Smart capacity planning: A machine learning approach

Master Thesis in Information Systems, Production and Logistics Management

by

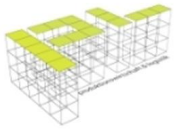
Sebastian Windmüller

submitted to the School of Managment,
Department of Information Systems, Production and Logistics Management

in partial fulfillment of the requirements
for the degree of Master of Science

supervisor: MMag. Stefan Haeussler, PhD,
Department of Information Systems, Production and Logistics Management

Innsbruck, 18 November 2020



Master Thesis

Smart capacity planning: A machine learning approach

Sebastian Windmüller (11703684)
sebastian.windmueller@student.uibk.ac.at

18 November 2020

Supervisor: MMag. Stefan Haeussler, PhD

Abstract

Manufacturing companies often use overtime or outsourcing of labor to handle unexpected increases in short-term demand. High demand levels can cause customer orders to ship late as not all orders can be processed in time. Not using overtime or outsourcing might lead to contractual penalties when shipments get delayed beyond their advertised shipping date. It is challenging to find a good balance between the costs of overtime and the costs of late shipments.

This thesis provides a model that allows planning the usage of overtime for a job shop production process with various Reinforcement Learning algorithms. In contrast to established methods from the literature, this model allows to adapt the decision-making dynamically to the system state. With Reinforcement Learning the model learns by itself and can therefore improve its performance over time and learn to adapt to new situations.

The model can have any Reinforcement Learning algorithm be applied to it and thus allows the comparison of various algorithms. This study compares Reinforcement Learning algorithms from literature and some naive heuristics on the model.

The results show that one Proximal Policy Optimization algorithm performed very well on the job shop model and achieved total production costs more than 10% lower than other algorithms and heuristics.

Key findings are that Reinforcement Learning can be a viable alternative to established heuristics for many decision problems, however more work is needed to evaluate Reinforcement Learning algorithms on other domains of operations research and manufacturing.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The production of physical goods requires a certain amount of planning and control, which is achieved with production planning and control (PPC) systems. Such systems are used to plan material requirements, manage demand, plan capacities and to schedule and sequence jobs. This leads to improvements for work in progress (WIP) inventory, shop floor throughput and flow times, stock holding costs and delivery date adherence as well as responsiveness to changes in demand. (Stevenson et al., 2005)
Out of the many aspects of production planning, this thesis takes a deeper look on production capacity planning. As Eppen et al. (1989) state, companies need to plan their production capacity before they know how much demand there will be. How far in advance the capacity must be planned depends on the company's specific industry. A manufacturing company that uses heavy and expensive machines with long delivery times might not be able to increase its capacity on the short term, whereas a service-oriented company that relies on manual labor might be able to recruit some staff on rather short notice.
There are many strategies on how to cope with production capacity constraints, but for long term planning, Eppen et al. (1989) name three major strategies: demand can be satisfied almost always, most of the time or not at all, with the last option giving the benefit of high capacity utilization.
On a macroeconomic level, production capacities are usually higher than the actual demand. From 1972 to 2019, the total industry production capacity of the USA was only utilized by on average 79.8%. This shows that most companies follow the strategy of trying to fulfill demand almost always, leading to unutilized capacity. Economic crises can lower the utilization even further, for example during the financial crisis in 2009 the US total industry capacity had a utilization of 66.7%, and during the Corona crisis in April 2020 there was a utilization of 64.2%. (Federal Reserve Bank, 2020)
Taking back the microeconomic perspective, we have established that companies need to have capacity available to fulfil demand, yet individual requirements of a company or industry as well as economic impacts can have great influence on the actual utilization level. Now let's assume that production capacity is fixed on the short term, that means a company can not obtain more capacity when there is a sudden rise in demand. As a practical example, we can consider the amount of production machines of a manufacturing company as its total capacity. Usually, the amount of machines can not be increased within a few days or weeks, sometimes not even within months.
This thesis focuses on short term production planning in such a situation, where the capacity that is usable is limited and can not be extended.
We now assume a fictional make-to-order company producing fictional goods in a job shop, which could be anything ranging from skincare products to airplanes. This company

manufactures for 16 hours a day, that is two shifts of workers (eight hours each) operating the machines or tools. As we know, the amount of machines is fixed, so if there is a sudden surge in demand, the company can either not meet the demand if it is too high, or it can try to find a more creative way of utilizing the existing resources, e.g. by temporarily running a third shift at night and paying the workers extra money. Depending on the demand, a full third shift might not be necessary and it could be sufficient to manufacture goods for just a few hours more each day.
In a deterministic world, where all demand and production flow times are known ahead of planning, this would be a mathematically solvable optimization problem (e.g. by linear (Eppen et al., 1989) or nonlinear programming (Ho and Fang, 2013)). However with machine failures, delays and other factors interrupting the manufacturing process, it would be beneficial to have a production planning system that continuously plans capacities based on the current state of the shop floor.
One of the main issues is the balance of the costs for using overtime and the costs for shipping goods late. High demand causes the production to be overwhelmed and goods get shipped late. Late shipments can cause contractual penalties, if their shipping date is later than planned (or guaranteed to the customer). It is therefore crucial to find a good balance between the usage of overtime and shipping goods late, and this balance can change over time.
This thesis tries to solve this decision problem by proposing a smart production planning software that controls the production capacity by dynamically adjusting the working hours in the production system in advance. The first component of the software is a simulation program that simulates the entire production process of the company, from receiving customer orders, manufacturing the goods up to shipping them to the customer. The production planning system inside the simulation has certain interfaces which allow an external program to control the amount of hours worked on the production shop floor. This external program is the second component, which accesses the production planning system through the aforementioned interfaces. The program consists of an agent that uses Machine Learning to find capacity control strategies on its own, and by learning from feedback given by the production system.
The thesis is structured as follows. Chapter 2 gives an overview of relevant literature about the topics of production planning and machine learning, providing the theoretical foundations for the model used later on. This model is described in Chapter 3, and the program functionalities and relevant algorithms are introduced. Chapter 4 analyses the performance of the proposed smart capacity planning system, comparing the status-quo with high demand and no overtime to multiple algorithms that aim to solve the decision problem by using overtime. The last chapter 5 concludes the thesis, summarizing the findings and gives an outlook on potential application of the proposed system and algorithms.

2 Theoretical background

This chapter is divided into two sections about workload control and machine learning, each investigating the current state of scientific literature as well as practical applications in the context of the capacity planning problem presented in the Introduction.

2.1 Production planning

As shown in the Introduction, production planning is important for a manufacturing organisation to plan material requirements, capacities and jobs. There are multiple production planning concepts that try to help companies solve the production planning problem, for example workload control (WLC), Manufacturing Resource Planning (MRP II), Constant Work In Process (CONWIP), Kanban and others (Zäpfel and Missbauer, 1993; Stevenson et al., 2005). This thesis uses a simulation model based on the workload control approach, as it has shown good performance in simulation studies (Stevenson et al., 2005; Bertrand and Van Ooijen, 2002). It was also employed by a similar simulation model which was used as a benchmark to verify the base simulation model of this thesis. In the upcoming sections, the following terms are used interchangeably: *order* and *job*, *machine* and *work center*, *inventory* and *queue*.

2.1.1 Workload control

Let's assume our company mentioned in the Introduction receives orders from its customers, assigns these orders a certain delivery date, manufactures the corresponding goods with machines on the shop floor and ships the goods to the customers. Workload control refers to a concept with the goal of having all these orders completed by their due dates while utilizing work center capacity as much as possible by controlling queues at an acceptable level (Fredendall et al., 2010; Kingsman and Hendry, 2002).
With workload control, the production environment can be seen as a queuing system. Before every machine or work center there is a queue of jobs waiting to be processed. Workload control is used to control the length of such queues by deciding to release or not to release jobs. A job remains in production or rather on the shop floor until all work on it has been completed. The amount of jobs on the shop floor influences the job release decision to avoid high inventory costs due to long queues. The time between release and completion of a job is referred to as the job's flow time. The planned time between entry in the order pool and completion of a job is called the job's lead time. If a job's actual flow time plus the waiting time in the order pool exceeds the planned lead time, the job

is considered late [1]. (Land et al., 1996)
Workload control is suitable for make-to-order companies which manufacture different goods after a customer places an order. The opposite of a make-to-order company (MTO) is a make-to-stock company (MTS) which makes standard products using a standardized process and keeps finished goods in an inventory as a buffer against variations in customer demand. Make-to-order on the other hand tries to use the existing capacity as much as possible and loses revenue from unused capacity. (Chen et al., 2009)
Price and delivery lead time are the most important factors for a make-to-order company when it has to compete with other similar companies, thus it is crucial to adhere to promised delivery dates as much as possible. Make-to-order companies are customer driven and have a high variability and uncertainty when it comes to planning their production process (Thürer et al., 2014).
Up to 90% of the time that an order is in production it has to wait inside some inventory until the next production step has free capacity. (Kingsman and Hendry, 2002)
As described in the Introduction, it is of course possible to follow the strategy of always having sufficient production capacity available, but this comes at the cost of most machines and workers sitting idle, which is financially challenging for a company that makes products to order and has to compete for every single order. That is why we assume a fixed production capacity which can not always fulfill all demand and operates on a rather high utilization level (the aforementioned 90% of idle time for orders infers that the machines have barely any idle time).
According to Kingsman and Hendry (2002), there are three levels where control over inventories or queues can be attempted: on the day-to-day shop floor control level (priority dispatching), on the short term production planning level (job release) or on the medium term production planning level (job entry).
One of the main aspects of workload control is a job release mechanism, which allows having a pool of unreleased jobs (order pool) waiting to be released to the shop floor. These jobs get released if a release wouldn't likely result in too many jobs having to queue on the shop floor (as queuing jobs on the shop floor increases production costs). This mechanism however might cause some jobs to be released too late and miss their delivery dates, if there are generally many orders on the shop floor and inside the order pool. Missing delivery dates imposes the danger of having to pay lateness fees to the customers. Therefore, an order release mechanism should also have a job entry stage to control the order pool itself. (Kingsman and Hendry, 2002)
The workload control approach provides some useful mechanisms at the job entry and job release stages to control the utilization of inventories and machines as well as to influence the lead times and delivery dates. Ultimately, WLC leads to short and stable queues. (Land et al., 1996)
The positive benefits of workload control mechanisms have been shown in various studies (Melnyk et al., 1991; Kingsman and Hendry, 2002; Chan et al., 2001; Ragatz and Mabert, 1988; Thürer et al., 2014, 2012).

[1] See Land et al. (1996) *figure 1: lead time components* for a visualisation of the difference between flow time and lead time

As discussed before, the WLC concept features control mechanisms on the job entry and job release level. The job entry mechanism allows a shop floor manager to prevent the queues from becoming too long due to a sudden increase in customer demand by just not accepting more orders into the order pool. While using this mechanism to keep the production environment lean makes sense from a shop floor manager's perspective, we also know that make-to-order companies often have to compete for customer orders and might not be able to afford denying too many orders, even if the order pool is already crowded.
Chen et al. (2009) suggest that managers of make-to-order companies ask themselves four questions before accepting an order. The questions are whether the company has the technical capability to handle the order, whether it has the production capacity to accommodate the order, whether it can complete the order in time and how high the profit generated from the order will be. The last three questions have direct impact on the short term capacity problem.
Long term changes in customer demand can be handled by increasing the production capacity through the acquisition of more machines , but on the short term it might be beneficial to look at other methods to increase capacity. The following section will take a look on the capacity planning problem briefly mentioned in the Introduction.

2.1.2 An introduction to the capacity planning problem

Generally, production is the most efficient when it operates at full capacity, while providing some idle capacity to serve as a buffer for problems arising during the manufacturing process like machine failures, necessary rework and more (Hammesfahr et al., 1993). Having not enough capacity results in lost sales, having excess capacity leads to higher per-unit costs, as machines and workers cost money despite being idle. One solution to reducing the chance of lost sales is to maintain a finished goods inventory, which would decrease in times of higher demand, also allowing the company to keep its production capacity at a fitting level (Hammesfahr et al., 1993). Depending on the company, it might be viable to keep a filled finished goods inventory, but this doesn't work for make-to-order companies. Expanding the production capacity by adding more work centers can also be very expensive, and running the production with lots of excess capacity can have dramatic consequences in a very competitive market.[2]
Chen et al. (2009) identified three tiers of capacity planning in terms of their planning horizon. Long-term capacity planning focuses, depending on the demand forecast, on yearly resource requirements for manufacturing. This can include plant locations and capacities, planning with suppliers and establishing new technologies or processes. Medium-term capacity planning focuses on monthly or quarterly resource requirements like the amounts of workforce, raw materials and inventories. Examples are the planning of hiring or laying off staff, ordering overtime or part-time as well as decisions regarding outsourcing and supplying materials. Two common medium-term planning strategies are *matching*

[2] See Hammesfahr et al. (1993) for an example of excess capacity in steel manufacturing leading to the demise of the US steel industry

demand and *level capacity*. Matching demand refers to keeping the production capacity exactly to the levels indicated by the demand forecast, with variations in forecasted demand being met with hiring or laying off workers. The level capacity strategy keeps the production capacity constant and variations of customer demand are handled with maintaining inventory, delivering late or the use of overtime, part time, temporary labor and outsourcing. However, according to the authors, most make-to-order companies use a hybrid approach and maintain a minimum amount of production capacity while handling demand variations with overtime and subcontracting. Short-term capacity planning focuses on planning the capacity on a daily or weekly basis, matching resources, work centers and jobs based on the specific job's requirements. (Chen et al., 2009)
Ultimately, the balance between lost sales on the one hand and excess capacity on the other hand needs to be found individually by every company. Some might decide depending on the market conditions and with accurate forecasting, others might try to come up with a numerical solution.
Hammesfahr et al. (1993); Chauhan et al. (2004) investigate strategic capacity planning, where the long term perspective on production capacity management and planning is considered.
Ho and Fang (2013) examined how to convert limited capacity into an optimal production strategy for maximizing profit using nonlinear programming. Chung et al. (2008) developed a model for companies with limited capacity that allows to find optimal solutions for seasonal demand.
Some research looked into the single-period problem or news-vendor problem, where there is a production season and a selling season, however the demand in the selling season is unknown when the production is planned (Chung et al., 2008; Petruzzi and Dada, 1999; Bradley and Arntzen, 1999; Khouja, 1999). Any excess goods which are not sold become close to worthless, just like a newspaper which hasn't been sold by the evening (but it also applies to industries like fashion and perishable foods). The uncertainty imposed on the production process by demand variations and process interruptions in a job shop production makes the problem examined by this thesis similar to the news-vendor problem, where the goal is to find a product's order quantity that maximizes the expected profit under probabilistic demand (Khouja, 1999).
Most solutions to the news-vendor problem however focus on industries where the manufactured goods become close to worthless after a while, but this usually doesn't apply to a make-to-order company. The problem for MTO companies is rather inventory cost and orders being very customer specific, but not decreasing value of the manufactured goods (after all, a customer asked specifically for that product), hence there must be a different solution to handle fluctuating demand under limited capacity constraints as a MTO company.
Chen et al. (2009) proposed a model for make-to-order companies which allows them to select only the most profitable orders while maintaining delivery date adherence. They suggest using overtime and outsourcing as ways to handle a fixed short-term production capacity. While the company in their model is allowed to refuse orders, late orders are not acceptable and therefore not considered in their model. The model's goal is to select

an optimal set of customer orders and to prescribe a schedule for each selected order so that they are completed before their due date, using mixed-integer programming. Their work could only solve small problems, as with an increasing amount of jobs that have to be considered by the problem solver program, the complexity and therefore the wall clock run time of the program increase drastically. The authors suggest that more efficient algorithms will be needed to tackle the capacity issue on an industrial scale. (Chen et al., 2009)

There are quite some research papers that use overtime as a way to overcome the problem of high lateness costs, but most authors use heuristics and mathematical programming for their solutions. The balance of overtime costs and lateness costs is a well studied topic. Yang et al. (2004) looked into the problem setting for a single machine shop floor environment that is purely deterministic. Their proposed priority algorithm for balancing the use of overtime and regular time showed close to optimal solutions, however is limited by the low shop floor complexity and the deterministic environment. A similar environment is considered by Jaramillo and Erkoc (2017), where a finite set of jobs runs in a single machine job shop. Their proposed heuristic managed to beat a mathematical model in both calculation run time and results and can therefore be applied to larger problems. Ornek and Cengiz (2006) employ multiple stacked linear programming models to design a dynamic production planning system that controls lot sizes, alternative job routing and overtime decisions and comes up with a capacity feasible material plan for a job shop environment. Yuan and Graves (2016) developed a production planning model that determines planned lead times for machines and production lot sizes for single parts to minimize inventory and overtime costs in a job shop production system.

All mentioned papers used mathematical programming methods to determine the use of overtime. These methods however have drastically increased run times the larger a production system becomes and therefore do not scale well for complex environments.

In the previous section we established the theoretical foundations of workload control and the capacity planning problem from the perspective of a make-to-order company. While there are multiple avenues for managing capacity, including numerical solutions based on mathematical programming, this thesis aims to explore a more modern approach which is based on machine learning techniques which are supposed to scale well even for complex production environments. Additionally, developing the mentioned mathematical models requires deep expert knowledge of both the mathematical modelling techniques and production management domain, whereas the proposed machine learning method requires no knowledge of the business domain at all.

The following section provides a brief introduction to the topics of artificial intelligence, machine learning and how they are applied in the field of operations research.

2.2 Machine Learning

This section lays out the details of how computers learn, what algorithms exist to make them learn and how some Machine Learning concepts can be applied to solve problems.

2.2.1 Learning, problem solving and intelligence

Learning is a hypothetical and invisible construct that refers to a (relatively) permanent change in behaviour, and is based on past experiences. While generally the word ”learning” is used with a specific end result in mind (*what is learned*), the psychological view focuses on the learning process. (Gross, 2015)
Learning is closely related to problem solving. Problem solving is required when a problem occurs, that is when a desired goal can not be reached. A subset of problem solving is decision making, as in the case of decision making we already have a set of possible solutions to choose from. (Gross, 2015)
When solving problems, humans (or animals) operate vastly different from computers. While humans use their cognitive capabilities (natural intelligence) which are limited, but inherently good at learning and solving problems of a certain scope, computers can only execute orders given to them by a program. Computers however have an inconceivably large processing power, which allows them to perform tasks that are too hard for humans (e.g. complex mathematical calculations or guessing a password). A ”simple” task like telling which color a shoe has would however be impossible for a computer, as that requires prior experience obtained from learning.
This difference between computers and natural intelligence gets smaller with the advent of artificial intelligence, which is the ”study of intelligent behaviour” or ”the science of thinking machines”. (Garnham, 2017)
A good problem setting to compare natural intelligence, conventional computer programming and artificial intelligence is the classification of spam e-Mails (a common example used in literature like Shalev-Shwartz and Ben-David (2014); Alpaydin (2014); Müller et al. (2016)). The task is to read an e-Mail and decide whether it is an undesired spam e-Mail or a normal one that should be delivered to the recipient. Most people will quickly notice based on all their past experiences (acquired through learning), that no prince from Nigeria would ever send them a large sum of money just like that and would correctly label such an e-Mail as spam. A conventionally programmed computer, being good at algorithmic tasks like the above mentioned mathematical calculations, will not perform well on classifying e-Mails, as there are no set rules that can tell if an e-Mail is genuine or spam. Such a task requires understanding natural language and then comparing the mail’s content with previous experiences. Early spam filters worked based on rules decided on by human experts, for example by using blacklists containing keywords that would get an e-Mail filtered out if it contained enough keywords (Müller et al., 2016).
While seemingly intelligent, such systems still relied on rules defined by human experience. Hand coded rules have two disadvantages, the first one being that the logic for a decision is specific to a domain and task and using it for a different task would require rewriting

the system, as such fixed rules can't generalise. The other disadvantage is that such fixed rules require a human expert to understand how a decision should be made, which only works as long as humans understand the subject.(Müller et al., 2016)
One domain where humans failed to give a computer a clear set of rules is face detection. Humans perceive vision entirely different than computers, as computers work with numbers and pixels, and a human can't describe his vision with millions of data points. Modern phones can now detect faces, something that was technically impossible about 20 years ago. (Müller et al., 2016)
Let's go back to the e-Mail classification example. If we have a large amount of data, e.g. millions of e-Mails and for each e-Mail a label if it is spam or not, we can use machine learning (a subset of artificial intelligence) and try to have the computer learn from that large set of already classified e-Mails. It can try to recognize patterns among millions of data points, a task that would be impossible for a human. While such a machine learning model might not operate perfectly (e.g. by classifying a genuine e-Mail as spam), it can give "a good and useful approximation" (Alpaydin, 2014). As long as the computer detects patterns in the large data set, these patterns can be seen as the new experience of the computer and we can make predictions based on this experience. This can be applied to the previously mentioned example of face detection, where a computer can detect the patterns that make up a human face by analyzing a large amount of images, each containing millions of pixels.
Two important concepts of Machine Learning are supervised and unsupervised learning. *Supervised learning* is learning from a training set of labeled examples provided by a knowledgeable external supervisor (Sutton and Barto, 2018). An example would be a data set containing two features, data on a credit card transaction and a label whether it is a fraudulent transaction or not. A supervised learning algorithm can then find patterns in the transaction data and learn if they belong rather to a genuine transaction or fraud. Broadly speaking, we give the solution and the algorithm finds a way to solve the problem for new, unseen data. Our e-Mail classification problem could be solved using supervised learning.
Unsupervised learning is about finding structure hidden in collections of unlabeled data. A practical example for unlabeled data would be sales records from a vendor. There is no clear label indicating the preferences of a customer, but by using unsupervised learning to find patterns in the customer records (e.g. purchase history), one might find a way to cluster customers into segments. Such a segmentation can then be used to tailor marketing efforts towards a well-defined target audience.
The previously discussed example of e-Mail classification is a simple one, as it is rather limited in complexity and scope. In a dynamic system with changing states that requires new decisions regularly (e.g. in the manufacturing of goods, where the state of the machines and products is constantly changing), we need more advanced applications of machine learning. Such a dynamic manufacturing system will be covered later in this thesis.

2.2.2 Artificial neural networks and deep learning

As discussed before, artificial intelligence has developed capabilities that can match or exceed the capabilities of natural intelligence for some tasks. Natural intelligence is not only the benchmark that artificial intelligence gets compared to in terms of performance, it also serves as an inspiration for AI researchers when it comes to designing machine learning applications. Artificial neural networks (ANN) are models of computation which are inspired by the structure of neural networks in the brain (Shalev-Shwartz and Ben-David, 2014). They are composed of a set of interconnected processing elements which are named neurons or nodes (Yao, 1999). This mimics the structure of an actual brain, where there are neurons connected by synapses. Both a real brain and an ANN transmit signals over synapses (edges) between the neurons (nodes), which can be described with a directed graph.
Foster (2019) describes Deep Learning as "a class of machine learning algorithm that uses multiple stacked layers of processing units to learn high-level representations from unstructured data". As this is a definition that one will probably have to read multiple times, let's discuss the concepts that were mentioned.
Structured data refers to the data being in the form of a table with columns and rows, similar to an Excel spreadsheet. An example could be data on people, with each row containing the data on one person and the columns representing the different features like birthday or name. Unstructured data refers to data that cannot be explained with a table, like images, videos or audio data. Those can be explained in the form of pixels, audio waves and time. Most common Machine Learning algorithms require structured data, so Deep Learning is needed to learn from unstructured data, as it "can learn how to build high-level informative features by itself, directly from the unstructured data". Deep Learning can also be applied to structured data as well, as we will see later in this thesis. [3] (Foster, 2019)
Most deep learning systems are (artificial) neural networks, so these terms are often used synonymous. A rather simple form of a neural network is a multilayer perceptron (MLP) that is used for classification [4] and regression [5] tasks. According to Müller et al. (2016), "MLPs can be viewed as generalizations of linear models [6] that perform multiple stages of processing to come to a decision". What makes a neural network important are the *multiple stages of processing*, which is best explained with some visualisations.
Figure A.1 shows a linear model, one of the most basic machine learning models. It features four input features and one output feature. A practical example would be giving the four inputs *education*, *age*, *address* and *nationality* and receiving the output feature *income*. Of course the model doesn't know the income by itself, so it has to be trained

[3] See *Figure 2-1. The difference between structured and unstructured data* in Foster (2019) for a good visualisation on the difference between structured and unstructured data
[4] In classification, the goal is to predict a class label from a predefined list of possibilities, e.g. classifying e-Mails as spam or not spam (Müller et al., 2016)
[5] For regression tasks, the goal is to predict a number, e.g. a person's annual income from their education, age and address (Müller et al., 2016)
[6] A linear model is one of the oldest and most basic techniques. It uses a linear function of its input features to make a prediction. (Müller et al., 2016)

with lots of structured training data before. The training set could be a table with five columns containing data on the four input features and the single output feature, as well as many rows so that the linear model can detect patterns (detecting patterns would be hard from just a few rows of data). The linear model assigns weights to each of the input features, indicated in the figure as *w[0]* and so on. These weights are learned coefficients, which (broadly speaking) indicate the "importance" or "influence" of an input feature at explaining the output. (Müller et al., 2016). One finding of the model could be that the input feature *education* has the largest influence on the predicted output *income*. Training a network refers to the process of finding these weights or coefficients (Foster, 2019).
The deep learning approach works similar to such a basic model, however it adds some processing layers between the input and output features. Figure A.2 shows such a model, a multilayer perceptron (so it's a neural network), with one hidden layer. A hidden processing layer adds lots of complexity to the model, as there are fare more coefficients to learn. As we can see, there is one coefficient between each input feature and each hidden unit, allowing the model to learn more complex relations between the features. This model can be scaled up with arbitrary many hidden layers, as seen in Figure A.3 where there are two layers. Depending on the model, there can be many more hidden layers, sometimes hundreds of layers for complex tasks like image recognition (Foster, 2019). This added "deepness" and complexity to the model is what we refer to as Deep Learning.

2.2.3 Reinforcement learning

After an introduction to what machine learning is, we now discuss the ML technique that is used in this thesis - Reinforcement Learning (RL).

What is Reinforcement Learning?

> "RL is a general class of algorithms in the field of machine learning that aims at allowing an agent to learn how to behave in an environment, where the only feedback consists of a scalar reward signal [...] The goal of the agent is to perform actions that maximize the reward signal in the long run."
> (Wiering and Van Otterlo, 2012)

Sutton and Barto (2018) state that learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence. Whether one is driving a car or talks to a person, there are reactions from the surrounding environment that influence further actions. Reinforcement Learning is a computational approach to learning from interaction and more focused on goal-directed learning from interaction than are other approaches to machine learning like supervised or unsupervised learning.(Sutton and Barto, 2018)
At the core of RL, there is a decision maker (called the agent) that is placed in an

environment. The agent undertakes actions and receives back a state and reward from the environment. Based on the information he just received, the agent comes up with a new action which in turn gives him a new state and a new reward. This goes on in a repeating loop from which the agent can gather experience and improve his decision making based on what the environment returns back to him (more on that in Section 2.2.3).
A practical example would be a chess game, where the player is the agent and the chess board is the environment. The environment is always in a certain state, which can change over time. Each state is part of a set of all possible states. In chess, the state could be the current position and type of all chess pieces on the board. The set of all possible states is an aggregate of all possible positions for all chess pieces on the board. If a player moves a chess piece, this changes the state to a different one even though all the other pieces remain on the same position.
The agent (player) has a set of possible actions which in the chess example are characterised by legal movements of chess pieces on the board (as defined by the game rules). After taking an action (moving a chess piece), the state of the environment changes. This sequence of action and obtaining a new state based on the previous action can be cycled through for a while, but like chess, it has to end at some point. Once the game ends, the agent receives a new piece of information: the reward. In chess, this can be one of several possible outcomes: a victory, a loss or a stalemate (there are actually many more outcomes, but for sake of simplicity let's assume only these three). In most games, there is a reward at the end which tells if one has one or lost the game. Depending on the problem, the reward(s) might come in different shapes like after every single action of the agent. While in most games, the reward comes only at the end, other problem sets can have rewards come in different shapes and even after every action of the agent. (Alpaydin, 2014)
By now it might have become clear that Reinforcement Learning is quite different from the other two major domains of ML, supervised learning and unsupervised learning. Where the other two domains analyze labeled and unlabeled data to provide information to the user, RL goes one step further and undertakes decisions on its own. This decision-making is what is necessary for a self-learning system that chooses the balance of lateness and overtime cost in a production system.
The theoretical foundation of the just described Reinforcement Learning process (using the example of chess) is called Markov Decision Process and will be described in the following section.

Markov decision processes

> "Reinforcement Learning can be formalized using dynamical systems theory, specifically with Markov decision processes (MDP). MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards."
> (Sutton and Barto, 2018)

Markov decision processes are the theoretical foundation of Reinforcement Learning. As stated above, the core elements of RL are an agent and an environment as well as states, actions and rewards in a discrete time frame. The interactions of these core elements will now be discussed.
Time is defined as a discrete number t (so it increases with integer numbers), S is the set of all possible states and S_t is the state of the agent at time t. A_t refers to the action that an agent selects at time step t. This decision is always based on state S_t. Once the agent has undertaken his action A_t, one time step passes and he will receive the numerical reward R_{t+1} and has a new state S_{t+1}. Once having received a new state, the agent will make a new decision, receive another state and reward and so on. In a finite Markov Decision Process, this sequence has a finite amount of individual elements, so there is a limit on the number of possible states, actions and rewards. The process is depicted below in Figure 2.1. (Alpaydin, 2014; Sutton and Barto, 2018) For the scope of this thesis

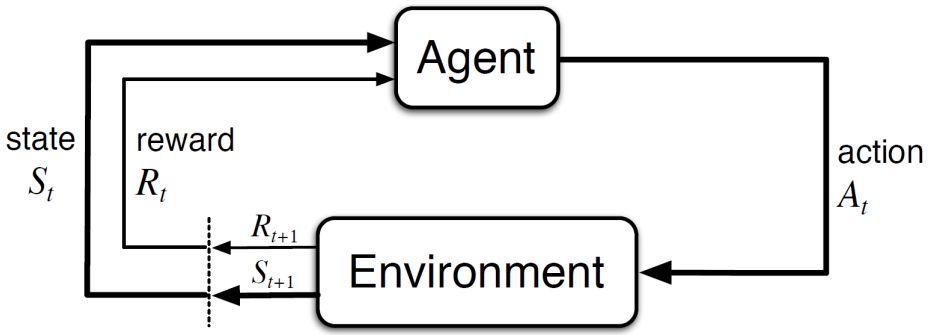


Figure 2.1: The agent–environment interaction in a Markov decision process (Sutton and Barto, 2018)

it is sufficient to understand the basic structure of a Markov Decision Process in that it is a constant loop of interactions between the agent and the environment. In the following Section 3.3 we will look at how a finite Markov Decision Process was implemented for the job shop capacity planning problem.

State of the art algorithms and applications

Reinforcement Learning has so far been applied only to a few real world problems. In contrast to established Machine Learning paradigms like supervised and unsupervised

learning, RL applications are far from routine (Sutton and Barto, 2018). With advancements in RL research, more real life applications emerge over time. RL has been used for resource provisioning to cloud applications (John et al., 2019), tuning cloud database settings (Zhang et al., 2019), controlling robots (Kormushev et al., 2013) or improving data center power efficiency both in terms of network routing (Sun et al., 2020) and cooling management (Li et al., 2019).
RL methods have been applied to games to a much greater extent. Just in the last years, RL has seen a dramatic improvements in playing games. Starting in 2013 with an RL algorithm (Deep Q-Learning or DQN) being used to play games from the Atari video console and beating human expert players in three out of six games tested (Mnih et al., 2013), the methods have quickly improved. In 2016, a research team trained a RL model to play the game Go with a mix of supervised learning from human expert games and RL from games played by the algorithm itself, achieving a 99.8% winning rate against other Go programs and defeating the human European Go champion by 5 games to 0 (Silver et al., 2016). In 2017, that same algorithm was improved to not require any human training at all, learning just by playing against itself and winning 100–0 against previous year's champion-defeating algorithm (Silver et al., 2017). In 2018, the algorithm has been used to teach itself other games like chess or shogi, beating established algorithms which were specialized on the specific games (Silver et al., 2018). In 2019, RL has been used in competitive computer games like Starcraft to rank higher than 99.8% of all human players (Vinyals et al., 2019) or in the game DOTA 2, managing to beat the world champion team (Berner et al., 2019).
Reinforcement Learning has developed the ability to learn the rules of any game and achieve super human capabilities in even the most complex games. But still there are only few applications in real life and beyond games. If we take the chess example from the beginning of this section, the Markov Decision Process would be rather simple to design. A chess board (the environment) has 64 squares, so the state could be designed as a vector with 64 values. Each single value could include information on which chess piece is situated on that specific square. The action space is a bit more complex, as different chess pieces can have different actions, and for each action the player can choose which chess piece should be moved next. After each move of the player, the reward would be 0 until the game ends. The final reward would be -1 for losing the match, and +1 for winning the match. So while formulating a MDP for chess is not trivial, the complexity of the MDP is somewhat limited by the game rules. Also, chess can be easily run as a computer program, which allows training an RL algorithm very quickly. This holds true for many games, thus researchers can focus more on the algorithm design. When taking RL to the real world, for example for controlling robots, there are quite some challenges in contrast to (computer) games. There are physical safety requirements such as not moving too fast and not breaking parts of the robot through random movements when exploring actions (Kormushev et al., 2013). Training RL models on real robots is quite inefficient as, in contrast to computer simulations, it takes a lot of time and resources to converge using trial and error (Zhu et al., 2017). In contrast to humans, common RL algorithms require more steps of training to achieve a similar performance,

alas the sample efficiency increases with new developments in RL research (Botvinick et al., 2019). Successful applications of RL to robotics are for example flipping a pancake using a robot arm that holds a frying pan (Kormushev et al., 2013), opening doors (Gu et al., 2017) or visual indoor navigation (Zhu et al., 2017).
As RL works best for problems that can be simulated on a computer, why hasn't RL been applied to decision support systems and other applications where decisions influence an economic outcome? One issue is that currently, most state of the art RL algorithms have been used for and tuned to playing games. As we know from the chess example, the reward for most games can be designed in such a way that every round of the game gives 0 reward and only in the end there is a reward of $+1$ or -1, depending on victory or defeat. This is also what most common algorithms are tuned for, so they optimize the agent's action as to maximize the obtained reward. But for economic problems it's not always possible to design the MDP and the reward in such a simple interval from -1 to $+1$. If an environment returns cost or profit as a reward, this cannot be formulated with just an integer range of three different numbers. Economic applications regularly return a non-zero reward after most training periods, which reduces the learning performance of some established RL algorithms (Schneckenreither, 2020). An addition to the established Q-Learning algorithm is provided by Schneckenreither (2020), which uses *Average Reward Adjusted Discounted Reinforcement Learning* to overcome the issues of standard discounted reinforcement learning algorithms when used on economic problems. The make-to-order company investigated by this thesis has production cost as its main success metric, so an RL environment built on such a job shop simulation would need to return a non-zero reward after almost every period. This thesis will later compare the algorithm of Schneckenreither (2020) to other algorithms on the capacity planning problem.
Finally, RL algorithms are domain-agnostic, that means they can be applied to any decision problem setting. The algorithm doesn't know what the logic or actual problem is behind the state and reward information that it receives, since these are only long lists of numbers. Therefore one (hypothetically perfectly tuned) algorithm could in theory be used for every problem that can be formulated using a MDP.

On the benefits of neural networks As stated before there are many different Reinforcement Learning algorithms. Some rather basic ones like Q-Learning (Watkins, 1989) use a tabular based approach, where the individual values that are relevant for deciding an action are being stored in a table with rows and columns. Such algorithms can certainly be used for most RL tasks, but all information that is passed from the environment to the agent needs to be accounted for in these tables. The more complex an environment or problem setting is, the larger the table must be in order to store all possible values.
If we look at the example of Q-Learning by Watkins (1989), the algorithm stores the discounted future reward for each combination of state and action in a table such as Table 2.1. For each possible combination of state and action there is one Q-Value stored inside the table.

	Action 1	Action 2	...	Action_m
State 1	Q_{11}	Q_{12}	...	Q_{1m}
State 2	Q_{21}	Q_{22}	...	Q_{2m}
...	...	...	...	...
State_n	Q_{n1}	Q_{n2}	...	Q_{nm}

Table 2.1: A Q-Learning table based on Zhang et al. (2019)

A small problem size like the game Tic Tac Toe is a perfect fit for a tabular Q-Learning approach. The game field has three rows and three columns, resulting in nine possible cells where a player can place one of two possible symbols. Including empty cells that have no symbol (yet) this allows each cell to have one of three possible values (symbol 1, symbol 2, empty). If we ignore game rules (like that one player wins once he has three of his symbols in one row or column), there are a total of $3^9 = 19683$ possible states. If we increased the problem size and added one row and one column to the game board, there would now be 3^{16} possible states - that is more than 43 million states. Just by adding a bit of complexity to the environment, the Q-table has dramatically increased in size and became harder to handle with traditional computer storage.
It becomes obvious that a tabular approach becomes less viable for larger problems, which is why neural networks are used with most modern Reinforcement Learning algorithms. Through the use of neural networks, the term Reinforcement Learning turns into Deep Reinforcement Learning (DRL). DRL is capable of storing and handling vast amounts of data for very complex problems by using neural networks.
The following chapter describes a model that applies Deep Reinforcement Learning to tackle the challenge of balancing overtime and lateness cost in a job shop environment.

3 Model

3.1 Simulating a job shop production

This chapter describes how a make-to-order company's production process is recreated as a computer simulation and then gets fitted to a structure that allows a Reinforcement Learning agent to learn from interacting with the production system. Additionally, multiple RL algorithms are implemented as agents for controlling the production environment.

3.1.1 Job shop model

In the design of the job shop model, the study follows closely the work of Haeussler et al. (2019), however the production system was changed from a flow shop to a job shop. The changes that were necessary for changing to a job shop environment are indicated below.

Basic structure The job shop of the make-to-order company contains an order pool, three work centers (each containing a queue and a machine), and a finished goods inventory. Once a customer order arrives, a job gets created inside the system and is added to the order pool. Despite following the workload control paradigm, no order acceptance mechanism is used in this study and all customer orders get accepted. The jobs inside the order pool get released at the beginning of a work day. After being released to the shop floor, the jobs follow their assigned routing depending on the product type. There are six different product types and all machines can process each product type. Each job has to go once through every machine before it is completed, but the routing differs between the six product types and there are no return visits. Thus, each job has to pass three production steps before it is finished.

Product type	Step 1	Step 2	Step 3
1	M1	M2	M3
2	M1	M3	M2
3	M2	M1	M3
4	M2	M3	M1
5	M3	M1	M2
6	M3	M2	M1

Table 3.1: Routing of product types

Table 3.1 shows the sequence in which the jobs have to pass through machines (M1 - M3), depending on their product type.
Once a job arrives in the designated work center according to the routing table, it gets placed inside the queue of that work center and waits until it can get processed inside the machine. Each machine can only process one job at a time. If there are multiple jobs inside a work center, the ones that are not being processed need to wait inside the queue. Once the machine has finished processing a job, the job which has arrived first to the queue gets moved to the machine for processing. The system employs a first-come-first-serve rule for the scheduling of jobs. Once a job has been finished, it gets moved to the finished goods inventory where it waits to be shipped at its designated due date. If the job is finished before its due date, it has to wait until the due date is reached and then gets shipped (early releases are prohibited). If the job is finished after its designated due date (thus being late), it gets shipped right away. Once an order has been shipped, it is gone from the shop floor and cannot influence it in any way.
Figure 3.1 shows a simplified structure of the shop floor with processing time distributions and example jobs inside machines and queues.

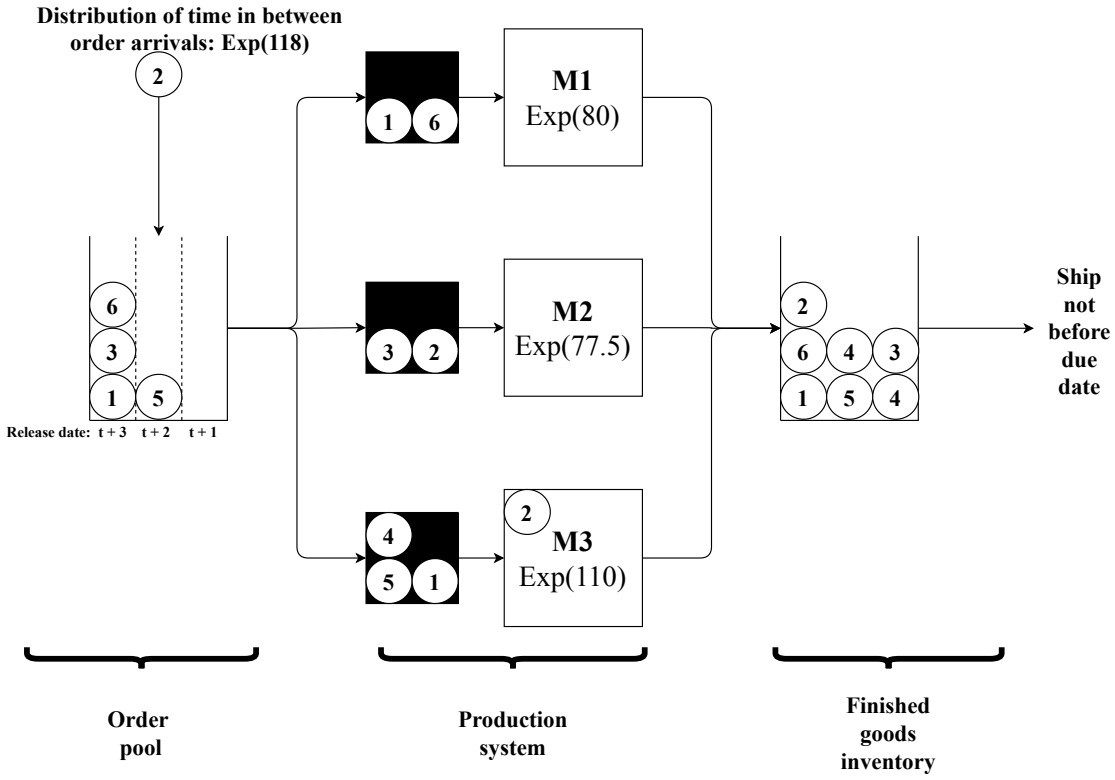


Figure 3.1: Structure of the job shop production system

Temporal considerations The job shop model considers time, so every action that happens during manufacturing happens at a certain time. A work day (also named period) is considered to be 16 hours (that is 960 minutes) long and is made up of two working shifts. At the beginning of each working day, jobs get released to the shop floor depending on the order release heuristic (as described below). Jobs that get processed in machines take a certain amount of time (measured in minutes). Lateness and earliness of orders are also calculated in minutes. One run of 8000 episodes or work days is called episode and refers to one full data sample, as the simulation needs to run for a while to generate meaningful results. (Haeussler et al., 2019)

Order generation Customer orders arrive to the system based on an exponential distribution Exp(118), which refers to the expected time between two events (in this case inter-arrival time of orders). In contrast to a uniform distribution, there is more variability in the data. This may result in times of almost no orders arriving, whereas at other times many orders may be placed by the customers within a short time frame. Exp(118) means that roughly every 118 minutes an order arrives.
Orders that arrive to the system are assigned a due date which indicates the time at which the order has to be shipped. The due date slack is always ten periods (or work days) after the order has arrived, so if an order arrives on day 1, it is due on day 11. Orders have a product type that is indicated by a number between 1 and 6 and the type assignment follows a uniform distribution. (Haeussler et al., 2019)

Order release Just like in the work of Haeussler et al. (2019) the job shop's order release mechanism has a backward infinite loading (BIL) policy which releases certain orders to the shop floor at the beginning of each period. Due to the BIL policy the order release mechanism doesn't release all orders from the order pool right away, but rather depending on a planned release date which is set for each order once it arrives in the order pool. The planned release date is calculated by subtracting the planned release date slack from the due date measured in periods from now. As an example, a job that is due in ten periods and has a planned release date slack of 2 periods would get released in eight periods. For a planned release date slack of 2 each job's planned release date is set to two periods before the job's due date. The order release mechanism releases all orders that have reached their planned release date at the beginning of the respective period. (Haeussler et al., 2019)

Processing jobs While the processing of jobs follows mostly the structure of Haeussler et al. (2019), some aspects had to be adapted due to the reduction to three work centers. The processing times of the machines follow an exponential distribution. All three machines have different processing times. Machine 1 is assigned Exp(80), machine 2 Exp(77.5) and machine 3 Exp(110).
Every time a new job gets placed inside a machine for further processing, the processing time for that order is calculated based on the exponential distribution mentioned above and a variable for the remaining processing time is assigned. As an example, the expected

processing time to finish one product in its current production step is 110 minutes for machine 3, but due to the variability introduced by the exponential distribution this may vary and at one time result in an actual processing time of 95, at another time it may result in an actual processing time of 113. As machine 3 has the highest expected processing time, it is considered the bottleneck of the production system. This makes it the most promising machine on which to try out some improvements later in this thesis. Whenever an order is inside a machine, it gets processed and the variable for the remaining processing time gets reduced by one. Once it reaches zero, the order is finished and gets removed from the machine.

Using overtime In addition to the base model of Haeussler et al. (2019) this study introduces the concept of overtime. Overtime means an increase in production capacity which can for example be achieved by having staff and machines work for more hours per day or by running the manufacturing process on more days per week or by outsourcing certain processes to external suppliers. How overtime is handled in reality is not evaluated here, but instead how overtime influences the production of the company. Overtime is the exchange of money (in form of production costs) for more production capacity.
Overtime is modelled as a decrease in processing time of the affected machines and it only affects the bottleneck machine (the "slowest" machine on the shop floor). Let's assume that a bottleneck machine has an exponential processing time distribution of Exp(110), that means that a job is expected to take 110 minutes to process on that machine, but might require a higher or lower amount of minutes. We further assume that a newly arrived job actually requires 100 minutes inside the bottleneck machine (due to the variability introduced by the exponential distribution). With no overtime, the job will take 100 minutes. If we however introduce 25% overtime, the job will take 25% less time to process, that is 75 minutes.
As one period or work day has 16 hours, an overtime of 25% refers to an additional 4 hours worked and an overtime of 50% refers to an additional 8 hours worked.
It would also be possible to model overtime in other ways, but all options yield the same result: the bottleneck machine's processing capacity increases by the percentage of the processing time reduction. This model uses the approach of reducing the required processing time when overtime is introduced. Overtime is applied to the production system as the variable $\mathcal{O}$ and can have the following values: 1.0, 1.125, 1.25 and 1.5. This means that the possible manifestations of overtime are either no overtime (1.0), 12.5%, 25% or 50% overtime, resulting in a processing time reduction or capacity increase of 0, 12.5, 25 or 50 percent.
Applying overtime to the bottleneck machine results in costs which are discussed below. The overtime multiplier $\mathcal{O}$ gets multiplied with the value of a machine's processing time reduction per minute. If there is no overtime, $\mathcal{O}$ is 1.0 and thus for every minute that would normally be subtracted from a job's remaining processing time, one minute gets subtracted. However if $\mathcal{O}$ is 1.25, the job's remaining processing time gets subtracted 1.25 minutes per actual minute. This results in the jobs being processed inside the machine for 25% less time.

Cost measurement Running a manufacturing system incurs certain costs beyond the material cost of the manufactured products, for example wages for workers, acquisition costs of machines and tools or setup costs. This is reflected in the job shop model with costs being incurred for all jobs that are being located somewhere on the shop floor. Any job that got released from the order pool up to the time it gets shipped is being counted, that means all work centers and the finished goods inventory are being considered. All costs are measured at the end of each period. Costs are equal for all product types and always constant.
The cost for jobs inside a work center (WIP cost) on the end of a period is €1 per job, and it doesn't matter if the job is inside a machine or a queue. The cost for (early) jobs waiting inside the finished goods inventory is €4 per unit. Late jobs incur contractual fees and therefore the back-order cost is €16 for each period that a job gets shipped late (so it's €32 if a job is two periods late). If overtime was used during a period, a base cost of €8 multiplied with the amount of extra hours is incurred to pay the additional hours that workers had to put in. As mentioned before, there are 12.5%, 25% and 50% overtime which refer to 2, 4 and 8 extra hours being worked in one period. Overtime is the only cost that is incurred globally as a flat-rate cost, instead of a cost per individual job.
As an example, if an overtime of 25% or $\mathcal{O} = 1.25$ is ordered, the overtime cost for that period is €8 × 4 hours = €32.
When running longer simulations over multiple periods (usually 8000 periods), a warm-up period gets introduced. The warm-up period takes place after 1000 consecutive periods and resets all cost to zero, as the first 1000 periods are used to get the production system fully running and filled with jobs. (Haeussler et al., 2019)
Let's consider a practical example for how costs get measured. At the end of a work day, two orders are in work center (WC) 1, three orders are in WC2, none in WC3, one is in the finished goods inventory and one order was shipped late by one period. There was overtime of 25% or four extra hours. This means that the final cost of that period is €57 (€5 for the work centers, €4 for finished goods, €16 for late goods, €32 for overtime).

Differences to Haeussler et al. (2019) The original study that the production system of this thesis is based on evaluated a flow shop with six machines, whereas here a job shop with three machines is investigated. In the flow shop model, processing times of machines and inter-arrival times of customer orders follow both uniform and exponential distributions, based on the desired variability of the process. The here presented job shop model uses only exponential distributions. Releasing orders happens dynamically through adaptive lead times in the original paper, whereas this thesis uses a static lead time approach by employing a backward infinite loading policy with two or three periods of planned due date slack. The processing times of the machines have been adapted to compensate for the lower amount of machines processing the same amount of orders. In addition to the flow shop model, this study introduces an overtime approach, allowing

the manufacturing process to have a higher production capacity. Cost rates for overtime were introduced additionally while the normal cost rates (shop floor, finished goods and lateness cost) stayed the same as in the original paper.

3.1.2 Implementation of the simulation

The implementation of the simulation has been done with the Python programming language in version 3.8. One simulation episode consists of 8000 periods (that is 8000 work days) and one simulation run consists of 30 episodes (that means 30 data samples). This results in 240000 simulated periods, which should be sufficient to ensure comparability between simulation runs.
The model supports two modes of operation, one being the job shop model with three machines as described above and the other being a simplified version of the job shop model which features only one machine. The single machine model makes it easier to try out new Reinforcement Learning algorithms, as having just one machine reduces the system complexity and makes it easier for the algorithms to learn. This thesis uses the full setup with three machines.
With a change of a parameter inside the settings file it is possible to switch between the two modes of operations before a simulation run. The settings file allows changing many more factors like cost rates, demand levels and policies or heuristics.
Overtime can be applied to the simulation by passing an action into the environment's step function, which is explained below in Section 3.1.3. When overtime is active, a global modifier gets updated with the respective percentage of overtime, resulting in the processing capacity increase as described above.
The simulation implementation is wrapped inside an environment that follows the structures defined by OpenAI Gym, which will be explained in the following section.

3.1.3 OpenAI Gym environment

What is a Gym environment? Gym is a toolkit for developing and comparing reinforcement learning algorithms provided by OpenAI. It serves as a collection of benchmark Reinforcement Learning problems and provides these in a convenient and accessible way. It uses a fixed structure to make these RL problems or tasks available to RL algorithms, which use common interfaces to interact with the simulated problems.(Brockman et al., 2016)
The two main concepts of Gym are environments and agents. The Gym structure that contains a RL task is called environment. Such an environment may contain a common task like balancing a pole on a cart or swinging a pendulum. Gym ships with multiple common benchmark tasks (or environments) included. These benchmark tasks have been discussed in the literature (Duan et al., 2016; Brockman et al., 2016). Gym natively supports more advanced environments based on 2D and 3D simulations which also consider the laws of physics, like teaching robots how to walk. Additionally, it is possible to have a RL algorithm play games from the Atari 2600 console.
The word agent refers to the implementation of a RL algorithm. Gym supports the

creation of custom environments, allowing researchers to come up with new RL tasks. While the common RL benchmark tasks and the ones that are delivered with Gym offer a great variety of different problem settings and allow for development of sophisticated RL algorithms, there are no common tasks for economic behaviour or operations management problems. This thesis builds a custom Gym environment based on the previously discussed job shop simulation, as the structures and interfaces of a Gym environment allow for convenient development of new RL algorithms and comparison to existing algorithms. Due to the flexible nature of a Gym environment, an algorithm that has been optimised for the job shop capacity planning problem can also be evaluated on the common RL benchmark problems.
To sum up, Gym provides a convenient way to either test one RL algorithm on multiple different RL tasks by switching environments, but also allows to test multiple algorithms on one specific RL task by switching the algorithms.

Basic structure of a Gym environment Gym environments are written in the Python programming language. This allows the job shop simulation to be converted to the structures imposed by a Gym environment rather easily.
Only a few things are mandatory to define a Gym environment. First, there must be an action space and an observation space. Additionally, a gym environment must always have at least two functions: `reset()` and `step()`.

Action space Actions are the method that an agent uses to assert control over an environment. These actions influence the environment in a certain way, and the effect of each action as well as the possible actions themselves need to be defined by the environment's creator.
The action space of a Gym environment refers to the set of valid actions that the agent can execute. It is defined by a set of numbers which indicate the boundaries of the possible actions. An action is usually a single number or a set of numbers of any form, like integer, decimal number, etc. A set of numbers can be an arbitrary structure of numbers like intervals, lists, tuples and so on. These numbers can be continuous or discrete.
In this simulation, the action space is defined as a discrete range of numbers in the interval $\{0, 1, 2\}$. This means that there are three possible actions, which are named action 0, action 1 and action 2. In the Analysis there are four different experimental designs which have two different sets of actions, named setup A and setup B. The action setups differ in the amount of overtime each action causes and the overtime values for each action and setup can be seen in Table 4.1. At the beginning of every new period of the simulation the agent must choose a new action, which will then influence the production capacity for that period. There is no restriction on how often an action can or must be used, but every period one single action must be chosen.

Observation space An observation is an environment-specific object representing an observation of the environment, which can for example be pixel data from a camera,

joint angles and joint velocities of a robot, or the board state in a board game (OpenAI, 2020). Such an observation refers to the state returned from the environment in a Markov Decision Process (see 2.2.3). A Gym environment's observation space is the structure of valid observations or states returned by the environment.
Just like actions, observations are defined by a set of numbers in an arbitrary format (like discrete number ranges, vectors, lists, and so on) and can span over multiple dimensions. Each observation in the here presented environment is made up of a collection of production metrics. A simplified definition would be that the observations contain for each stage of production the amount of jobs inside the respective stage, separated by product type. In total, an observation contains 132 data points. If we look back at the description of the simulation base model, there are nowhere as many stages of production (namely order pool, work centers 1 to 3, finished goods inventory) and only six product types. The large number of observations stems from the fact that for some production stages there are multiple data points. This gets more clear when looking at Table A.1 in Appendix A.
Each product type has 22 data points, resulting in an observation with a total of 132 data points. This observation is defined inside the Gym environment as a one-dimensional array which gets returned after every action of an agent. Table A.1 displays the observation in a tabular way for human readability, but the actual observation is an array of 132 numbers.
The table shows full mappings only for product type 1, but the structure and mix/max values are the same for all other product types. The "Min" and "Max" columns indicate the minimum and maximum value that the respective data point can have. For data point 1 this indicates that the number of orders in the pool that are due in one period must be in the interval $[0, 15]$. Technically, there is no limit on how many orders can be inside the order pool or any other stage of production, but a maximum of 15 or 30 was given for each data point to obtain a boundary that will usually not be exceeded, but is small enough that observations can be handled well. It is assumed that not more than 15 jobs are inside each stage of production, with the exception being the amount of orders inside the finished goods inventory which are early by 1 period. Experience shows that this value can on rare occasions exceed 15, thus 30 was chosen as a safe boundary that will usually not be reached.
An example observation could look as follows:

[1 5 1 1 2 2 0 3 0 0 0 0 0 0 0 0 1 1 0 0 0 0 2 3 2 0 3 0 3 2 0 0 0 0 0 0 0 0 0 2 0 0 0 3 1 0
0 1 1 3 2 2 0 0 0 0 0 0 0 0 2 0 1 0 0 1 1 1 0 0 0 2 4 2 0 0 0 0 0 0 0 0 1 0 0 0 3 1 2 1 3 3
0 2 2 0 0 0 0 0 0 0 0 0 1 0 0 0 1 2 3 1 1 1 2 1 2 0 2 0 0 0 0 0 0 0 1 0 0 0]

Figure 3.2: Example observation from the environment

Using Table A.1, we can map each number to a certain data point's description. The first number in Figure 3.2 refers to data point 1 in Table A.1, the second number to data point 2 and so on. As the example observation starts with the numbers 1 and 5, we conclude that there is one job of product type 1 inside the order pool and due in one

period as well as five jobs of product type 1 inside the order pool and due in two periods. This mapping process can be applied to all of the 132 data points.
After every action undertaken by an agent, the state of the environment gets updated and the environment returns its new state in the form of an observation back to the agent.

Reset and step Both the `reset()` and `step()` functions are called on an instantiated Gym environment to interact with the environment.
The `reset()` function resets the entire environment to the default state and returns an observation. For the job shop simulation this means that the entire production is reset to an empty state. All inventories, machines and lists get cleared and all orders are deleted. Parameters and metrics are reset to their default values. The time is reset to 0. After that, the environment returns an observation of the default state, which is always the same after each call of `reset()`. The observation of the empty default state contains the value 0 for each individual element of the observation, thus resulting in a list of 132 zeros.
The `step()` function takes an action as input and steps the environment ahead by one step, influencing the production system with the given action. Stepping ahead refers to advancing the job shop production simulation by one period or working day. An input action is valid if it is a part of the action space. After the environment has stepped forward, the function always returns four values: `observation`, `reward)`, `done` and `info`. Returning these four values is mandated by the Gym structure and the values always refer to the period that just has passed. So if we assume that a period has passed in the job shop simulation, the observation and reward refer to the system state of the period end and not at the beginning of a new period.
The `observation` parameter returns an observation from the environment, namely its current state at the end of the period that just has passed. The observation must always be a part of the observation state.
The `reward` is the reward that was collected on the end of the period, which in our case is the total cost that was incurred in the period.
`done` returns a Boolean value (either true or false) and indicates whether the current simulation episode is completed. This is mostly used for environments where an agent can fail, e.g. when teaching a robot how to walk and it fell of a cliff. The agent then knows it must call the `reset()` function and start over. The job shop simulation is however continuous and can not fail, each episode just ends after 8000 periods. Thus the `done` parameter is always returned as false for the first 7999 periods of each episode, and only after the final period 8000 it is returned as true. This then usually results in the agent resetting the environment (depending on the agent's implementation).
`info` can be used for passing human-readable text to an agent, allowing a developer to receive additional information. It is however not used in the implementation of the job shop environment.

Considerations for using Gym and RL in a shop floor simulation As argued before, there are not many applications of Reinforcement Learning in the context of economic

behaviour or operations management. While it is no problem to model such a task with a Gym environment from a technical perspective, some issues arise when common RL algorithms are supposed to interact with such an environment that returns observations and rewards diverging from the expected norm.
Section 2.2.3 described that designing rewards for a Markov Decision Process is not trivial for problems that periodically return a non-zero reward. While this is an issue that can be solved by using differently designed algorithms, a general issue of Deep Learning algorithms is the handling of numbers larger than 1 and smaller than -1. This is because neural networks expect their input data to have a mean of 0 and a variance of 1. Such issues can however be solved with rescaling or normalizing the data that is fed into the neural network. (Müller et al., 2016)
Both the rewards and observations returned by the job shop Gym environment are not in an ideal format for being passed into a neural network. Rewards are far below 0 every period, as production cost is always negative. Observations are represented as the number of jobs inside specific production steps, and thus are always 0 or higher. It is certainly possible to feed these "real" values directly into a neural network, but the learning performance will not be satisfactory. It is therefore necessary to rescale the data, as the environment's `step()` function only returns unnormalized rewards and observations. Returning only unnormalized and "real" data was a conscious design decision, as the developers of new algorithms should not be constrained in how the data returned by the Gym environment gets normalized. For that reason, any normalization needs to happen outside of the environment. Possible solutions are software wrappers around the environment or normalizing the data within the algorithm itself. A software wrapper takes the input to and output from the environment and adjusts the values according to a desired logic. It serves as an interface between the algorithm and the environment and translates between them. A normalization wrapper could for example change all rewards to be a number between 0 and -1, keeping their relative differences intact and only impacting the absolute values of the rewards. A practical example would be that when the reward per period is usually between 0 and -300, a normalization wrapper could divide all rewards by 300, resulting in a normalized reward between 0 and -1. The calculations need to be designed depending on the individual requirements of the used algorithms. Software wrappers have the advantage that existing algorithms don't need to be adapted to the structure of the job shop environment, because the wrapper serves as a translation interface.
The same normalization logic can also be applied directly inside an algorithm, which allows for more customization, but has the disadvantage that is has to be implemented for every algorithm that gets used.
This thesis compares multiple algorithms on their capacity planning performance in the same environment, but uses a mixed approach depending on the individual algorithm. Some algorithms that are used in this study were created by other researchers and come with a software wrapper already applied, whereas one algorithm had received some major changes by the author and thus normalizes its inputs on a deeper level inside the algorithm.

3.2 Decision making with agents

In the previous section we established an understanding of the simulation base model (the job shop simulation) and the Gym environment that wraps around it. This section discusses the ways that allow an external agent to access and control the job shop's production floor using the interfaces as defined by the environment.
It is important to know the difference between an agent and an algorithm. An agent refers to any abstract entity that controls the environment through a defined interface. This could be a shop floor manager making decisions in a real world production process, but also a virtual piece of software that runs certain actions, e.g. a program that creates production schedules or plans the capacity. In theory, the interfaces as imposed by Gym could also be used by a real person to control the production process if they were accessible with a user interface. For this study we will hover only use software agents that read and control the environment an run on a computer.
An agent can control the environment using its `step()` and `reset()` functions and gets returned rewards and observations. It can choose to make use of the returned values, but doesn't need to. An agent can implement a sequence of instructions which is called algorithm. Such an algorithm can use the observations and rewards given by the environment to come up with instructions on what to do based on the information. The agent serves as an interface for the algorithm to interact with the Gym environment, which in turn serves as an interface towards the shop floor production simulation. This process is depicted in Figure 3.3. It combines Figure 2.1 (which shows the structure of a Markov Decision Process or MDP) with the actual implementation in this thesis. The environment part of a common MDP consists of a Gym environment wrapping around the base simulation model and the agent part contains an agent that implements an arbitrary algorithm. It is possible, but not necessary to have the agent implement a Reinforcement Learning algorithm. In fact, it makes sense to have some rather basic agents that serve as a baselines which the Reinforcement Learning algorithms have to surpass in terms of maximizing the total sum of rewards over multiple evaluation episodes. Such a basic agent would not consider the rewards and states returned by the environment, but rather have its own way of coming up with actions to execute inside the environment. This could be fixed actions or random actions, allowing a researcher to ensure that a RL algorithm can beat pure chance or the status quo of the production process.
Additionally, it would be possible to implement an algorithm that uses other non-RL methods like mathematical programming. It should however be considered that the basic structures of a Gym environment are suited only towards the application of Reinforcement Learning. For implementing mathematical optimization methods or other non-RL algorithms it might be beneficial to develop a different wrapper for the job shop simulation that is more suitable for these methods.

A basic agent Using the basic structures of the Gym environment that we discovered so far, we can create a very basic agent as seen in Algorithm 1. It doesn't use Reinforcement Learning and instead has a fixed, predefined action which it executes every step.

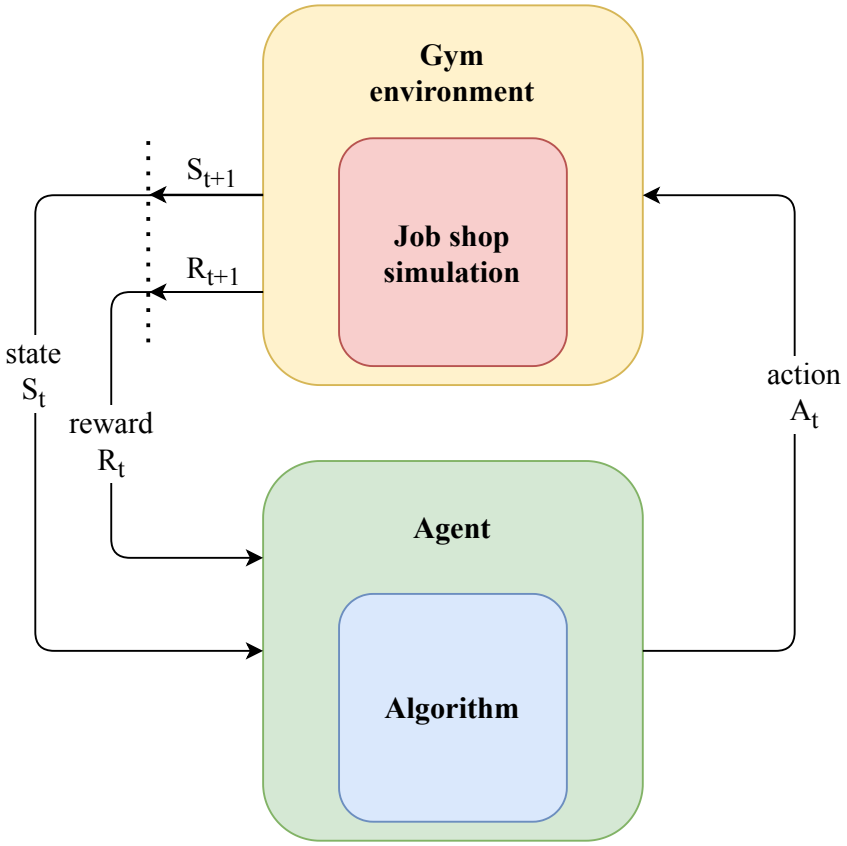


Figure 3.3: Implementation of a Markov Decision Process

For this agent, we assume that it keeps the production capacity at the default levels, that is at 100% of the normal capacity. This results in the agent always using action 0, which keeps the processing time multiplier at a constant 1.0. The agent runs the environment for one episode, that is 8000 periods. For this basic agent we do not consider the observations that are returned by the `step()` function, thus entirely ignoring the state of the environment.
The results of this agent can also be achieved by not using a Gym environment at all and just running the simulation as it is (since Gym doesn't contain any business logic and only serves as a structured interface). Any agent that considers observations and perhaps even follows the structures imposed by Gym will however be easier to apply to the job shop simulation. The algorithm for the basic agent doesn't have any output, but as its result we use the sum of rewards over all periods. As we simulate economic behaviour inside a production environment, the major metric to be considered is production cost. While other metrics such as lateness, machine utilization and others are important and will be considered in the evaluation of the agents, all algorithms will be compared on the main goal of improving the production cost.

Algorithm 1: A basic agent using a fixed action

```
Result: Sum of rewards
action ← 0;
sum of rewards ← 0;
observation ← environment.reset();
while done is false do
    observation, reward, done, info ← environment.step(action);
    sum of rewards ← sum of rewards + reward;
end
```

The sum of rewards returned by Algorithm 1 serves as the main benchmark throughout this study. By not using any overtime at all, the basic agent simulates the status quo of the production system. Other algorithms need to beat this benchmark, but are allowed to use overtime (which however incurs costs as well, so its use should be balanced carefully). This basic agent can also run with other pre-defined actions (action 1 or 2). That gives the possibility to run a quick comparison of a shop floor with no overtime against a shop floor that only uses overtime.
The outcome of an agent using only overtime can give the user some basic intuition on how well overtime would be suitable for the production process before any (computationally expensive) machine learning methods are employed. As a fixed usage of overtime that completely ignores the system state adds unnecessary production cost (overtime cost), it should be outperformed by a Reinforcement Learning agent that balances the individual production costs like overtime cost and lateness cost (otherwise there would be no point in using RL).

A basic agent using random actions Following the structure of the basic agent from Algorithm 1 we can write a basic agent that uses random actions. This allows to compare the results of purely random agent behaviour with a Reinforcement Learning agent's results. Therefore the action is recalculated in every step of the agent by obtaining a new random number with a uniform distribution in the interval $\mathcal{U}(0,2)$ as seen in Algorithm 2. The environment's state and reward are still ignored.

Algorithm 2: A basic agent using random actions

```
Result: Sum of rewards
sum of rewards ← 0;
observation ← environment.reset();
while done is false do
    action ← a random number in the interval U(0,2);
    observation, reward, done, info ← environment.step(action);
    sum of rewards ← sum of rewards + reward;
end
```

Algorithm 3: A custom heuristic that considers the load of the bottleneck work center

```
while condition for termination is not fulfilled do
    /* Calculate next action using load of work center 3 */
    load ←amount of jobs inside work center 3;
    if load < 4 then
        /* No overtime */
        action = 0;
        else if 4 ≤ load < 8 then
            /* Use overtime */
            action = 1;
        else
            /* Use more overtime */
            action = 2;
    observation, reward, done, info ← environment.step(action);
end
```

A custom heuristic that considers the load of the bottleneck work center Algorithm 3 shows a custom heuristic that decides on the usage of overtime depending on the amount of orders inside work center 3 (that is the bottleneck machine and its inventory). It uses two threshold values (4 and 8) to determine the use of overtime. If there are less than four jobs inside work center 3, there is no overtime. If there are more or equal than four, but less than eight jobs inside work center 3, there is an overtime of 12.5% (setup A) or 25% (setup B). For eight or more jobs there is an overtime of 25% (setup A) or 50% (setup B).
The custom heuristic has no background in literature and was developed by the author of this study.

The Reinforcement Learning agent The Reinforcement Learning agent is implemented in a way that allows using and comparing multiple algorithms on the job shop environment. The user can run the agent and give text commands to choose a specific algorithm for training or evaluating a Reinforcement Learning model.
Most of the algorithms that are used in this study come from the Python package Stable Baselines 3 (Raffin et al., 2019). This package contains some state of the art Reinforcement Learning algorithms derived from established research papers. It follows the structure of Gym environments, so the algorithms of the Stable Baselines 3 package can be used with the job shop environment without any major customization. The used algorithms are explained in Section 3.3.
The Reinforcement Learning agent has two major modes of operation, namely for training a model or for evaluating a model. The training mode builds a model based on the agent's interactions with the environment. As we know from Section 2.2.2, a model is (in simple terms) a collection of weights that indicate the "importance" or "influence" of an input feature at explaining the output (obtained through experience). This concept

is easier to understand when considering a simple example like the spam filtering from Section 2.2.2. It is quite intuitive that there are some input features that are better suitable for classifying spam than others (for example knowing the time when the e-Mail was received will probably not help much). The same goes for Reinforcement Learning, where the state and reward serve as a model's input features and the agent's next action is the output feature. However as a human, it is much harder to understand how an algorithm decides on a specific action just knowing the current environment state and potentially a reward. But just like when playing chess, where the positions of the chess pieces serve as the state, a human also arrives at some decision on which piece to move where. In both cases, neither the human chess player nor the Reinforcement Learning algorithm can give a clear and correct explanation of why that specific action was chosen. The action was rather conceived as a consequence of experience that was turned into a model (although humans and algorithms build their models quite differently).
Now that we have established how both humans and computers can build models based on experiences from interaction, we get a grasp of how the model training process of the Reinforcement Learning agent functions. Based on constant exploration, the model assigns weights to input features and builds certain "paths" (e.g. which action works best in which situation). Model performance usually increases with more training time (among other factors), just like for humans (learning something for one hour versus learning it for weeks). Once a predefined limit of training steps has been reached, the training phase gets stopped and the agent saves a model file on the computer's local file system.
This model file can be loaded and used by the evaluation mode to evaluate the agent's performance or even use it in a productive setting, e.g. for actually controlling machines. Algorithm 5 depicts how a trained model file gets used in evaluation mode of the Reinforcement Learning agent.
Note that the model depends on the chosen algorithm and the Reinforcement Learning agent supports multiple algorithms. Each algorithm creates its own model file and the models are not interchangeable between algorithms, as the algorithms have different strategies and structures when building their models.
The following section explains the algorithms that were implemented for being used by the RL agent.

3.3 Implemented algorithms

This thesis implements two Reinforcement Learning algorithms and compares them with each other as well as with the aforementioned "basic" agents. The algorithms are pretty different in how they learn from interactions with the environment, but they are both compatible with the structures imposed by Gym and are therefore easily interchangeable.

3.3.1 Average Reward Adjusted Discounted Reinforcement Learning

Average Reward Adjusted Discounted Reinforcement Learning or ARA-DiRL is based on the Q-Learning algorithm by Watkins (1989) and Sutton and Barto (2018) and

aims to improve the performance of Q-Learning on problems or environments that periodically return non-zero rewards (Schneckenreither, 2020). The base Q-Learning algorithm is well described in Sutton and Barto (2018) and will not be discussed in this thesis. The ARA-DiRL algorithm exists because traditional Q-Learning performs well for environments that return rewards with an average of zero and a variance of 1, but shows worse performance on problems that differ from this reward design. Such environments where Q-Learning performs well are for example games like chess. Economic problems like the here presented job shop that incurs periodic costs as its reward are inappropriate for the standard discounted reinforcement learning framework which is used by Q-Learning (see 2.2.3). ARA-DiRL adapts the standard Q-Learning algorithm and optimizes it for being used on economic problems that return non-zero rewards.
The ARA-DiRL algorithm implemented in this thesis is based on Schneckenreither (2020) (Algorithm 1 on page 13), but is modified so that it ignores the exponential smooting introduced in the original ARA-DiRL algorithm. Only the tabular case from Schneckenreither (2020) requires exponential smoothing. This thesis uses a neural network and so the exponential smoothing happens inside the neural network and not in the algorithm logic itself. Therefore the exponential smoothing learning rate γ and the average reward adjusted discounted state-value $X^{\pi}_{\gamma_1}(s_t, a)$ are completely omitted (see Algorithm 4.
This modified ARA-DiRL variant that is built using a neural network performs similarly to the original algorithm of Schneckenreither (2020) and will therefore be called ARA-DiRL during the evaluation of this study.
The implementation of the (modified) ARA-DiRL for this thesis is built as an extension module to the DQN algorithm of the Python package Stable Baselines 3 (Raffin et al., 2019). Raffin et al. (2019) is a reference implementation of the DQN algorithm of Mnih et al. (2013), which performed very well on various ATARI games using Deep Q-Learning (thus it's called Deep Q-Network or DQN). Based on this reference implementation of the DQN algorithm, this thesis adds the average reward discounting functionality of the modified ARA-DiRL (Algorithm 4) as an extension to the original DQN algorithm (Mnih et al., 2013). The source code of the implementation of Algorithm 4 can be obtained from the repository provided in the Appendix C.

3.3.2 Proximal Policy Optimization

The Proximal Policy Optimization or PPO algorithm of Schulman et al. (2017) comes from OpenAI[1], who are also the creators of Gym (which is used in this thesis). In contrast to ARA-DiRL which is based on the discounted reinforcement learning framework, the PPO algorithm uses policy gradient methods to learn from its interaction with the environment. Just like the ARA-DiRL implementation, the actual implementation of the PPO algorithm is based on the Python package Stable Baselines 3 (Raffin et al., 2019). The PPO implementation however is identical to the one from Stable Baselines 3 and is not changed at all. During a preliminary comparison of various baseline algorithms

[1] www.openai.com

from Stable Baselines 3, PPO showed the most promising results on the job shop environment and is therefore used as an adversary to ARA-DiRL. As opposed to ARA-DiRL, the PPO algorithm is not optimized specifically for economic problems. According to its developers, its focus lies on ease of implementation, low sample complexity and ease of tuning. This ease of use allows employing the PPO algorithm for this thesis without any further configuration of hyperparameters. Future research might try to find optimal hyperparameters for the job shop capacity planning problem to further increase performance of the PPO algorithm.

3.3.3 Training and evaluation of the algorithms

The training of the algorithms is done using the same environment configuration for all algorithms and using the same configuration for both training and evaluation. The job shop model with three work centers is trained for 1 million time steps (that is 1 million periods). The implementation of the base simulation allows an alternative mode of operation with just one single work center (everything else remains identical to the version with three work centers), but that mode is not used for the evaluation of this thesis. The single work center mode is recommended for trying out new algorithms, as the problem size is smaller and therefore an untuned algorithm can achieve better learning performance.
The ARA-DiRL algorithm has two different sets of hyperparameters for controlling the learning during the training phase, depending on the amount of work centers in the base simulation (one or three work centers). The used hyperparameters as well as trained model files are available in the code repository of Appendix C and can be used directly for evaluation purposes without having to train a model.
Figure A.4 shows the rewards per period during the 1 million training time steps of the ARA-DiRL algorithm. Note that the rewards are normalized inside the algorithm (see 3.1.3). The purple symbols indicate the single rewards (1 million rewards in total) and the green symbols indicate the average reward as defined by ρ in Schneckenreither (2020). It is clearly visible that during the beginning of the training phase the average reward decreases at first, since the algorithm has not yet derived a successful policy from its experiences. Over time it manages to learn successfully and receives an increasing average reward that remains stable once the learning phase has faded.
For evaluation, the RL agents initialize a pre-trained model file, receive the current environment state and then come up with a new action that the algorithm deems optimal for the current situation. Algorithm 5 shows an example of how this thesis implements the evaluation of RL agents using the Stable Baselines 3 Python package and syntax (Raffin et al., 2019).

4 Analysis

This chapter discusses the results obtained from the various agents and algorithms that are implemented and used on the base simulation to improve the balance of lateness and overtime costs in a job shop production system.

4.1 Experimental design

First, let's recapitulate the basics of how time gets tracked in the base simulation. The smallest time unit is a minute, and one work day consists of 16 hours (that is 960 minutes). One work day is also named period. When running experiments on the simulation, one data sample consists of 8000 simulated periods or work days. Each experiment consists of 30 data samples or $30 \times 8000 = 240000$ work days. These 30 data samples are then analyzed and validated using statistical tests (see Section 4.2).
There are four different experimental designs which are different in terms of their order release mechanism configuration and the usage of overtime.

Name	Planned release date slack (periods)	Action 0 overtime	Action 1 overtime	Action 2 overtime
BIL2-A	2	0 hours	2 hours	4 hours
BIL2-B	2	0 hours	4 hours	8 hours
BIL3-A	3	0 hours	2 hours	4 hours
BIL3-B	3	0 hours	4 hours	8 hours

Table 4.1: Experimental design

Table 4.1 shows the four different setups get compared in this chapter. These setups' names consist of two parts, one refers to the planned release date slack (BIL2 or BIL3) and the other indicates the possible extent of overtime (A or B). The planned release date slack refers to the amount of periods that a job gets released from the order pool before its due date (due date − planned release date slack = planned release date, also see Section 3.1.1). The overtimes of the three actions refer to the amount of extra hours per period that are incurred by the usage of overtime. The base amount of hours per period is always 16. Executing action 2 in the experimental setup BIL2-A would result in an additional four hours of overtime, that is 20 working hours in total.
The two different settings for the planned release date slack BIL2 and BIL3 should differ in their costs due the fact that a higher planned release date slack results in jobs getting

released to the shop floor earlier. Jobs that get released earlier incur more earliness costs inside the finished goods inventory, but have a higher chance of getting finished in time and thus reducing the lateness cost. Changes to the planned release date slack do not incur any costs.
The different overtime parameters A and B indicate a higher or lower usage of overtime. Setup A has overtime of two and four extra hours per period, which results in a 12.5% or 25% capacity increase, and setup B has four and eight extra hours of overtime that result in an extra 25% and 50% capacity. All overtime adds cost according to 3.1.1
Before comparing the performance of the different agents that are implemented, a baseline needs to be established. That baseline is the so called status-quo of the job shop simulation, that means the "current" situation in the manufacturing company in which there is high demand, high lateness costs and no overtime. This is done using the basic agent of Algorithm 1 where the action is set to 0. This means that the agent always runs the same action 0, meaning there is no overtime at all. Any new measure for the manufacturing company should improve the total costs in comparison to the status-quo, otherwise it is not seen as beneficial.
Table 4.2 displays the structure of the costs as described in Section 3.1.1 which make up the total cost of an experiment that gets used as the main metric for comparison.

Costs per job and period		Flat-rate costs per period	
WIP	1€	0 hours overtime	0€
Finished goods	4€	2 hours overtime	16€
Late goods	16€	4 hours overtime	32€
		8 hours overtime	64€

Table 4.2: Cost structure

4.2 Statistical validation

To verify the statistical significance of the results, the Wilcoxon signed-rank test is applied to the most relevant pairs of samples using a confidence level of 1% (Wilcoxon et al., 1970). The main metric considered by this thesis is the total cost incurred by each of the agents, and thus the Wilcoxon signed-rank test is used for these data points of each experiment.
The results tables indicate insignificant differences with a light grey cell background. Results that are not significantly different from each other are marked with the same color. If two groups of insignificant results are placed below each other but there is a significant difference between the groups then one group is colored with a darker grey.

4.3 Comparison of the agents

Now that the baseline which the agents have to beat is established, all agents and algorithms get compared on the costs which are incurred during the experiments. The tables that show the results are sorted by total costs.
The agents or algorithms that get compared on the four experimental designs are the following:

- No overtime (status-quo or baseline)
- Always 2h/4h overtime (setup A/B)
- Always 4h/8h overtime (setup A/B)
- Random overtime
- ARA-DiRL (Algorithm 4)
- PPO (Schulman et al., 2017)
- Custom heuristic (Algorithm 3)

4.3.1 BIL2-A and BIL2-B

Table 4.3 displays the costs and order amounts that are achieved by the various agents on the two BIL2 experimental setups (that is a planned release date slack of two periods) for both overtime levels, A and B. The results for the planned release date slack of two periods show that in comparison to the baseline with no overtime, the usage of overtime leads to strong improvements in terms of the total costs. Both overtime setups A and B show improvements, with B leading to slightly better results for the RL agents but worse results for the naive agents. Setup B makes use of more overtime than setup A and as long as there is considerable lateness cost, the usage of more overtime tends to be beneficial. This additional overtime needs to be paid for, and so it is only viable to use a lot of additional overtime if the system load justifies it.
The PPO algorithm shows the best performance among all evaluated measures and together with the custom heuristic is the only agent that can score total costs of well below 500,000€. Is is interesting to note that the PPO algorithm seems to learn much better from its interaction with the environment than the ARA-DiRL algorithm. On all experimental setups, PPO has considerably more late orders and lateness cost than ARA-DiRL, but still leads on overall performance. In terms of overtime cost, PPO performs best as it manages to use overtime to a lesser extent than the other agents. PPO's relationship of lateness and overtime cost indicates that it performs well at balancing individual costs to minimize total cost. PPO's performance is only matched by the custom heuristic and a Wilcoxon signed-rank test shows that PPO and the custom heuristic have no significant differences in terms of their total cost (with a confidence level of 1%). Interestingly, the custom heuristic outperforms the ARA-DiRL algorithm

(even though it doesn't learn and makes naive assumptions).
ARA-DiRL performs much closer to the non-RL agents which use random or fixed actions without any learning. For the less overtime setup A, ARA-DiRL gets even beaten by random actions, however it manages to perform better than chance in overtime setup B. For setup BIL2-A, ARA-DiRL performs identical to the agent that always runs four hours of overtime, however this is not the case for BIL2-B. With the lesser overtime setting of setup A, ARA-DiRL might have derived from its experience that a constant four hours of overtime might be the best course of action (it is not as the results of PPO show).
Looking at the total amount of shipped orders we do not see much difference between the different agents and a Wilcoxon signed-rank test shows that there is no significant difference between the agents in terms of total orders. Early and late orders are of course influenced by the usage of overtime, that means that applying overtime more often or for more hours per day increases the amount of early orders and reduces the amount of late orders. This becomes clear in Table 4.3, where for example the baseline with no overtime has very few early orders and many late orders, whereas the agent that runs a fixed overtime of 8 hours per day obviously achieves the most early orders and the least late orders. Both ways are of course not desirable, since they incur unnecessary costs which are not balanced and therefore are beaten by the "smart" approaches.
The most interesting agents to compare using the Wilcoxon signed-rank test are the custom heuristic, ARA-DiRL and PPO. When testing the total cost for significant differences, the following results are obtained.
PPO and the custom heuristic have no significant differences in terms of total cost, but both perform significantly better than ARA-DiRL (for both cost setups A and B). ARA-DiRL performs better than chance for BIL2-B, but for BIL2-A there is no significant difference between the random agent and ARA-DiRL. Only with a confidence level of 5%, the random agent scores significantly better than ARA-DiRL on BIL2-A.

Setup	Agent	Total Cost	Finished goods cost	Lateness cost	Overtime cost	WIP cost	Total orders	Early orders	Late orders
BIL2-A	PPO	465,912	96,288	263,239	66,919	39,464	64,954	46,051	18,839
		100%	21%	56%	14%	8%	100%	39%	61%
BIL2-A	Custom	469,842	93,357	274,572	61,416	40,497	65,051	45,413	19,569
		100%	20%	58%	13%	9%	100%	70%	30%
BIL2-A	Always 2h overtime	522,526	103,131	266,972	112,000	40,424	65,048	45,943	19,052
		100%	20%	51%	21%	8%	100%	71%	29%
BIL2-A	Random overtime	533,763	100,833	278,767	112,027	42,136	65,049	45,235	19,757
		100%	19%	52%	21%	8%	100%	70%	30%
BIL2-A	Always 4h overtime	535,287	133,200	151,308	224,000	26,780	65,055	54,220	10,787
		100%	25%	28%	42%	5%	100%	83%	17%
BIL2-A	ARA-DiRL	535,287	133,200	151,308	224,000	26,780	65,055	54,220	10,787
		100%	25%	28%	42%	5%	100%	83%	17%
BIL2-A	No overtime	699,791	50,868	550,179	0	98,744	64,932	25,593	39,292
		100%	7%	79%	0%	14%	100%	39%	61%
BIL2-B	Custom	429,904	107,953	202,209	86,223	33,519	65,032	50,580	14,389
		100%	25%	47%	20%	8%	100%	78%	22%
BIL2-B	PPO	434,825	107,200	204,203	89,629	33,791	65,086	50,430	14,590
		100%	25%	47%	21%	8%	100%	77%	22%
BIL2-B	ARA-DiRL	531,631	141,267	119,375	247,530	23,458	65,046	56,498	8,503
		100%	27%	22%	47%	4%	100%	87%	13%
BIL2-B	Always 4h overtime	535,287	133,200	151,308	224,000	26,780	65,055	54,220	10,787
		100%	25%	28%	42%	5%	100%	83%	17%
BIL2-B	Random overtime	554,510	127,301	173,545	224,475	29,187	65,078	52,666	12,363
		100%	23%	31%	40%	5%	100%	81%	19%
BIL2-B	No overtime	699,791	50,868	550,179	0	98,744	64,932	25,593	39,292
		100%	7%	79%	0%	14%	100%	39%	61%
BIL2-B	Always 8h overtime	706,538	160579	80150	448,000	17,809	64,992	59,246	5,712
		100%	23%	11%	63%	3%	100%	91%	9%

Table 4.3: Costs (in €) and order amounts for BIL2, sorted by total costs

4.3.2 BIL3-A and BIL3-B

Table 4.4 shows the results for the BIL3 setup, which has a planned release date slack of three periods. In comparison with the BIL2 setup, this leads to less lateness overall, since the jobs get released one period earlier from the order pool, resulting in less late orders. This reduction of lateness leads to different results and costs.
The baseline costs are a lot lower as the orders get released one period earlier. The Reinforcement Learning agents are however not able to lower their total costs in the same way as the costs get lower for the baseline with no overtime. While the baseline costs sink by 16% through the earlier order release, the PPO agent lowered its total cost by almost 11% (setup A) or 5.5% (setup B).
The custom heuristic shows mixed results for the change to three periods of planned release date slack. While for setup BIL3-A it performs almost 8% better than with BIL2-A, with setup BIL3-B it performs worse by having a total cost that is more than 5% higher than the cost of BIL2-B. This result seems unintuitive, as the overall less lateness in the BIL3 setup should mostly lead to lower costs. But when looking at the lateness cost, it becomes clear that the bad performance of the custom heuristic for BIL3-B comes from the fact that it has very little lateness cost and very high overtime cost. The custom heuristic appears to not be suitable for dynamic environments with changing parameters (e.g. dynamic order release mechanisms).
It is interesting to note that ARA-DiRL performed exactly identical on both BIL3 setups A and B. This leads to the conclusion that it has found strategies that result in exactly the same cost for both variants. On a closer look, it becomes clear that in setup BIL3-A the ARA-DiRL algorithm performs identical to the agent that always uses four hours of overtime and in setup BIL3-B it performs identical to the agent that also uses constantly four hours of overtime. It is very likely that for the BIL3 setting, ARA-DiRL has found a constant four hours of overtime to be optimal (which it is not, as PPO shows).
The Wilcoxon signed-rank test applied to the total costs achieved on the BIL3 setups shows the following results. Random overtime performs significantly better than ARA-DiRL on BIL3-A, but there is so significant difference on BIL3-B. PPO and the custom heuristic perform significantly better than ARA-DiRL on both setups. PPO scores significantly better than the custom heuristic on both setups for BIL3.

We conclude that the best overall performance is achieved by the PPO algorithm, since it adjusts well to changing parameters and for all experimental designs it is among the best scoring agents. On the BIL2 setup it performs evenly with the custom heuristic, but on BIL3 it beats all other agents by a significant margin.
It becomes clear that overtime is indeed a viable solution for solving the problem of high demand on the short term. Reinforcement Learning has shown its benefits when there are dynamic changes to an environment, e.g. from changing the order release mechanism.

Setup	Agent	Total Cost	Finished goods cost	Lateness cost	Overtime cost	WIP cost	Total orders	Early orders	Late orders
BIL3-A	PPO	417,765	264,294	66,369	37,874	49,228	65,021	60,255	4,745
		100%	63%	16%	9%	12%	100%	93%	7%
BIL3-A	Custom	433,331	295,714	35,005	61,722	40,890	65,139	62,637	2,489
		100%	68%	8%	14%	9%	100%	96%	4%
BIL3-A	Always 2h overtime	510,609	300,218	57,585	112,000	40,806	65,051	60,970	4,068
		100%	59%	11%	22%	8%	100%	94%	6%
BIL3-A	Random overtime	512,461	297,130	61,735	111,810	41,785	65,043	60,625	4,402
		100%	58%	12%	22%	8%	100%	93%	7%
BIL3-A	No overtime	587,826	173,770	312,342	0	101,715	65,008	42,820	22,163
		100%	30%	53%	0%	17%	100%	66%	34%
BIL3-A	Always 4h overtime	614,832	347,913	15,572	224,000	27,347	65,070	63,966	1,097
		100%	57%	3%	36%	4%	100%	98%	2%
BIL3-A	ARA-DiRL	614,832	347,913	15,572	224,000	27,347	65,070	63,966	1,097
		100%	57%	3%	36%	4%	100%	98%	2%
BIL3-B	PPO	411,234	259,577	66,896	34,450	50,312	65,106	60,334	4,748
		100%	63%	16%	8%	12%	100%	93%	7%
BIL3-B	Custom	454,502	322,280	12,436	86,266	33,521	65,031	64,134	891
		100%	71%	3%	19%	7%	100%	99%	1%
BIL3-B	No overtime	587,826	173,770	312,342	0	101,715	65,008	42,820	22,163
		100%	30%	53%	0%	17%	100%	66%	34%
BIL3-B	Random overtime	614,014	340,668	19,972	224,183	29,190	65,063	63,630	1,426
		100%	55%	3%	37%	5%	100%	98%	2%
BIL3-B	ARA-DiRL	614,832	347,913	15,572	224,000	27,347	65,070	63,966	1,097
		100%	57%	3%	36%	4%	100%	98%	2%
BIL3-B	Always 4h overtime	614,832	347,913	15,572	224,000	27,347	65,070	63,966	1,097
		100%	57%	3%	36%	4%	100%	98%	2%
BIL3-B	Always 8h overtime	853,620	384,561	3,261	448,000	17,798	64,999	64,766	232
		100%	45%	0%	52%	2%	100%	100%	0%

Table 4.4: Costs (in €) and order amounts for BIL3, sorted by total costs

4.3.3 Process metrics

Table 4.5 shows various metrics of the production process, namely service level, bottleneck utilization and mean flow time in minutes for each of the experimental setups.

Service level refers to the punctuality of shipments and is specified by the percentage of orders that are shipped in time. Overall, BIL3 achieves higher service levels than the BIL2 setups. This is obviously due to the fact that orders get released earlier and therefore finish earlier.
When looking at BIL2, the service level of the baseline with no overtime at all is very low at 39%, so only 39% of all orders are shipped in time. All other agents achieved far higher levels of punctuality, especially the ones that run more overtime. This is to be expected, since the baseline agent suffers from high lateness cost, which gets alleviated through the use of overtime. Out of all the agents that use overtime, PPO and the custom heuristic achieved the lowest service level even though they both have by far the best results in terms of total cost. This indicates that PPO learned quite well how to find a good balance between overtime cost and lateness cost and sometimes chose to let jobs run late instead of ordering higher overtime. The custom heuristic has shown previously that it performs well in an environment with high lateness (e.g. BIL2).
For BIL3, the advantage of PPO decreases, while the custom heuristic still performs strong with 96% and 99% punctuality. This punctuality is however paid for with higher costs in comparison to the PPO agent.

Bottleneck utilization is the utilization in percent of the bottleneck machine (in this case, that is machine 3). This metric is calculated by dividing the amount of hours that the bottleneck machine was actively processing jobs by the total amount of available hours. For example, if the machine ran for eight hours during a 16 hour period, its utilization is 50%.
The results in Table 4.5 show that the baseline agent has a very high utilization of the bottleneck machine (93% on all setups) in comparison with the agents that are allowed to run overtime. This shows that high service levels indicate a lower bottleneck utilization and vice versa. ARA-DiRL always has around 75% utilization, while PPO has between 84% and 90% utilization. The custom heuristic also performs similar for all setups. Only the fixed action agents that always run a certain amount of overtime have strongly varying bottleneck machine utilizations.

Mean flow time is measured in minutes and refers to the average time that a job is on the shop floor before it gets finished. This metric shows that in the status-quo, an average job takes roughly 2200 minutes to be processed (including wait times inside intermediate inventories). As the jobs have the same processing times for every experimental setup, this shows clearly that the high flow time must stem from wait times inside inventories. As using overtime improves the capacity of the bottleneck machine, we can conclude that using overtime reduces the mean flow time of jobs drastically. All agents that

use overtime have flow times which are less than half than those of the baseline agent. Interestingly, the ARA-DiRL algorithm constantly achieves low flow times in comparison to PPO and the custom heuristic, but cannot manage to beat them on total cost.

Overall, the rather simple custom heuristic has shown to perform very well even compared to modern Reinforcement Learning algorithms. ARA-DiRL has not managed to derive distinctive strategies except for the BIL2-B setup. On all other setups it just ran the same action constantly, resulting in identical performance to the agents with fixed actions. PPO has learned its own strategies which performed very well in comparison with all other measures.

Setup	Agent	Service level	Bottleneck utilization	Mean flow time (minutes)
BIL2-A	ARA-DiRL	83%	75%	973
BIL2-A	Always 4h overtime	83%	75%	973
BIL2-A	PPO	71%	86%	1,179
BIL2-A	Always 2h overtime	71%	83%	1,199
BIL2-A	Random overtime	70%	83%	1,223
BIL2-A	Custom	70%	87%	1,193
BIL2-A	No overtime	39%	93%	2,199
BIL2-B	Always 4h overtime	91%	62%	829
BIL2-B	ARA-DiRL	87%	73%	919
BIL2-B	Always 2h overtime	83%	75%	973
BIL2-B	Random overtime	81%	76%	1,011
BIL2-B	PPO	78%	84%	1,082
BIL2-B	Custom	78%	84%	1,077
BIL2-B	No overtime	39%	93%	2,199
BIL3-A	ARA-DiRL	98%	75%	980
BIL3-A	Always 8h overtime	98%	75%	980
BIL3-A	Custom	96%	87%	1,198
BIL3-A	Always 4h overtime	94%	83%	1,200
BIL3-A	Random overtime	93%	83%	1,220
BIL3-A	PPO	93%	89%	1,342
BIL3-A	No overtime	66%	93%	2,209
BIL3-B	Always 8h overtime	100%	62%	828
BIL3-B	Custom	99%	84%	1,076
BIL3-B	ARA-DiRL	98%	75%	980
BIL3-B	Always 4h overtime	98%	75%	980
BIL3-B	Random overtime	98%	76%	1,012
BIL3-B	PPO	93%	90%	1,358
BIL3-B	No overtime	66%	93%	2,209

Table 4.5: Process metrics sorted by service level

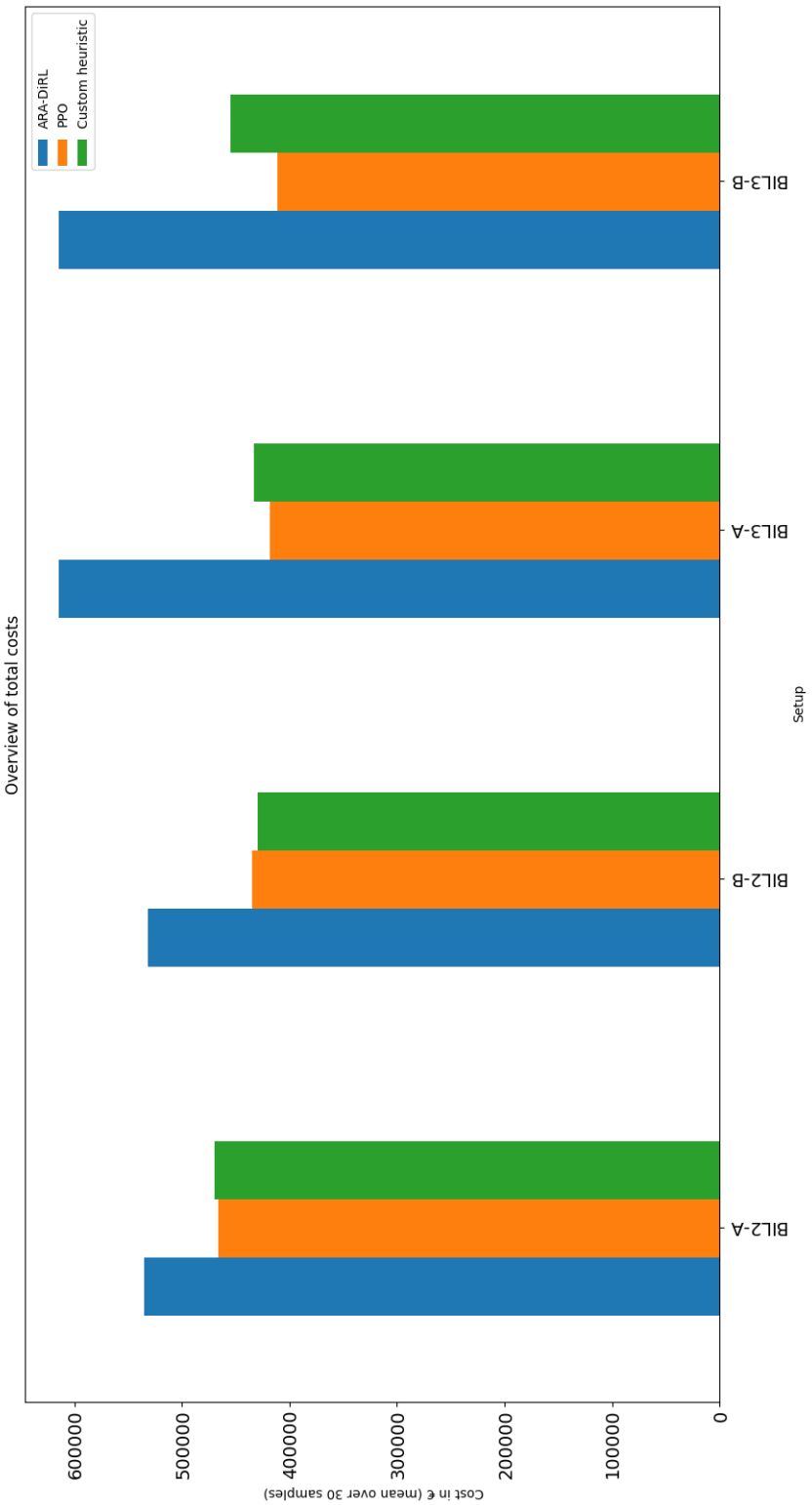


Figure 4.1: Overview of the total costs

5 Conclusions

5.1 Summary

In manufacturing, sudden surges in demand are often hard to deal with on the short term. While the use of overtime helps against having to decline customer orders, its cost needs to be balanced with other costs that occur in times of high demand (e.g. backorder cost due to late shipments). Research has found various heuristics that aim to find a good balance between overtime and backorder cost, however these heuristics often only work in a deterministic world or require the knowledge of experts to be developed and customized to each manufacturing environment.
This thesis implemented the simulation of a job shop manufacturing process with three work centers and a surrounding environment that allows Reinforcement Learning algorithms to learn from interaction with the production process and subsequently control the balance of lateness cost and overtime cost in the job shop.
The results show that Reinforcement Learning algorithms are able to perform well on the capacity planning problem and managed to reduce total production costs by 25% and up to 38% in comparison to a baseline without any overtime and also beat a random heuristic and a custom heuristic by a significant margin.

5.2 Limitations and future research

The capacity planning study conducted in this paper provided useful insights on how Reinforcement Learning can be applied to solve a multitude of problems in the operations research domain without any expert knowledge in operations research or without the need to build complex heuristics. There are however some limitations to the study, as it takes part in a new stream of research and has yet to be applied to common problems from the literature.
While two promising algorithms have been applied in this thesis, there are many more Reinforcement Learning algorithms that show potential for promising results on a production planning system. Additionally, even the currently implemented algorithms can be optimized in terms of their hyperparameters which influence the learning performance. Despite the large tuning efforts necessary especially for the ARA-DiRL algorithm, there is lots of potential for better results on the job shop capacity planning problem.
For this study, the design of the environment's state focuses only on the number of orders inside certain stages of production. Future research should focus on improving the design of the state, as it might be beneficial to include additional information on the production process such as remaining processing times of jobs inside machines or even predictions of

flow times using supervised learning. The design of the reward could be changed to a positive reinforcement with the implementation of profit. By allowing shipped orders to generate revenue, a reward over zero could be realized for most periods.
Most research that examined the balancing of overtime costs and lateness costs focused on developing heuristics that use mathematical programming approaches. Such approaches function entirely different from the here presented Reinforcement Learning framework, but nonetheless it is necessary to compare the RL approach with established heuristics. New research could add to the Gym environment of this study and use the interface provided by Gym to apply established heuristics instead of Reinforcement Learning to the capacity planning problem. While the here presented custom heuristic goes into a good direction with its performance being almost as good as the PPO Reinforcement Learning algorithm, it is a very simple one without any background in the literature.
Heuristics that get applied to the job shop simulation should take into account that while the demand process and the processing times of machines follow an exponential distribution, due to the order release mechanism and a fixed due date slack the demand is rather deterministic. A production process with more variability could introduce a dynamic due date slack or machine failures. To obtain a truly dynamic job shop spontaneous order arrivals could be added.
The business logic of the job shop system itself could be adapted in future research so that other configurations of various parameters can be examined. This can include the planned release date slack of the BIL order release mechanism, the cost structures, the scheduling heuristic and other settings.
Finally, the here presented ARA-DiRL and PPO algorithms should be evaluated on other problem settings from the operations research (OR) domain. Many problem settings in the OR field have some kind of economic impact and would therefore be viable environments to evaluate the performance of RL in real-world problems. Such problem settings could be scheduling, setting parameters in some kind of computer system, planning inventories, managing networks and so on.

5.3 Challenges, success factors and learnings

Based on the experience from this study, the following challenges, success factors and learnings were identified. The design of the reward and state are crucial for Reinforcement Learning success and also its biggest challenge. A researcher should determine carefully which information the agent will receive, as this can make or break the performance of the RL agent. Returning too much unnecessary information from the environment can decrease the learning performance, but also giving too little or useless information is not beneficial. when designing an environment, one can not always estimate which information will be useful for the agent. It is therefore recommended to constantly evaluate changes to the state design with training sessions of a RL algorithm. This allows the researcher to see if an algorithm can still learn from a newly designed state.
With that in mind, it is advisable to start with small, less complex state designs and run

RL algorithms on these. Only once an established RL algorithm learns well on a small problem should the problem complexity be increased.
Returning non-zero rewards periodically provides an additional hindrance which can be overcome by normalizing data that is returned to the agent, choosing suitable algorithms or customizing the environment or the algorithm.
Hyperparameter tuning can be very important depending on the algorithm. Modern algorithms like PPO require very little tuning, whereas older algorithms like Q-Learning (or its derivation ARA-DiRL) need lots of trial and error. While there are software packages that automate the finding process of optimal hyperparameters, this can be impossible for more complex problems like the here presented job shop with three machines. With training times ranging from two to four hours of real time for just one training run and potentially tens of thousands of combinations of hyperparameters to tune, it becomes clear that it is not viable to find the parameters automatically.
This leads to the last challenge of Reinforcement Learning for real-world problems, namely sample complexity. Training takes hours of real time and can often not be parallelized. This makes trial and error for designing the environment or for tuning hyperparameters very expensive for complex models. Such high training costs can ruin the benefit of RL not requiring expert knowledge on a problem, because it might at some point be cheaper to actually gather that expert knowledge instead of further optimizing the state or some algorithms.

5.4 The bottom line

Modern Reinforcement Learning algorithms are very versatile and perform well without any customization. The vast ecosystem for RL software allows researchers to quickly come up with solutions for established problems and in general a high speed of development and ease of implementation. In this thesis, the Reinforcement Learning method has been proven to be applicable to real-world economic problems, which opens up exciting new possibilities for automated decision making in problem settings for many domains. In the future, more and more systems will use Reinforcement Learning instead of traditional heuristics for making decisions. Due to the domain-agnostic nature of RL algorithms, there is the potential for a few powerful algorithms to be used on most problems around the world without any customizing to the individual problem setting. Companies that provide such algorithms will have the potential for immense profits when allowing the world to use these algorithms for solving all kinds of problems. Just like supervised and unsupervised learning, RL benefis immensely from the relatively cheap computing power that is available since the last decade and will soon be applied in more and more real world scenarios.

Bibliography

Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2014.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

JW M Bertrand and HPG Van Ooijen. Workload based order release and productivity: a missing link. *Production Planning & Control*, 13(7):665–678, 2002.

Matthew Botvinick, Sam Ritter, Jane X Wang, Zeb Kurth-Nelson, Charles Blundell, and Demis Hassabis. Reinforcement learning, fast and slow. *Trends in cognitive sciences*, 23(5):408–422, 2019.

James R Bradley and Bruce C Arntzen. The simultaneous planning of production, capacity, and inventory in seasonal demand environments. *Operations Research*, 47(6): 795–806, 1999.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

FTS Chan, P Humphreys, and TH Lu. Order release mechanisms in supply chain management: a simulation approach. *International journal of physical distribution & logistics management*, 2001.

Satyaveer S Chauhan, Rakesh Nagi, and Jean-Marie Proth. Strategic capacity planning in supply chain design for a new market opportunity. *International journal of production research*, 42(11):2197–2206, 2004.

Chin-Sheng Chen, Siddharth Mestry, Purushothaman Damodaran, and Chao Wang. The capacity planning problem in make-to-order enterprises. *Mathematical and computer modelling*, 50(9-10):1461–1473, 2009.

Chia-Shin Chung, James Flynn, and Ömer Kirca. A multi-item newsvendor problem with preseason production and capacitated reactive production. *European Journal of Operational Research*, 188(3):775–792, 2008.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

Gary D Eppen, R Kipp Martin, and Linus Schrage. Or practice—a scenario approach to capacity planning. *Operations research*, 37(4):517–527, 1989.

Federal Reserve Bank. Industrial Production and Capacity Utilization - G.17, July 15 2020 , 2020. URL https://www.federalreserve.gov/releases/g17/20200715/g17.pdf. Last accessed 10 September 2020.

David Foster. *Generative deep learning: teaching machines to paint, write, compose, and play.* O'Reilly Media, 2019.

Lawrence D Fredendall, Divesh Ojha, and J Wayne Patterson. Concerning the theory of workload control. *European Journal of Operational Research*, 201(1):99–111, 2010.

Alan Garnham. *Artificial intelligence: An introduction.* Routledge, 2017.

Richard Gross. *Psychology: The science of mind and behaviour 7th edition.* Hodder Education, 2015.

Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3389–3396. IEEE, 2017.

Stefan Haeussler, Manuel Schneckenreither, and Christoph Gerhold. Adaptive order release planning with dynamic lead times. *IFAC-PapersOnLine*, 52(13):1890–1895, 2019.

RD Jack Hammesfahr, James A Pope, and Alireza Ardalan. Strategic planning for production capacity. *International Journal of Operations & Production Management*, 1993.

Jyh-Wen Ho and Chih-Chiang Fang. Production capacity planning for multiple products under uncertain demand conditions. *International Journal of Production Economics*, 141(2):593–604, 2013.

Fernando Jaramillo and Murat Erkoc. Minimizing total weighted tardiness and overtime costs for single machine preemptive scheduling. *Computers & Industrial Engineering*, 107:109–119, 2017.

Indu John, Aiswarya Sreekantan, and Shalabh Bhatnagar. Efficient adaptive resource provisioning for cloud applications using reinforcement learning. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 271–272. IEEE, 2019.

Moutaz Khouja. The single-period (news-vendor) problem: literature review and suggestions for future research. *Omega*, 27(5):537–553, 1999.

Brian Kingsman and Linda Hendry. The relative contributions of input and output controls on the performance of a workload control system in make-to-order companies. *Production planning & control*, 13(7):579–590, 2002.

Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013.

Martin J Land, Gerard Gaalman, et al. Workload control concepts in job shops a critical assessment. *International journal of production economics*, 46(47):535–548, 1996.

Yuanlong Li, Yonggang Wen, Dacheng Tao, and Kyle Guan. Transforming cooling optimization for green data center via deep reinforcement learning. *IEEE transactions on cybernetics*, 50(5):2002–2013, 2019.

Steven A Melnyk, Gary L Ragatz, and Lawrence Fredendall. Load smoothing by the planning and order review/release systems: a simulation experiment. *Journal of Operations Management*, 10(4):512–523, 1991.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Andreas C Müller, Sarah Guido, et al. *Introduction to machine learning with Python: a guide for data scientists*. " O'Reilly Media, Inc.", 2016.

OpenAI. Gym documentation, 2020. URL https://gym.openai.com/docs/. Last accessed 09 October 2020.

AM Ornek and O Cengiz. Capacitated lot sizing with alternative routings and overtime decisions. *International Journal of Production Research*, 44(24):5363–5389, 2006.

Nicholas C Petruzzi and Maqbool Dada. Pricing and the newsvendor problem: A review with extensions. *Operations research*, 47(2):183–194, 1999.

Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. https://github.com/DLR-RM/stable-baselines3, 2019.

Gary L Ragatz and Vincent A Mabert. An evaluation of order release mechanisms in a job-shop environment. *Decision Sciences*, 19(1):167–189, 1988.

Manuel Schneckenreither. Average reward adjusted discounted reinforcement learning: Near-blackwell-optimal policies for real-world applications. *arXiv preprint arXiv:2004.00857*, 2020.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

Mark Stevenson, Linda C Hendry, and Brian G Kingsman. A review of production planning and control: the applicability of key concepts to the make-to-order industry. *International journal of production research*, 43(5):869–898, 2005.

Penghao Sun, Zehua Guo, Sen Liu, Julong Lan, Junchao Wang, and Yuxiang Hu. Smartfct: Improving power-efficiency for data center networks with deep reinforcement learning. *Computer Networks*, page 107255, 2020.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Matthias Thürer, Mark Stevenson, Cristovao Silva, Martin J Land, and Lawrence D Fredendall. Workload control and order release: A lean solution for make-to-order companies. *Production and Operations Management*, 21(5):939–953, 2012.

Matthias Thürer, Mark Stevenson, Cristovao Silva, Martin J Land, Lawrence D Fredendall, and Steven A Melnyk. Lean control for make-to-order companies: Integrating customer enquiry management and order release. *Production and Operations Management*, 23 (3):463–476, 2014.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575 (7782):350–354, 2019.

Christopher John Cornish Hellaby Watkins. Learning from delayed rewards, 1989.

Marco Wiering and Martijn Van Otterlo. *Reinforcement learning*, volume 12. Springer, 2012.

Frank Wilcoxon, SK Katti, and Roberta A Wilcox. Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test. *Selected tables in mathematical statistics*, 1:171–259, 1970.

Bibo Yang, Joseph Geunes, and William J O'Brien. A heuristic approach for minimizing weighted tardiness and overtime costs in single resource scheduling. *Computers & Operations Research*, 31(8):1273–1301, 2004.

Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

Rong Yuan and Stephen C Graves. Setting optimal production lot sizes and planned lead times in a job shop. *International Journal of Production Research*, 54(20):6105–6120, 2016.

Günther Zäpfel and Hubert Missbauer. New concepts for production planning and control. *European Journal of Operational Research*, 67(3):297–320, 1993.

Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432, 2019.

Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3357–3364. IEEE, 2017.

A Additional figures

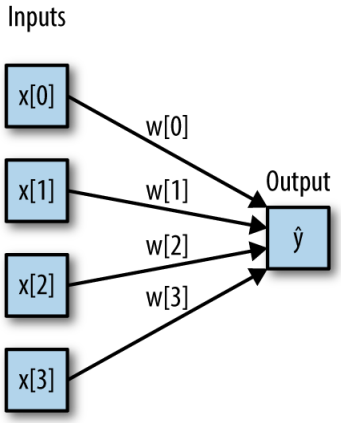


Figure A.1: Visualisation of logistic regression (Müller et al., 2016)

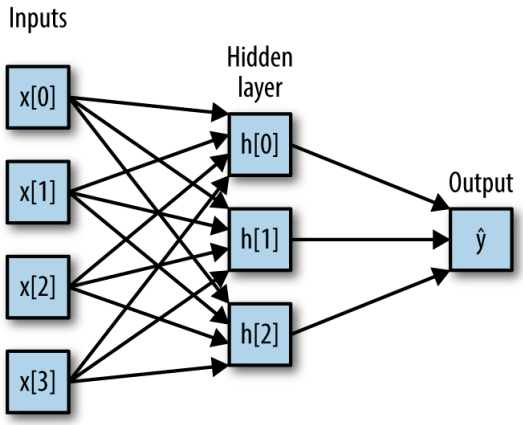


Figure A.2: A neural network with one hidden layer (Müller et al., 2016)

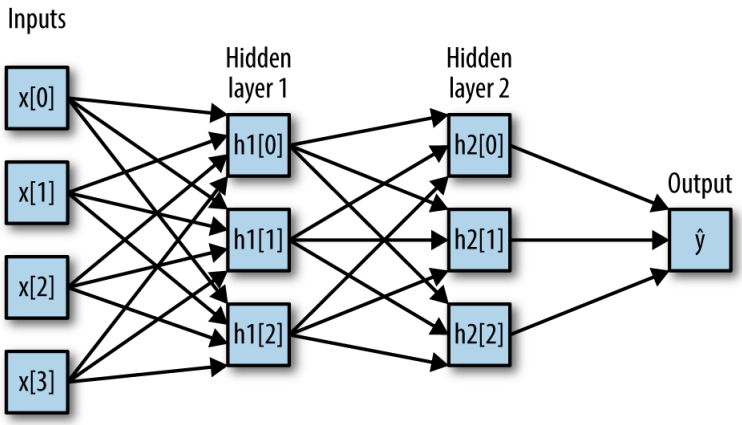


Figure A.3: A neural network with two hidden layers (Müller et al., 2016)

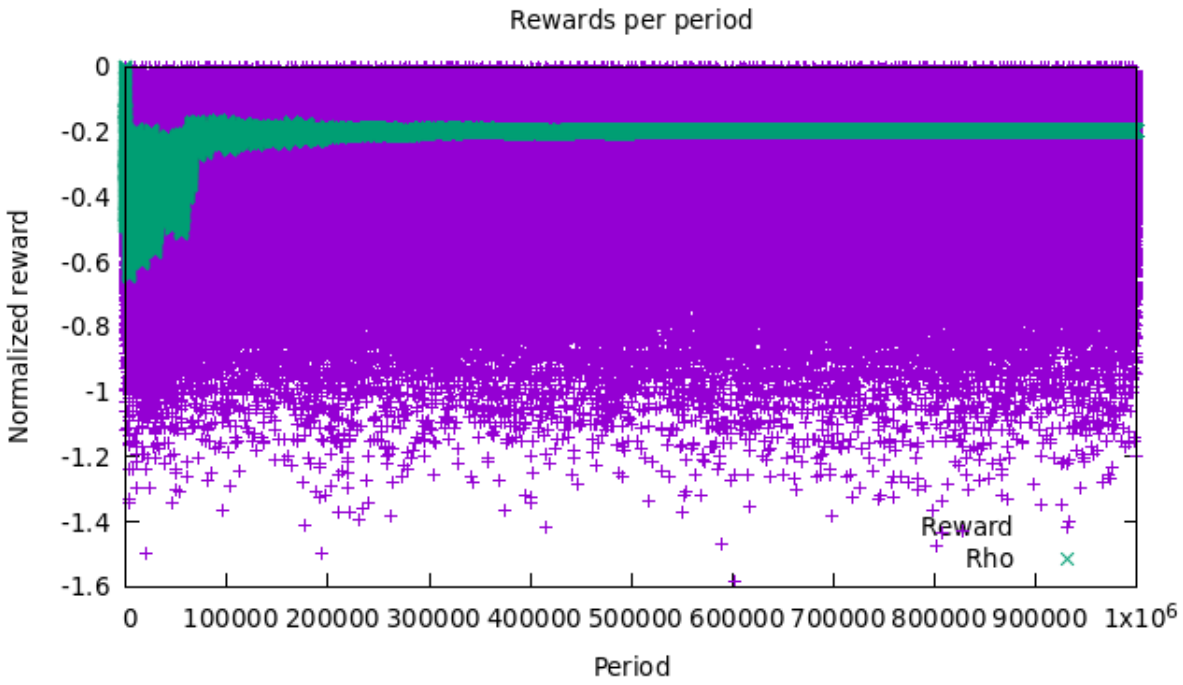


Figure A.4: Normalized rewards per period during the training procedure

Data point	Number of orders...	Product type	Min	Max
1	in order pool due in 1 period	1	0	15
2	in order pool due in 2 periods	1	0	15
3	in order pool due in 3 periods	1	0	15
4	in order pool due in 4 periods	1	0	15
5	in order pool due in 5 periods	1	0	15
6	in order pool due in 6 periods	1	0	15
7	in order pool due in 7 periods	1	0	15
8	in order pool due in 8 periods	1	0	15
9	in order pool due in 9 periods	1	0	15
10	in order pool due in 10 periods	1	0	15
11	in work center 1	1	0	15
12	in work center 2	1	0	15
13	in work center 3	1	0	15
14	in FGI early by 1 period	1	0	30
15	in FGI early by 2 periods	1	0	15
16	in FGI early by 3 periods	1	0	15
17	in FGI early by 4 or more periods	1	0	15
18	shipped in time	1	0	15
19	shipped late by 1 period	1	0	15
20	shipped late by 2 periods	1	0	15
21	shipped late by 3 periods	1	0	15
22	shipped late by 4 or more periods	1	0	15
23	in order pool due in 1 period	2	0	15
24	in order pool due in 2 periods	2	0	15
...	...	...	...	...
45	in order pool due in 1 period	3	0	15
...	...	...	...	...
67	in order pool due in 1 period	4	0	15
...	...	...	...	...
132	shipped late by 4 or more periods	6	0	15

Table A.1: Tabular overview of the observation space

B Additional algorithms

Algorithm 4: Modified ARA-DiRL algorithm based on Schneckenreither (2020)

Initialise state $s_0, \rho^\pi = 0$, set an exploration rate $0 \leqslant p_{exp} \leqslant 1$, exponential smoothing learning rate $0 < \alpha < 1$, and discount factor $0.5 < \gamma_1 \leqslant 1$, where $\gamma_1 = 1$ is usually a good choice.

while *the stopping criterion is not fulfilled* **do**

With probability p_{exp} choose a random action and probability $1 - p_{exp}$ one that fulfills
$\max_a \preccurlyeq_\epsilon X^\pi_{\gamma_1}(s_t, a)$.

Carry out action a_t, observe reward r_t and resulting state s_{t+1}.

if *a non-random action was chosen* **then**

$$\rho^\pi \leftarrow (1-\alpha)\rho^\pi + \alpha[r_t + \max_a X^\pi_{\gamma_1}(s_{t+1}, a) - X^\pi_{\gamma_1}(s_t, a_t)]$$

Update the average reward adjusted discounted state-values.

$$X^\pi_{\gamma_1}(s_t, a_t) \leftarrow r_t + \gamma_1 \max_a X^\pi_{\gamma_1}(s_{t+1}, a) - \rho^\pi$$

Set $s \leftarrow s'$, $t \leftarrow t + 1$ and decay parameters

Algorithm 5: A Reinforcement Learning agent for evaluating a model following the syntax of Stable Baselines 3

```
Result: Sum of rewards
model ←model from the desired algorithm;
sum of rewards ← 0;
/* Get new observation from environment's reset() method */
observation ← environment.reset();
while done is false do
    /* Predict next action using current observation */
    action ← model.predict(observation);
    /* Done gets returned by the environment once 8000 periods have been
       reached */
    observation, reward, done, info ← environment.step(action);
    sum of rewards ← sum of rewards + reward;
end
```

C Source code and results

The source code as well as the results for this thesis can be found at the following repository on the internet. https://github.com/sebwindm/masterarbeit

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Ich erkläre mich mit der Archivierung der vorliegenden Masterarbeit einverstanden.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift