UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

HAW HAMBURG

TFG del Grado en Ingeniería Informática

Annex B: Theoretical Backgrounds and Experimentation

Presented by Helen Haase
at Universidad de Burgos
January 17, 2022
Tutor: Bruno Baruque Zanón

Contents

List of Figures

List of Tables

Chapter 1

List of Acronyms

A2C Advantage Actor Critic

A3C Asynchronous Advantage Actor Critic

AGV Automated Guided Vehicle

DRL Deep Reinforcement Learning

DQN Deep Q-Network

EDD Earliest Due Date

ERM Experience Replay Buffer

FIFO First In First Out

FMS Flexible Manufacturing System

FJSP Flexible Job Shop Problem

IID independent and identically distributed

JSP Job Shop Problem

KPI Key Performance Indicator

KBGA Knowledge Based Genetic Algorithm

MDP Markov Decision Process

ML Maximal Lateness

MP Matrix Production

NN Neuronal Network

NTJ Number of Tardy Jobs

ODBC Open Database Connectivity

OPC UA Open Platform Communications Unified Architecture

PPO Proximal Policy Optimization

RL Reinforcement Learning

SGA Simple Genetic Algorithm

SRPT Shortest Remaining Process Time

SPS Siemens Plant Simulation

SPT Shortest Processing Time

TD Temporal Difference

TCT Total Completion Time

TL Total Lateness

TRPO Trust Region Policy Optimization

TT Total Tardiness

UU Uptime Utilization

WIP Work in Progress

XML Extensible Markup Language

Chapter 2

Introduction

This appendix serves the purpose of showing the theoretical concepts, that are related to the project. That is why first of all the weaknesses of a typical line production are displayed, which then motivated the design of a matrix production or Flexible Manufacturing System (FMS). Then the most relevant vocabulary is defined and the general concepts of this production system is explained. After showing current challenges of this production system the chosen problem for this project is defined.

In order to find possible solutions the current research is examined and required concepts of Reinforcement Learning are explained.

Lastly the performed experiments of the prototype, that is covered in Annex A, are evaluated and the experimental results are presented.

Chapter 3

Theoretical Concepts

3.1 Flexible Modular Production Systems

3.1.1 Line Production and the Demands on Modern Production Systems

The basic idea of line production is a conveyor belt connecting the different production steps. Every step should have a uniform cycle time meaning all tasks are synchronized. Upon the termination of a production step, the product can be transported via conveyor belt to the next processing step unnecessary waiting times. While this setup works well for production processes with a constant portfolio, it does not work well when this is not the case.

The temporal transition between the relation of volume and complexity of products, shows that the demands on these production systems are different. Figure 3.1 shows that in 1913, when the line production using a conveyor belt was introduced, the goal was to build as many identical products as possible. Nowadays there is a need to manufacture customized product in a broad portfolio [8]. In this context the complexity of the product increases [7].

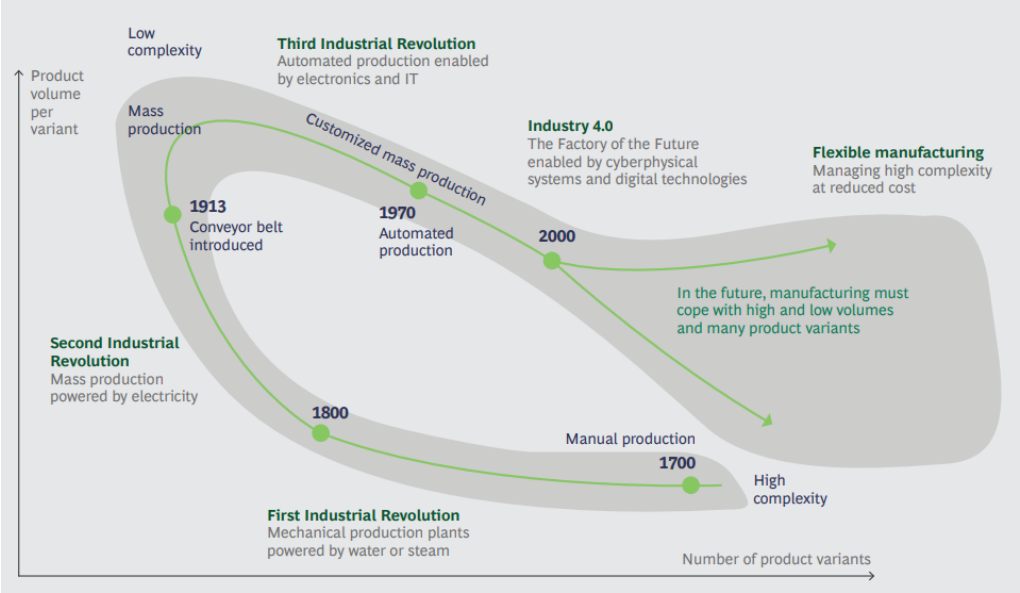

Figure 3.1: Relation between product complexity and volume over time; taken from Küppel et al.[18]

However there are further framework conditions that cause this development. These are shorter market cycles, the increase in planning due to supply chain and the globalization changes, region-specific demographic change as well as legal and social requirements [7]. Hence a production system should show a high flexibility with regards to a changing market (market flexibility), the range of products (production flexibility), its scale (volume flexibility), its possibility to adapt or renew products (product flexility) as well as its ability to rearrange and reconfigure the system (process flexibility) [8].

A regular line production might not be able to answer all of these demands. Its biggest disadvantage is that the system is easily affected in case of breakdown or a damage of the conveyor belt, as these incidents result in a partial of complete failure of the conveyor belt happens.

Consequently a new manufacturing process appears necessary.

3.1.2 Matrix Production

The main difference between line production and matrix production or FMS is the elimination of the equal cycle time. As the main goal is to maximize

the workload of the entire system, a process should be completed within one cycle. The process needs to be synchronous, as being faster leads to waiting times, while being slower leads to disruption of the following cycles. In an optimal scenario processing time for one production step equals cycle time. As this is nearly impossible to realize an individual cycle time is used for each subsystem or each work center, instead of a uniform one for the entire system [8, 7].

In order to gain a better understanding of the system in building a well-defined vocabulary, the following section will introduce the key components of a matrix production.

3.1.2.1 Key Components of a Matrix Production or FMS

Matrix production and FMS are used interchangeably, as the developed prototype system is simplified as to fall under both categories and the additional demands for Matrix Production (MP) in comparison to FMS can be neglected.

The production system consists of a variety of working stations, that are connected by Automated Guided Vehicles (AGVs) to transport the product and a cart, which includes the required materials or tools. Each working station has a buffer area, a worker, tools and stationary material. As simplified approach only includes one processing machine, a working station will be generally referred to as a machine. In this setup they also do not include buffers and AGVs. Further constraints of the simulation are defined in chapter 4.5 of the Annex A.
Working packages are processed at the working stations.These include a number of tasks, that are performed in the individual cycle time. The smallest dividable tasks are considered as processing steps. In this work, working packages will be referred to as processing steps, as a working package always only includes one processing step. Processing time describes the time required to complete a processing step. A part is used to describe the product that is still in the production process.
The wording is oriented on the work of Greschke [7].

3.1.2.2 Current Challenges

As Matrix production is still in its infancy there are still some challenges to overcome. These challenges among others include the allocation of work centers (layout planning), product design, production control, production planning, route planning and material flow as well as optimized scheduling. It is important to consider that these aspects often intersect each other, so that certain design choices influence others.

3.1.3 The Flexible Job Shop Problem

As scheduling directly impact the efficiency of a production system and its costs, this work follows the Flexible Manufacturing System (FMS) approach. The Job Shop Problem (JSP) describes an optimization problem, in which various manufacturing jobs are assigned to a time slot of an already chosen machine while trying to minimize the makespan. The Flexible Job Shop Problem (FJSP) extends the JSP insofar as not only one machine can be selected but some or all offering the required operation [42] of a processing step. This way the choice of the machine is added to the problem.

The FJSP is strongly NP-hard, meaning that it can not be solved within a deterministic polynomial time. This implies, that common exact solving methods can not be used [6]. The research community's strategies use hyper-heuristic and meta-heuristic, including genetic algorithms or reinforcement learning. Therefore the prototype developed as part of this project is used to solve a FJSP applying reinforcement learning.
The required concepts will be explained in the following chapter.

3.2 Reinforcement Learning

3.2.1 Introduction

Machine Learning defines a number of approaches to improve the performance of a specific task by learning from experience. Reinforcement Learning (RL) is part of this context, as it learns from interacting with an environment. Nowadays it finds application in many areas, including autonomous driving, stock trading, computer games and robotic as well as industry automation [29].

RL is used in autonomous driving, in tasks such as controlling path planning, trajectory optimization or the development of high-level driving policies for complex navigation tasks [14]. In computer games artificial intelligence as well as more recently RL has found application, *games often provide interesting and complex problems for agents to solve* [33]. RL is used for strategic decision making, micromanagement training bots as enemy or ally of the player.

Reinforcement Learning is neither a supervised nor an unsupervised algorithm but rather intersects with both. While supervised learners know the correct solution and therefore receive direct feedback through their chosen answer. Unsupervised learners do not get any feedback. Reinforcement Learning is settled on a middle ground, receiving a reward or a punishment depending on the state of the environment after performing an action instead of having a supervisor that knows what is right or wrong and therefore is a clear feedback. Another substantial difference is, that the order of the chosen actions has a relevance of the sequential. Reinforcement algorithms also perform actions to change an environment and thus modify data and include a state, representing the current state of the environment [34].

The following sections will show the formulation of problems as a Markov Decision Process, as well as the interaction of an agent with its environment. The learning process of the agent will be explained, the influence of the reward system explored and different forms of RL algorithms presented. This chapter is largely based on the book by S. Sutton and G. Barto [38], the lecture series of David Silver [34] and the Spinning Up OpenAI resource [1].

3.2.2 Markov Decision Process

RL is a framework, used to solve problems, that can be expressed in a Markov Decision Process (MDP). In order to apply RL its problem first has to be as a MDP.

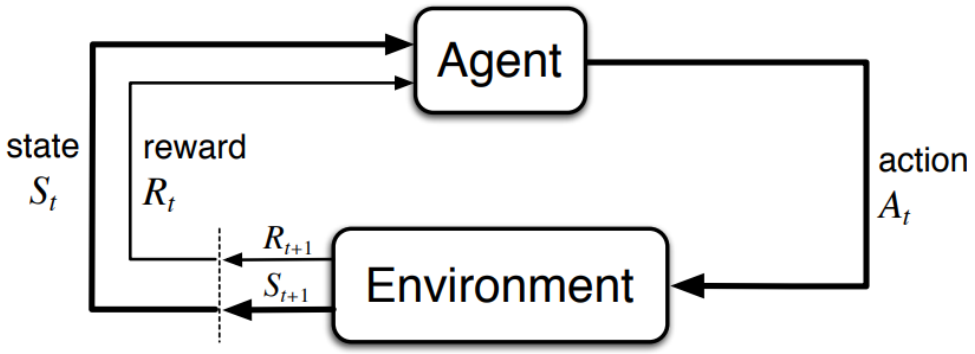

Figure 3.2: The agent-environment interaction in a Markov decision process; taken from Sutton and Barto [38])

A MDP generally consists of a tuple ($\mathcal{S} = statespace, \mathcal{A} = actionspace, \mathcal{R} = reward, \mathcal{P} = probability, \gamma = discountfactor$). Each tuple component will be explained throughout this section.
The process itself is shown in figure 3.2. The agent is the component, making decisions and learning from them. It tries to find optimal behaviour when interacting with an environment.
MDPs are used in situations, where decision making can influence the environment but the outcome also depends to some point on random.

The agent may be for instance be a human and the environment an environment maze in the dark. Obviously the human tries to exit the maze. At any time the he can try moving left, right, straight or backwards. Let us pretend, that the human chooses to take a step to the left in order to try and leave the maze. Now three different situations may occur. Either there is a wall to left, so running into it hurts the human physically (punishment) or the position is free and the human can move (reward) or the position is free and is the exit to the maze (higher reward). Either way the human gets feedback, the so called reward for its action. Due to the change of position the current state of the environment changes. The human chooses another action, receives a reward and a new state. This process is continues, until the human finds a way out of the maze. Afterwards the human will be reset into the maze and the entire process will be repeated, until the human has learned how to escape this maze at an acceptable speed.

This sequence of steps is called a trajectory and starts as follows

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2... \tag{3.1}$$

where S is a state from all possible states $\mathcal{S}$ (compare the MDP tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$), so here each position in the maze could be depicted as one state of the environment. The states in a MDP have to fulfill the Markov property, which defines, that only knowing the current state leads to the same decisions as much as knowing all past states

$$Prob[S_{t+1}|S_t] = Prob[S_{t+1}|S_t, S_{t+1}..] \tag{3.2}$$

This means, that the current state includes all necessary information and that the process has no memory.
a is an action of the action space $\mathcal{A}$, that in the example would include $\{left, right, straight, backwards\}$. R is the reward, which is a numerical value, lying in $\mathcal{R}$.
In this work the author will concentrate on MDPs that only have a finite state and action space, as the RL agent trained only consists of these kinds of spaces.

$\mathcal{P}$ represents the state transition probability matrix. This matrix contains a probability for reaching the next state when performing a chosen action in the current state ($\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$). In the context of the maze example, you may assume, that the human has a left-right weakness, so in every state when the human chooses to turn left, chances are 70% that he turns left and 30% of the cases that he turns right. Depending on the algorithm not every agent needs to know $\mathcal{P}$, as it could also learn this from experience.

3.2.2.1 The Reward System

Now that the general interaction of agent and environment has been shown, the question arises on how the agent exactly chooses an action. Starting point is understanding the motivation of the agent. While a human might have sufficient reasons to leave a maze, an agent does not. To give the agent a reason to try and find optimal behaviour for a given problem, a reward or punishment system is used.

The goal of the agent is to maximize the total reward it receives at each time step t. The total reward or total return is in its simplest case defined as:

$$G_t \doteq R_{t+1} + R_{t+2} + \ ... \ R_T \tag{3.3}$$

where R are the rewards for each time steps and T is the final time step. This definition proves useful as long as the interaction of the agent and the environment can be split well in so called episodes. This is the case for the maze example, as whenever the human reaches the exit an episode can end. The key is having a well defined terminal state, such as leaving the maze. However not every problem set includes a specific terminal state.

The hypothetical case assumed is that the reward depends on the direct route between the human and the exit of the maze. A big reward will be given, when the human escapes.
What if in order to escape the human has to take a detour that at first seems at a longer distance to the exit? If the human takes this route, the first couple future rewards would be lower, than when those received when taking a route that leads to a dead end. So taking a detour will first result in lower rewards, but in future show much higher rewards when leaving the maze. To model situations requiring the agent to choose actions that are more far-sighted and not only consider an immediate reward, the discount factor γ is used. It balances short and long term decisions. This is the last parameter of the tuple defining the MDP.

It is important to mention, that the discount factor is not only a way of balancing of future rewards' relevance, but also to bound. That means, that if no discount factor was used and the episodes would last infinitely long time, the problem of unboundedness would cause the sum of the reward to keep growing. When performing infinite steps within an episode there is no converging policy would and therefore it is not possible to define an optimal one.

Combining these characteristics, the discount factor can be viewed *as the probability that the game [or episode] will be allowed to continue after the current move* [20]. It therefore defines the urgency to find a solution

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + ... \ = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \gamma \in [0, 1] \tag{3.4}$$

As shown in equation 3.4, the discount factor is multiplied with each of the following rewards. If the discount factor is close to zero, the focus lies on gaining high short term rewards, while when the discount factor is close to one, the evaluation is more far-sighted.

3.2.2.2 The Choice of Actions

Now, that the motivation of the agent is clear, it can be applied to understand how the agent chooses actions, which at the end is the most pressing question. To do so the concepts of the policy π, the value function $v_\pi(s)$ and the Q-function $q_\pi(s, a)$ have to be introduced.

A policy describes a strategy that the agent follows and therefore fully outlines the agents behaviour. It defines the probability choosing an action a when being in state s.

$$\pi(a|s) \doteq P[A_t = a|S_t = s] \tag{3.5}$$

Note that difference to $\mathcal{P}$ is that it predicts the chance of *transitioning* into a state s_{t+1} from a state s_t when performing an action a. The policy however indicates the probability of *choosing* a specific action when being in state s_t. In the maze example the policy could be to always randomly choose an action. So for each state the probability of choosing to go left would be 25%, the same as going right, straight or backwards.

When the agent finds itself in state s and from then on follows the policy π, the value function indicates the expected return. So the function shows how good it is for the agent to be in the present state.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \tag{3.6}$$

$\mathbb{E}_\pi$ is the expected value of the rewards.

Finally there is the Q-function or action-value function, indicating the expected return, when taking an action a being in a state s and then following the policy π.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{3.7}$$

In other words this function shows how successful an action is when being in state s. All these concepts come together.

The final goal of the agent is to learn the optimal policy π^*, that performs better or equal in every given state than any other policy. So the optimal policy is defined as the policy for which it is valid, that $\pi^* \geq \pi$ if and only if $v_{\pi^*} \geq v_\pi$ for all $s \in \mathcal{S}$. Associated to an optimal policy is the optimal value function v^*, defined as $v*(s) = \max_\pi v_\pi(s)$ for all state s. Similarly it includes an optimal Q-function, that is defined as $q^*(s,a) = \max_\pi q_\pi(s,a)$ for all $s \in \mathcal{S}$ and all actions $a \in \mathcal{A}$.

The vital property for q^* is fulfilling the Bellman optimality equation. The equation 3.8 shows that the expected return is the sum of returns for the current action a in state s and the best possible discounted reward by following the optimal Q-function thereafter.

$$q^*(s,a) = E[R_{t+1} + \gamma \max_{a'} q^*(s',a')] \tag{3.8}$$

To answer the question on how the agent chooses the best action, you can generally say that in the long run the agent will choose the actions, that lead to the highest reward.
But in the beginning of the learning process the agent does not know, which actions are the the most remunerated. In order to explore which actions are best compensated, the agent has to learn from its experience. Therefore different algorithms can be used in the training process.
The following section will focus on giving an overview of the different types of algorithms.

3.2.3 Algorithms

The taxonomy shown in figure 3.3 gives an overview of the best known algorithms and their categorization. Note, that this overview is not complete and a clear subdivision of the categories is not clearly definable, even if it proves to be a helpful outline.

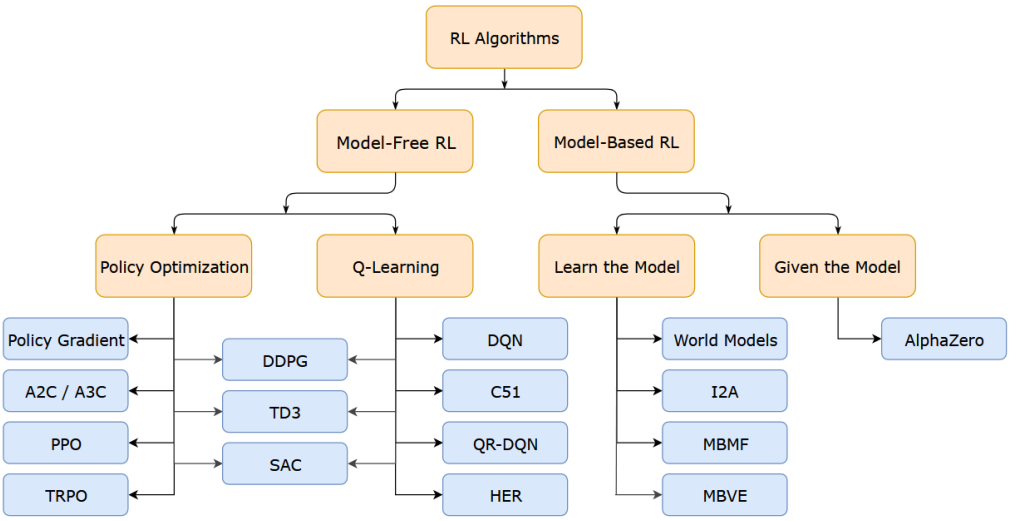

Figure 3.3: Taxonomy of algorithms in modern RL (non-exhaustive), taken from "Spinning Up in Deep Reinforcement Learning" [1]

RL algorithms are classifiable into model-free and model-based algorithms. While model-based algorithms have access to or learn an environment model, model-free approaches work altogether without the knowledge of the models. In model-based RL the agent presumes to have knowledge about the probability transition matrix $\mathcal{P}$ the state space $\mathcal{S}$, the action space $\mathcal{A}$ and the reward space $\mathcal{R}$. Having all this information a decision tree can be modelled and evaluated. This procedure is often only possible when the action and state space are reasonably small so that the computational costs do not explode [10].

In contrast, model-free algorithms do not provide knowledge of the model. Instead an approximation of the expected return is achieved by sampling actions in the environment. Model-free approaches are more popular, as they are easier to implement and tune. Often a model of the environment is not available. If an agent learned a model, the model specific information might bias the agent which makes it more difficult to perform well in other scenarios where the current models would not be accurate.

This scope of this work only includes model-free approaches as they are found more often in current research. Model-free algorithms are split into two subdivisions. On one hand there are algorithms that perform Q-learning

and on the other there are policy optimization algorithms.

Q-learning follows the strategy of learning a Q-function that satisfies the Bellman equation. It usually works off-policy, which means that the data that has been calculated with past policies can be included in current calculations. When the Q-function has been obtained the optimal policy can be derived from it by choosing the actions with maximum value. A well known example is DQN. Refer to the work of the Mnih et al. for more information [24].

Q-learning has a few weaknesses, leading to the need to advance to other algorithms. The reasons are that the performance is poor when the action space is large or infinite. As Q-learning iterates over all possible actions in order to improve, the computation costs get too demanding.
It is also less stable, as the indirect way of learning leads to more failure modes. But as not only the data of the current policy is used, the sample efficiency is higher.

In contrast policy optimization specifically has a representation of the policy $\pi_\theta(a|s)$. In this way it directly optimizes the objective it aims for. In the learning process either the parameter θ is optimized by performing gradient ascent on the performance objective or indirectly by maximizing local approximations of the performance objective. Policy optimization are on-policy referring to the fact that they only update on the data collected with the present policy.

Typical algorithms that use policy optimization are Advantage Actor Critic (A2C) [23], Asynchronous Advantage Actor Critic (A3C) [23], PPO [31] or Trust Region Policy Optimization (TRPO) [30]. For more information the reader may refer to references.

3.2.4 Hyperparameter Tuning

In order to optimize the algorithms displayed, a process called hyperparameter tuning is performed. This procedure includes finding an optimal Machine Learning model architecture by finding the ideal hyperparameter configuration [44]. Hyperparameter are the parameters chosen prior to the training process.
Hyperparameter tuning is still considered an open issue, as there is no *a*

priori way of knowing the optimal configuration of the parameters. Each computation run is time intensive. Especially as the size of data sets for supervised Machine Learning problems is growing, as well as the complexity of the model results [12]. It is considered a difficult task because it requires a very deep understanding of the different parameters, their influence and dependencies among each other.

Typical approaches for tuning are manual search, grid search and random search. Even though there are more sophisticated approaches like Bayesian optimization techniques, these are not part of the scope of this work. Manual search is perhaps the easiest attempt, as the configurations are chosen by intuition. Random search lacks the intuition of the investigator, as it combines the parameters in a random combination within a defined interval. Grid search, as the name says combines every possible combination of the parameters, which has the weakness of giving each parameter the same importance [32].

Finally typical hyperparameters are the number of episodes, learning rate, batch size, discount factor or exploration rate. A more in depth overview of the chosen parameters for the experiments and its responsibilities are given in chapter 6.2.1. Additionally to these parameters, the learning algorithm, the reward design and the action space play a significant role in the design choices and are explained in more depth in chapter 5.

Chapter 4

Related Work

4.1 Exhaustive Searching Algorithms

Finding an optimal solution for a given problem-set is a long and often difficult task.

From the 1950s to the 1980s mathematical programming methods were commonly used to automatically find optimal solutions. While these methods can provide optimal solutions, the computational costs explode when increasing the search space due to the NP-hard complexity of the Flexible Job Shop Problem (FJSP). Other methods like Branch-and-Bound methods strove to decrease the search space by strategically cutting out solutions, performing worse than the current one. This slightly improved the computational cost of these algorithms yet is still not applicable for larger problems.

The general problem remains. If an optimal solution is required, **exhaustive searches** have to be used to iterate over the entire search space. This means the requirement on solutions had to be slightly modified and researchers started to request a nearly-optimal solution. This small change in the requirements opened new doors to explore search space using **approximate searches**.

4.2 Approximate Searching Algorithms

Approximate searches include different algorithms like priority dispatch rules, insert algorithms or bottleneck based heuristics [42].
As an example for these algorithms, dispatching rules will be presented

following an heuristic approach. As heuristics makes use of gathered domain knowledge in the context of production systems possible dispatching rules are:

- *First In First Out (FIFO): The rule describes simple queuing. The lot with the longest waiting time is processed first*
- *Shortest Processing Time (SPT): The lot with the shortest processing time has the highest priority. This policy reduces the Work in Progress (WIP) in front of the machine.*
- *Shortest Remaining Process Time (SRPT): This policy reduces the total WIP in the production.*
- *Earliest Due Date (EDD): The job with the earliest delivery date has the highest priority.[40]*

There are also variation such as look-ahead rules that consider information about future job arrivals to generate a good schedule in advance. If the setup of machines is considered be able to perform certain production steps, it makes sense to not demount it after each process but keep future requests in mind.
Advantages of heuristics are that they are often easy to comprehend on operational level. Their implementation is often simple and computational lighter, which makes them capable of being used in real-time settings.
But as they rely on domain knowledge, they only find problem specific solutions and are not adaptive, learning or flexible [40, 4]. Research has shown, that *in summary[...] any single dispatching rule is unlikely to yield satisfactory performance with respect to multiple performance measures*[11]. This raises the question on how the optimization process can be improved. Obvious approaches appear to be combining different heuristics or generate new ones.

4.2.1 Hyper-heuristic and Meta-heuristic approaches

This is where Hyper-heuristic and Meta-heuristic approaches come into play.Hyper-heuristics *use selection and generation mechanisms to automatically design an optimisation algorithm by selecting or generating suitable heuristics during the search*[36], which usually aims to gain more generally applicable methodologies. By comparison to that meta-heuristics*use pre-designed algorithmic frameworks comprised by specific parameters and operators to solve a problem*[36]. They search in search space of solutions finding a solution for a single specific problem.

To show the performance differences of these algorithms, the results of the work of Lee and Kim [19] will be introduced. They compared priority scheduling and hybrid algorithm. Their working scope comprises the route selection problem and Job Shop Problem (JSP), which are the two decomposable problems in the context of Flexible Manufacturing System (FMS). The JSP includes the problem of splitting the sequence of operations on each machine, while the route selection defines which machine to choose for performing the current operation. The objective is to minimize the make span. Therefore two different approaches have been compared. The first one is priority scheduling and the second one a hybrid algorithm.

The priority scheduling uses two types of rules. One is aimed at machine selection and at part selection. Machine selection rules determine which machine is used when more than one is available. In this context there are two options. Part selection rules determine which of the waiting parts of the waiting queue are selected. In this context eight possible rules including all possible combinations are compared.

The hybrid algorithm has a very similar starting point, yet also uses the meta-heuristic Simulated Annealing.

Simulated Annealing [15] is one of the most popular meta-heuristics making it a good example. An initial solution generated by the priority scheduling algorithm, is used as baseline. The local search begins and similar solutions in the neighborhood will be analysed, until the termination condition is met.

Results showed that hybrid algorithms perform better than the priority scheduling algorithms because of the above exploration. This generates the potential in using heuristics that perform well and then tuning them using different meta-heuristics. Hyper-heuristics come into play when well performing meta-heuristics for certain problem sets are found. Then Hyper-heuristics can sequence meta-heuristics in a more powerful order.

Other examples for meta-heuristic approaches are swarm-based algorithms, tabu search or genetic algorithms.
In a FMS scheduling problem context genetic algorithms have been well researched since the 1980s.

While there are multiple examples of the application of genetic algorithms, general limitations and weaknesses of these are often neglected. Prakash, Chan and Deshmukh [28] have addressed this issue by using a knowledge base in combination with the genetic algorithm. They compared a Knowledge Based Genetic Algorithm (KBGA) with a Simple Genetic Algorithm (SGA), a Decision Tree as well as other Priority Rules in the problem setting of Su and Shiue from 2003 [37]. The model consists of few machines with buffer, AGVs and load and unload stations.

The premise of the work was that genetic algorithms, even though they are global search algorithms, perform poorly when the initial population is very weak and the genetic operators' selection, mutation and crossover are poorly chosen for the context. So they introduced a knowledge base that collected and filtered information about the production system like machine information, parts information or operation information. When using the knowledge to decide on the genetic operators, the following results were gathered as shown in table 4.1.

Scheduling strategy	TP		MF	
	Mean	SD	Mean	SD
KBGA	5265.83	18.86	1025.61	117.42
SGA	5248.73	19.8771	1066.48	206.40
Decision tree (DT)	5246.76	19.81	1151.1685	270.78
First in first out (FIFO)	5226.90	56.96	1769.90	541.35
Shortest processing time (SPT)	5243.53	22.70	1086.01	221.68
Shortest imminent operation time (SIO)	4236.96	37.87	1237.88	286.81
Shortest remaining processing time (SRPT)	5245.50	22.50	1165.33	358.70
Critical ratio (CR)	5200.73	48.94	1669.06	304.98
Dynamic slack (DS)	5219.96	72.81	1163.14	346.71
Earliest due date (EDD)	5231.40	56.29	1536.28	585.81

Figure 4.1: Comparative results, TP = throughput, MF = mean flow time; taken from Prakash et al. [28]

The KBGA outperformed, even if not significantly, all other approaches. The graphs 4.2 and 4.3 is that, KBGA converges much faster than SGA. This has been checked by using a knowledge base for another 10 problems, which returned similar results. Another result is the importance of the parameter choice. When choosing different crossover probabilities (on the right) the performance varies. So knowledge bases may be useful tools, but the hyperparameter tuning will remain an important step when optimizing the schedule.

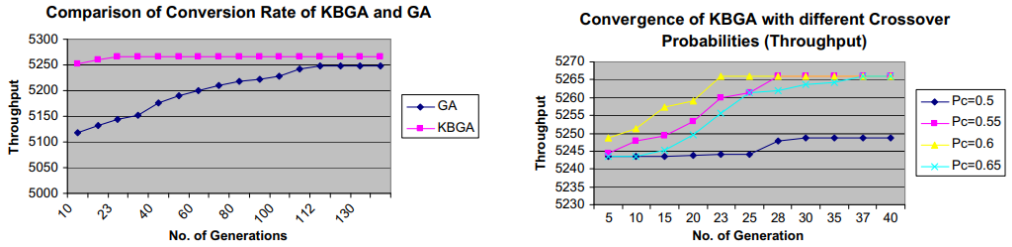

Figure 4.2: Comparison of convergence rate of KGBA and SGA with performance indicator throughput; taken from Prakash et al. [28]

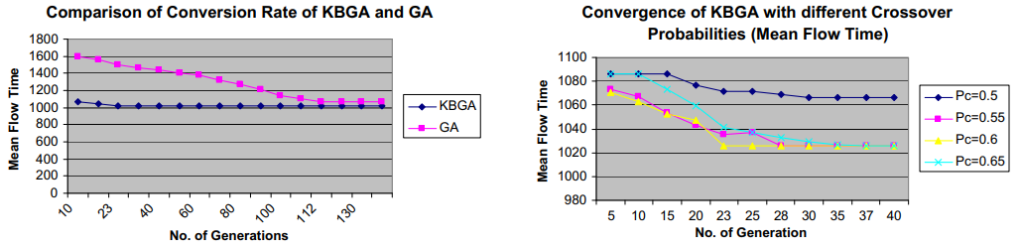

Figure 4.3: Comparison of convergence rate of KGBA and SGA with performance indicator mean flow time; taken from Prakash et al. [28]

4.3 Reinforcement Learning

Even though there are a variety of applied Machine Learning approaches, the scope of this work includes only RL. In the last couple of years the importance of RL has increased. Only few years ago humans still were clearly superior in certain tasks, but now agents using RL perform better. A popular example are the Atari computer games that are often used as reference for comparing RL to humans capabilities and show that RL already has beaten humans in quite a few games [24].

Another recent example is the game Go, which is strategy board game for two players. *Humankind has accumulated Go knowledge from millions of games played over thousands of years, collectively distilled into patterns, proverbs and books. In the space of a few days, starting tabula rasa, AlphaGo Zero was able to rediscover much of this Go knowledge, as well as novel*

strategies that provide new insights into the oldest of games[35].

So why shouldn't RL also provide new strategies and heuristics that humans have not yet been mentally incorporate?

A recent literature review [27] stated, that *in total, 89% of the benchmarked Deep Reinforcement Learning implementations increased the scheduling performance and reached lower total tardiness, higher profits, or other problem-specific objectives.*

The general interaction between a RL algorithm and scheduling problem itself is described in the following. A simulation is used to try out the suggested actions. At first the simulation returns the current state to the RL agent. Based on this information the agent decides on an action. The simulation performs this action and transitions into the new state. It gives a reward to the agent, which is used to improve further decision making. This process is repeated until a termination condition is met.

As presented in chapter 3.2, there are different learning algorithms in the field of RL. Furthermore the approaches range from using single agents to multi-agent systems, from being cooperative to being competitive. There are also combinations of different strategies.

In the following part of the section different ideas for modelling the agent will be presented. Afterward the subsections are split into different topics, that will find application when designing the algorithm for this problem. Structuring the section like this offers the ability to focus on specific ideas of the design questions instead of comparing entire agents.

4.3.1 General approaches

One of the most classical approaches is Q-learning and Q-learning variations that find application in research [43, 2, 41, 26]. Aydin and Öztemal [3] for example developed Q-III, an extension of the traditional Q-learning algorithm that improves still existing generalization problems. In their approach in 2000 they used an RL agent that performs dynamic scheduling by letting the RL agent learn to choose dispatching rules fitting the circumstances best.

Some combine RL with neuronal networks to predict actions. Expert knowledge may be incorporated using an event handler like the Job Shop Management by Waschneck et al. [41] do. They also added rules for certain scenarios. In case 80% of the Work in Progress are in queues a switch to heuristics is performed to avoid a crash of the system. This change is then penalized by the reward function.

Another idea is the usage of bidding models like May et. al [21], where multiple different agents bid on which decisions to make.

A further market-based approach is presented by Csáji et al.. They use resource and order agents. Each job and each resource get an agent assigned to select the presumably best schedule. In this approach three levels of learning are applied. The top level uses a Simulated Annealing algorithm, followed by the RL algorithm, that uses Temporal Difference (TD) learning. The last level applies a Neuronal Network (NN) function approximator [5].

Mayer's et al. used a Deep Reinforcement Learning (DRL) agent. They tried to find the optimal policy with a policy gradient method combined with a NN function approximator. However they noticed that when they only used one agent, the agent could either learn how to create a balanced system or how to keep the balance within a production system. So they suggested to train more than one agent. One agent would be responsible to establish an equilibrium of the distributed workload while the other would learn how to keep the equilibrium over time [22].

4.3.1.1 Reward Design

The reward design is required to translate the processing targets into a reward. These targets can be one or many of the following:

- *Total Completion Time (TCT) so far*
- *Average Work in Progress (WIP)*
- *Total Tardiness (TT) so far*
- *Total Lateness (TL) so far*
- *Maximal Lateness (ML) so far*
- *Number of Tardy Jobs (NTJ) so far [43]*

The reward is generally divided into two parts: the immediate reward given by each state transition and the accumulated reward. While the immediate

reward works towards the short-term effect of an action, the accumulated reward represents the long-term effects. It is often important to balance short-term and long-term reward portions in the reward function.

The reward function often considers a negative reward component when performing invalid actions and for each time step that has passed. For example Mayer et al. [22] used the following reward function (see 6.1), where $-t_T, s_{t-1}$ is the negative reward portion assigned to the passed time. This is done to encourage the agent to finish the production quickly. R_{bonus} is given, when a product is finished by using this action, while R_{punish} is a negative reward portion for trying to apply invalid actions.

$$R_t := \begin{cases} -t_T, s_{t-1} \rightarrow s_t + R_{bonus} & \text{if } A_{t-1} == A_{sink} \in A_{t-1,val} \\ R_{punish} & \text{if } A_{t-1} \notin A_{t-1,val} \\ -t_T, s_{t-1} & \text{otherwise} \end{cases} \tag{4.1}$$

Another approach is to use the Uptime Utilization (UU) of the production system. UU is defined by the division of the number of slots in process and total capacity. In the work of Waschneck et al. [41] the reward phase is split into two phases. In the first phase the UU in the work center is added to the negative rewards for taking invalid action.In the second phase the UU of the entire line is added to the negative reward.

Altemüller et al. implemented a single agent DQN, that punished invalid actions with a reward of -1. They also used a local and global reward component. The local reward was given for choosing urgently needed parts from a line. The global reward re-compensated low numbers of average time constraint violations. During the training they noticed that the agent was reducing the order releases so that the system was less loaded, making it easier to handle time constraints. To counterbalance this behaviour the second global reward portion was rewarding adherence to a predefined WIP level [2].

4.3.1.2 State Design

The state space contains all possible states. The state, in which an agent needs to make a decision is called a decision state.

The state design is considered a crucial point for the performance of the RL algorithm. The state should fulfill the Markov property and different papers suggest similar state vectors.

$$S_t := [\underbrace{4}_{ss} \; \underbrace{0\,1\,2}_{ib_1\ p_1\ ob_1} \; \underbrace{3\,1\,0}_{ib_2\ p_2\ ob_2} \; \underbrace{1}_{vo} \; \underbrace{1\,0\,0\,0\,0\,0}_{vp_1\ \dots} \; \underbrace{0\,1\,0\,0\,0\,0}_{va_1\ \dots}]$$

Source M_1 M_2 V_1 V_1 location Valid actions

Figure 4.4: State vector as presented by Mayer et al. [22]

As shown in figure 4.4 the vector could be a one-hot vector encoding the items and their location. Each machines (M_n) is modelled by three different integers, the input buffer (ib_n), the current process (p_n) and the output buffer (ob_n). Each Automated Guided Vehicleis modelled by a single integer (V_n), 1 signifying that it is currently occupied and 0 meaning the AGV is free. Additionally the location is encoded (V_n *location*). For each position a boolean that can be 1 or 0 is used. If the AGV is at the certain position it is set to 1. Finally the state contains the information, which actions are currently valid and which are not.

Papers like May et. al [21] also add general production information. As an economic bidding model is applied the state also includes market information like previous quotes for parts and all current quotes for stations and AGV. Even individual agent information is added.

Bao-An Han and Jian-Jun Yang [9] suggest to focus on suitable features, when using a value-function approximator. These are;

1. main features of the scheduling environment including global and local information
2. information related to the scheduling objective

It is also recommendable to represent the state in a common feature set. The features should be a numerical representation of the state that are easy to calculate and normalize.

In conclusion it has proven positive to design the state as minimalistically as possible. Xie, Zhang and Rose stated, that from their results they found, *that unnecessary inputs can only make the state space even larger and worsen the performance. So, taking only the necessary information form states and actions is always the right step* [43].

4.3.1.3 Action Design

The action space usually starts including all possible actions. Then invalid action are either masked [21], punished when executed [22] or receive a zero-valued reward [16]. The last approach included a counter to avoid the agent always choosing this option.

In this context the action space often contains two types of actions like *pick* and *drop*. The pick and drop action can be arbitrarily complicated. It ranges from choosing whether to pick or drop a part up to choosing the correct machine (machine selection) or choosing the best part from a queue [2] (part selection) in order to optimize the production process. So the action usually contains parameters indicating where the source and destination of this action is and also indicating which item will be transported.

Kuhnle et al. suggested also to add a waiting action to give the system a chance to wait for a part to finish processing [16].

Chapter 5

Design choices of Reward, State Space and Action Space

After gaining a better understanding of RL, it was possible to design the main characteristics of the agent. The following subsections explain the reasoning and the choices for the design of the reward, state space, action space and the algorithms of the agents.

It should be mentioned, that the considered time steps for choosing an action were not equidistant, since the state is only transferred from the simulation to the agent, when an action is required. This avoids communication traffic that is not useful, as invalid actions cannot be performed and therefore can be rewarded on the software side of the agent.

5.1 Action Space

The action type that an agent can choose is called *move* and includes the origin and the target as well as the item that shall be transported. Following the defined work plan the two different product types result in possible actions, that are depicted in figure 5.1 and 5.2.

Possible action	Source1	M1	M2	M3	M4	M5	M6	M7	M8	Drain1	Drain2
Source1	x	✓	x	x	x	x	x	x	x	x	x
M1	x	x	x	✓	✓	✓	x	x	x	x	x
M2	x	x	x	x	x	x	x	x	x	x	x
M3	x	x	x	x	x	x	x	✓	✓	x	x
M4	x	x	x	x	x	x	x	✓	✓	x	x
M5	x	x	x	x	x	x	x	✓	✓	x	x
M6	x	x	x	x	x	x	x	x	x	x	x
M7	x	x	x	x	x	x	x	x	x	✓	✓
M8	x	x	x	x	x	x	x	x	x	✓	✓
Drain1	x	x	x	x	x	x	x	x	x	x	x
Drain2	x	x	x	x	x	x	x	x	x	x	x

Table 5.1: Possible actions for product type A, ✓shows possible action, x shows impossible actions, left side positions show the origin, top positions show the target

Possible action	Source1	M1	M2	M3	M4	M5	M6	M7	M8	Drain1	Drain2
Source1	x	x	✓	x	x	x	x	x	x	x	x
M1	x	x	x	x	x	x	x	x	x	x	x
M2	x	x	x	✓	✓	✓	x	x	x	x	x
M3	x	x	x	x	x	x	x	x	x	x	x
M4	x	x	x	x	x	x	x	x	x	✓	✓
M5	x	x	x	x	x	x	x	x	x	✓	✓
M6	x	x	x	x	x	x	x	x	x	✓	✓
M7	x	x	x	x	x	x	x	x	x	x	x
M8	x	x	x	x	x	x	x	x	x	x	x
Drain1	x	x	x	x	x	x	x	x	x	x	x
Drain2	x	x	x	x	x	x	x	x	x	x	x

Table 5.2: Possible actions for product type B, ✓show possible action, x show impossible actions, left side positions show the origin, top positions show the target

Therefore the action space includes the cross product of origin, target and product type, so

$$11 * 11 * 2 = 242$$

actions, from which 24 are considered possible. Note, that in this context possible actions are defined as actions that are following the working plan. Valid actions are defined as the sub-set of possible actions that are able to be performed during the simulation in a specific point in time. These are the conditions that need to be fulfilled in order to call an action valid:

- The origin from which a part shall be moved is occupied

- The origin contains a part that matches the part type of the action
- The target is not occupied
- The action is in the set of possible actions
- An action is also considered valid, if the origin contains a part, that has not finished its processing step yet. The system will then wait for its termination before performing the action

At first sight it is not possible to distinguish valid from invalid actions, so this will also be one of the main challenges of the agent, especially as it is not aware of these constraints directly.

5.2 Reward

A well defined reward helps the agent to optimize. Often reward functions are close to optimization criteria, which in this case is the minimal makespan for a defined demand.

As the agent has to learn which actions to perform at which point in time, it is common to punish invalid actions.

That is why it seemed suitable to use the reward function as shown in 5.1, suggested by Mayer et al. [22].

$$R_t := \begin{cases} -t_T, s_{t-1} \rightarrow s_t + R_{bonus} & \text{if } A_{t-1} == A_{sink} \in A_{t-1,val} \\ R_{punish} & \text{if } A_{t-1} \notin A_{t-1,val} \\ -t_T, s_{t-1} & \text{otherwise} \end{cases} \tag{5.1}$$

Figure 5.1: Reward function type *mayer*

where $-t_T, s_{t-1}$ is the negative reward portion assigned to the passed time. This is done to encourage the agent to finish the production fast. R_{bonus}

is given, when a product is finished by using this action, while R_{punish} is a negative reward portion for trying to apply invalid actions.

Throughout the experimentation the weights of each part of the reward function have been varied.

5.3 State Space

The state design can have a major influence on the learning performance that can be reached [38]. It is derived from an environment, that is either fully or partially observable. In this case it can be fully observable, as all information can be given to the agent.

In general it is recommendable to encode the information numerically in a one-hot vector. The information should only be included, if it appear important and can be pointed out by duplication. Furthermore the change of the state due to an action should be transparent [17].

Despite an extensive literature review no further information regarding the best practises for designing a state space was encountered.

It was decided to design the state space small and to avoid too much information. The goal was to make it easier for the agent to learn the influence of only a few but relevant variables. A smaller state space may also reduce the computation time. In chapter 5.2 of Annex A an overview over the state XML-files is given.

The state space includes all machines each with the property *occupied*, represented through a boolean *part type* which can include an element of the set PartA, PartB, None and the remaining processing time in seconds. Empty machines have a processing time of -1.

This information however is only used to check internally, whether an action is valid or not, since it would significantly increase the different variations of the state. That is why only occupation is considered part of the state. So each machine can either be occupied or not, which results in

$$2^{11} = 2048$$

states.

The state does not directly include an indicator as to whether they are terminal state. It is not encoded here as it would increase the state space dramatically. The termination of an episodes is reached when the defined demand of products has been produced or the maximum of steps per episode have been tried out.

5.4 Agent Type

Two agent types have been chosen to be evaluated during the experimental phase. These are Proximal Policy Optimization (PPO) and double Deep Q-Network (DQN). The following section explains the reason for this choice.

PPO is based on Trust Region Policy Optimization (TRPO), which is a natural gradient method, that uses trust regions. Trust regions are constraints on how far the updated policy can differ from the last one. However this increases the overhead of the optimization problem.

While PPO also aims to limit or *clip* the difference between the old and the current policy, it adds this constraint directly to the objective function, as it assures that the difference between the old and the updated policy are not large[13, 39]. If they were too big this could lead to irrecoverable harm. So PPO algorithms *have some of the benefits of trust region policy optimization (TRPO), but they are much simpler to implement, more general, and have better sample complexity (empirically). Our experiments test PPO on a collection of benchmark tasks [... and] show that PPO outperforms other online policy gradient methods, and overall strikes a favorable balance between sample complexity, simplicity, and wall-time*[31]. A high sample complexity means the problem requires a very high amount of transition to obtain good results.

On the other hand a value based algorithm was chosen, as these do not directly improve the policy but the Q-function, from which the policy can be derived. However Q-learning with function approximators are often not able to solve simple problems and are generally poorly understood [31].

There are two main issues for value based algorithms. The first issue is that the target values are not stationary. This means that the target, which is applied to improve the internal network change, is also calculated using

this network [39, 25]. The second issue is the non-compliance with the IID assumption. This means that the data used for the training is not independent and not identically distributed.

DQN algorithms solve the first issue by evaluating current parameters of the NN based on a prior setup of the target network. The target network is updated infrequently so that the target stays constant for a long period of time, which makes it more stationary and easier to optimize towards.

The second issue is solved by introducing an Experience Replay Buffer (ERM), which works as a larger buffer for prior transitions. Whenever the buffer is full with transitions, they are replacing the ones that are currently used in training. Now they can be sampled randomly to train the NN and do not have be used directly and thus not in order.

These solutions however slow down the training process, since the training uses old transitions for a long time. After a long training time they usually yield good results [39].

Lastly it is to say, that the double DQN was selected, because their training using two neural networks is considered more stable.

Chapter 6

Experiments

6.1 Introduction

In order to collect information about the potential of Reinforcement Learning in a production system setting, the theoretical concepts have been explained (compare chapter 3.2) and the related work has been analyzed as presented in chapter 4.

To analyse the question from a more practical point of view, a prototype software has been developed allowing the user to perform different experiments in relation to the training process of an agent in a FMS environment. The following chapter deals with these experiments and the results derived. The methodology will be explained, the performed experiments will be listed and the chosen performance indicators introduced. Finally, the results of the experiments will be presented.

6.2 Methodology and Setup

The main goal for the agent is to reduce the timespan required to produce the demand. Therefore the *totalTime* of each production episode is stored. To obtain good results the agent has to learn to differentiate between valid and invalid actions.

The training environment for the agent was set up under the configurations described in Annex A in chapter 4.5.

The presented data has been collected during the training and evaluating process of the agent. Every experiment was performed three times in order to ensure the correctness and robustness of the data. However the author could not guarantee that the agent always uses the same seed, as the used library does not offer this feature. This causes additional random originating from the experiments to the built-in random of the agent.

Originally a grid search was planned to gain a deeper understanding of the dependencies and influences of the different hyperparameters. Before starting a grid search, a few trial and error runs were performed. It showed that experiments with 20 episodes took 20-30 minutes, experiments with 100 episodes around 2 hours and experiments with 200 episodes around 4 hours. For this reason the grid search was restructured and split into subcategories to decrease the number of experiments by reducing the cross product of the different configurations.
The best performing combination of parameters of one category were used for the next experiment iteration. This obviously is a simple assumption, as it is possible that other combinations would have performed better.
To reduce this risk the parameters regarding the training duration were tuned first, as these are the parameters that dictate the speed of learning and show the convergence of the agents training process. It is not necessary to continue training an agent, if its personal best is reached. Furthermore slowing down the training of an agent does perhaps not have positive side effects, so that these experiments do not have to be performed.

Afterwards different reward functions have been tested that balance the formula differently.

Finally different algorithms have been tried including another agent type and different discount factors. Table 6.1 shows the different categories of experiments as well as the tuned parameters.

In the end of the first experiment set it was planned to last around 20 days of computation as the experiments were performed three times.
However the other experiments required a significantly shorter computation time, as only few parameters were changed. For the second 4 days and for the last experiment 2 days of computations were estimated per run.

6.2.1 The Hyperparameter

The chosen hyperparameters are shown in table 6.1.

Category	Parameter	Interval
Training duration	Episodes	10 - 200 [num]
	Learning Rate	0.001 - 0.1 [%]
	Batch Size	1 - 10 [num]
	Exploration Rate	0 - 0.01 [%]
Reward function	Reward Function	mayer, mayer_-_100_-5_, mayer_-_10000_-5_, mayer_-_10000_-25_, mayer_+_100_-5_, mayer_+_10000_-5_, mayer_+_10000_-25_
Agent configuration	Agent Type	PPO,double DQN
	Discount Factor	0.5 - 0.99 [%]

Table 6.1: Tuned parameter and their intervals: [%] indicating a percentage value, [num] any other numeric value

The amount of **episodes** indicates the time spent to train the agent. An episode ends either when a terminal state is reached or when the maximal amount of actions (in this case 25000) has been exceeded. Both conditions are defined by the problem set. It is important that the amount is high enough as to enable the agent to learn patterns and behaviour. At the same time it should be small enough to perform the experiments on a commercial computer within reasonable time.

The **learning rate** represents how fast the agent shall learn. Small learning rates lead to a slower but more thorough learning process, while a higher learning rate leads to more quick decisions.

The **batch size** influences the speed of adjustments similarly to the learning rate. It determines after how many episodes the agent shall update.

The **exploration rate** defines the balance between exploration and exploitation and is the fourth and last parameter that will be investigated in the first set of experiments.

The second category are problem specific configurations. In this case the focus was set on the **reward function** (compare chapter 3.2), as this is a crucial point for the agent's performance.
For these experiments the reward function is based on the work of Mayer et al. [22]. A slightly adapted version is shown in figure 6.1. The only difference here is, that the $-t_T, s_{t-1}$ is added instead of subtracted. The explanation of the different parts of the formula can be taken from chapter 4.3.1.1. The reason for choosing this reward function is given in chapter 5.2.

$$R_t := \begin{cases} t_T, s_{t-1} \rightarrow s_t + R_{bonus} & \text{if } A_{t-1} == A_{sink} \in A_{t-1,val} \\ R_{punish} & \text{if } A_{t-1} \notin A_{t-1,val} \\ t_T, s_{t-1} & \text{otherwise} \end{cases} \tag{6.1}$$

Figure 6.1: Reward function type *mayer*

The last series of experiments include the concrete agent configurations. This comprises the **agent type**, so the learning algorithm used for training. The agents are trained to use either Proximal Policy Optimization or double Deep Q-Networks. The choices of these agents are explained in chapter 5.4 of the Report.

In this category also the **discount factor** is significant. It is a factor, that balances long term against short term decisions. A small discount factor means that the agent acts towards gaining high returns short term, while a high discount factor lets the agent prefer beneficial actions in the long term.

The **demand** is a problem specific parameter. It indicates how long an episode lasts by representing the amount of products that have to be manufactured within one episode. Since it is impossible to have a representative demand due to computational cost a value of 100 has been showed. The number should be small enough to avoid exorbitant calculations, yet big enough to enable the agent not only to learn the initial phase of building an equilibrium in the system, but also to maintain its balance.

6.2.2 The Experiments

This section lists the different experiments that were executed per category. Table 6.2 includes the experiments regarding the training duration, table 6.3 the experiments, examining the reward function. Table 6.4 shows the experiments including the agent configurations.

Learning Rate	Episodes	Exploration Rate	Batch Size
0.001	20	0	1
0.001	20	0	10
0.001	20	0.001	1
0.001	20	0.001	10
0.001	20	0.01	1
0.001	20	0.01	10
0.001	100	0	1
0.001	100	0	10
0.001	100	0.001	1
0.001	100	0.001	10
0.001	100	0.01	1
0.001	100	0.01	10
0.001	200	0	1
0.001	200	0	10
0.001	200	0.001	1
0.001	200	0.001	10
0.001	200	0.01	1
0.001	200	0.01	10
0.01	20	0	1
0.01	20	0	10
0.01	20	0.001	1
0.01	20	0.001	10
0.01	20	0.01	1
0.01	20	0.01	10
0.01	100	0	1
0.01	100	0	10
0.01	100	0.001	1
0.01	100	0.001	10
0.01	100	0.01	1
0.01	100	0.01	10
0.01	200	0	1
0.01	200	0	10
0.01	200	0.001	1
0.01	200	0.001	10
0.01	200	0.01	1
0.01	200	0.01	10

Table 6.2: List of Experiments type **Training duration**: Static parameters are demand (100), discount factor (0.99), reward type (*mayer*), agent type (PPO) and maximal steps of an episode (25.000); each experiment includes a training and an evaluation cycle

The best two configurations of the first experiment used to analyse the different reward functions.

Reward Type	Exploration Rate
mayer	0
mayer	0.01
mayer_-_100_-5	0
mayer_-_100_-5	0.01
mayer_-_10000_-5	0
mayer_-_10000_-5	0.01
mayer_-_10000_-25	0
mayer_-_10000_-25	0.01
mayer_+_100_-5	0
mayer_+_100_-5	0.01
mayer_+_10000_-5	0
mayer_+_10000_-5	0.01
mayer_+_10000_-25	0
mayer_+_10000_-25	0.01

Table 6.3: List of Experiments type **Reward function**: Static parameters are demand (100), discount factor (0.99), agent type (PPO), maximal steps of an episode (25.000), episodes (100), learning rate (0.001) and batch size (1); each experiment includes a training and an evaluation cycle

The different configurations can be read from the name of the reward function. *mayer* indicates the origin of the reward function, the first "-" or "+" indicates, whether the t_T, s_{t-1} was added or subtracted. The following number 100 or 10000 are the divisors of the t_T, s_{t-1} indicating the importance of this term. Lastly -5 or -25 indicate the punishment for an invalid action.

The last experiment again applies the two best performing setups and evaluates agent type and discount factor.

Agent Type	Discount Factor	Exploration Rate
PPO	0.5	0
PPO	0.5	0.01
PPO	0.99	0
PPO	0.99	0.01
DDQN	0.5	0
DDQN	0.5	0.01
DDQN	0.99	0
DDQN	0.99	0.01

Table 6.4: List of Experiments type **Agent configuration**: Static parameters are demand (100), maximal steps of an episode (25.000), episodes (100), learning rate (0.001) and batch size (1), reward function *mayer_-_10000_-5*; each experiment includes a training and an evaluation cycle

6.2.3 The Key Performance Indicators KPIs

Four performance indicators were chosen: *totalTime*, *return*, *totalSteps*, *validActions*.

totalTime describes the time in seconds of the simulation required to produce the demand. This is the main indicator of the experiments and direct objective, that needs to be minimized.

Return represents the total reward given withing one episode to the agent. If the reward function supports the optimization goal, *return* should increase over time, as *totalTime* decreases.

While the agent aims to reduce *totalTime* it also has to learn which actions are valid. Therefore the *totalSteps* indicator is used. It represents the total amount of actions that the agent chooses. The goal is to decrease the *totalSteps* meaning the agent learned to find which actions are valid. *validActions* should stay constant, as the required action to produce a demand are constant as well. It is possible that they decrease slightly meaning the agent found a way of finding a solution requiring even less steps to produce the demand.

For the analysis of the experiment a python script has been used, that also plotted the following graphs.

6.3 Analysis of the Experiments

The three experiment series are evaluated in this section. Evaluating the first experiment a more detailed overview over the KPIs was given, to show their interdependence. For the later experiments this was reduced and only the most significant takeaways were shown.

6.3.1 Training duration

The following section shows the results of the experiments regarding the training duration. Instead of analyzing all 36 configurations individually, similar configurations are combined and a trend line for each KPI is shown. The best performing experiments were than found by looking into the most promising trend lines. At the same time the underlying experiments were shown, when the trend lines were surprising.
For each performance indicator a prediction is given and the actual performance explained.

6.3.1.1 TotalTime

It is expected, that *totalTime* is increasing in the training process, as the agent improves. In the evaluation *totalTime* should be constant, as the agent applies the trained policy.

The results are shown in figure 6.2.

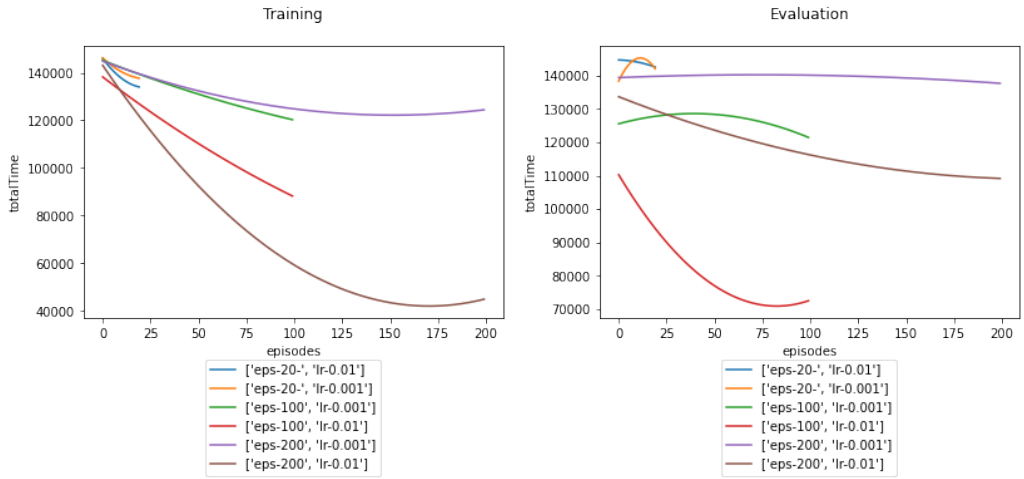

Figure 6.2: Trend lines of *totalTime*, experiment **Training Duration**; each trend line is based on six underlying experiments

The predicted decreasing tendency is visible during the training process. On first sight trends using a learning rate of 0.01 have a faster falling trend. However when analyzing all individual graphs it seems evident, that this is the case, because in these situations an early stopping mechanism was applied. This is an implementation detail defining as that after a number of three identical returns are reached, the training process is interrupted, as the agent is probably learning poor behaviour. The computation time was thus slightly reduced. All following KPIs values are set to 0, which in return is the reason for the falling trend lines.

Figure 6.3 shows all individual underlying experiments. On the left side are graphs of the experiments with a constant of 200 episodes and a learning rate of 0.001, while on the right side the experiments with 200 episodes and a learning rate of 0.01 are shown. It stands out that the graphs with the higher learning rate fluctuate much more and that four out of six experiments have an early stop, resulting in a descending trend line.

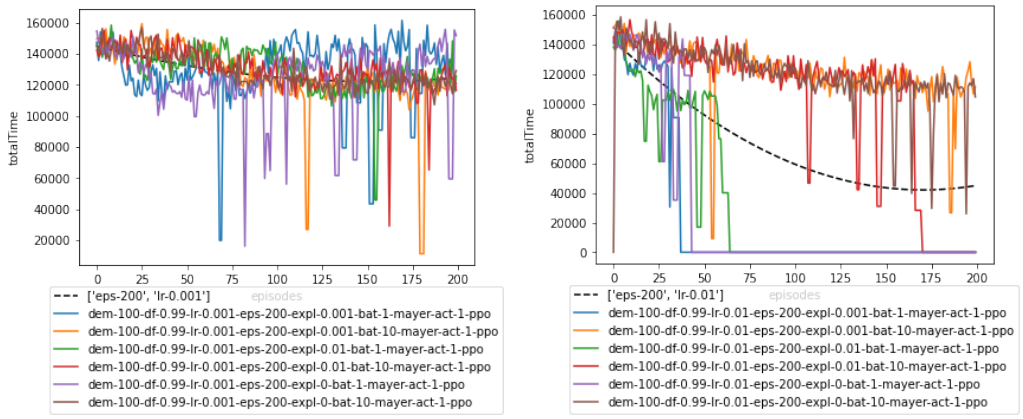

Figure 6.3: Underlying experiments for the *totalTime* trend line [eps-200,lr-0.001] and [eps-200,lr-0.01] during training; experiment **Training Duration**

At first sight **totalTime** during the evaluation process does not seem to be constant. However this is again traceable to early stoppings resulting in descending trend lines. The learning is that reduced learning rates result in a more consistent training. While the amount of episodes did not have a significant influence, it seems that 20 episodes were not enough to improve the system, while 100 episodes with a learning rate of 0.001 and a batch size of 1 has led to the best performance with around 120900 seconds.

6.3.1.2 Return

The author foresees that the *return* increases over time in the training process as the agent tries to maximize it, indirectly improving *totalTime*. In the evaluation process *return* should stay quite constant around the individual maximal value of the training process.

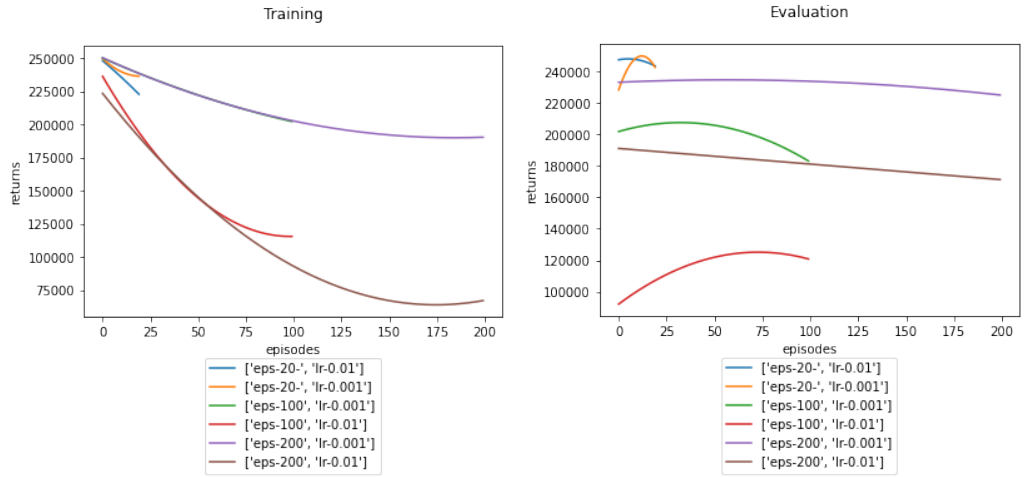

Figure 6.4: Trend lines of *return*; experiment **Training Duration**

Figure 6.4 shows the results. *Return* decreases over time during the training process. Even though high gradients are again traceable to the early stopping mechanism, analyzing the first and last values of each individual graph shows that *return* decreases significantly.This happens, because the current time is added to the results. As time decreases, this value also decreases.

Ignoring the early stoppings in the evaluation, the graphs can be considered constant, which is the result predicted.

6.3.1.3 TotalSteps

While the agent should decrease *totalTime*, it also has to learn to differentiate between valid and invalid actions. The prediction is, that over time the number of *totalSteps* decreases in the training process and is constant in the evaluation process.

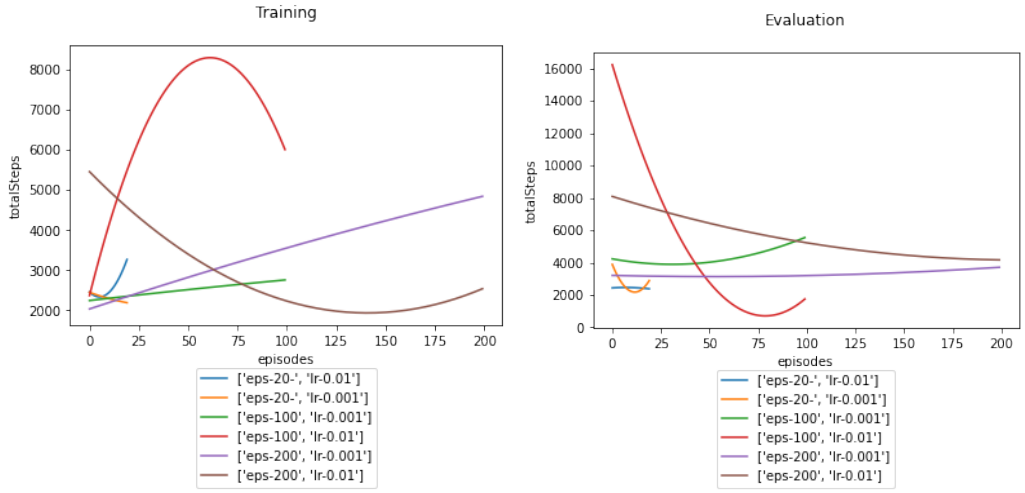

Figure 6.5: Trend lines of *totalSteps*; experiment **Training Duration**

In figure 6.5 the results are shown. It seems that the trend lines with 20 episodes did not have enough time to learn the difference and therefore do not decrease significantly. The graphs with 200 episodes are due early stoppings (for [eps-200,lr-0.01]) while high fluctuation is visible in the other trend line ([eps-200, lr-0.001]). The graph [eps-100, lr-0.01] takes a noticeable course. Looking at the underlying experiments the fluctuation was small in the beginning but increases over time. Additionally one experiment that at first gave very high values but had an early stop, explaining why the graph decreases when reaching 50 episodes.

Considering early stoppings the evaluation can be considered as predicted.

6.3.1.4 ValidActions

The prediction for the course of *validActions* is that they should slightly decrease in the training process and are constant in the evaluation.

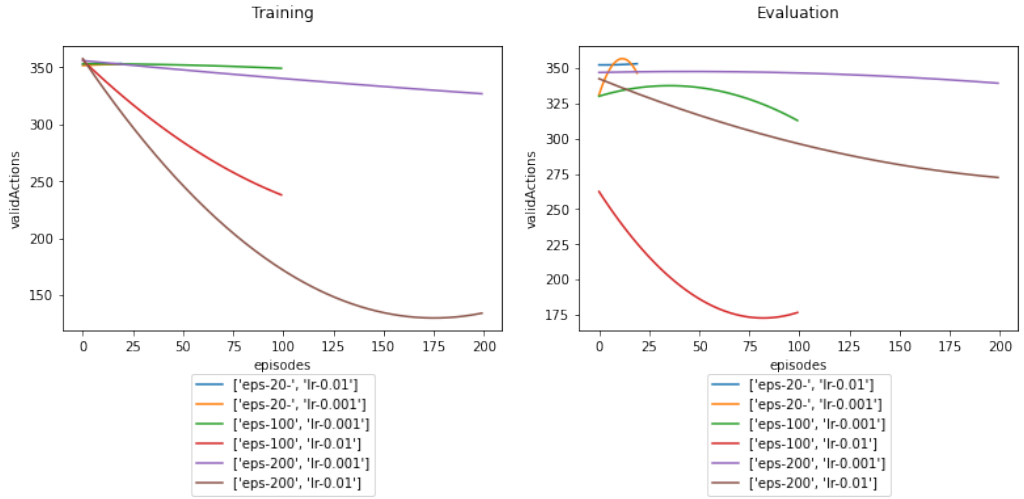

Figure 6.6: Trend lines of *validActions*; experiment **Training Duration**

The results are shown in figure 6.6. As predicted the graphs are slightly decreasing. The graphs including experiments with no early stops ([eps-20,lr-0.01],[eps-20,lr-0.001],[eps-100,lr-0.001], [eps-200,lr-0.001]) are more or less constant and all of these trends reach a value of around 350 actions, necessary to produce the required demand.

The predicted tendency is visible in the evaluation. Only graph [eps-100,lr-0.01] has a large decrease, due to the underlying experiment with a batch size of 1 and an exploration rate of 0, that very often have very bad results, meaning the maximum amount of actions have been chosen, but only a very small number of actions were valid, lowering the average significantly.

The best two experiments are found in the underlying experiments of the trend line [eps-100, lr-0.001], as they did not fluctuate or stop early due to bad results. A batch size of 1 performed better by around 20000 seconds, so the following two configurations were chosen for the next experiment series.

1. dem-100-df-0.99-lr-0.001-eps-100-expl-0-bat-1-mayer-act-1-ppo
2. dem-100-df-0.99-lr-0.001-eps-100-expl-0.01-bat-1-mayer-act-1-ppo

For the next series of experiments, the reward function will be adapted.

6.3.2 Reward function

The following section shows the results of the experiments involving the reward function and in that context the balance between reducing *totalTime* and learning to differ between valid and invalid actions. First of all an overview of the performance of each reward function is given. In comparison to the last experiment the results are not presented following the KPIs in the order. Instead a table highlights the performance and early stoppings. This allows are fast overview of the best performing configuration, used for the last experiment.

Note, that the returns now are no more directly comparable anymore, as the function used to calculate the returns has changed.

Each reward function was tested with the best two configurations of the first series of experiments. They differ in their values of the exploration rate. The blue graph always shows the setup with an exploration rate of 0.01 (short: [expl-0.001]), while the orange graph uses an exploration rate of 0 ([expl-0]). The black dotted graph shows the interpolated trend line of both individual experiments.

6.3.2.1 Prediction

The author predicted that using the reward functions subtracting a percentage of the passed simulation time will perform better. This is assumed, as punishing the agent for taking a long time would lead the agent to try and reduce the *totalTime*, while adding the value would encourage the agent to slow down the production.

The second assumption was that using a weight of 10000 instead of 1 or 100 would perform better, as this balances the importance of *totalTime* and the learning process for valid actions the most equally. For instance *mayer_-_10000_-5* would give -5 as a punishment for choosing an invalid action, while working for five hours would lead to -180 as a reward.

6.3.2.2 Result

The following table shows the average *totalTime* of each experiment run over all episodes during the evaluation.

Reward Function	expl	Mean Run 1	Mean Run 2	Mean Run 3
mayer	0.01	115808s	123547s	6562s
mayer	0	121804s	121276s	126707s
mayer_+_100_-5	0.01	15863s	14851s	15086s
mayer_+_100_-5	0	5496s	20933s	11708s
mayer_+_10000_-5	0.01	106984s	103883s	118125s
mayer_+_10000_-5	0	101721s	116519s	110602s
mayer_+_10000_-25	0.01	49554s	137769s	5810s
mayer_+_10000_-25	0	141062s	127208s	133801s
mayer_-_100_-5	0.01	2200s	3547s	15614s
mayer_-_100_-5	0	7626s	65777s	955s
mayer_-_10000_-5	0.01	22484s	138066s	59186s
mayer_-_10000_-5	0	154402s	149531s	127486s
mayer_-_10000_-25	0.01	141878s	25765s	149254s
mayer_-_10000_-25	0	133143s	138940s	126604s

Table 6.5: Overview of the *totalTime* of the different reward functions and runs, blue numbers indicate an early stopping, red ones a continuous choice of 25.000 steps

The blue and red coloured *totalTimes* are not to be considered as valid runs, as they either stopped early or achieved low *totalTimes* only by not finding enough *validActions*. The *mayer_-_100_-5* and *mayer_+_100_-5* only had runs excluded from further evaluation. Supposedly this is the case, because the importance of *totalTime* was too high leading the agent to avoid learning to distinguish valid and invalid actions.
Better performing were *mayer_-_10000_-25* and *mayer_+_10000_-25*. Only in the setup with an exploration rate of 0.01 early stopping was encountered. The best results were achieved when using the *mayer_+_10000_-5* function. Even though it was predicted that using a 10000 divisor would balance the importance, it was surprising that the function adding the *totalTime* was performing best. This however shows one of the main takeaways of this experiment series. It was predicted that it would counterbalance the agent if he received a reward for long times. However it could also be interpreted as giving a high reward for finding a solution at all, as the *totalTime* is significantly higher in those cases. This shows that the human biases the system even unintentionally when choosing components of the reward function.

The best two configurations are consequently:

1. dem-100-df-0.99-lr-0.001-eps-10000-expl-0.01-bat-1-mayer_+_10000_-5-act-1-ppo
2. dem-100-df-0.99-lr-0.001-eps-100-expl-0-bat-1-mayer_+_10000_-5-act-1-ppo

6.3.3 Agent Type

The last experiment compared two different agent types, being double DQN and PPO as well as using a discount factor of 0.5 [df-0.5] and a discount factor of 0.99 [df-0.99].

6.3.3.1 Prediction

The author foresees that the PPO agent performs better than the double DQN agent, as the parameters were specifically chosen for this agent. Perhaps the DQN agent finds some solutions, but the over all performance should be worse.

6.3.3.2 Result

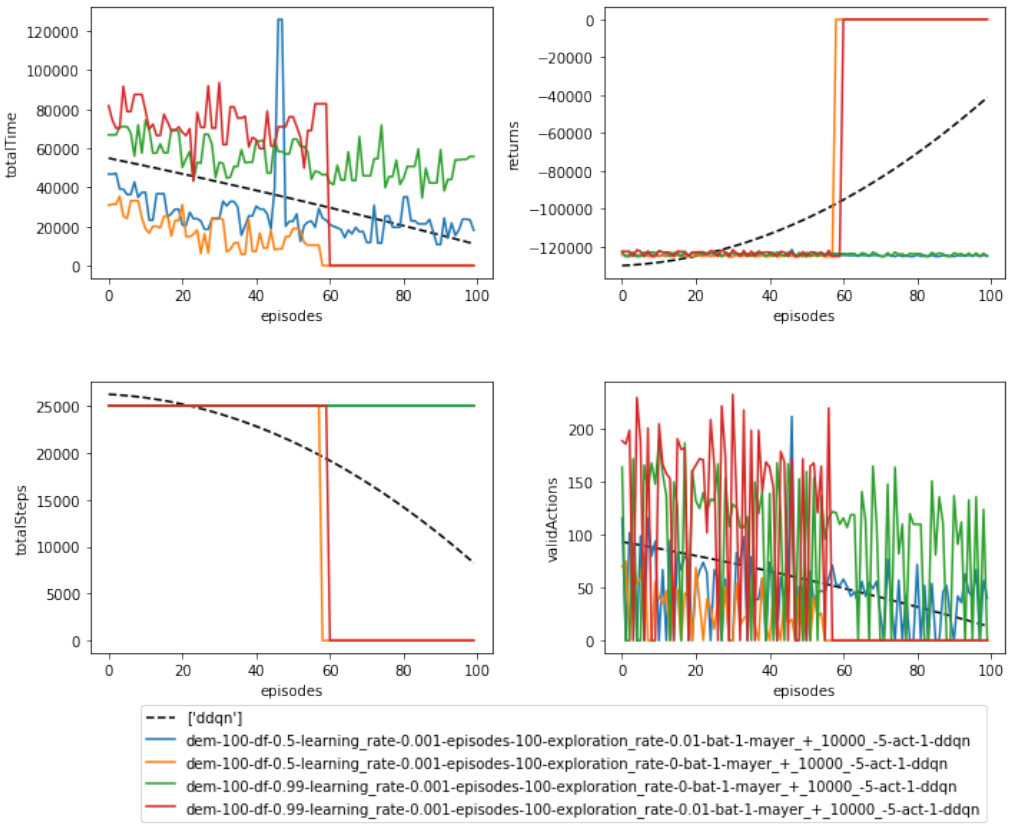

Figure 6.7: Training process of all double DQN agents

During the double DQN agents' training processes depicted in figure 6.7 it was noticeable, that no agent learned enough to distinguish valid from invalid actions. Two of them stopped early and the other two always required 25000 steps, but did not find enough valid actions to produce the demand in time. Consequently the evaluation (shown in figure 6.8) of the agents reflects that not solution was found. Probably the current parameters were not suitable for the double DQN, as these were chosen considering only the PPO. This confirms the prediction, that the combination of different parameters is important and directly influences the performance of the agent. However due to the high computation time it was not possible to extend the amount of experiments.

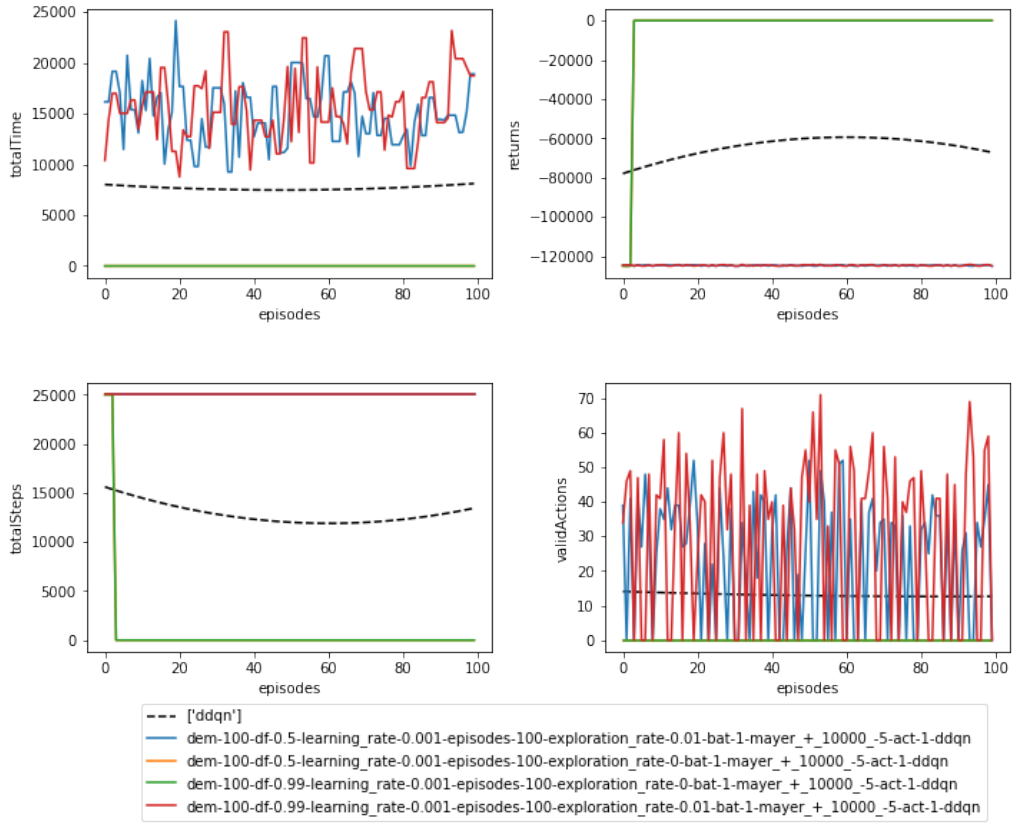

Figure 6.8: Evaluation process of all double DQN agents

Taking into account that the current configuration has been chosen for the PPO algorithm, the results - as expected - are better. Interesting is the impact of the discount factor. The training process of the PPO is shown in figure 6.9. The graphs showing *return* and *totalSteps* have been zoomed in to focus on the differences caused by the different discount factors.

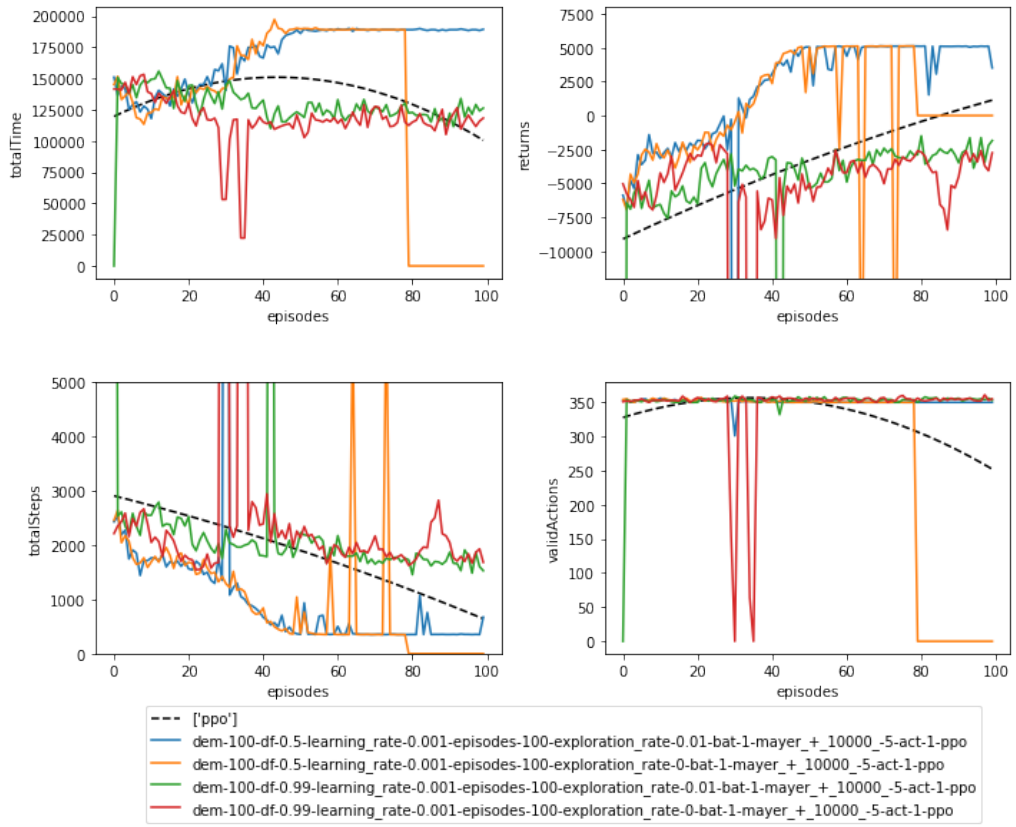

Figure 6.9: Training process of all PPO agents

Comparing the *totalSteps* of the experiments with a discount factor of 0.5 ([df-0.5]) and a discount factor of 0.99 ([df-0.99]), it is noticeable, that both graphs with the smaller discount factor led to significant decrease of the *totalSteps*. In fact the average is 1027, and the average of the experiments with the higher discount factor is 2704. It appears that a more short term decision making ([df-0.5]) lets the agent focus on reducing the steps while a more long-term decision making urges the agent to reduce the *totalTime*.

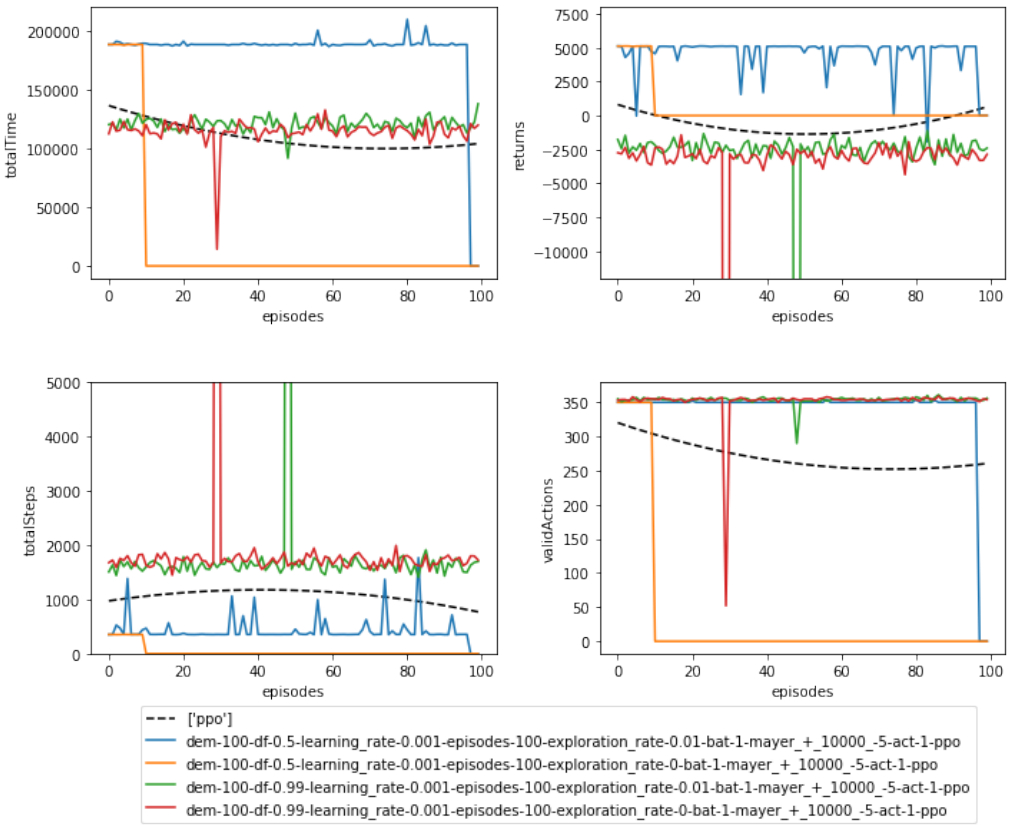

Figure 6.10: Evaluation process of all PPO agents

At last the evaluation process shown in figure 6.10 will be analysed. The yellow graph ([df-0.5,expl-0]) has stopped early which happened, because the return was three times in a row the same value. However this was not caused by high numbers of *totalSteps*, but rather a coincidence. Being aware of this fact, the experiments with a discount factor of 0.5 were very consistent. The blue graph [df-0.5,expl-0.01] returns a average *totalTime* of 183870s. Even though this is a very high value, it is impressive, that the agent on average only tried 424 *totalSteps* to find enough valid steps to finish the episode.

The experiments with the higher discount factor on the other hand had to try on average 1857 ([df-0.99, expl-0.01]) and 1950 ([df-0.99, expl-0]) *totalSteps*. However the average *totalTime* is significantly lower being 121241s and 115003s.

This shows the interesting fact, that the discount factor appears to have an influence of the focus on either the *totalTime* or the *totalSteps*. The agent seemed to classify the reduction of *totalSteps* as a short term goal, while being aware that the decrease of the *totalTime* is the long term goal.

It is to say that the configurations with a discount factor of 0.99 returned the best results. The final two best configurations are:

1. dem-100-df-0.99-lr-0.001-eps-100-expl-0.01-bat-1-mayer_+_10000_-5-act-1-ppo
2. dem-100-df-0.99-lr-0.001-eps-100-expl-0-bat-1-mayer_+_10000_-5-act-1-ppo

6.4 Conclusion

During the course of the experiments different observations have been made. The observations are not necessarily universal but can only be formulated regarding these experiments. They are summarized in the following list:

1. A small (0.001) learning size stabilizes the learning process and reduces jittering.
2. In the second experiment set it was striking that a higher exploration rate more often led to early stopping effects.
3. To improve the main objective of reducing time, it is necessary for the agent to learn to distinguish valid and invalid actions. This requires the reward function to balance the different goals accordingly. When the agent focused too much on *totalTime* (for example for *mayer_-_100_-5*), he did not learn the difference and returned bad results.
4. The author became aware of her own bias due to assumptions. It was for instance predicted, that punishing the agent for having high *totalTimes* would encourage the agent to reduce them. Instead adding the value helped the agent to learn that finding a solution and therefore having a high *totalTime* is worth it. This shows how difficult it is to not bias an agent.
5. The discount factor seemed to support the balance of the importance of the objectives. It seemed as if the agent recognized the long and

short term goal of the training. This was noticeable when a lower discount factor, so a more short term decision making, encouraged the agent to focus on valid steps.

6. Another observation is that the parameters are very specific to the applied algorithm. Even though it can not be said, whether double DQN would perform well for this scenario, the poor behaviour is probably related to the parameters supporting the PPO algorithm.

7. Approximately 350 steps are required to find a solution for this problem. This is the sum of fifty products following the processing of PartA and fifty following the processing of PartB. Even though the required time varied, the agent learned this information and often yieled valid solution within 350 to 355 actions.

Bibliography

[1] Joshua Achiam. Spinning up in deep reinforcement learning. , 2018. Last accessed: 12-01-2022.

[2] Thomas Altenmüller, Tillmann Stüker, Bernd Waschneck, Andreas Kuhnle, and Gisela Lanza. Reinforcement learning for an intelligent and autonomous production control of complex job-shops under time constraints. *Production Engineering*, 14(3):319–328, 2020.

[3] M.Emin Aydin and Ercan Öztemel. Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, 33(2):169–178, 2000.

[4] Juergen Branke, Su Nguyen, Christoph Pickardt, and Mengjie Zhang. Automated design of production scheduling heuristics: A review. *IEEE transaction and evolutionary computation*, 20(1):110–124, 2016.

[5] Balázs Csáji, Laszlo Monostori, and Botond Kádár. Reinforcement learning in a distributed market-based production control system. *Advanced Engineering Informatics*, 20:279–288, 2006.

[6] M.K. Marichelvam; P. Manimaran; M. Geetha. Solving flexible job shop scheduling problems using a hybrid lion optimisation algorithm.

[7] Peter Greschke. Matrix-produktion als konzept einer taktunabhängigen fließfertigung. pages 8–15, 97–99, 2016.

[8] Peter Greschke, Malte Schönemann, Sebastian Thiede, and Christoph Herrmann. Matrix structures for high volumes and flexibility in production systems. *Procedia CIRP*, 17:160–165, 2014.

[9] Bao-An Han and Jian-Jun Yang. Research on adaptive job shop scheduling problems based on dueling double dqn. *IEEE Access*, 8:186474–186495, 2020.

[10] Quentin J. M. Huys, Anthony Cruickshank, and Peggy Seriès. Reward-based learning, model-based and model-free. pages 1–10, 2014.

[11] Shihui Jia, Douglas Morrice, and Jonathan Bard. A performance analysis of dispatch rules for semiconductor assembly & test operations. *Journal of Simulation*, 13:163–180, 2019.

[12] Hadi S. Jomaa, Josif Grabocka, and Lars Schmidt-Thieme. Hyp-rl : Hyperparameter optimization by reinforcement learning, 2019. pages in journal unknown.

[13] Sergios Karagiannakos. Trust region and proximal policy optimization (trpo and ppo). `https://theaisummer.com/TRPO_PPO/`, 2019. Last accessed: 12-01-2022.

[14] Bangalore Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–18, 2020.

[15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[16] Andreas Kuhnle, Jan-Philipp Kaiser, Felix Theiß, Nicole Stricker, and Gisela Lanza. Designing an adaptive production control system using reinforcement learning. *Journal of Intelligent Manufacturing*, 32:866–876, 2021.

[17] Schäfer Louis Stricker Nicole Lanza Gisela Kuhnle, Andreas. Design, implementation and evaluation of reinforcement learning for an adaptive order dispatching in job shop manufacturing systems. *Procedia CIRP*, 81:234–239, 2019.

[18] Daniel Küpper, Christoph Sieben, and Kristian Kuhlmann und Justin Ahmad. Will flexible-cell manufacturing revolutionize carmaking? `https://www.bcg.com/publications/2018/flexible-cell-manufacturing-revolutionize-carmaking`, 2018. Last accessed: 12-01-2022.

[19] Dong-Ho Lee and Yeong-Dae Kim. Scheduling algorithms for flexible manufacturing systems with partially grouped machines. *Journal of Manufacturing Systems*, 18(4):301–309, 1999.

[20] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. page 157–163. Morgan Kaufmann Publishers Inc., 1994.

[21] Marvin Carl May, Lars Kiefer, Andreas Kuhnle, Nicole Stricker, and Gisela Lanza. Decentralized multi-agent production control through economic model bidding for matrix production systems. *Procedia CIRP*, 96:3–8, 2021.

[22] Sebastian Mayer, Tobias Classen, and Christian Endisch. Modular production control using deep reinforcement learning: proximal policy optimization. *Journal of Intelligent Manufacturing*, 32(8):2335–2351, 2021.

[23] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. page 8, 2013.

[25] Miguel Morales. Grokking deep reinforcement learning, 2020.

[26] Jorge Palombarini and Ernesto Martínez. Smartgantt – an intelligent system for real time rescheduling based on relational reinforcement learning. *Expert Systems with Applications*, 39(11):10251–10268, 2012.

[27] Marcel Panzer and Benedict Bender. Deep reinforcement learning in production systems: a systematic literature review. *International Journal of Production Research*, pages 1–26, 2021.

[28] Anuj Prakash, Felix Chan, and S G Deshmukh. Fms scheduling with knowledge based genetic algorithm approach. *Expert Syst. Appl.*, 38(4):3161–3171, 2011.

[29] José Salvador, João Oliveira, and Mauricio Breternitz. Reinforcement learning: A literature review (september 2020). page 10, 2020.

[30] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017. pages in journal unknown.

[31] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. pages in journal unknown.

[32] Bobak Shahriari, Kevin Swersky, Ziyun Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1).

[33] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A survey of deep reinforcement learning in video games. *CoRR*, abs/1912.10944:1, 2019.

[34] David Silver. Lectures on reinforcement learning. URL: `https://www.davidsilver.uk/teaching/`, 2015. Last accessed: 12-01-2022.

[35] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

[36] Heda Song, Isaac Triguero, and Ender Özcan. A review on the self and dual interactions between machine learning and optimisation. *Progress in Artificial Intelligence*, 8(2):143–165, 2019.

[37] C.-T Su and YR Shiue. Intelligent scheduling controller for shop floor control systems: A hybrid genetic algorithm/decision tree learning approach. *International Journal of Production Research - INT J PROD RES*, 41(12):2619–2641, 2003.

[38] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, second edition edition, 2018.

[39] Julien Vitay. Deep reinforcement learning. `https://julien-vitay.net/deeprl/NaturalGradient.html#sec:natural-gradients`. Last accessed: 12-01-2022.

[40] Bernd Waschneck, Thomas Altenmüller, Thomas Bauernhansl, and Andreas Kyek. Production scheduling in complex job shops from an

industrie 4.0 perspective: A review and challenges in the semiconductor industry. *SAMI@iKNOW*, 2016. pages in journal unknown.

[41] Bernd Waschneck, André Reichstaller, Lenz Belzner, Thomas Altenmüller, Thomas Bauernhansl, Alexander Knapp, and Andreas Kyek. Optimization of global production scheduling with deep reinforcement learning. *Procedia CIRP*, 72:1264–1269, 2018.

[42] J. Whang, G. Ding, Y. Zou, and all. Review of job shop scheduling research and its new perspectives under industry 4.0. *Journal of Intelligent Manufacturing*, 30(4):1809–1830, 2017.

[43] Shufang Xie, Tao Zhang, and Oliver Rose. Online single machine scheduling based on simulation and reinforcement learning. *Simulation in Produktion und Logistik 2019*, pages 59–68, 2019.

[44] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020.