frac{\partial v_l}{\partial v_j} \\
&= \sum_{l \in S(j)} -\delta_l \frac{\partial v_l}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial v_j} \\
&= \sum_{l \in S(j)} -\delta_l w_{lj} \frac{\partial \hat{y}_j}{\partial v_j} \\
&= \sum_{l \in S(j)} -\delta_l w_{lj} \hat{y}_j (1 - \hat{y}_j)
\end{aligned} \tag{2.56}$$

Finally we can express $-\frac{\partial L_h}{v_j}$ as δ_j which results in:

$$\delta_j = \hat{y}_j (1 - \hat{y}_j) \sum_{l \in S(j)} \delta_l w_{lj} \tag{2.57}$$

Finally we can express the rule for stochastic gradient descent as:

$$\Delta w_{ji} = \eta \delta_j x_{ji} \tag{2.58}$$

Algorithm 2.3 Gradient Descent

1: **procedure** BACKPROPAGATION($\mathcal{D}, \eta, n_{in}, n_{out}, n_{hidden1}, \ldots, n_{hidden,n}$)
2: nn ← initialize_network($n_{in}, n_{out}, n_{hidden1}, \ldots, n_{hidden,n}$)
3: **while** !stopping_criteria **do**
4: **for all** $\langle \mathbf{x}, y \rangle \in \mathcal{D}$ **do**
5: $\hat{\mathbf{y}} \leftarrow$ nn.predict($\mathbf{x}$) // Forward propagation step
6: **for all** $l \in n_{out}$ **do** // Backpropagate the output units
7: $\delta_l \leftarrow \hat{y}_l(1 - \hat{y}_l)(y_l - \hat{y}_l)$
8: **end for**
9: **for all** $n \in [n_{hidden-1}, \ldots, n_{hidden,n}]$ **do** // Propagate the hidden units
10: **for all** $r \in n$ **do**
11: $\delta_r \leftarrow \hat{y}_r(1 - \hat{y}_r) \sum_{s \in S(j)} w_{lr} \delta_s$
12: **end for**
13: **end for**
14: $\Delta w_{ji} = \eta \delta_j x_{ji}$
15: $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$
16: **end for**
17: **end while**
18: **end procedure**

Now we continue to describe the algorithm in pseudocode and give some more insight to the procedures. Algorithm 2.3 start by initializing a neural network with a predefined architecture. Afterwards the network is trained until a certain stopping condition is met.

Possible criteria are a fixed number of iterations, a specific error threshold the algorithm needs to surpass or a rising error on a separate validation set. Within the main loop, for each available training example, the network generates the predicted outcome according to the input. For each output node l the error term δ_l is calculated according to equation (2.54). The error term for the hidden units is not as straight forward because the neurons cannot be linked to a specific prediction $\hat{y}$. To compute δ_r for a hidden neuron k we use the error term δ_s of each neuron which is directly influenced by k. Meaning δ_k is computed using all δ_s from the next deeper layer. After each processed training example the weighs w_{ji} are adapted according to the computed errors.

2.2.2.3 Optimization Techniques

We will now present several common optimization techniques for neural networks following [9]. These techniques fortify the presented backpropagation to get faster conversion rates as well as better results.

Mini-Batch:

Currently we described batch learning, where the algorithm iterates over all available training data and updates the weights accordingly. Secondly we described stochastic gradient descent. In this method the algorithm performed a weight update after each processed training example. Both methods have their respective disadvantages. Batch learning is slow and vulnerable to local minima as well as saddle points. Stochastic gradient descent is more expensive to calculate and due to the high frequent updates the gradient becomes noisy. This on the one hand can avoid local minima but also makes it hard to settle on the global minimum. Mini-batch utilizes a mix of both methods. The training data is divided in different mini-batches of size n. In training the weights are updated after each mini-batch is processed. This method combines the advantages of the upper mentioned methods. It converges stably to the global minimum, it conserves the computational resources and provides fast learning. The only downside is an additional hyperparameter to be considered.

Input Scaling:

Consider a network with two different input nodes. The first node receives inputs in a range [100,1000] and the second one within a range [0.001, 0.01]. In this setting the classifier has a hard time adjusting the weights accordingly as an error on the second input parameter will go down under the huge error values provided by the first input. However, the classifier should not distinguish between the different inputs and should value them evenly. To overcome this issue we can normalize an standardize the data. Normalization scales all inputs to fit into a range [0,1] and is done via min-max normalization. Considering a training set $\mathcal{D}$ we define x_{max} as the highest input value of the training set and

x_{min} as the lowest. We then can use the min max normalization as follows:

$$x_{norm} = \frac{x_{input} - x_{min}}{x_{max} - x_{min}} \tag{2.59}$$

where x_{input} is the original input value and x_{norm} its respective normalized one.
When using standardization we rescale the input to have the properties of a normal distribution. We calculate the mean μ_x and the standard deviation σ_x of the given training input to utilize the so called Z-score normalization:

$$x_{std} = \frac{x_{input} - \mu_x}{\sigma_x} \tag{2.60}$$

which results in the standardized value x_{std} for each training input. The training set afterwards features the input values within a range [-1,1] with $\mu = 0$ and $\sigma = 1$.

Number of Hidden Units

The number of input and output units is already predefined by the problem. The number of input units however must be chosen. Generally the number of hidden units defines the expressive power of the network and is therefore dependent of the learning problem. Using too many hidden units might result in a higher likeliness of overfitting whereas with to too few units the net can not adapt to the training data. A method to determine the number of hidden units to use is to start with a large number and prune them afterwards.

Initializing Weights

Needless to say that if we would initialize the weights at 0 we would not be able to train at all as of equation (2.55). We need to set the initial weights in a way that the training can be fast as well as uniform. Meaning that all weights hit their desired value at the same moment. We use a predefine probability distribution to initialize the weights randomly. We chose a distribution of the form $-\tilde{w} < w < +\tilde{w}$. If $\tilde{w}$ is set too high, the weights might saturate much too early. Otherwise if they are chosen too low, the net activation will be too small and tend to a linear model. For a hidden unit which gets input from d input units the net activation will be $\tilde{w}\sqrt{d}$. Provided we use standardized inputs with $\sigma = 1$. As our inputs range within -1 and 1 we also want our net activation to be within this range. This results in $\tilde{w} = \frac{1}{\sqrt{d}}$ for the weights of the input units. Therefore we initialize them randomly within $-\frac{1}{\sqrt{d}} < w_{ji} < +\frac{1}{\sqrt{d}}$. In a same way we can define the interval for the weights between hidden units and output units which results in the interval $-\frac{1}{\sqrt{n_h}} < w_{ji} < +\frac{1}{\sqrt{n_h}}$ with n_h as number of hidden units.

Learning Rate

The learning rate determines the speed we move towards the minimum. However this

holds only if the learning rate is chosen sufficiently small enough to ensure convergence. Since in practice the training is most likely not proceeded until the network converges to the minimal training error the learning rate is relevant for the shape of the final network. If during learning the training is slow or the loss is not reduced at all we might increase the learning rate. On the other hand if the loss diverges the learning rate is too high. A common practice is to reduce the learning rate as learning progresses.

Momentum

Momentum is used to overcome plateaus on the error surface. To use momentum the weight update rule is altered to:

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1) \tag{2.61}$$

which represents a weight update during the nth iteration. The constant $0 \leqslant \alpha \leqslant 1$ is called momentum. The second term in this equation, called momentum term, ensures a constant movement of the gradient. Using this, the algorithm might overcome local minima. Additionally convergence is faster because the algorithm has an increased step size in regions with a similar gradient.

Early Stopping

During training it can happen that the network is trained to fit the training data rather than the general data and its respective distribution. Therefore training should be stopped before gradient descent is finished. A common method used is withhold a separate validation set and stop training once the error reaches a minimum on this set.

2.3 Deep Reinforcement Learning

Now we will resume with reinforcement learning, where we left off in section 2.1.6 at function approximation using a linear function with feature weights $\mathbf{w}$. The disadvantage of this linear approach is that the linear function might not be sufficient to model the action-value function we will use in this thesis. To overcome this, we can use a non-linear function approximator like an MLP can be used [10].

In the recent years, breakthroughs were made using Deep Neural Networks (DNN) as function approximation for deep reinforcement learning (DRL). DNNs are neural networks containing more than one hidden layer [11]. For the sake of completion we want to note that many of these DRL approaches use different classes of DNNs such as Convolutional Neural Networks or Recurrent Neural Networks. However, they are beyond the scope of this thesis.

To utilize DRL we describe the deep-Q-learning algorithm. The basic idea of this al-

gorithm is to save observed state transition in a so called memory replay and use this data to train a DNN which predicts Q values based on possible actions. We describe our implementation of this algorithm in detail in chapter 3.
This algorithm provides several advantages over the classical Q learning approach. First, it provides a better usage of the data as we might use one data point in the replay memory in several future weight updates for the DNN. The memory replay also allows the retrieval of a random mini-batch containing uncorrelated state transitions [12].

2.3.1 Target Network

The above mentioned approach uses the same DNN to predict the $\hat{Q}$ values of the current and the next state. This happens when the network is trained on a transaction of a retrieved mini-batch. This transaction contains a state s_t. We predict the $\hat{Q}_{t+1}$ estimate of a proceeding state to determine the maximum possible $\hat{Q}_{t+1}$. We use this estimate and the reward r_t from the transition to find the maximum possible Q_{target} value for the initial state (see equation (2.14)). Once we performed this for all transitions within the mini-batch, we utilize the DNN again to get the current prediction $Q_{predicted}$ to then calculate the error as difference to Q_{target} and subsequently the loss.
This approach might lead to an instability during the training. To overcome this, we use two DNNs: a main network, which we use to calculate $Q_{predicted}$ and a target network for Q_{t+1}. The weights of the main network are updated every time a terminal state is reached during exploration, while the target network is only updated after several iterations.

3 Implementation

3.1 Deep Q-Learning Approach

3.1.1 Construction of the MDP

Based on section 2.1.2 we will now introduce our MDP model for the learning task. Note that in the provided pseudocode $\mathbf{x}$ represents a vector or array and $\mathbf{X}$ a matrix. Global configuration parameters are written in capital letters (e.g.: SOME_CONFIG). An instance object is written as `Object`.
The result of the system analysis indicates that the role shift leader seems most suitable to be used as an agent in our model. The shift leader is in charge of allocating the operators to the related process steps. We identified the following agent actions:

- Send operators home early
- Make operators work longer
- Reallocate an operator

At the beginning of the shift, the shift leader is given a total operator pool O. Using the actions stated above, the shift leader is able to manage his system with the given operator pool at the current time interval $O_t \subset O$ based on the current performance for each of the $|W|$ process steps and the incoming orders. Due to the possibility of sending operators home the available operators $o \in O_t$ can differ for different time intervals t.
Now we present our MDP modelled in order to represent the real life situation given in section 1.2. For our model we distinguish between two state spaces S and S^*. S represents the complex state space containing all given information. We use S for the simulation of activity data (see 3.2) and calculation of the reward. S^* is the compressed state space used for training (see 2.1). S and S^* are connected via the compression function $h(S) = S^*$. This is done to reduce the amount of parameters the learning algorithm is confronted with. However, as mentioned above, we want to use the more detailed information in some cases.
Actions: $A = \left\{a_1, \ldots, a_{|A|}\right\}$. These represent the actions identified in the beginning of this section. Additionally an action representing doing none of the decisions mentioned above is also included. This action can be used in case the current state is considered

optimal by the policy. Further, the action a_{stay} is removed because its effect cannot be mapped into the state space described later. This action will be depicted implicitly. The default behaviour of a shift is altered to an artificial setting, where operators stay until the end of the total workday, including overtime. Meaning per default all operators will work a maximum of overtime if not actively sent home earlier. This leads to the following action space:

- $a_{\text{leave}}(n)$: Make n operators leave early
- a_{nothing} : Do nothing
- $a_{\text{swap}}(o, w)$: Reallocate operator o to process step w

The amount of actions $|A|$ varies depending on the current state. It is important to note, that for an action $a_{\text{leave}}(n)$ only the amount of operators which will be sent home can be chosen. This helps to reduce the amount of possible actions and is therefore beneficial for the runtime of the training. This process is described in detail in section 3.2.2.2
States: $S = \left\{s_1, \ldots, s_{|S|}\right\}$. Every state is defined by the current allocation $\alpha_{w,t}$ of operators to process steps, the filling of the order buffers $\beta_{w,t}$ and the maximum throughput $\gamma_{w,t}$ at each process step. This results in $s = \left[\alpha_{1,t}, \ldots, \alpha_{|W|,t}, \beta_{1,t}, \ldots \beta_{|W|,t}, \gamma_{1,t}, \ldots \gamma_{|W|,t}\right]$. In regards to the allocation and the maximum throughput, recalling 1.2, it is important to highlight that every operator has an individual skill set which defines the possible work capacity for each process step. Therefore the allocation $\alpha_{w,t} \subset O_t$ defines the available work capacity at each process step w for a certain time interval t during the shift. We define the work capacity at w provided by operator o as $l_{w,o}$.
The maximum throughput $\gamma_{w,t}$ for a process step w is defined as follows:

$$\gamma_{w,t} = \max_{O_i \subset O_t : m_{w,min} \leqslant |O_i| \leqslant m_{w,max}} \sum_{o \in O_i} l_{w,o} \tag{3.1}$$

The filling of the buffer $\beta_{w,t} = |X_{w,t}|$ is given by the amount of orders $|X_{w,t}|$ the respective buffer of process step w contains at the time interval t. Note that $\beta_{1,t}$ is always filled directly by the incoming orders at the time interval t.
Regarding the compressed state space $S^* = \left\{s_1^*, \ldots, s_{|S^*|}^*\right\}$ we continue by defining the compression function $h(S) = S^*$. A compressed state is defined by the total available work capacity $\omega_{w,t}$, the accumulated workload $\beta_{w,t}$ and the maximum throughput $\gamma_{w,t}$, each for one process step w. We get $\omega_{w,t}$ for a process step by adding up the possible work output of each assigned operator:

$$\omega_{w,t} = \sum_{o \in \alpha} l_{o,w}. \tag{3.2}$$

The maximum throughput $\gamma_{w,t}$ and the buffer $\beta_{w,t}$ will not be compressed any further as it is already defined by only one value per w. In total this leads to $3|W|$ parameters per state.

Rewards: We designed our reward function to return the profit gained after the end of every interval t. The profit is calculated by comparing the achieved revenue with the costs incurred. The profit is influenced by the total number of orders processed at the last process step and therefore considered fulfilled. Hence we define $\beta_{|W|,t}$ as finished workload on the last process step. The constant $c_{revenue}$ defines the revenue for a finished workload equivalent to a single order of the amount one. Now the complete revenue can be stated as:

$$revenue = \beta_{|W|,t} * c_{revenue}. \tag{3.3}$$

To calculate the profit, the revenue of the finished workload has to be reduced by the costs for operators. For simplicity we define the cost per operator c_{nt} during normal time for $t = 15$min to be constant and independent of the individual operator o and their skills. Therefore the total cost for $|\alpha|$ assigned operators is:

$$|\alpha| * c_{nt}. \tag{3.4}$$

Further, we define the additional costs for overtime in the same way using c_{ot} as additional cost per operator per time interval:

$$|\alpha| * c_{ot}. \tag{3.5}$$

This leads for the reward function $r(s, a)$ to be defined as follows:

$$r_t(s, a) = \beta_{|W|,t} * c_{revenue} - |\alpha| * c_{nt} - [|\alpha| * c_{ot}]_{ifovertime} \tag{3.6}$$

However, to provide better feedback and give priority to the completion of all orders we add an additional *penalty* for not finished orders. Starting from one and a half hours into overtime we calculate the *penalty* based on the total sum of remaining workload over all given process steps. The constant $c_{penalty}$ defines the penalty for a workload containing only a single order of the amount of one:

$$penalty = \sum_{w \in W} \beta_{w,t} * c_{penalty} \tag{3.7}$$

This leads to the complete reward function:

$$r_t(s, a) = \beta_{|W|,t} * c_{revenue} - |\alpha| * c_{nt} - [|\alpha| * c_{ot}]_{ifovertime} - \sum_{w \in W} \beta_{w,t} * c_{penalty}. \tag{3.8}$$

Transition Probabilities: We now describe how a transition between two states hap-

pens in our model. At first, depending on the action taken, operators are either shifted from one process step to another, or a chosen amount of operators are removed from the system (i.e.: sent home). Which operators are sent home is determined by their current position in the leave priority cue. In our implementation the order of the operators in the leave priority cue is set randomly before the start of the shift. Once the changes to the operators assignments are carried out, the number of orders completed in a time interval t is passed from the buffer of one process step to the corresponding buffer of the next process step.
It is important to note that the number of orders at the first process step (i.e.: new incoming orders) are set randomly. The probability distribution that controls the number of incoming orders is covered in section 3.2.

3.1.2 Q-learning with function approximation by an MLP

We will now use the model described above to outline the implementation of our reinforcement learning algorithm in detail. The implementation is inspired by [13] and [14] and adapted to our needs. We used Python 3 with TensorFlow and Keras for the neural networks supported by NumPy. The UML diagram in figure 3.1 provides an overview of the classes used. After a short overview the detailed implementations are described.

- The `ExplorationExploitationScheduler` handles the balance between exploring new actions and exploiting profitable ones.
- The `MemoryReplay` acts as a data storage for experienced transitions by the agent. Further it returns the data in mini-batches which are used to train the neural network
- The MLP provides the architecture for the neural networks.
- The `Agent` manages the upper three classes. It performs the actual learning by calculating the loss and applying gradient descent to the network. It also updates the target networks accordingly
- The `TrainingLoop` uses the `Agent` to train the neural network according to preconfigured parameters

3.1.2.1 MLP

Based on the detailed explanation in section 2.1 we now propose our adapted neural network. We decided to use an MLP over a Convolutional Network as it is a better fit for our data. We also use a different network for each time interval t. This results in a total of 49 different MLPs. Each one of these networks receives a compressed state s^*

Figure 3.1: UML diagram of the reinforcement learning algorithm using an MLP

Name	**Value**	**Description**
MAX_IT	30×10^6	Defines the maximum of total iterations in training
MAX_EPISODE_LENGTH	1 shift	Defines the length of one training episode
TOTAL_ITS_BETWEEN_EVAL	50×10^3	Defines the number of iterations in-between evaluations
EVAL_LENGTH	100 shifts	Defines how many shifts are evaluated
TN_UPDATE_FQ	10×10^3	Defines the frequency of target network updates
ONLY_EXPL	50×10^3	Defines how many iterations only exploration takes place
GAMMA	1	Defines the discount factor
BATCH_SIZE	32	Defines the size of a mini-batch
LEARNING_RATE	0.0001	Defines the learning rate used for Adam
DO_NOTHING_PREFERENCE	0.95	Defines the probability of $a_{nothing}$ during exploration
USE_TARGET_NETWORK	True | False	Defines whether the agent uses the target network for learning or not
C_PENALTY	90	Defines the penalty for unfinished orders in overtime

Table 3.1: Configured hyperparamters for the training loop

as input and is expected to output the value $V(s^*)$. Based on those values the agent calculates the respective Q-function in order to choose the action to take. To achieve this we used a rather simple network architecture. In our model the input layer is a one dimensional array containing the parameters of the compressed state defined in section 3.1. Then follows twice the combination of a hidden layer with 400 neurons with ReLU activation followed by a batch normalization layer. The output layer has only one neuron as the network is intended to output only the value of the final state. This final layer uses a linear activation function. After establishing the layers, the network is compiled for the Adam optimizer with the given LEARNING_RATE and using Huber loss. Using the Huber loss has the advantage that it's not as sensitive for outliers as a squared error loss and is defines as follows:

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2, & \text{for } |a| \leqslant \delta \\ \delta \cdot (|a| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \tag{3.9}$$

The Adam optimizer was used instead of the proposed RMSProp used by Google DeepMind because Adam is less sensitive to different learning rates [15].

3.1.2.2 Exploration-Exploitation-Scheduler

In section 2.1.1 we already introduced the problem of choosing between exploration and exploitation when taking actions. We will now address this challenge in detail and provide possible solutions for it. A simple algorithm to control exploration and exploitation is called ε-greedy policy. A random action is taken with probability ε and the action yielding the maximum $\hat{Q}(s,a)$ value with probability $(1-\varepsilon)$.
We use ε-greedy with an varying ε. At first the agent only explores and ε is kept at $\varepsilon_{initial} = 1.0$. Then the agent starts to exploit more and more as ε is now decreased linearly to $\varepsilon_{halfway} = 0.1$. Finally ε decreases to $\varepsilon_{final} = 0.01$ according to [16]. The decay of ε is shown in figure 3.2.

3.1.2.3 Memory Replay

The `MemoryReplay` functions as storage for training data. It stores state transitions, the action that led to them and the received reward. The variable n_{total} determines the maximum number of transitions that are stored. In our case n_{total} was set to 10^6 records. The parameter $i_{current}$ holds the number of transitions currently stored. This is needed to determine the upper limit of the index which could possibly be returned. When saving a new transition, i_{next} specifies the next index to write to. As described in section 2.2.2.3 we use mini-batches to train the neural network. The size of these batches is set via n_{batch}. From the transition we store the initial state s_t, a snapshot of the `System` object before the action is taken and all possible actions $\mathbf{a}_{possible}$ at this state. We do not store

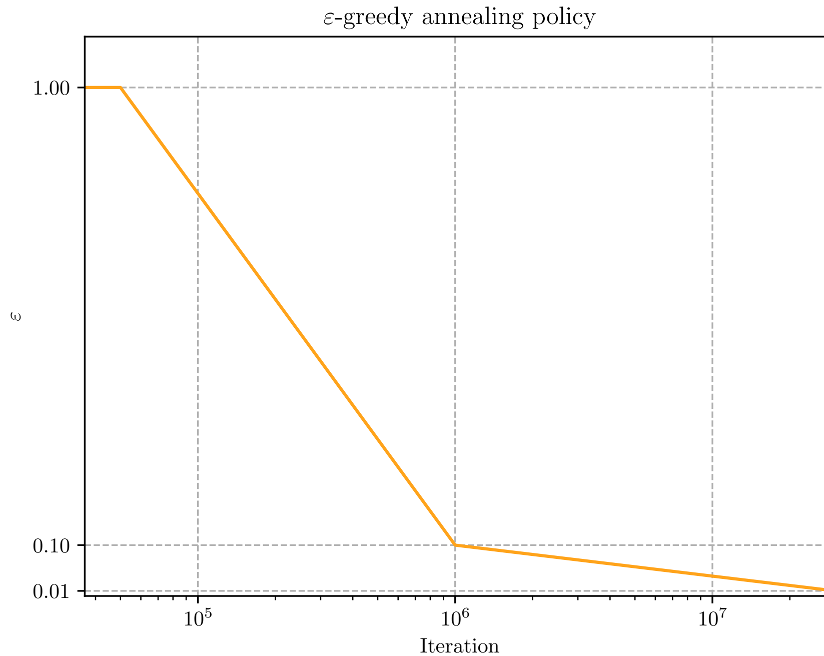


Figure 3.2: Linear decay of ε over time

information like the subsequent state s_{t+1} or the action taken explicitly as these are not needed in our case as we are able to gain this information by using the saved `System`.

3.1.2.4 Agent

The `Agent` class combines all the classes above mentioned together. The proposed ε-greedy policy is used by the method `get_action()`. In order to calculate ε the `Exploration-ExploitationScheduler` is utilized to select an exploitative or an exploratory action. When the choice is exploitation we try all possible actions a_t in the current state s_t to find the best $\hat{Q}(s,a)$. To do so we must perform action a, observe the new state s_{t+1}, predict $V(s_{t+1})$ and calculate $\hat{Q}(s,a)$ for all possible actions. Finally, we return the action with the highest $\hat{Q}$ estimate. However, if exploration is made and the action is chosen completely random with no additional boundaries we might face a bad overall distribution of states. This is caused by the action $a_{leave}(n)$ which limits the action space of all following states. There is also only one action of doing nothing in contrast to sending operators home early actions, as the amount of these actions scale with the number of operators currently working in the `System`. To balance this inequality the parameter DO_NOTHING_PREFERENCE was established. This parameters defines a hard coded preference for the action $a_{nothing}$. Each time the agent chooses exploration a second random number is generated. Only if this number is lager than the aforementioned parameter the `System` is influenced by a randomly chosen action. Otherwise $a_{nothing}$ is

Algorithm 3.1 Train the neural network

1: **procedure** LEARN(`System`)
2: $\mathbf{v}_{true} \leftarrow []$
3: $\mathbf{S}_t, \mathbf{system}_t, \mathbf{A}_{possible} \leftarrow$ replay_buffer.get_minibatch()
4: $i \leftarrow 0$
5: **while** $i <$BATCH_SIZE **do**
6: $\mathbf{s}_{t+1}, \mathbf{r}_t, \mathbf{terminal} \leftarrow []$
7: **for all** $a \in \mathbf{A}_{possible}[i, :]$ **do**
8: $s_{t+1}, r_t, terminal \leftarrow$ **system**[i].get_next_state_save(a)
9: $\mathbf{r}_t << (r_t)$
10: $\mathbf{s}_{t+1} << (r_t)$
11: $\mathbf{terminal} << (terminal)$
12: **end for**
13: **if** USE_TARGET_NETWORK **then**
14: $\mathbf{v}_{t+1} \leftarrow$ target_nn.predict($\mathbf{s}_{t+1}$)
15: **else**
16: $\mathbf{v}_{t+1} \leftarrow$ main_nn.predict($\mathbf{s}_{t+1}$)
17: **end if**
18: $\mathbf{v}_{true} << \max(\mathbf{r}_t + \text{GAMMA}\mathbf{v}_{t+1}(1 - \mathbf{terminal})$
19: $i \leftarrow i + 1$
20: **end while**
21: $\mathbf{v}_{pred} \leftarrow$ main_nn.predict($\mathbf{s}_t$)
22: $\mathbf{error} \leftarrow \mathbf{v}_{pred} - \mathbf{v}_{true}$
23: $loss \leftarrow \text{Huber}(\mathbf{v}_{pred}, \mathbf{v}_{true})$
24: main_nn.apply_gradients($loss$)
25: **return** $loss, \mathbf{error}$
26: **end procedure**

chosen.
The method `predict()` utilizes the respective neural network to predict the value based on the input state.
Finally the method `learn(System)` retrieves a mini-batch and trains the main network as seen in algorithm 3.1. Note that every mini-batch represents the `System` in a certain time interval t. First the algorithm retrieves a mini-batch from the `MemoryReplay`. The mini-batch is processed by two loops. The outer loop iterates through the batch itself. The inner loop selects each possible action which can be taken by the current `System` of this mini-batch. Each of these possible actions is taken and the resulting state s_{t+1} as well as the reward r_t are received. The value of state s_{t+1} is predicted by the network. For each sate of the batch the value V_t is calculated as:

$$V_t = r_t + \max V_{t+1} \tag{3.10}$$

The `Agent` can either use the target network to predicts V_{t+1} or the main network. Additionally, V_{t+1} is set to zero in the case that s_{t+1} is the terminal state of the `System`. The

main network is then used to predict the values V_{pred} for the respective states s_t in the mini-batch. Now the error for each predicted value V_{pred} can be calculated:

$$error = V_{pred} - V_{true} \tag{3.11}$$

Finally, we utilize the Huber function to compute the total loss of the predictions on the mini-batch based on the error. The loss is utilized to apply gradient descent on the current main network to adapt the weights within.

3.1.2.5 Training Loop

The class `TrainingLoop` is the core element of the learning algorithm and brings all classes mentioned before together. It is initialized with a `System` and an `Agent` containing the main network, target network, `MemoryReplay` and `ExplorationExploitationScheduler`. The state number i_{curr} is set to 0. The method `run()` represents the main method of the entire learning algorithm (see algorithm 3.2). The main loop, in which the whole training is embedded, runs until the pre-configured limit is reached, (see table 3.1.2 for details on configured parameters). Within this loop, the training is divided into epochs. Each epoch contains a pre-configured number of episodes. The `System` is reset at the beginning of each episode. An episode lasts until the end of a shift. At each iteration of an episode an action a_t is chosen and performed. The received transition is then stored in the `MemoryReplay`. If one of the configured update frequencies is reached, the main network is trained or the weights of the target network are updated. These updates are only performed after the initial phase of total exploration. At the end of each epoch the algorithm evaluates the current network. For evaluation the same procedure of receiving an action, taking it and saving the reward. This is repeated until the configured evaluation iteration limit is reached. After each evaluation loop the evaluation results are displayed.

3.2 Simulation

This section will describe the implementation of a simulated system used in this thesis, as described in section 1.2. At first we give a brief explanation why an extensive simulation is used. Then the implemented simulation is described in more detail.

3.2.1 Why Simulation is Needed

The process of training a reinforcement learning policy necessitates data in the form of state transitions. The first intuition might be to generate the data upfront and feed it to the system during training. With this approach the data would also be extendable with data collected from a real life situation. However, this method has disadvantages which

Algorithm 3.2 The Training Loop

1: **procedure** RUN
2: $i_{curr} \leftarrow 0$
3: **while** $i_{curr} <$ MAX_IT **do**
4: $i_{epoch} \leftarrow 0$
5: **while** $i_{epoch} <$ TOTAL_ITS_BETWEEN_EVAL **do**
6: `System`.reset()
7: **while** True **do**
8: $a_t \leftarrow$ `Agent`.get_action(s, `System`)
9: $s_{t+1}, r_t, terminal \leftarrow$ `System`.perform_action(a_t)
10: $i_{curr} \leftarrow i_{curr} + 1$
11: $i_{epoch} \leftarrow i_{epoch} + 1$
12: $i_{episode} \leftarrow i_{episode} + 1$
13: `Agent`.add_experience(s_t, `System`, $\mathbf{a}_{possible}$)
14: **if** *terminal* && `Agent`.replay_buffer.size > ONLY_EXPL **then**
15: `Agent`.learn()
16: **end if**
17: **if** i_{curr}mod TN_UDPATE_FQ == 0 &&i_{curr} >ONLY_EXPL **then**
18: `Agent`.update_target_network()
19: **end if**
20: **if** *terminal* **then**
21: **break**
22: **end if**
23: **end while**
24: **end while**
25: `System`.reset()
26: $r_{sum} \leftarrow 0$
27: *terminal* $\leftarrow$ False
28: $\mathbf{r}_{eval} \leftarrow []$
29: **while** $i_{eval} <$ EVAL_LENGTH **do**
30: $a_t \leftarrow$ `Agent`.get_action(`System`)
31: $_, r_t, terminal \leftarrow$ `System`.perform_action(a_t)
32: $r_{sum} = r_t + r_{sum}$
33: **if** *terminal* **then**
34: $\mathbf{r}_{eval} << r_{sum}$
35: `System`.reset()
36: $r_{sum} \leftarrow 0$
37: *terminal* $\leftarrow$ False
38: break
39: **end if**
40: **end while**
41: $\overline{r}_{eval} \leftarrow$ mean($\mathbf{r}_{eval}$)
42: **print** $\overline{r}_{eval}$
43: **end while**
44: **end procedure**

would artificially limit the reinforcement learning policy. Depending on the amount of data and the algorithm used to generate it, the agent will take actions which lead to states not present in the data. This is also a key disadvantage which rules out training only on real life data. In real life only data of one specific way of handling the system would be present. However, the reinforcement learning policy also needs data apart form this given way to learn a good strategy. Therefore, a simulation which communicates dynamically with the reinforcement learning policy was chosen. The main functionality of this simulation is to receive an action and calculating the resulting state. The simulation also handles the reward calculation and the initial assignment of the operators at the beginning of the shift. These details of this simulation are presented in the following chapter. The simulation was designed to be as general as possible in order to be able to be reused in future experiments.

3.2.2 Simulation Approach

3.2.2.1 Initialization

The simulation is initialized with several parameters. First the simulation is given an order simulation object which generates orders based on an predefined probability distribution. The order generation and its options are presented in more detail in the subsection 3.2.3. For now we can view the order generation as an object that outputs an array containing the incoming orders for each time interval t. The next initialization parameter is a matrix $\mathbf{M}_{minmax}$ containing information about the minimum and maximum assignable operators per process step in the system:

$$\mathbf{M}_{minmax} = \begin{bmatrix} m_{1,min} & m_{1,max} \\ \vdots & \vdots \\ m_{|W|,min} & m_{|W|,max} \end{bmatrix} \tag{3.12}$$

where $|W|$ is the number of process steps the system contains. Next the maximum operator performance for each operator and process step defined as matrix:

$$\mathbf{L}_{perf} = \begin{bmatrix} l_{1,1} & l_{1,2} & \cdots & l_{1,|W|} \\ l_{2,1} & l_{2,2} & \cdots & l_{2,|W|} \\ \vdots & \vdots & \ddots & \vdots \\ l_{|W|,1} & l_{|W|,3} & \cdots & l_{|W|,|O|} \end{bmatrix} \tag{3.13}$$

using the already proposed notation $l_{w,o}$ for the performance of an operator o at process step w. The input array **times** represents the length of the normal sift and the overtime:

$$\textbf{times} = \begin{bmatrix} times_{nt} \\ times_{ot} \end{bmatrix} \tag{3.14}$$

where $times_{nt}$ describes the duration of the normal time and $times_{ot}$ the duration of the overtime of a shift. Finally the cost and revenue parameters are defined as follows:

$$\mathbf{c} = \begin{bmatrix} c_{nt} \\ c_{ot} \\ c_{revenue} \end{bmatrix} \tag{3.15}$$

whereas c_{nt} describes the cost of an operator for a time interval during normal time and c_{ot} the cost during overtime.
Now we continue with the actual initialization of the simulated system. First, an array representing the leave priority queue is created. The leave priority is used to determine which operator should be send home next if a $a_{leave}(n)$ action is taken. Therefore, the leave priority queue contains all operators in a random order. The simulation then performs an initial assignment of the given operators to the process steps. We start with a simple assignment where we loop $|O|$ times through the process steps and assign the operator with the highest performance to the respective process step. Then, the assigned operators or rather their performance is summed to determine the total workforce available at each process step. This represents half of the values of the state space. To get the other half we call the order simulation to generate an order array. The first order is then immediately pushed into the buffer of the first process step completing the initial state of the system. Finally, the remaining shift time is set according to the respective input parameter.

3.2.2.2 State and Action Management

As mentioned before the core functionality of the simulation is the calculation of a new state based on an action input. The procedure starts by determining which kind of action needs to be executed. If an action a_{leave} or a_{swap} is chosen, the assignment is changed accordingly. Either an operator and its performance are moved from one process step to another or the chosen amount of operators are removed entirely from the system. The simulation then performs the actual state transition. First, each process step outputs the amount of processed orders which are then either put into the buffer of the subsequent process step or counted as finalized workload (if the last process step is passed). Then the new orders are populated as orders into the buffer of the first process step. Afterwards the reward is calculated based on the finished orders and the costs incurred as described in paragraph 3.2.2.3. The algorithm continues by checking if the shift has ended. If so, a variable *trerminal* is set to true. Otherwise the remaining shift time is reduced. Finally the system sets the resulting state by performing the chosen action as current state.

The system can then output all actions that can be taken with respect to the current state. Assuming the current state is not a final one the logic works as follows:

- $a_{\text{leave}}(n)$: This action set provides the availability to sent home any amount of operators that are currently present.
- a_{nothing} : This action is available anytime without any further constraints.
- $a_{\text{swap}}(o, w)$: This action is only available within the given constraints of $\mathbf{M}_{minmax}$. Meaning if a process step only has $m_{w,min}$ operators currently assigned no operator can be removed. Consequently, if already $m_{w,max}$ operators are assigned, no further operators can be assigned to this process step.

A system in a terminal state supports no further actions.

3.2.2.3 Reward Calculation

The calculation of the reward for a performed state transition is also an essential feature of the simulation as it provides direct feedback to the reinforcement learning policy about the chosen action. The reward function is already defined in equation (3.8). To recap the reward is calculated by subtracting the operator cost for normal time and overtime from the generated revenue by finalized workload. After a configured amount of time into overtime an additional penalty is subtracted for not completed workloads.

3.2.2.4 Cloning and Resetting

The `System` is used throughout the whole training process, therefore it must be a reset upon reaching a terminal state. The reset triggers a new initialization. If a probabilistic order generation is enabled, new orders are generated accordingly. The leave priority as generated new at random.

3.2.3 Order Generation

The order generation method was already briefly mentioned in the section above. Now we give a more detailed explanation on how the order influx is generated. The basic behaviour we wanted to simulate are orders that come in early at the beginning of the shift. To achieve this behaviour a Poisson distribution was chosen to simulate the incoming orders. The Poisson distribution is generally used to determine the number of events within an interval. The distribution uses the mean value λ to determine the amount of events denoted as k. Figure 3.3 shows distributions for different values of λ and their associated event distribution.

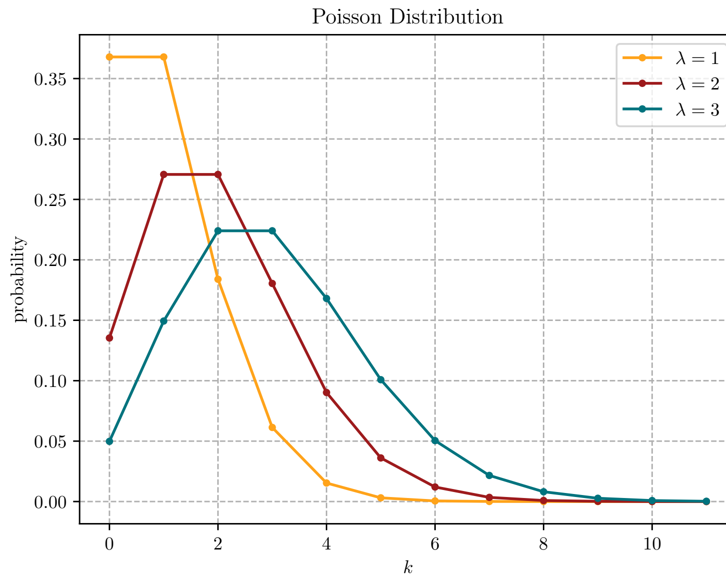


Figure 3.3: Poisson Distribution with different values of λ

In our implementation we wanted to use a fixed amount of orders and distribute them over several time intervals. We initialize the order generation with a constant $|X|$ which determines the total order influx over the whole shift and λ to configure the Poisson distribution. As seen in algorithm 3.3 we then simply loop $|X|$ times and assign orders of the amount one to this time interval according to the probability distribution. This is repeated until $|X|$ is reached. The order generation then outputs the array **o**, containing the orders for each time interval of the shift.

Algorithm 3.3 Generate Order

```
 1: procedure GET_ORDERS(λ, |X|)
 2:     x ← []
 3:     for all i ← 1, |X| do
 4:         x ← poisson(λ)
 5:         if x ⩽ |X| then
 6:             x[i] ← x
 7:             |X| ← |X| − x
 8:         else
 9:             x[i] ← |X|
10:             |X| ← 0
11:         end if
12:     end for
13:     return o
14: end procedure
```

4 Validation and Results

This chapter will give an overview, how we will validate our proposed reinforcement learning approach. The validation will be based on comparison against an upper and a lower bound. The upper bound will be used to measure the calculated results of the proposed reinforcement learning policy against will be provided by a linear program. The linear program will set the optimistic optimum. However, we will not be able to model certain points of the system like for example the startup behaviour with this approach. For the lower bound a naive algorithm to control the system is proposed. We want to show that it is worthwhile to utilize the reinforcement learning policy, although it is more effort than a simple heuristic. After describing the upper and lower bound in detail, we will describe the results for different scenarios that are used to validate the reinforcement learning policy. As these scenarios are based on real-life scenarios encountered during warehouse operations, they are a valid test to estimate the feasibility of our proposed reinforcement learning approach.

4.1 Upper Bound: Linear Program

We briefly introduce the concept of linear programming. The general problem for an instance of linear programming is defined by a matrix $A \in \mathbb{R}^{m \times n}$ and the column vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$. The goal is to find a column vector $x \in R^n$ that maximizes $c^\top x$ while $Ax \leqslant b$. Note that $c^\top x$ denotes the scalar product of the vectors. For vectors the notation $x \leqslant y$ means that the inequality holds for each component within x and y. A linear program can be simply written as $\max\{cx : Ax \leqslant b\}$. A vector x with $Ax \leqslant b$ is a feasible solution for the linear program. An optimal solution is considered a feasible solution that obtains the maximum. However, a linear program does not have a feasible solution if it is infeasible. A linear program P is infeasible if i.e. $P := \{x \in \mathbb{R}^n : Ax \leqslant b\} = \emptyset$. The most common algorithm to solve a linear program is called simplex-algorithm. However, given the scope of the current thesis, we will not discuss the detail of how the simplex algorithm or others work [17].

4.1.1 Definition of the Linear Program for Validation

We designed our linear program to minimize the total labour costs by using objective function:

$$\sum_{i=0}^{|O|}\sum_{j=0}^{|W|}(x_{i,j} + x'_{i,j}c_{ot}) \tag{4.1}$$

where $x_{i,j}$ denotes how long an operator i works on process step j during normal time. As a result an operator who works the whole normal shift creates at total cost of 1. Equivalently $x'_{i,j}$ defines the amount of work done in overtime. The parameter $|O|$ defines the total amount of operators available and $|W|$ the total number of process steps. The resulting costs are increased by a constant parameter c_{ot}. We continue by defining the problem constraints. First we need to limit the total working time of operator i over all process steps to 1:

$$\sum_{j=0}^{|W|} x_{i,j} \leqslant 1 \tag{4.2}$$

In the same way we limit the presence for overtime to 0.5, as overtime lasts only half as long:

$$\sum_{j=0}^{|W|} x'_{i,j} \leqslant 0.5 \tag{4.3}$$

We continue by defining

$$\mathbf{o} = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_{|W|} \end{bmatrix} \tag{4.4}$$

as order influx were o_j represents the amount of orders for process step j. Note that due to the fact that we want to emulate a linear system each order amount must be the same: $o_1 = o_2 = ... = o_{|W|}$. Now we introduce one boundary for each process step which ensures that at each process step all orders are completed:

$$\begin{gathered} \sum_{i=0}^{|O|} x_{i,1}l_{i,1} + x'_{i,1}l_{i,1} \geqslant o_1 \\ \sum_{i=0}^{|O|} x_{i,2}l_{i,2} + x'_{i,2}l_{i,2} \geqslant o_2 \\ \vdots \\ \sum_{i=0}^{|O|} x_{i,|W|}l_{i,|W|} + x'_{i,|W|}l_{i,|W|} \geqslant o_{|W|} \end{gathered} \tag{4.5}$$

In this equation $l_{i,j}$ denotes the total performance an operator i at process step j can provide for the whole shift. This definition is different from previous one, where this

performance was always related to a certain time interval within a shift. We conclude our linear program by adding the non-negative conditions:

$$x_{i,j}, x'_{i,j} \geqslant 0 \qquad \forall_{i \in \{0,\ldots,|O|\}}, \forall_{j \in \{0,\ldots,|W|\}} \tag{4.6}$$

The implementation of this linear program was done using AMPL. The chosen solver was MINOS [18].
As we now see, using this approach we cannot depict the system in the exact same way as with the simulation presented in section 3.2. The linear program does not feature probabilistically distributed orders, as all orders are present within the constraints in equation (4.5) at the beginning. Further the linear program is not faced with the start up behaviour as the reinforcement learning policy. For the linear program, work at each process step can start immediately at the beginning. Whereas due to the linearity of the process steps an order can only be finished after $|W|$ time intervals, when the first order can possibly reach the last process step. Therefore its not expected from the reinforcement learning policy to achieve a result as good as the linear program. However, to evaluate the efficiency of the reinforcement learning policy we are interested how close its results come to the optimistic optimum set by the linear program.

4.2 Lower Bound: Naive Algorithm

To get a justification for the effort of training the reinforcement learning policy we design a simple naive algorithm to control the system. If the trained policy performs worse than the lower bound set by this naive algorithm, we will be able to tell that it would not be worth the effort to solve this assignment problem with the proposed reinforcement learning policy.
For the naive algorithm we start with the initial assignment as already describe in section 3.2.2.1. For shifting operators we shift an operator from the process step $w_{smallest}$ with the smallest buffer to the process step $w_{highest}$ with the largest one. The operator which has the highest working capacity on $w_{highest}$ is chosen to be taken away from $w_{smallest}$. If at a process step the buffer runs empty and no more orders are expected to arrive we first try to shift these operators to other process steps still having capacity for more operators. Once this is not possible anymore, we send all operators assigned to this process step home.

$w \backslash m$	m_{min}	m_{max}
w_1	0	6
w_2	0	6
w_3	0	6
w_4	0	6
w_5	0	6

Table 4.1: Scenario 1: Process step capacities

4.3 Validation Scenarios

4.3.1 General Remarks

We will always use a shift length of 480 minutes and 240 minutes of overtime. When using intervals of 15 minutes, this results in 32 intervals in normal time and 16 intervals in overtime, totalling 48 intervals.
During training we save all MLPs during the evaluation phase. Once the training is completed, we will use the model yielding the highest mean evaluation reward for further analysis. However, we want to emphasise, that the validation will always be performed on newly generated test data. Further, we might stop training earlier if the evaluation reward does not improve anymore over several training episodes.

4.3.2 Validation Scenario 1: Target VS Main Network

For the first experiment we crated a scenario with a high order influx concentrated at the beginning of the shift. We loaded so many orders, that they were only able to be processed completely just before the end of the shift. Further we trained twice. One agent uses only the main network (see section 3.1.2.4). The other agent uses an additional target network. We used this approach to check whether using a target network has an influence on the convergence and stability during the training or not. Further, we wanted to get a proof of concept and check how our network performs against the proposed linear program and the naive algorithm.
Process Steps:
As seen in table 4.1, our system contained five process steps. On each process step six people can work in parallel at once. Further, it is also possible to remove all assigned operators from the process step.
Operators:
The system featured 20 operators. The possible work capacity at each process step follows a specific pattern. Each operator has one process step which is considered his main process step. There he is able to process 0.2 orders in 15 minutes. Further, he has another process step where he can provide 90% of the work capacity of his main process step. For

$o\backslash w$	w_1	w_2	w_3	w_4	w_5
o_1	0.2	0.18	0.1	0.1	0.1
o_2	0.2	0.1	0.18	0.1	0.1
o_3	0.2	0.1	0.1	0.18	0.1
o_4	0.2	0.1	0.1	0.1	0.18
o_5	0.18	0.2	0.1	0.1	0.1
o_6	0.1	0.2	0.18	0.1	0.1
o_7	0.1	0.2	0.1	0.18	0.1
o_8	0.1	0.2	0.1	0.1	0.18
o_9	0.18	0.1	0.2	0.1	0.1
o_{10}	0.1	0.18	0.2	0.1	0.1
o_{11}	0.1	0.1	0.2	0.18	0.1
o_{12}	0.1	0.1	0.2	0.1	0.18
o_{13}	0.18	0.1	0.1	0.2	0.1
o_{14}	0.1	0.18	0.1	0.2	0.1
o_{15}	0.1	0.1	0.18	0.2	0.1
o_{16}	0.1	0.1	0.1	0.2	0.18
o_{17}	0.18	0.1	0.1	0.1	0.2
o_{18}	0.1	0.18	0.1	0.1	0.2
o_{19}	0.1	0.1	0.18	0.1	0.2
o_{20}	0.1	0.1	0.1	0.18	0.2

Table 4.2: Scenario 1: Work capacity of the operators

the remaining process step the operator has a working capacity of 50% compared to the main process step. We designed our operators so that for every process step there are four operators with a working capacity of 100% and four with a working capacity of 90%. Further details on the operators can be found in table 4.2.

Cost Parameters:

The costs for an operator is 0.12 per 15 minute interval. During overtime the costs are doubled to 0.24. A revenue of 25 is earned for each completed order.

Order Influx:

The probabilistic order arrivals are modelled with a Poisson distribution as described in section 3.2.3 using $\lambda = 4$. A total order influx of 22 orders is configured. This order amount is chosen based on the fact that if we consider an equally distributed order assignment at each process step, 0.8 orders can be processed within one time interval. To process all orders 27 time intervals are needed. However, if we additionally consider that it takes five time intervals for an order to be completed from start to finish we can expect all orders to be finished after 32 time intervals. Based on this approximation all orders can be finished just in the 32 intervals of normal shift time without the need to transition to overtime.

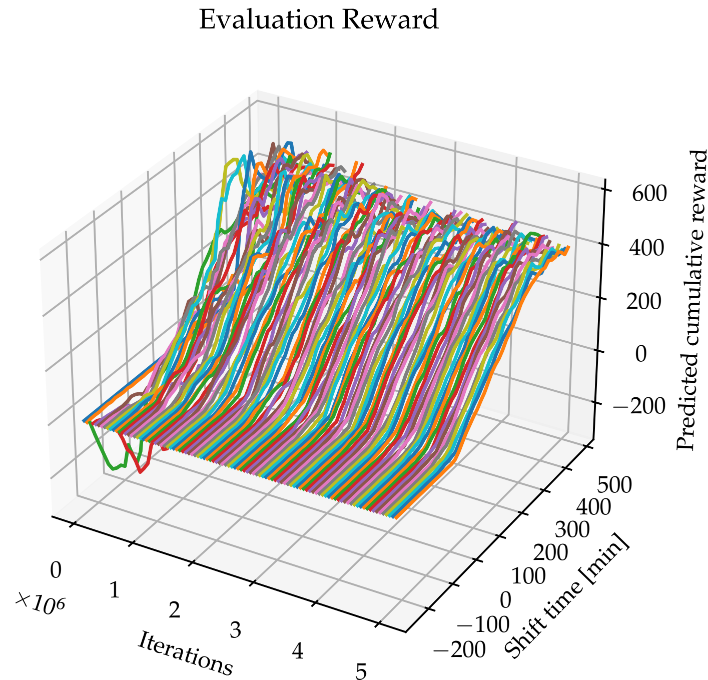


Figure 4.1: Scenario 1: Predicted cumulative reward for each time interval during training process using only the main network

4.3.3 Results Scenario 1: Target VS Main Network

Now we present the results of the experiment described in section 4.3.2. We trained the two reinforcement learning policies as described above.

4.3.3.1 Training

In the figures 4.1 and 4.2 we show the progress during training with respect to the predicted accumulated reward at every time interval t during a shift. When comparing the data, one can clearly see, that using a target network provides additional stabilization. In the target network the lines representing the expected rewards are smoother and more stable over several evaluations. The figures 4.3 and 4.4 show the predicted cumulative reward in detail for the best performing network respectively. In figure 4.4, which corresponds to the approach using the target network, the three different parts of a shift can be clearly differentiated. In the first part, from the the initial start until the first order is completely processed, the expected reward rises only slightly. This is caused by the personnel costs, which result in a negative reward. Once the first orders are completed, the line is decreasing constantly, as the orders are processed and the remaining possible reward lowers. Once all orders are processed, the operators are released and no further reward is expected. The line flattens out again. In the figure 4.3, corresponding to the

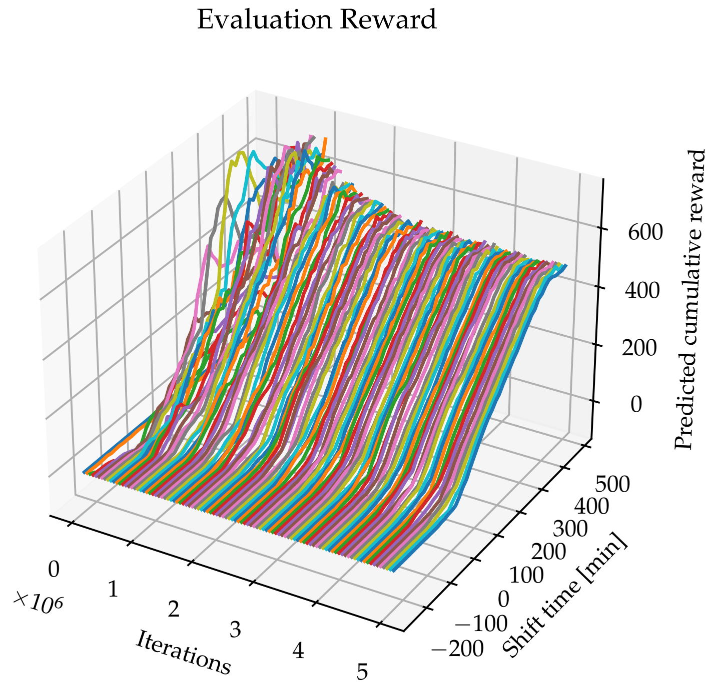


Figure 4.2: Scenario 1: Predicted cumulative reward for each time interval during training process using the target network

main network, we see that the first part was not predicted as good as using the additional target network. First we see, that the cumulative reward falls and then rises again before it changes into a steadily falling line. Further we can observe, that using the target network at the beginning the prediction of the reward was correctly at approximately 475 while the prediction using only the main network was underestimating the reward.

Figure 4.5 shows the reward received during evaluation for each reinforcement learning policy respectively. The training of the reinforcement learning policy using only the main network was stopped after approximately 5×10^6 iterations. This was done as the reward stalled for nearly 1×10^6 iterations as can be seen in figure 4.5. For the sake of comparison we trained the reinforcement learning policy using the additional target network for the same amount of iterations although it converged even faster. Using the main network, the highest achieved evaluation reward was 475.3 after 4.5×10^6 iterations. Adopting the target network the highest achieved evaluation reward was 476.5 after 4.85×10^6 iterations. For further analysis we will use the networks at the point of achieving the highest evaluation reward respectively.

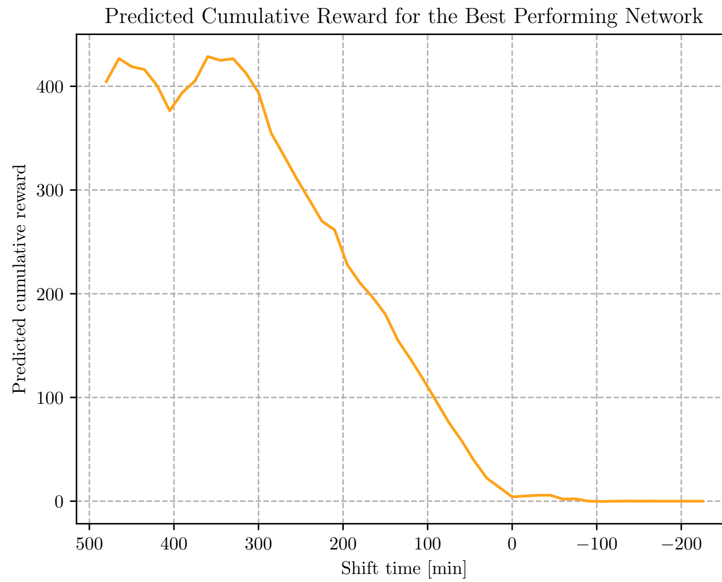


Figure 4.3: Scenario 1: Predicted cumulative reward of the best performing network using only the main network

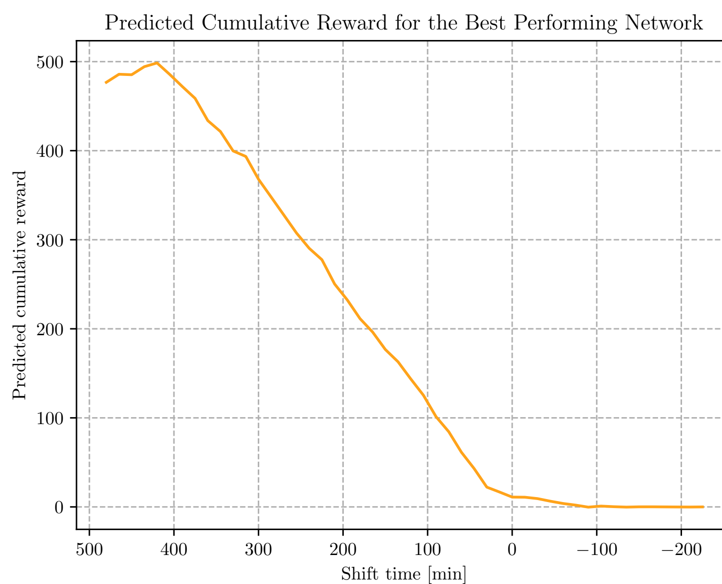


Figure 4.4: Scenario 1: Predicted cumulative reward of the best performing network using an additional target network

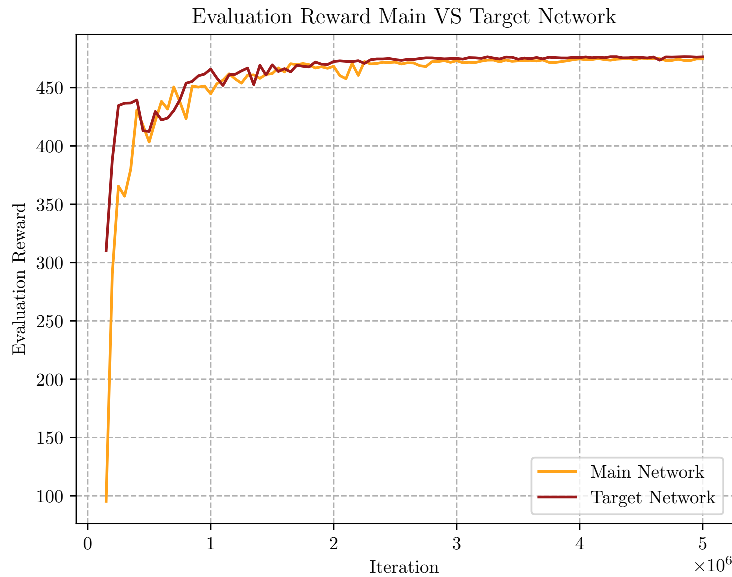


Figure 4.5: Scenario 1: Comparison of mean evaluation rewards during training

4.3.3.2 Result

The final evaluation is performed after the training on newly generated orders and leave queues. We evaluate each reinforcement learning policy for 100 shifts and average the result. Further we utilize the linear program from section 4.1 and the naive algorithm in section 4.2 to compare our results.
The reinforcement learning policy using only the main network generated an average reward of 475.4 while the policy using an additional target network achieved a reward of 476.5. Therefore, the difference between those two approaches was 1.1 or 0.2%. If we compare this difference to the operator cost it is equivalent to the saving the cost of 9.2 operators within one time interval or the cost of one operator working for 137 Minutes. Therefore we will consider using a target network for further scenarios.
If we compare the better of both results to the lower bound, we see that the reinforcement learning approach achieved a total of 18.9 or 4.1% more reward than the naive algorithm. In operator context this refers to the saved cost of 4.92 operators working a normal shift without overtime. Note that the naive algorithm was also evaluated for 100 runs using the mean value as final result. Finally in contrast to the upper bound given by the linear program, reinforcement learning gained a 1.35% smaller reward by 6.5 in total. However this was expected due to the limitations of the linear program mentioned in section 4.1.1.

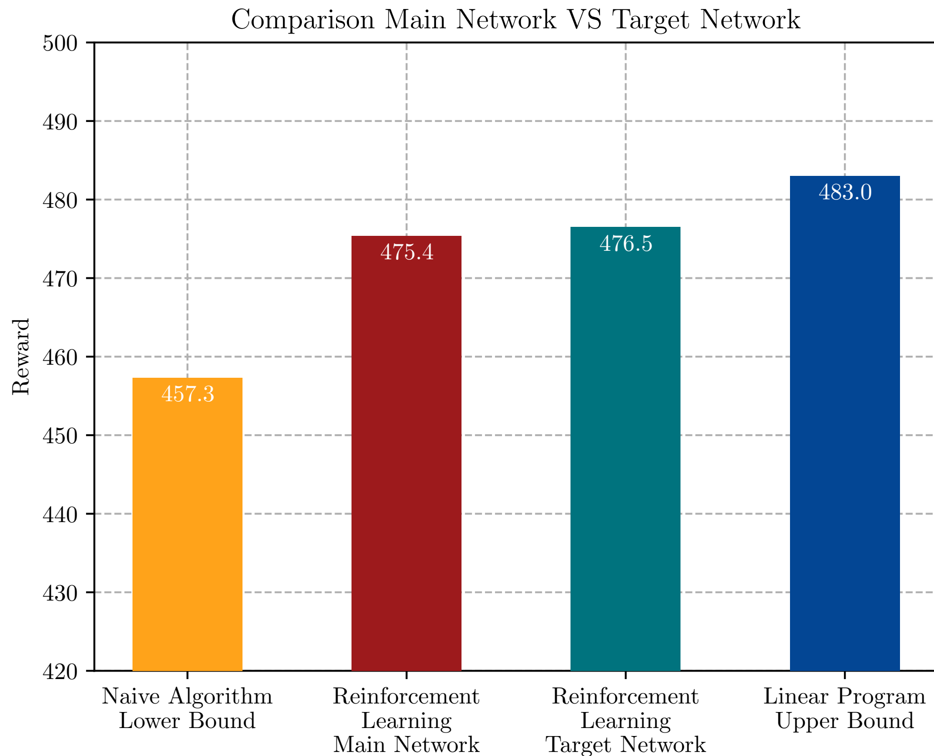


Figure 4.6: Scenario 1: Achieved rewards comparing main network and target network

4.3.4 Validation Scenario 2: Existing Network with different Operators

The first validation scenario featured a fixed set of operators and competences as can be seen in table 4.2. However, it is much more common in day to day warehouse operation scenarios, that the set of available operators varies from time to time. We therefore tested the already trained network from scenario 1 using different sets of operators. For the first set we reduced the overall working capacity to 90% compared to the capacity used in scenario 1. This results in an operator set as depicted in table 4.3.
At the same time, we are also interested in investigating how the network reacts if the working capacity of the operators is increased. Therefore, we also introduce an operator set with 150% working capacity compared to scenario 1. The details can be seen in table 4.4.

4.3.5 Results Scenario 2: Existing Network with different Operators

The reinforcement learning policies as well as the naive algorithm were evaluated for 100 shifts. For the first test, using operators with 90% working capacity compared to scenario 1, the network seemed to adapt well and a reward of 465.3 was achieved sd er can see in figure 4.7. In comparison to the naive algorithm, which produced a reward of 431.0,

$o \backslash w$	w_1	w_2	w_3	w_4	w_5
o_1	0.18	0.162	0.09	0.09	0.09
o_2	0.18	0.09	0.162	0.09	0.09
o_3	0.18	0.09	0.09	0.162	0.09
o_4	0.18	0.09	0.09	0.09	0.162
o_5	0.162	0.18	0.09	0.09	0.09
o_6	0.09	0.18	0.162	0.09	0.09
o_7	0.09	0.18	0.09	0.162	0.09
o_8	0.09	0.18	0.09	0.09	0.162
o_9	0.162	0.09	0.18	0.09	0.09
o_{10}	0.09	0.162	0.18	0.09	0.09
o_{11}	0.09	0.09	0.18	0.162	0.09
o_{12}	0.09	0.09	0.18	0.09	0.162
o_{13}	0.162	0.09	0.09	0.18	0.09
o_{14}	0.09	0.162	0.09	0.18	0.09
o_{15}	0.09	0.09	0.162	0.18	0.09
o_{16}	0.09	0.09	0.09	0.18	0.162
o_{17}	0.162	0.09	0.09	0.09	0.18
o_{18}	0.09	0.162	0.09	0.09	0.18
o_{19}	0.09	0.09	0.162	0.09	0.18
o_{20}	0.09	0.09	0.09	0.162	0.18

Table 4.3: Scenario 2a: Work capacity of operators

$o \backslash w$	w_1	w_2	w_3	w_4	w_5
o_1	0.3	0.37	0.15	0.15	0.15
o_2	0.3	0.15	0.37	0.15	0.15
o_3	0.3	0.15	0.15	0.37	0.15
o_4	0.3	0.15	0.15	0.15	0.37
o_5	0.37	0.3	0.15	0.15	0.15
o_6	0.15	0.3	0.37	0.15	0.15
o_7	0.15	0.3	0.15	0.37	0.15
o_8	0.15	0.3	0.15	0.15	0.37
o_9	0.37	0.15	0.3	0.15	0.15
o_{10}	0.15	0.37	0.3	0.15	0.15
o_{11}	0.15	0.15	0.3	0.37	0.15
o_{12}	0.15	0.15	0.3	0.15	0.37
o_{13}	0.37	0.15	0.15	0.3	0.15
o_{14}	0.15	0.37	0.15	0.3	0.15
o_{15}	0.15	0.15	0.37	0.3	0.15
o_{16}	0.15	0.15	0.15	0.3	0.37
o_{17}	0.37	0.15	0.15	0.15	0.3
o_{18}	0.15	0.37	0.15	0.15	0.3
o_{19}	0.15	0.15	0.37	0.15	0.3
o_{20}	0.15	0.15	0.15	0.37	0.3

Table 4.4: Scenario 2b: Work capacity of operators

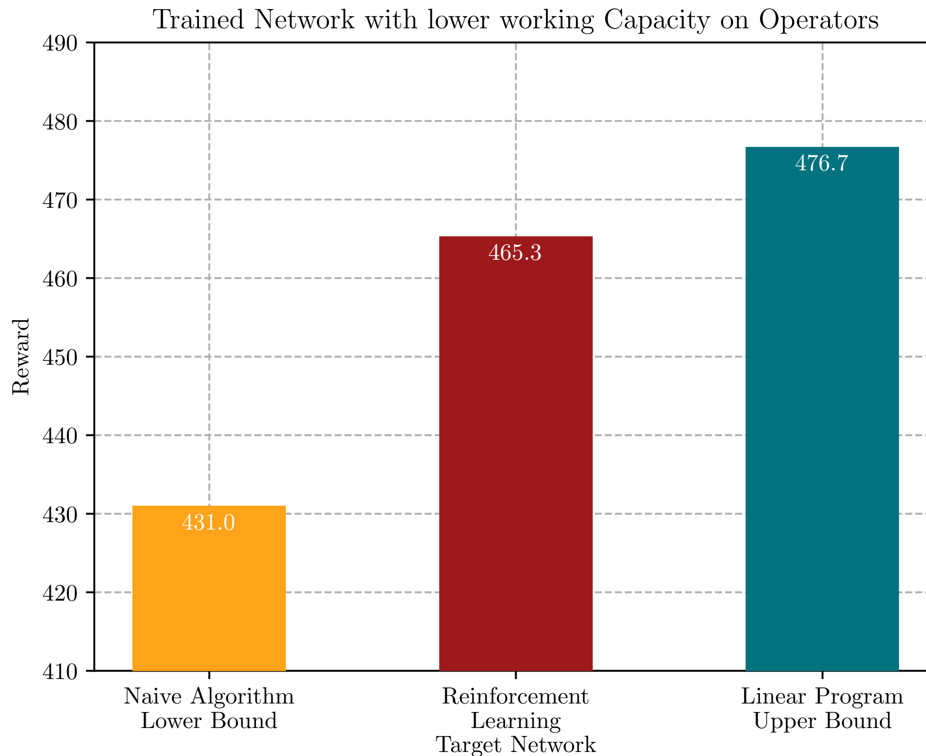


Figure 4.7: Scenario 2a: Decreased operator working capacity to 90% on trained network

the reinforcement learning policy achieved a 7.4% higher reward. This is an even bigger difference than in the first scenario on which the network was initially trained on. In comparison to the linear program the reinforcement learning policy achieved on average 11.4 or 2.4% less reward. Hence overall the reinforcement learning policy adapted quite well wo the new operator set.
However, the policy was not able to adapt to the operator set with higher working capacities. Looking at figure 4.8 we see that the average reward of 476.2 achieved on this set was the same as in the first scenario. To better understand the reasoning behind these differences, we looked at the actions taken by the reinforcement learning policy in detail. The results show that the reinforcement learning policy did not send operators home earlier although all orders were already processed. In comparison, the naive algorithm achieved an average reward of 491.8 in scenario 2 with 150% working capacity, outperforming the reinforcement learning policy by 3.2%. Looking at the upper bound, the linear program earned 29.8 more reward in total, surpassing the pre trained network by 6.1%. In this case, the reinforcement learning policy was not able to adapt to the new conditions at all. This happened as the neural network learned that these operators take longer to finish all orders.

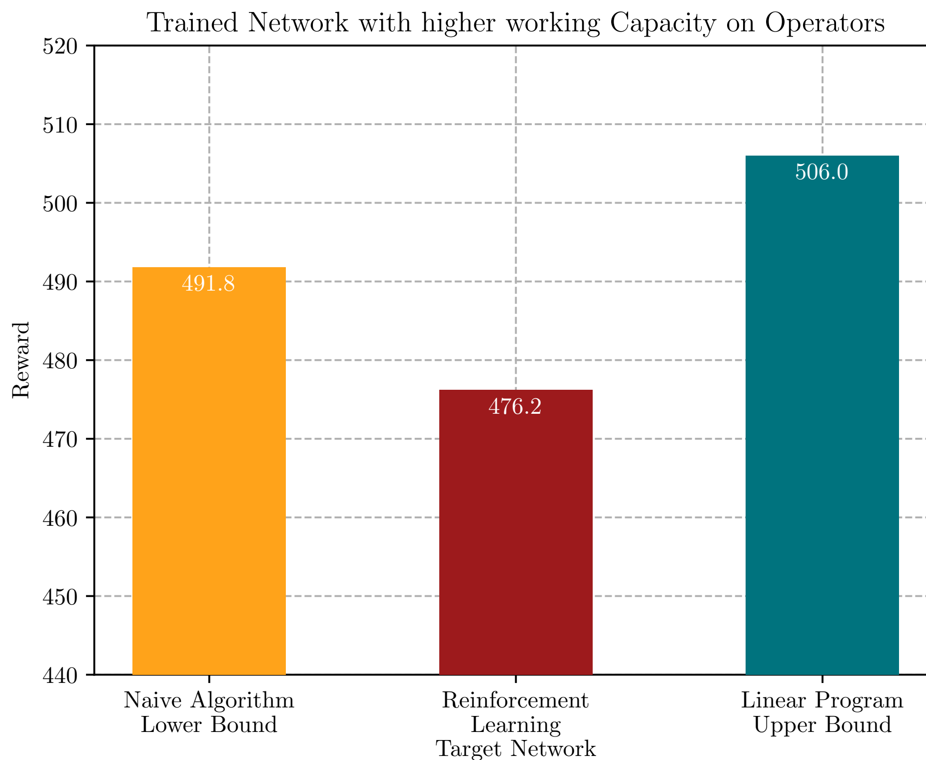


Figure 4.8: Scenario 2b: Increased operator working capacity to 150% on trained network

4.3.6 Validation Scenario 3: Retraining with Higher Workload

As we saw in the previous scenario the reinforcement learning policy did not adapt well to the operator set presented in table 4.4. We therefore investigated if we could achieve a better result when retraining the network. In this scenario we only used the target network as it produces more promising results and converged faster in scenario 1. All other parameters were kept the same as in the first two scenarios.

4.3.7 Results Scenario 3: Retraining with Higher Workload

4.3.7.1 Training

The predicted accumulated reward over the shift time figure 4.9 shows as similar result as in scenario 1. The estimated reward at the beginning of the shift settled after approximately 2×10^6 iterations and did not fluctuate anymore. Looking at figure 4.9 we can see, that the orders are now finished faster and therefore the operators can be sent home earlier compared to scenario 1.
The same behaviour can be observed in figure 4.10. For the first 2×10^6 iterations, the evaluation reward increased. For the remaining iterations the reward did not improve anymore, so the training was stopped. For further evaluation we again selected the best performing network during training. This was the network at 2.45×10^6 iterations.

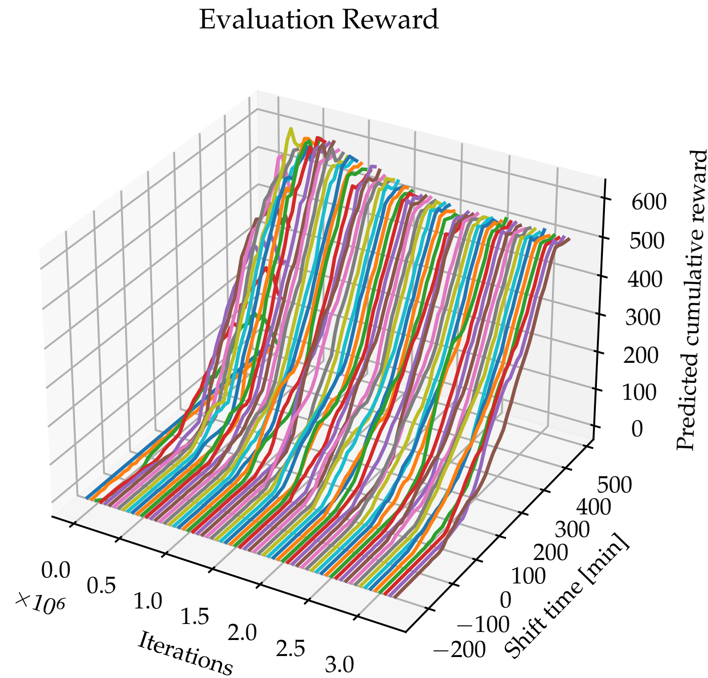


Figure 4.9: Scenario 3: Evaluation reward during training

4.3.7.2 Results

When using the best performing training network from scenario 3 and using 100 newly generated order distributions, we were able to achieve an average reward of 498.3. Compared to the result in scenario 2b, this is an improvement of 22.1 or 4.5% of the average reward. When comparing this result to the naive algorithm, the best performing reinforcement learning algorithm achieved, on average, a 6.5 or 1.3% higher reward. This is in stark contrast to the greater difference (18.9 points) between the naive and the best performing reinforcement learning algorithm in Scenario 1. This difference is most likely explained by the premise set for scenario 1. There we set out to investigate the performance when operating at the threshold between normal time and overtime. In Scenario 1 the naive algorithm performed poorly and dragged the system into overtime, which in consequence, led to higher operator cost, and therefore a lower reward.
Lastly we compare the new policy learned for scenario 3 against the upper bound. This policy was able to produce 7.7 or 1.5% less reward on average.
In the context of operator cost, this approach saves the amount of 5.75 operators working a whole shift, compared to the approach of scenario 2. Compared to the naive algorithm the equivalent of 1.7 operators working a whole shift is saved.

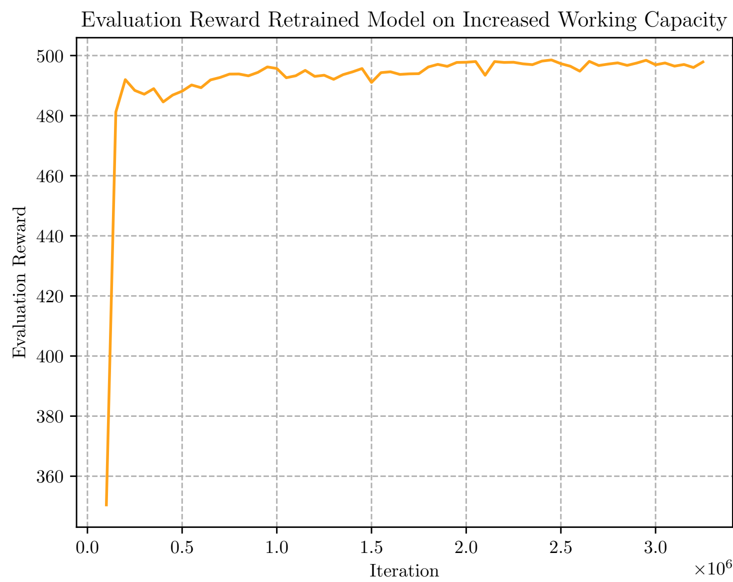


Figure 4.10: Scenario 3: Evaluation reward during training

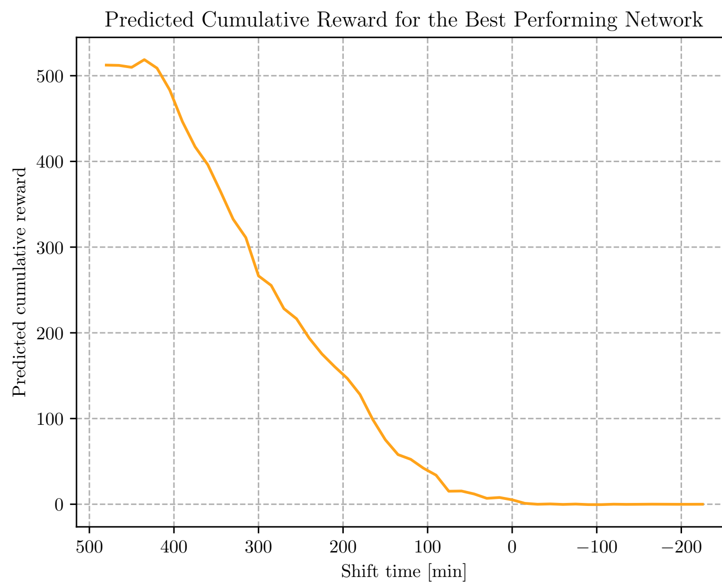


Figure 4.11: Scenario 3: Predicted cumulative reward of the best performing retrained network on 150% working capacity

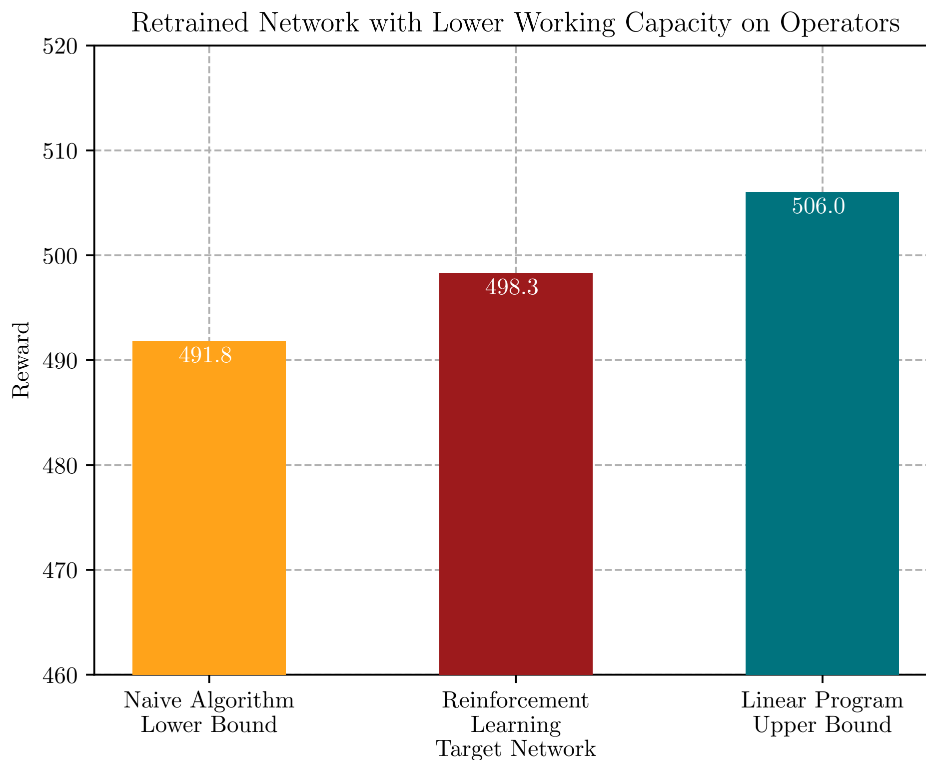


Figure 4.12: Scenario 3: Increased operator working capacity to 150% on A retrained network

4.3.8 Validation Scenario 4: Retraining with Reduced Workload

Scenario 2 showed that a trained network was able to adapt to a set of operators with a reduced workload. We wanted to test the limits of the reinforcement learning policy and see if the policy would still be able to complete all orders when working capacity was even further lowered to 75% of the values initially set in table 4.2. Given the reduced working capacity, we expect to see both algorithms, the previous algorithm and the newly trained one, to use operators in overtime in this scenario. As we did not want to over penalize the expected overtime, we adapted the artificial penalty (see equation (3.7)) from starting after 90 minutes to starting after 165 minutes. This number was chosen as a compromise between over penalization and and still favouring completing all orders.

4.3.9 Results Scenario 4: Retraining with Reduced Workload

4.3.9.1 Training

As expected, the fulfilment of orders took noticeable longer in Scenario 4 (see Figure 4.13). The part of the shift in which orders are processed now extended into overtime. Nevertheless, the network converged in a similar way as we observed in the previous results. As we can see in Figure 4.15, the evaluation reward did not noticeably improve

$o \backslash w$	w_1	w_2	w_3	w_4	w_5
o_1	0.15	0.135	0.075	0.075	0.075
o_2	0.15	0.075	0.135	0.075	0.075
o_3	0.15	0.075	0.075	0.135	0.075
o_4	0.15	0.075	0.075	0.075	0.135
o_5	0.135	0.15	0.075	0.075	0.075
o_6	0.075	0.15	0.135	0.075	0.075
o_7	0.075	0.15	0.075	0.135	0.075
o_8	0.075	0.15	0.075	0.075	0.135
o_9	0.135	0.075	0.15	0.075	0.075
o_{10}	0.075	0.135	0.15	0.075	0.075
o_{11}	0.075	0.075	0.15	0.135	0.075
o_{12}	0.075	0.075	0.15	0.075	0.135
o_{13}	0.135	0.075	0.075	0.15	0.075
o_{14}	0.075	0.135	0.075	0.15	0.075
o_{15}	0.075	0.075	0.135	0.15	0.075
o_{16}	0.075	0.075	0.075	0.15	0.135
o_{17}	0.135	0.075	0.075	0.075	0.15
o_{18}	0.075	0.135	0.075	0.075	0.15
o_{19}	0.075	0.075	0.135	0.075	0.15
o_{20}	0.075	0.075	0.075	0.135	0.15

Table 4.5: Scenario 4: Working capacity of operators

after 2.5×10^6 iterations. Looking at the detailed view in figure 4.13 we can now clearly see, that the policy keeps operators for overtime as the orders are finished just before the end of the whole shift. As in previous Scenarios, this Scenario was also validated using the network with the highest evaluation reward.

4.3.9.2 Results

In order to compare the results of the network, we again created 100 new orders. Feeding these orders into the shifts controlled by the newly trained network resulted in an average reward of 438.0 as we can see in figure 4.16. The system controlled by the policy from scenario 2, using this network achieved an average reward of 434.7. So this policy only achieved a reward 3.3 points, or 0.75% lower than the retrained network. This shows, that the reused network was still able to adapt to the new operator set. The naive algorithm was not able to finish all orders until the end of the shift and therefore only produce an average reward of 256.0 which is a deviation of 52.4% from the result achieved using the retrained network. The comparison with the linear program showed that the reinforcement learnings policy using the newly trained network did achieve 5.3% less reward which is a total of 24.1. This is a bigger gap between the reinforcement learning policy and the linear program than we observed in the previous scenarios. This difference can partially be explained by the fact that both solutions were now (at least partially) working within

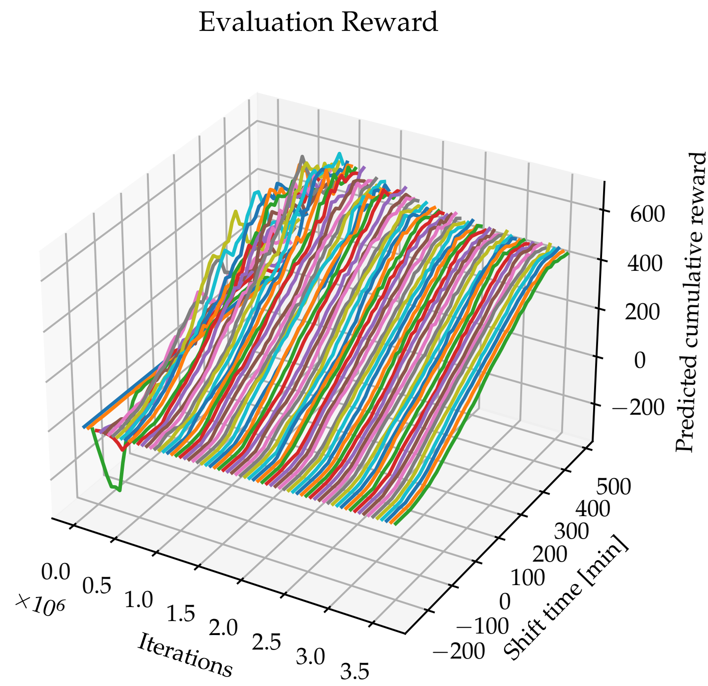


Figure 4.13: Scenario 4: Evaluation reward during training

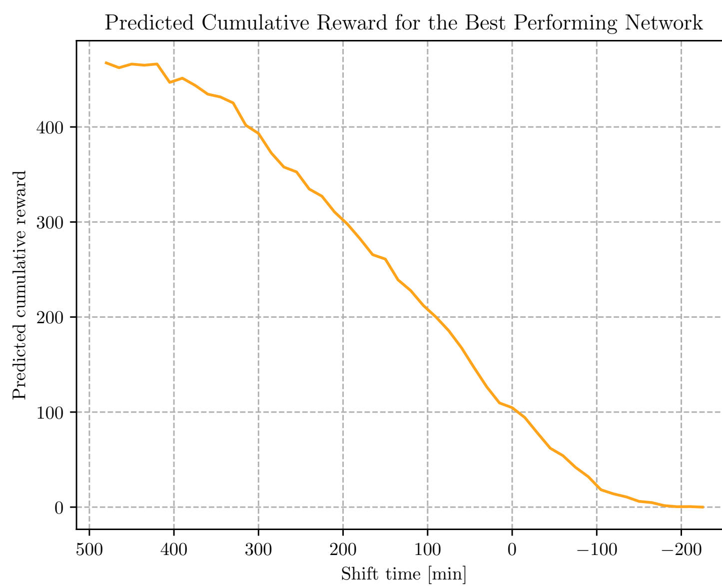


Figure 4.14: Scenario 4: Predicted cumulative reward of the best performing retrained network on 75% working capacity

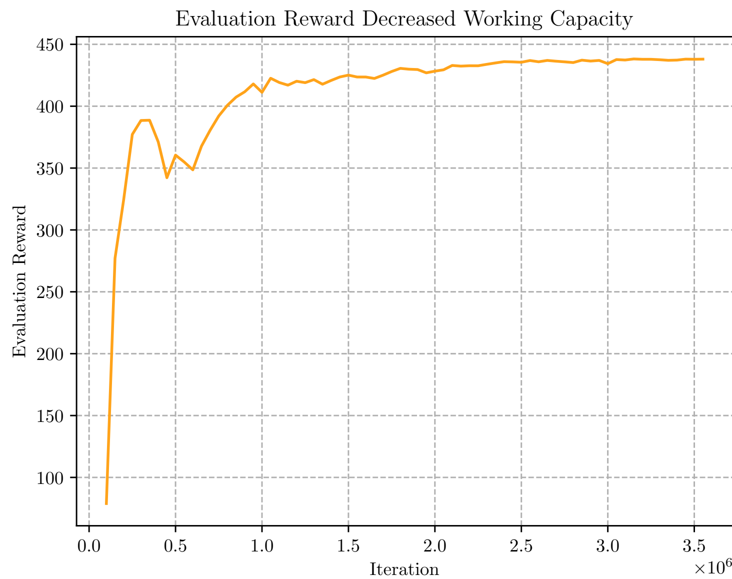


Figure 4.15: Scenario 4: Evaluation reward during training

overtime, in which the operator time costs twice as much as during normal time. This means, that our proposed reinforcement learning policy is still able to produce viable results while utilizing overtime. In the operator cost context the linear program saved the equivalent of 6.3 operators working a normal shift costs. Comparing the use of the already trained network to the newly trained, the network of scenario 1 only produced the equivalent of 1.2 operators working a normal shift more cost, which was not expected upfront.

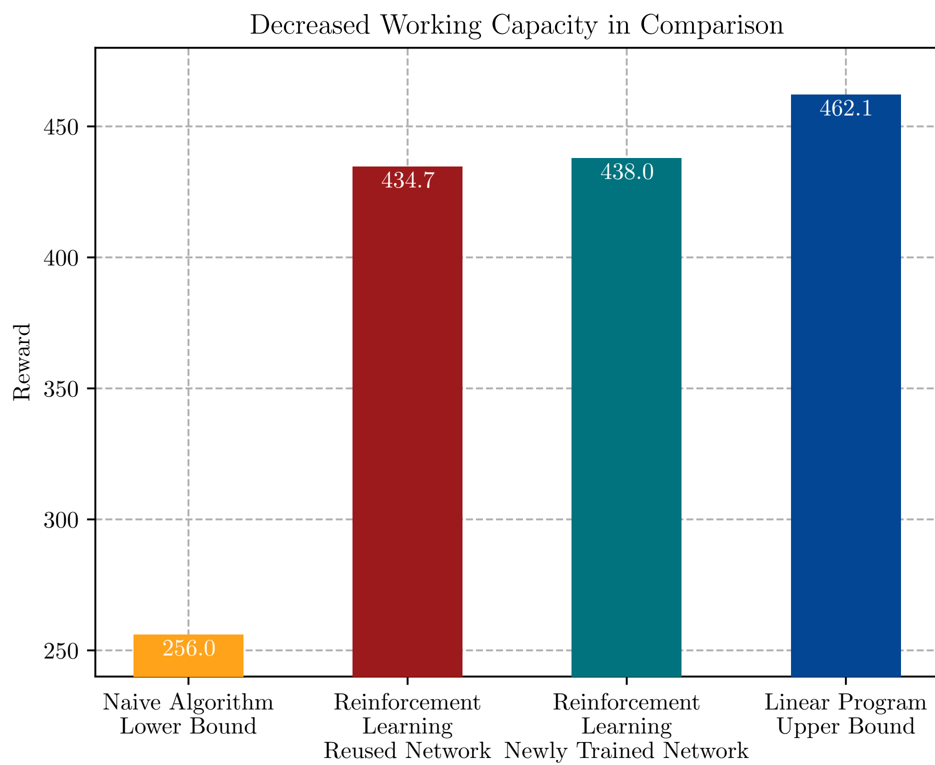


Figure 4.16: Scenario 4: Decreased operator working capacity to 75% on A retrained network

5 Summary and Outlook

5.1 Summary

The goal of this thesis was to investigate if reinforcement learning can be used to support warehouse operations and reduce the occurring personnel costs by managing work hours. To do so, we developed a reinforcement learning policy which used a neural network as function approximation. For the learning model an MDP was chosen. As training solely on real-life data or even on pre-generated date was not sufficient, a dynamic simulation was created in order to provide on-demand training data for the policy. The reinforcement learning policy was validated using four different scenarios. These scenarios were based on real-life scenarios which could be expected in modern warehouse operations. In all scenarios, the results of the reinforcement learning policy were compared to an upper and lower bound which was provided by a naive control algorithm and a linear program respectively. The first scenario featured a moderate order influx completable slightly within the normal working time of a shift. In this first scenario, the reinforcement learning policy performed better than the naive algorithm and achieved a 4.1% higher reward on average. The reinforcement learning policy performed similarly to the upper bound set by the LP, only achieving a 1.4% lower reward on average. In the second scenario, we investigated the adaptability of the trained networks to operators of different working capacities. The results showed that the networks were able to adapt very well to operators with a lower working capacity and were still able to achieve good results (7.4% better than the naive approach; 2.4% lower than the linear program) under these conditions. However, the policy was not able to adapt as well to operators with higher working capacity. The results stayed in line with the results of operators with lower capacity from the previous scenario, achieving a reward 3.2% lower than the naive approach and 6.1% lower than the linear program. The results stem from the fact that the network did not release the operators from the job, even when their work was finished and thereby continued to accumulate operator costs. In the third scenario we then retrained the network using the operators it had failed to adapt to in scenario 2 in order to proof, that a retrained network could achieve a competitive result.

When we then used the retrained network in a scenario otherwise similar to scenario 2, the reinforcement learning policies was able to achieve results (1.3% better than the naive approach; 1.5% lower than the linear program) comparable to the results achieved in

scenario 1. So we concluded that the results of this scenario were of even quality to the ones in scenario 1. In scenario 4 the operator working capacity was lowered even further than the reduction in scenario 2. in order to test the behaviour of the policy in situations where overtime is needed to fulfil all open orders. In addition to training a new network, we also tested the performance of the network from scenario 1 in order to see how well it could adapt to this change. The results showed that the network trained in scenario 1 was able to adapt very well (average reward of 434.7) and was only outperformed slightly by the newly trained network (average reward of 438.0, or 0.8% higher). The naive algorithm was not able to finish all orders until the end of the day (incl. maximum overtime) and was therefore easily beaten by both reinforcement learning policies. The reinforcement learnings policies were not able to come as close to the upper bound set by the LP as in the other scenarios. This can be at least partially explained by the different cost structure during overtime, giving negative rewards doubled emphasis.
Overall we were able to show that reinforcement learning using neural networks as function approximation is a viable approach to reduce personnel costs and support warehouse executives in their daily decision making.

5.2 Outlook

Although our approach showed promising first results, further research is needed in order to investigate if it is able to function as well in more advanced and more complex real world scenarios. For example, although in real world warehouse operations, managers have to make decisions about different sets of personnel on a daily basis, our current approach showed some limits when it came to adapting to new operators with different skill sets. Investigating potential solutions is beyond the scope of this thesis. Future research is needed to investigate alternative forms of structuring the learning of the networks when it comes to utilizing different operators or sending them home.
The current approach used in this master's thesis is characterised by a high implementation and computation effort. Further research should therefore investigate if the use of different heuristics could save time and effort without affecting the quality of the results.

Bibliography

[1] Mikko Varila, Marko Seppänen, and Petri Suomala. "Detailed Cost Modelling: A Case Study in Warehouse Logistics". In: *International Journal of Physical Distribution & Logistics Management* 37 (2007), pp. 184–200. DOI: 10.1108/09600030710742416

[2] James Aaron Cooke. "INVENTORY VELOCITY ACCELERATES". In: *Logistics Management* (2003)

[3] DSLV Bundesverband Spedition und Logistik. *DSLV-Kostenindex Sammelgut: Stückgutkosten weiter im Steigflug.* 2022. https://www.dslv.org/dslv/web.nsf/id/li%7B%5C_%7Dfdihcd7eed.html (visited on 05/12/2022)

[4] Teun van Gils et al. "The use of time series forecasting in zone order picking systems to predict order pickers' workload". In: *International Journal of Production Research* 55 (2017), pp. 6380–6393

[5] Frada Burstein, ed. *Handbook on decision support systems.* International handbooks on information systems. Berlin [u.a.]: Springer, 2008, pp. 83, 121, 367. ISBN: 978-3-540-48712-8

[6] Tom M. Mitchell. *Machine learning.* 8. Boston, Mass. [et.al.]: WCB/McGraw-Hill, 2002. ISBN: 9780070428072;0070428077;

[7] Richard S Sutton and Andrew Barto. *Reinforcement learning: an introduction.* 2. Cambridge, Massachusetts, London: The MIT Press, 2018. ISBN: 9780262039246

[8] Charu C Aggarwal. *Neural Networks and Deep Learning.* Springer, 2018, p. 497. ISBN: 978-3-319-94462-3. DOI: 10.1007/978-3-319-94463-0

[9] Richard O Duda, Peter E Hart, and David G Stork. *Pattern Classification.* 2nd. Wiley-Interscience, 2001, pp. 305–318

[10] David Pool and Alan Mackworth. *SARSA with Linear Function Approximation.* 2010.
http://artint.info/html/ArtInt_272.html
(visited on 05/21/2022)

[11] Radoslaw M. Cichy and Daniel Kaiser. "Deep Neural Networks as Scientific Models". In: *Trends in Cognitive Sciences* 23.4 (2019), pp. 305–317. ISSN: 1364-6613. DOI: https://doi.org/10.1016/j.tics.2019.01.009.
https://www.sciencedirect.com/science/article/pii/S1364661319300348

[12] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning.* 2013. DOI: 10.48550/ARXIV.1312.5602.
https://arxiv.org/abs/1312.5602

[13] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533. DOI: 10.1038/nature14236

[14] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning.* London, UK, 2016. arXiv: 1511.06581 [cs.LG]

[15] Matteo Hessel et al. *Rainbow: Combining Improvements in Deep Reinforcement Learning.* 2017. arXiv: 1710.02298 [cs.AI]

[16] Szymon Sidor and John Schulman. *OpenAI Baselines: DQN.* 2017.
https://openai.com/blog/openai-baselines-dqn/
(visited on 05/12/2022)

[17] Bernhard H Korte et al. *Combinatorial optimization.* Vol. 4. Berlin Heidelberg: Springer, 2011, pp. 49–50. ISBN: 978-3-540-71843-7. DOI: 10.1007/978-3-540-71844-4

[18] B.A. Murtagh and M.A. Saunders. "Large-Scale Linearly Constrained Optimization". In: *Mathematical Programming* 14 (1978), pp. 41–72