h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbmn
FACHBEREICH MATHEMATIK
UND NATURWISSENSCHAFTEN

Hochschule Darmstadt

Fachbereiche Mathematik
und Naturwissenschaften
& Informatik

Reward Machines for Reinforcement Learning based Gantry Robot Scheduling

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M. Sc.)

im Studiengang Data Science

vorgelegt von

Patricia Christin Coberger

Matrikelnummer: 769437

Referent: Prof. Dr. Horst Zisgen
Korreferent: Prof. Dr. Frank Bühler

Ausgabedatum: 01. August 2022
Abgabedatum: 12. Januar 2023

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 12. Januar 2023

Zusammenfassung

In den vergangenen Jahren wurden Reinforcement Learning Agenten vermehrt zur Steigerung der Effizienz und Produktivität in Produktionssystemen eingesetzt. Bis dahin wurden regelbasierte Heuristiken verwendet, die jedoch bei komplexeren Problemen schnell an ihre Grenzen geraten und dadurch keine optimale Lösung finden können. Im Gegensatz dazu erzielen Reinforcement Learning Agenten durch ihre zustandsabhängige Flexibilität bessere Ergebnisse, insbesondere für komplexere Probleme. Allerdings beschränkte sich der Einsatz von Reinforcement Learning Techniken bisher größtenteils auf endliche Aufgaben. In der realen Welt steht man jedoch häufiger vor Problemen ohne klar definiertes Ende (kontinuierliche Probleme). Reward Machines beschreiben ein Konzept, das die Struktur der Reward-Funktion offenlegt und dieses Wissen nutzt, um den Trainingsprozess eines Reinforcement Learning Agentens zu beschleunigen. Dieses Konzept fokussiert sich jedoch ebenfalls hauptsächlich auf die Verwendung für endliche Aufgaben. Diese Masterarbeit untersucht, ob sich das Konzept der Reward Machines auch für kontinuierliche Aufgaben anwenden bzw. übertragen lässt. Einerseits beschäftigt sich diese Arbeit damit, ob das theoretische Konzept der Reward Machines eine Nutzung für kontinuierliche Aufgaben zulässt. Andererseits wird analysiert, ob der Algorithmus zur Beschleunigung des Trainings auch für unendliche Probleme funktioniert. Die Analyse des theoretischen Konzepts zeigt, dass Anpassungen notwendig sind, da Reward Machines als endliche Automaten nur endliche Eingabesequenzen erlauben. Deshalb sind Anpassungen insbesondere in der Definition der akzeptierenden Zustände notwendig. Diese Änderungen sind zwar notwendig, um eine korrekte Definition zu erhalten, jedoch haben sie nur wenig Einfluss auf das tatsächliche Training. Darüber hinaus werden Anpassungen am Algorithmus vorgenommen. Diese sind zwar technisch nicht zwingend notwendig, jedoch erscheinen sie durch den unendlichen Charakter kontinuierlicher Probleme sinnvoll. Diese Anpassungen werden in Form von Experimenten an Portalrobotersteuerungen, welche als kontinuierliches Problem einzustufen sind, getestet. Die Experimente zeigen, dass die Frage nach der Schnelligkeit des Lernens für die untersuchten Ansätze nicht leicht zu beantworten ist, da die Antwort maßgeblich durch die Definition von Geschwindigkeit beeinflusst ist. Einerseits verdeutlichen die Experimente, dass sowohl der ursprüngliche als auch der angepasste Algorithmus schneller eine Policy erlernen kann als ein Lernansatz ohne Reward Machines, wenn man die Schnelligkeit über die Anzahl der beobachteten Aktionen beziehungsweise die Anzahl der verwendeten Episoden zugrunde legt. Andererseits identifizieren die Experimente den Lernansatz ohne Reward Machine als effizienter in Bezug auf die Datennutzung, was auch als Aspekt der Geschwindigkeit betrachtet werden kann. Jedoch wird auch deutlich, dass der ursprüngliche Ansatz Schwierigkeiten hat, den Umgang mit selten auftretenden Zuständen zu erlernen. Diese Zustände können durch stochastische Aspekte, wie z.B. Maschinenausfälle, beeinflusst sein. Im Gegensatz dazu ist der angepasste Algorithmus in der Lage, mit solchen Herausforderungen umzugehen.

Schlagworte - Reinforcement Learning, Reward Machine, Automatentheorie, Portalrobotersteuerung, Produktionssysteme

Abstract

During the last decades, reinforcement learning agents have found their way into production systems. They have been used to increase the efficiency and productivity of a factory, as reinforcement learning agents are able to outperform rule-based heuristics due to their great state-dependent flexibility. So far, most efforts have been made to apply reinforcement learning techniques to episodic tasks. However, most real-world problems are continuous problems that need to be solved. Reward machines represent a concept that reveals the structure of the reward function and exploits this knowledge to accelerate the training process of a reinforcement learning agent. Since this concept focuses on the use in episodic tasks, this master thesis investigates whether the concept of reward machines is applicable or transferable to continuous tasks. On the one hand, the research is concerned with whether the theoretical foundations of reward machines allows the use in continuous tasks. On the other hand, it is analysed whether the algorithm for exploitating the reward machines also works for continuous tasks. The analysis of the theoretical foundations shows that adjustments are necessary because reward machines, as finite state machines, only work for finite input sequences. Therefore, adjustments are required above all in the definition of acceptance states. However, these changes are necessary but only relevant for a sound definition and have little impact on the actual training. Furthermore, changes in the algorithm are not technically necessary, but seem reasonable due to the infinite nature of a continuous task. The experiments show that the question of speed of learning for the examined approaches cannot be easily answered, since the definition of speed is relevant. The original approach as well as proposed adaptions are tested by means of experiments on the use case of gantry robot scheduling, which can be considered as a continuous problem. On the one hand, the experiments indicate that both the original and the adjusted algorithm are able to learn a policy faster than a learning approach without reward machines in terms of number of actions and the number of episodes, respectively. On the other hand, the approach without reward machines is identified as data-efficient which can be considered as a factor of speed. However, it becomes clear that the original reward machine approach has difficulty learning how to behave in rare states, e.g. influenced by stochastic aspects such as machine failures. In contrast, the adjusted approach is able to deal with these challenges.

Keywords - Reinforcement learning, reward machine, automata theory, gantry robot scheduling, production system

Contents

1 Introduction **1**
1.1 Running Example 2
1.2 Contributions 3
1.3 Structure of the Thesis 3

2 Foundations of Scheduling **5**
2.1 Gantry Robot Scheduling 5
2.2 Complexity 6

3 Automata Theory **8**
3.1 Definition of Automaton 8
3.2 Classes of Automata 9
3.3 Infinitiy in Automata Theory 9
3.3.1 Infinite Words 10
3.3.2 ω-Automata 10
3.3.3 ω-Pushdown Automata 11

4 Neural Networks as Function Approximator **12**

5 Foundations of Reinforcement Learning **13**
5.1 Definition of Reinforcement Learning 13
5.2 Problem Setup 13
5.3 Core Elements 15
5.4 Learning Approaches 16
5.4.1 TD-Learning 17
5.4.2 Deep Reinforcement Learning 19
5.5 Reward Function Design 24

6 Reward Machines **26**
6.1 Definition 26
6.2 MDP with Reward Machine 28
6.3 Approaches for Reward Machines 33
6.3.1 Counterfactual Experiences for Reward Machines 33
6.3.2 Hierarchical Reinforcement Learning for Reward Machines . . 37
6.3.3 Automated Reward Shaping 37

7 Related Work 39

8 Exploitation of Reward Machines for Continuous Tasks 47
8.1 Reward Machine Design . . . 47
8.2 Reward Machine Definition . . . 49
8.3 CRM Learning Approach . . . 51
8.4 Reward Machine Visualisation . . . 56
8.5 Summary of Adjustments . . . 58

9 Experiment 59
9.1 Experimental Setup . . . 59
9.1.1 Experimental Design and Implementation . . . 59
9.1.2 Production Environment . . . 62
9.2 Reward Machine . . . 68
9.2.1 States . . . 68
9.2.2 Acceptance Criteria . . . 72
9.2.3 Input Alphabet . . . 73
9.2.4 State-Transition Function . . . 75
9.2.5 State-Reward Function . . . 75
9.3 Application of Learning Approaches . . . 76
9.3.1 Adjusted Reward Machine Approach . . . 76
9.3.2 Original Reward Machine Approach . . . 86
9.3.3 Basic Learning Approach without Reward Machine . . . 94
9.3.4 General Observations over all Learning Approaches . . . 103

10 Discussion 106

11 Conclusions 113

Appendix 116

List of Figures 124

List of Tables 126

Bibliography 128

1 Introduction

Nowadays robots play an important part in manufacturing. They are used for different aspects of the production system such as processing and material handling. These robots facilitate a higher automation of the manufacturing process. At first, heuristics were applied to control robots for material transportation. However, in order to increase the efficiency and productivity of a factory, a rather complex and sophisticated control and scheduling system is essential. In complex manufacturing systems scheduling rules based on heuristics may not lead to optimal results. However, machine learning techniques like reinforcement learning offer an opportunity to cope with these challenges.

During the last decade, efforts have been made to apply reinforcement learning in production scheduling problems. Further, advances in reinforcement learning research led to improvements in the application in production context.

However, an important aspect in reinforcement learning is the design of the reward function. It affects not only the quality of a policy but also how fast such a policy can be discovered.

Moreover, most reinforcement learning problems tackled in the context of production scheduling dealt with finite tasks. A typical example is the execution planning of previous known jobs. As soon as these jobs are done, the reinforcement learning problem is solved. Especially in production systems, this is far from reality. Production companies want to be able to respond to dynamic situations such as jobs that are not known in advance. Further, they do not stop production after one batch of jobs and start planning a new batch, but theoretically continue production forever.

This master thesis concentrates on finding optimal policies faster using reward machines for a continuous task in the context of production scheduling, in particular gantry robot scheduling. Here, faster refers to a comparison with standard learning approaches.

Reward machines introduced by Icarte et al. [17] represent an approach that addresses the problem of designing reward functions and exploiting their structure in order to find an optimal policy faster. Since this approach focuses on episodic tasks and covers continuous tasks only slightly, this master thesis investigates the suitability of the general reward machine concept to continuous tasks. Furthermore, the applicability of algorithms developed for the use of reward machines to complex continuous problems is analysed. These applicability aspects are examined by means of experiments on a use case related to gantry robot scheduling.

1.1 Running Example

Figure 1.1 visualises the setting of a production system which employs gantry robots for material transport. This simple example is used to motivate the relevance of using reward machines for gantry robot scheduling.

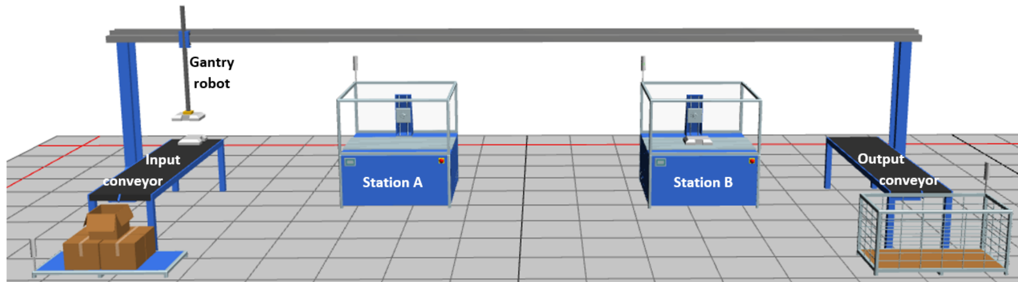


Figure 1.1: Example of a gantry robot in a production context for material transportation.

The presented example consists of an input conveyor, two machines executing the same processing steps independently, an output conveyor and a gantry robot. The example production system proceeds as follows: An element arrives via the input conveyor in the production system in order to perform an operation like drilling a hole. This processing step can be either executed by station A or station B, but both machines work in the same way. To process the operation, the element must be carried from the input conveyor to any of these machines. When a machine has completed the operation, the finished element must be passed on the output conveyor. The gantry robot must move from its current position to the input conveyor or machine to deliver the goods to the next station. Since elements can be waiting for handling at each station, scheduling this gantry robot becomes a complex task which can be solved by reinforcement learning.

Reinforcement learning requires positive or negative rewards for actions depending on the current state. In the given example, a reinforcement learning agent could collect a reward for each element that correctly reaches the output conveyor. In this case, however, the agent would receive information about its behaviour only at the end. Because a reinforcement learning agent learns a policy through a trial-and-error process, the agent has to explore many different states and actions to reach a state in which it receives a reward. A final reward can only be achieved if the element has been processed and delivered to the output conveyor. However, this state can be only observed if all necessary pre-steps have been performed. The item has to be picked up at the input conveyor, then a gantry has to carry the object to station A or station B, unload it and pick it up after completing processing. Then, the gantry has to deliver the product to the output conveyor and perform an unloading. Thus, many different actions have to be performed to achieve at least a throughput of one in a given time. Therefore, training by trial-and-error takes a long time and may also lead to suboptimal policies. However, instead of using

trial-and-error to figure out which action will yield a high reward in a given state, the general concept of the reward function could be revealed and consequently its internal structure could be exploited. This could accelerate the learning process and still lead to an optimal policy. Reward machines describe such a way to find an optimal policy faster.

In the remainder of this thesis, an approach is described that focuses on determining an optimal scheduling strategy of gantry robots more quickly, as illustrated by the running example.

1.2 Contributions

This thesis contributes to the use of reward machines for continuous reinforcement learning problems. On the one hand, the applicability of the existing concept of reward machines for continuous tasks is investigated and difficulties are identified. On the other hand, this work extends existing approaches to provide useful insights for learning reward machines for continous tasks.

1.3 Structure of the Thesis

The remainder of this master thesis is structured as follows:

- Chapter 2 provides a brief introduction into scheduling and gantry robots since this is the main application field of this thesis. Furthermore, a brief look is taken at the assignment of gantry robot scheduling to complexity classes.
- Chapter 3 summarises automata theory which provides a foundation for the theoretical concept of reward machines. Moreover, because of the connection with the definition of reward machines, a consideration of infinity in automata theory is already made in preparation for a later discussion of continuous tasks in reward machines.
- Chapter 4 briefly introduces the ability of specific neural networks to serve as function approximators, which is a relevant characteristic for deep reinforcement learning.
- In Chapter 5, reinforcement learning lays out the core concept of this thesis.
- Chapter 6 presents the second core concept of this thesis - reward machines.
- In Chapter 7, this master thesis is placed in the context of existing work on reinforcement learning in production scheduling. Furthermore, the state-of-the-art of reinforcement learning in automated guided vehicle scheduling in general and gantry robot scheduling in particular is summarised.

- Chapter 8 explains why the fundamental reward machine concept is not applicable for complex continuous tasks and how it has to be adapted to work for complex continuous tasks as well.
- Chapter 9 illustrates what production environment is used to demonstrate the capabilities of the modified approach. Moreover, approaches with and without reward machines are tested by means of experiments.
- The results derived from the experiments will be discussed in Chapter 10. Furthermore, limitations of the approach and the experiments are explained.
- Chapter 11 summarises the adjusted reward machine approach for complex continuous tasks and outlines meaningful findings. Finally, an outlook for future work is given.

2 Foundations of Scheduling

Scheduling describes the problem of allocating resources like machines to tasks. In this process, the allocation is optimised regarding a given objective. In general, Pinedo et al. [33, pp. 1] identify scheduling decisions as the main success factors for the performance of production systems.

Pinedo et al. [33, p. 13, 245] distinguish two different production models. On the one hand, deterministic models describe a finite number of jobs and machines. On the other hand, stochastic models consider uncertainties as they occur in real world production systems. According to Wang et al. [42], these uncertainties can cause machine breakdowns and variance in operation and transportation times.

Production environments can be designed differently and therefore vary in complexity. Pinedo et al. [33, pp. 13] describe seven different scheduling problems, four of which deal with shop scheduling. A flow shop scheduling problem specifies a production system in which each job has to be processed on each machine in the same order. A more general variant is flexible flow shop scheduling. This production environment consists of work centers with multiple machines executing the same operations in parallel. In contrast to flow shop scheduling, job shop scheduling allows different machine sequences for each job. Further, flexible job shop scheduling describes an extension of job shop by introducing work centers as in flexible flow shops.

Flexible job shop systems are considered as the most difficult systems since they produce many different products of different batch sizes. Furthermore, according to Wang et al. [42], stochastic models complicate these problems due to uncertainties. Therefore, assumptions are often made for simplicity. In research, for example, it is often assumed that machines are available without any failures. Furthermore, a common simplification consists in ignoring any transportation time. While schedules indicate which job is to be executed next on a particular machine, they do not address how a job is delivered to that particular machine.

2.1 Gantry Robot Scheduling

Gantry robots belong to the group of serial robot systems. A relevant aspect in robotics consists in the degrees of freedom. Siegert et al. [37, p. 20] define degrees of freedom of a robot as number of possible independent movements. According to Pott et al. [35, p. 19], a gantry robot has three degrees of freedom. Such a robot can move along a portal (x-axis), move up and down (y-axis), and use an arm to pick up or unload elements

(z-axis).

Since gantry robots are installed to supply machines with items, this system is particularly suitable for multi-machine processes. However, this requires all machines to be arranged in such a way that the robot can operate them from the gantry.

Gantry robot scheduling is about which machine to serve next to achieve a given goal. A gantry robot scheduling problem belongs to the class of production scheduling problems. However, in contrast to previously described scheduling problems, gantry robot scheduling takes material handling into account. These scheduling problems do not need to ignore transportation subjects. However, this impacts the performance of a production system. Ou et al. [28] note that in systems that use gantry robots for material handling, the performance of the system depends not only on the machines but also on the actions of the gantry.

Like general production scheduling tasks, gantry robot scheduling can be part of flexible job shops. Ou et al. [28] extend a regular production system with gantry robots as described in the following. Such a production system can be divided into multiple work cells, each of which consists of several machines that process the same operation in parallel or different operations in series. Each work cell includes at least one gantry robot. A gantry robot is employed to load material into a machine, unload a processed part from it, and transfer this part to another machine or conveyor. A machine is blocked as long as an element is present. Only when a gantry robot picks up this element the machine can be loaded with a new unprocessed part.

Running Example The running example as described above constitutes a gantry robot production system with one work cell and one gantry robot. Furthermore, the running example can be classified as a flexible flow shop scheduling problem, since only one type of processing is considered, which can be performed on two different machines in parallel.

2.2 Complexity

Complexity indicates how many resources are needed to solve a problem algorithmically. Problems and their algorithms are divided into different classes based on their complexity. Problems of complexity class *NP-hard* are considered to be non-deterministic solvable in polynomial computing time.

MacCarthy et al. [21] assign job shop scheduling problems to the class of *NP-hard* complexity. Methods that are able to produce an optimal schedule work only for small instances in polynomial time. Furthermore, heuristics approximate optimal solutions in polynomial time.

Heger et al. [13] consider gantry robot scheduling as a more complex optimisation problem than job shop scheduling. This results from the additional handling of material transport within the general scheduling problem. Therefore, gantry robot scheduling is

also considered to be *NP-hard*. Dynamic aspects in an environment, e.g. machine failures, lead to a further increase in complexity.

3 Automata Theory

Since the concept of reward machines is based on automata theory, a brief overview of key definitions is given. First, this area is placed in the context of computer science. Then, a selection of relevant terms from the field of automata theory is defined. To provide a solid basis for the later discussion on the applicability of continuous tasks in reward machines, a special look at infinity in automata theory is given.

Vossen et al. [41, p. 1] classify automata theory in the field of theoretical computer science. Automata formally describe machines and how they operate. Such an automaton analyses whether or not a given input sequence is accepted.

3.1 Definition of Automaton

According to Hopcroft et al. [15, pp. 38], an automaton A can be formally defined over a tuple of five elements $(Q, \Sigma, \delta, q_0, F)$.

- Q describes a set of states of the automaton. A finite state machine holds only a finite set of Q.
- The alphabet Σ represents a finite, non-empty set of symbols. The alphabet describes what input symbols are accepted by a given automaton. Σ^* describes a set of finite words of this alphabet.
- The transition function of a given automaton is denoted as $\delta : Q \times \Sigma \rightarrow Q$. This function maps a tuple of state and input sequence to a state of the automaton.
- The initial state of an automaton is marked as q_0.
- The set of accepting states of the automaton is represented by F which is a subset of Q. An accepted state is reached if an input sequence starting at the initial state transitions to a state of F.

An automaton receives a sequence w, also called a word, as input which consists of symbols of Σ. The set of accepted input sequences defines the accepted language of a given automaton.

3.2 Classes of Automata

Vossen et al. [41, pp. 11] divide automata into different groups according to the complexity of the language they can represent.

A finite state machine limits Q to a finite set of states. Thus, the language this type of automaton can recognise is called a regular language. Additionally, a regular language can be specified by a regular expression. In general, there are two types of finite state machines which can be distinguished by the nature of their transition function. A deterministic finite state machine admits only one possible subsequent state, while a non-deterministic finite state machine accepts different following states despite the same initial state and the same input sequence. Further, Vossen et al. [41, pp. 118] introduce a special class of finite state machines that contain an output function $\lambda : Q \times \Sigma \to \Omega$. This output function assigns an output to each tuple of state and input. A *Mealy automaton* [41, pp. 122] is a deterministic finite state machine with such an output function. In contrast, a *Moore machine* [41, pp. 132], also a deterministic finite state machine, has an output function $\lambda : Q \to \Omega$ that determines the output based only on a state and not an input.

A pushdown automaton constitutes an extension of a non-deterministic finite state machine. It comes with a stack that can store input sequences of stack symbols. This gives a pushdown automaton the ability to store information in last-in-first-out order. With each input symbol, the top element of the stack is read and a stack symbol is written back to the stack. Formally, the general definition of an automaton must be extended to include a stack alphabet Γ and an initial stack symbol $\perp$. Furthermore, the transition function δ must be adapted to account for the stack. Thus, the transition function of a pushdown automaton is defined as $\delta : Q \times \Sigma \times \Gamma \to 2^{Q \times \Gamma}$. This modified δ provides for the special handling of the stack when reading an input symbol. The pushdown automaton allows to recognise context-free languages which describes a more complex language than the one recognised by finite state machines.

The turing machine describes a category of automata capable of recognising more advanced languages than a finite state machine and a pushdown automaton. However, this type of automaton goes far beyond what is necessary for this work. Therefore, we will not go into further details.

3.3 Infinitiy in Automata Theory

All these automata have in common that they only accept finite words. However, infinity is a relevant aspect in automata theory since some systems are not supposed to terminate. Furthermore, an automaton which can evaluate the success only after processing a whole word cannot work in the same way with infinite words, as they have no end and hence never reach a terminal state. Therefore, a specific class of automata has to be considered, which make adjustments necessary, especially with regard to the acceptance criteria.

3.3.1 Infinite Words

According to Thomas [40], an infinite word, also called ω-word, is defined like a finite word over a finite alphabet Σ. However, since Σ is finite, it follows that an infinite word must contain at least one symbol that occurs infinitely often. Moreover, in the context of words, infinite strictly means countably infinite.

The set of infinite words is denoted as Σ^ω and $L \subseteq \Sigma^\omega$ represents the ω-language consisting of infinite words.

While finite words can be concatenated and still belong to the group of finite words, this is not true for infinite words. Hofmann et al. [14, pp. 61] explain that concatenation of an infinite word with any other word (finite or infinite) is not possible because the end of the first word is never reached and therefore no transition to the second word can take place. However, the only possible concatenation with an infinite word consists in the concatenation of a finite and an infinite word, because the first word, being a finite input sequence, can reach its last symbol and thus a transition to the second word can occur. This concatenation would result in an infinite word.

3.3.2 ω-Automata

An ω-automaton extends the finite state machine for infinite input sequences. The formal definition of an ω-automaton differs slightly from the general automaton definition presented above. Farwer [8] defines ω-automata as a quintuple $(Q, \Sigma, \delta, q_0, Acc)$. Hereby, Q, Σ, δ and q_0 are similar to the general automata definition. Like finite state machines, an ω-automaton can have a deterministic or non-deterministic transition function δ. However, while F as set of accepting states was part of the automata definition above, ω-automata require with Acc a more general acceptance component.

Farwer [8] distinguishes different acceptance components. Most of these definitions require a function $inf : \Sigma^\omega \to 2^Q$ that receives a word w as input and returns a set of states that are visited infinitely often for that input word.

Büchi Acceptance The Büchi acceptance component [8, pp. 5] is a set of states F. It follows that $Acc = F$. A word $w \in \Sigma^\omega$ is accepted by the automaton A only if there is at least one element in F that is visited infinitely often by the input word w. The Büchi acceptance condition can be summarised as follows:

$$\exists f \in F : f \in inf(w)$$

Muller Acceptance The Muller acceptance criteria [8, pp. 6] is a set of state sets $\mathcal{F} \subseteq 2^Q$ which is a subset of the power set of Q. An infinite input sequence w is accepted by the automaton A only if the set of infinitely recurring states at the sequence of w is an element in $\mathcal{F}$. This can be formalised as:

$$inf(w) \in \mathcal{F}$$

Rabin Acceptance The Rabin automaton [8, pp. 8] accepts infinite words that satisfy a pairs condition. The acceptance component is defined as pair of two sets. The acceptance set $\Omega = \{(E_1, F_1), ..., (E_k, F_k)\}$ with $E_i, F_i \subseteq Q$ distinguishes accepting and rejecting states. From this it follows that a word w is accepted by the automaton if there is a pair (E_i, F_i) in which the elements of E_i do not occur infinitely often in w and elements of F_i belong to the infinitely recurring states in w. This can be formalised as:

$$\exists(E_i, F_i) \in \Omega : (inf(w) \cap E_i = \emptyset) \wedge (inf(w) \cap F_i \neq \emptyset)$$

As a contrast to the Büchi and Muller acceptance criteria, the Rabin automaton defines not only accepting but also rejection criteria.

Streett Acceptance The Streett automaton [8, pp. 9] is generally defined similarly to the Rabin automaton, however it differs in its acceptance condition. The Streett automaton accepts only those words w for which each pair of (E_i, F_i) satisfies the condition that if states in F_i are visited infinitely often then there have to be states in E_i that occur infinitely often as well. This can be formally summarised as follows:

$$\forall(E_i, F_i) \in \Omega : (inf(w) \cap F_i \neq \emptyset) \rightarrow (inf(w) \cap E_i \neq \emptyset)$$

Parity Acceptance A Paritiy automaton [8, pp. 9] extends the Rabin automaton by the requirement of set inclusion. That means each E_i has to be a real subset of F_i while all elements of F_i in turn must also be elements in E_{i+1}. It follows that $E_1 \subset F_1 \subset ... \subset E_m \subset F_m$. Furthermore, a priority is assigned to each state depending on its belonging to E_i, F_i. This priority value depends on how many sets of states E_i, F_i contain this certain state. States in E_1 receive the lowest priority value because it occurs in all set of states. On the other hand, states that are only included in the biggest superset F_m obtain the highest priority value k. In general, this assignment can be expressed by function $c : Q \rightarrow \{1, ..., k\}$ that maps a state of the automaton to a number. A word w is considered valid only if the minimal $c(q)$ of all states q recurring infinitely often in w is even. Formally, this acceptance condition requires that

$$min\{c(q) | q \in inf(w)\} \text{ is even}$$

All of these acceptance criteria are interchangeable and all recognise the same ω-language.

3.3.3 ω-Pushdown Automata

Cohen and Gold [6] have extended the general pushdown automaton for infinite input sequences. The result is an ω-pushdown automaton capable of recognising context-free ω-languages. Similar to ω-automata, the main extension consists of adjusting accepting states. Thus, F defines which states mark terminal states. In addition, the same acceptance criteria as for the ω-automaton can be applied.

4 Neural Networks as Function Approximator

Artificial neural networks play an important part in the field of artificial intelligence and were theoretically formalised in 1954 by Marvin Minsky for the first time. Du et al. [7, pp. 83] define feedforward networks as specific subset of artificial neural networks that use only inputs from the previous layer to compute the output value of the current perceptron. Single-layer and multi-layer networks belong to this class of artificial neural networks. While single-layer networks consist only of an input and an output layer, multi-layer networks contain at least one so-called hidden layer in between.

Du et al. [7, pp. 83] identify function approximation and classification of patterns as key goal of multi-layer feedforward networks. The activation function also plays an important role. A non-linear activation function is required to approximate non-linear functions by the neural network. Otherwise, a linear combination would take place over several layers, which corresponds to a linear function.

Especially multi-layer feedforward networks are crucial for function approximation. While single-layer feedforward networks can only represent linear functions, no matter which activation function is used, Goodfellow et al. [11, pp. 164] emphasise the strength of multi-layer feedforward networks through their ability to learn non-linear functions as well.

In 1989, Hornik et al. [16] proved that multi-layer feedforward networks are a class of universal approximators. This means that any function from one finite-dimensional space to another one can be approximated by a feedforward network. However, the degree of accuracy depends on the architectural design of such a neural network. This includes the number of hidden layers as well as the amount of units per layer.

Since reinforcement learning problems require function approximations for huge state spaces, multi-layer feedforward networks are often employed as universal function approximators. Details on the use of neural networks in reinforcement learning are explained in the next chapter.

5 Foundations of Reinforcement Learning

Bishop et al. [3, p. 3] split the field of machine learning into supervised, unsupervised and reinforcement learning. While supervised learning requires a training dataset consisting of features and a target in order to predict a category or a continuous value, an unsupervised learning problem focuses on revealing groups of similar elements from a dataset. In contrast, reinforcement learning does not need any data beforehand. Instead it receives information by interacting with an environment.

5.1 Definition of Reinforcement Learning

According to the standard work on reinforcement learning by Sutton and Barto [39], reinforcement learning trains an agent how to act in different situations in order to achieve a certain goal. In doing so, the agent pursues the maximisation of a returned reward. However, no strategy is implemented by a developer for this. Therefore, the agent must learn which actions are beneficial through a trial-and-error search. Since the state-space is rather unknown at the beginning, the agent has to find out which actions are most rewarded in certain states. However, the impact of actions can only be assessed in the long run.

The question of whether the best choice is to follow a known action with the highest current reward or to explore more unknown states to find better actions represents an important trade-off in reinforcement learning, called the exploration-exploitation trade-off. Often a so-called ε-greedy approach[1] is used to achieve this compromise.

After training, the reinforcement learning agent obeys a policy derived from the observed rewards.

5.2 Problem Setup

Figure 5.1, extracted from [39, p. 48], visualises the setup of a reinforcement learning problem. Such a reinforcement learning problem describes the interaction between an agent and an environment. At each time step t, an agent chooses an action A_t based on the current state of the environment S_t. The chosen action affects the environment and leads

[1] The ε-greedy approach chooses a random action in ε percent of the cases. Often an ε-decay is used to realise a shrinking ε-value.

to a transition to state S_{t+1} with a certain transition probability. Furthermore, a numerical reward R_{t+1} is computed based on the observed state-action-state transition (S_t, A_t, S_{t+1}). As a consequence, the agent receives the subsequent state S_{t+1} and the computed reward R_{t+1} as input.

After one iteration, the interaction continues with the next action chosen by an agent. The choice of these actions follows a policy which guides the agent to a maximum return. However, the agent must be properly trained until it has developed a policy that can guide it accordingly.

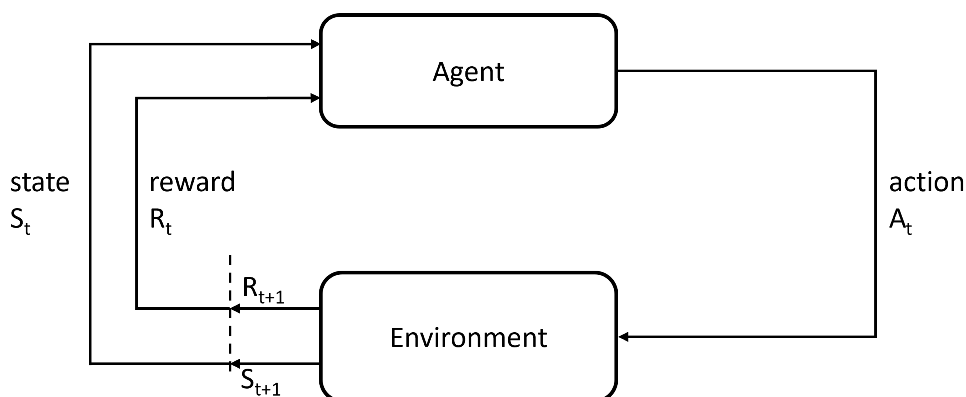


Figure 5.1: The agent-environment interface [39]

Li [19, p. 13] explains that this interaction either goes on forever, which defines a process as continuous, or it ends when a terminal state of the environment is reached. In the latter case, the reinforcement learning problem is called episodic.

Due to the characteristics of a reinforcement learning problem, Sutton and Barto [39, p. 48] indicate that the next state depends only on the current one, but no previous states. Thus, reinforcement learning problems satisfy the Markov property which implies that only the current state determines the next state. By fulfilling this requirement, reinforcement learning in general can be formalised as a Markov Decision Process (MDP). MDPs are defined as a tuple

$$MDP = (S, A, R, p)$$

where S describes the set of states and A the set of actions. The reward function R maps the current state, the chosen action and the resulting state on a numerical value. So, an agent receives a reward at each time step. Furthermore, the probability function p represents the state-transition probabilities.

Running Example For the running example, an agent-environment interaction as shown in Figure 5.1 can be given. As an example interaction, a ride from the input conveyor to station B can be considered. The location of the loader forms the state of the environment S_t, in this case the input conveyor. The chosen action A_t *Move to station B*

implies a change of state from the current state S_t (input conveyor) to the loader's new location and thus to the state S_{t+1} (station B). For this transition, the agent receives a predefined reward R_{t+1}.

The agent of the given example selects the next action either randomly from the entire action space or based on previous observations regarding the current state, which corresponds to the policy derived so far.

The given example can be viewed as both an episodic and a continuous reinforcement learning problem. In the episodic case, the task would be defined as the processing of e.g. one job. Once this job is processed and reaches the output conveyor, the task is considered completed. In the continuous case, on the other hand, not only one job but an infinite and previously undetermined number of jobs are considered.

5.3 Core Elements

In order to understand how an agent is trained, further elements of reinforcement learning need to be introduced. The formulas as presented in the following align to the definitions and equations used by Sutton and Barto in [39, pp. 54 - 64].

The training of an agent aims to learn a policy π. Such a policy provides guidelines on how to act in a given situation. As Equation (5.3.1) illustrates, a policy corresponds to the probability of choosing an action A_t if a state S_t is given.

$$\pi(S_t) = P(A_t|S_t) \tag{5.3.1}$$

However, the training aims not only to learn any policy, instead an optimal policy shall be discovered. A policy is considered optimal if it leads to the greatest expected return. The return G_t describes the cumulated sum of rewards (Equation (5.3.2)).

$$G_t = \begin{cases} \sum_{k=0}^{T} R_{t+1+k}, & \text{if episodic} \\ \sum_{k=0}^{\infty} \gamma^k \cdot R_{t+1+k}, & \text{if continuous} \end{cases} \tag{5.3.2}$$

In case of a continuous reinforcement learning problem, a discount rate $\gamma < 1$ is necessary to ensure the convergence of an infinite reward sequence. In addition, discounting evaluates all future rewards at time step t.

Further, the return is used to quantify how valuable a certain state is expected to be. This evaluation takes place at two successive points in time. The state-value function V_π calculates the expected return G_t of a certain state S_t assuming a policy π.

$$V_\pi(S_t) = E_\pi[G_t|S = S_t] \tag{5.3.3}$$

$$V_\pi(S_t) = E_\pi[R_{t+1} + \gamma \cdot G_{t+1}|S = S_t] \tag{5.3.4}$$

$$V_\pi(S_t) = E_\pi[R_{t+1} + \gamma \cdot V_\pi(S_{t+1})|S = S_t] \tag{5.3.5}$$

Equations (5.3.3), (5.3.4) and (5.3.5) illustrate that applying the definition of the return and the expectancy value on the return of the next state, the state-value function can be considered as a recursive formula.

Furthermore, the evaluation can also be made after an action is already chosen. The action-value function $Q_\pi(S_t, A_t)$ estimates a value with similar meaning as the state-value function. While the state-value function determines the expected return in a certain state before taking a certain action, the action-value function considers the action A_t in a given state S_t as fixed.

$$Q_\pi(S_t, A_t) = E[G_{t+1} | S = S_t, A = A_t] \tag{5.3.6}$$

$$Q_\pi(S_t, A_t) = E[R_{t+1} + \gamma \cdot G_{t+1} | S = S_t, A = A_t] \tag{5.3.7}$$

$$Q_\pi(S_t, A_t) = E[R_{t+1} + \gamma \cdot Q_\pi(S_{t+1}, A_{t+1}) | S = S_t, A = A_t] \tag{5.3.8}$$

Like the state-value function, the action-value function also works recursively. Both recursive formula are called the Bellman equation.

In addition to evaluating the quality of a state or state-action pair, an optimal value function (state-value or action-value function) can be determined independently of any policy. The optimal value function holds the greatest expected return. Due to the self-consistency condition of the Bellman equation ((5.3.5) and (5.3.8)) an optimal value function of state S_t has to be based on the optimal value function of the next state S_{t+1} (Bellman optimality equation).

It follows that an optimal policy is based on an optimal state- and action-value function, and vice versa. Therefore, an optimal policy can be easily derived from a given optimal value function, and the optimal value function can be calculated using a given optimal policy. However, often neither a policy nor state-values are given.

There are many different ways of identifying an optimal policy with benefits for different reinforcement learning problems. Since this thesis focuses on designing a reward function by means of reward machines, only Q-Learning and Deep-Q-Networks (DQN) are examined in more detail.

5.4 Learning Approaches

Learning methods are used to find an optimal policy. These approaches require different levels of prior knowledge of an environment. Sutton and Barto [39, p. 23, 73] distinguish between model-based and model-free learning methods. A model-based learning method uses a model of the environment to plan the environments' behaviour. For example, a dynamic programming approach postulates a model of the environment and the transition probabilities. In contrast, a model-free learning method determines an optimal policy through trial-and-error. The temporal-difference (TD) learning approach, for example, updates values episodically based on values one time step ahead.

In the following, Q-Learning as a tabular learning algorithm and a variant of TD-Learning is presented. As an extension, Deep-Q-Networks (DQN) are introduced which approximate Q-values.

5.4.1 TD-Learning

In [39, pp. 119], temporal-difference (TD) learning is presented as a combination of Monte-Carlo learning with dynamic programming. While the Monte Carlo technique formalised in Equation (5.4.1) requires receiving a final outcome G_t to estimate the state-value function of a certain state, the dynamic programming approach from Equation (5.4.2) calculates the estimates based on the estimated value of the next state. However, dynamic programming requires knowledge of the transition probabilities in order to estimate the state-value function, as $p(S_{t+1}, R_{t+1}|S_t, A_t)$ in Equation (5.4.2) indicates.

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)) \tag{5.4.1}$$

$$V(S_t) \leftarrow \max_{A_t} \sum_{S_{t+1}, R_{t+1}} p(S_{t+1}, R_{t+1}|S_t, A_t)[R_{t+1} + \gamma \cdot V(S_{t+1})] \tag{5.4.2}$$

A merge of both of these methods results in the TD-Learning approach. This method uses the estimate of the next step - like in dynamic programming - and applies it to the Monte Carlo state-value function. Like the Monte Carlo state-value function, the TD-Learning approach in Equation (5.4.3) does not take into account the entire temporal difference error (TD-error) defined as $R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)$. Instead this TD-error is shrunk by a learning rate α, as Winder explains in [45, pp. 60].

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)] \tag{5.4.3}$$

The use of the state-value function V at two different points in time - t and $t+1$ - , as can be seen in Equation (5.4.3), gives the TD-learning approach its name.

Furthermore, Sutton and Barto [39, pp. 119] emphasise that TD-Learning is an iterative process and improves the estimation over multiple episodes.

5.4.1.1 Q-Learning

The Q-Learning approach developed in 1989 by Watkins [44] can be described as an extension of TD-Learning. It transfers the idea of TD-Learning from the state-value function in Equation (5.4.3) to an action-value function, as one can see in Equation (5.4.4). Similar to general TD-Learning, Q-values are updated by a delta which represents the difference between the actual value $R_{t+1} + \gamma \cdot \max_{A_{t+1}} Q(S_{t+1}, A_{t+1})$ (target value) and the previous estimate $Q(S_t, A_t)$. Furthermore, the actual value results from the optimal Q-value of the next step which represents a greedy approach. However, the optimal value

depends on the chosen action. Looking one-step ahead and choosing the next action greedy and not based on a given policy π defines Q-Learning as an off-policy approach.

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \cdot \underset{A_{t+1}}{max} Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)] \quad (5.4.4)$$

The detailled procedure of how Q-Learning is implemented is summarised in Algorithm 1 [45, p. 62]. Since the estimate of each state-action pair is stored in a table (line 2 in Algorithm 1), Q-Learning can be classified as tabular method. Moreover, this characteristic constrains the Q-Learning approach to reinforcement learning problems of discrete state- and action-spaces. First, in line 6 in Algorithm 1 an action is chosen according to the ε-greedy approach. The ε-greedy approach balances the trade-off between exploration and exploitation. A random selection occuring ε-times enables an exploration of unknown states. However, depending on the trainings' progress exploration becomes less important. In that case, the next action A_t is chosen by the maximal Q-value for the given state S_t as $\underset{A_t}{max} Q(S_t,A_t)$. The actual Q-Learning as explained in Equation (5.4.4) takes place in line 8-11. As line 5 in Algorithm 1 indicates, the current episode runs until a terminal state in the environment is reached and thus, the task is completed. While this loop-condition corresponds to the exit condition of an episodic task, this would not work for a continuous task. In a continuous task, the episode could therefore terminate after a certain period of time.

Algorithm 1 Q-Learning algorithm [45, p. 62]

1: Algorithm parameters: policy π, learning rate $\alpha \in (0,1]$, discount rate $\gamma < 1$
2: Initialise $Q(s,a)$, for all $s \in S, a \in A$
3: **for** each episode **do**
4: Initialise state S_t
5: **while** S_t is not terminal **do**
6: Choose action A_t accoring to ε-greedy approach
7: Observe reward R_{t+1} and subsequent state S_{t+1}
8: **if** S_{t+1} terminal **then**
9: $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} - Q(S_t,A_t)]$
10: **else**
11: $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \cdot \underset{A_{t+1}}{max} Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)]$
12: **end if**
13: $S_t \leftarrow S_{t+1}$
14: **end while**
15: **end for**

5.4.2 Deep Reinforcement Learning

Deep Reinforcement Learning applies deep learning methods such as neural networks to reinforcement learning problems as explained by Winder in [45, p. 91]. Especially when the state-space becomes overly complex, deep reinforcement learning comes into play. In this case, a tabular method like Q-Learning reaches its limits and approximate solution methods become necessary. An approach approximating a function receives a subset of the state-space and uses the results to estimate values for unknown states. According to Sutton and Barto [39, pp. 195], the ability to generalise beyond observed behaviour is a key success factor.

However, Mnih et al. [26] emphasise that deep reinforcement learning often suffers from non-linear function approximations that compute Q-values. These result in instability and divergence as common problems. In addition, function approximations can be considered as supervised learning problems that assume independent inputs. Yet, reinforcement learning generates a sequence of states and thus correlated inputs. Furthermore, in reinforcement learning, function approximation must estimate not only the predicted Q-value but also the true Q-value (target), whereas in regular supervised learning problems the true target data is available and fixed in advance. The approximation of the predicted Q-value and the true Q-value is achieved using the same approximation function. However, this leads to a correlation between predicted Q-values and target values. A third difficulty also arises from the unavailable knowledge of true Q-values. Since the target value must be approximated and the same approximation function is used as for the predicted values, changes in the approximation function also affect the target values. This problem is often referred to as moving target.

In the following, Deep-Q-Networks (DQN) will be introduced which address these challenges.

5.4.2.1 Deep-Q-Networks

Deep-Q-Networks (DQN) developed in 2013 by Mnih et al. [26] represent a major advancement in the field of reinforcement learning. DQNs employ deep neural networks as a non-linear approximation in order to receive estimates of the action-value function. It represents a variant of the Q-Learning approach with the difference that DQNs can handle a continuous state-space. In contrast to previous applications of neural networks in reinforcement learning, the DQN approach deploys two neural networks of the same architecture but with different weights θ and θ^{-}, a Q-network $Q(S_t, A_t; \theta)$ and a target-network $Q(S_t, A_t; \theta^{-})$. As Figure 5.2 illustrates, both networks receive a state as input and return the Q-values of this state. While the Q-network computes the Q-value for all possible actions and returns only the Q-value for an observed pair of action A_t and state S_t, the target-network determines this value for each action in the action-space and returns only the greatest Q-value. However, both networks receive different states as input. The Q-network gets state S_t, while the input of the target-network is the successive state

S_{t+1}. This results from the Q-Learning function (5.4.4). As a consequence, the loss of the current observation of S_t, A_t, S_{t+1} and R_{t+1} is calculated based on the output of both networks. The Q-network inputs the predicted Q-value for the current state-action pair (S_t, A_t). On the other hand, the target network contributes the Q-value for the subsequent state-action pair (S_{t+1}, A_{t+1}) and thus, provides a Q-value that is assumed to be the true Q-value. However, this Q-value of the subsequent time step $t+1$ has to be assessed at time step t which is why the reward of the current state-action pair R_{t+1} has to be taken into account. The evaluation at time step t is relevant, in order to be able to determine the mean squared error of the prediction and the assumed truth value at time step t.

As Figure 5.2 illustrates, the Q-network outputs the predicted value and is updated after each iteration, the target Q-network returns the actual Q-value and remains unchanged for a defined number of iterations C.

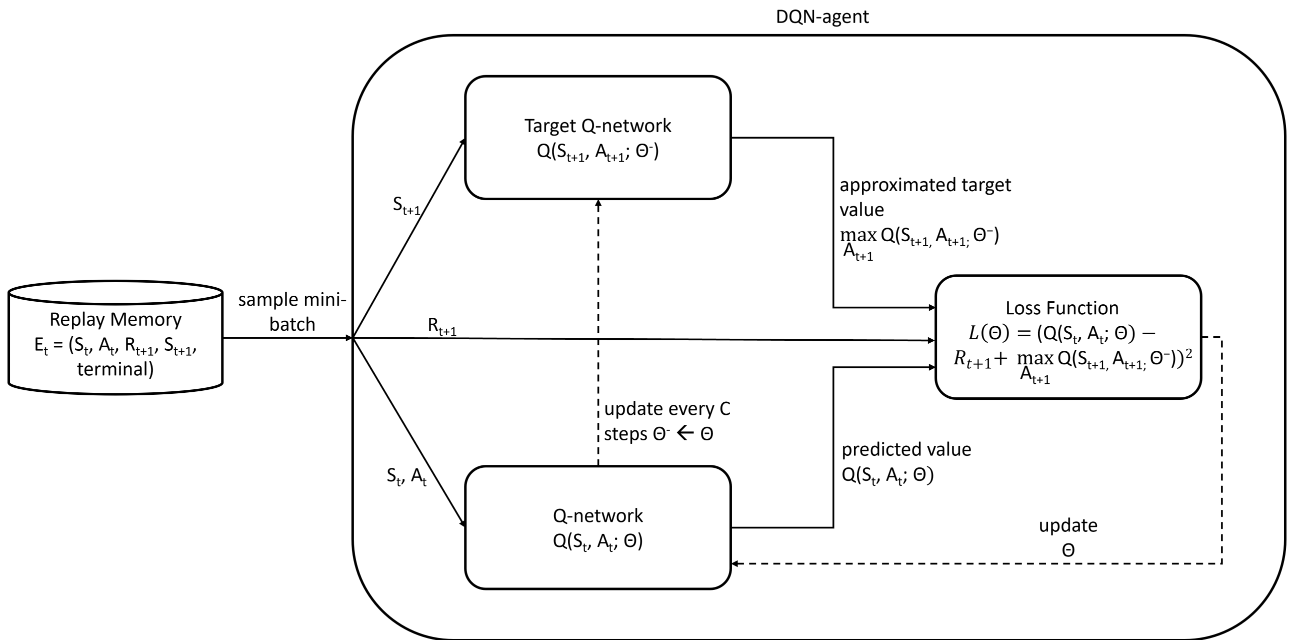


Figure 5.2: Training of a DQN-agent

In addition to that, another key difference in learning with DQNs consists in the concept of experience replay, first studied by Lin [20]. The experience replay stores the experiences gained from interactions with the environment in a so-called replay memory as shown in Figure 5.3 which extends the previous agent-environment interaction. For each experience E_t, a tuple $(S_t,\ A_t,\ R_{t+1},\ S_{t+1},\ terminal)$ including the current state S_t, a chosen action A_t, a received reward R_{t+1} as well as the next state S_{t+1} is recorded. Furthermore, information about whether this transition reaches a terminal state or not is included by a *terminal*-flag.

As indicated at the beginning of this chapter, reinforcement learning unlike supervised learning methods, has no data from which to learn. However, the experience replay collects data at runtime. These experiences are used to train the Q-network by drawing random batches.

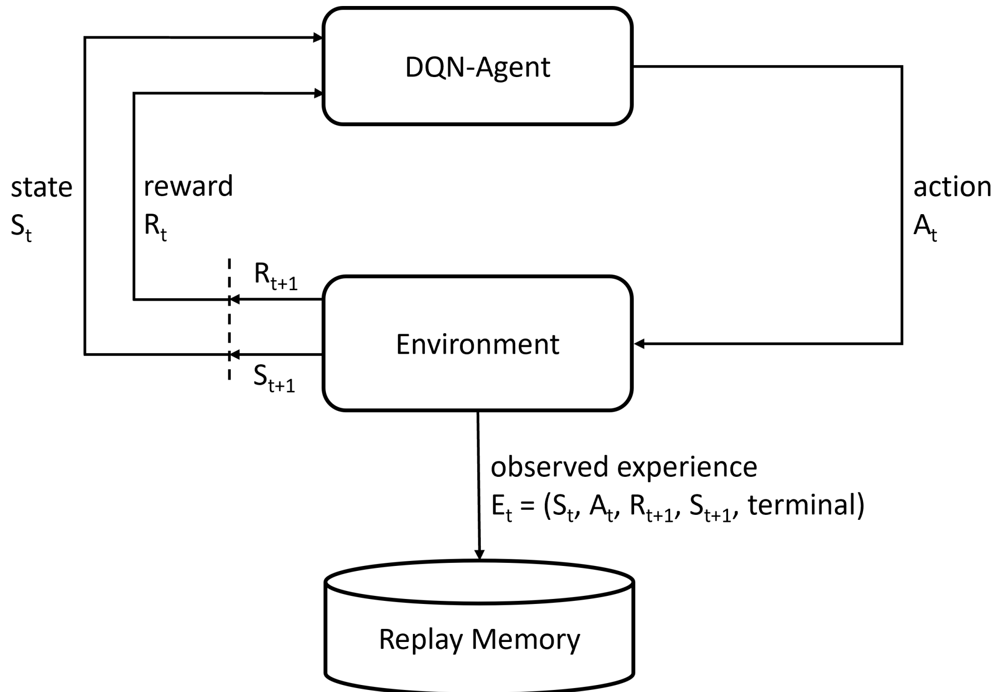


Figure 5.3: DQN-agent-environment interaction

Lei [18] argues that the distinction between a target- and a prediction-network (Q-network), as well as the concept of experience replay, affect the stability of the training in a positive way. A temporary disconnection between a target- and a Q-network reduces the correlation between both outputs. Furthermore, since the update of the target-network happens less frequent, the problem of moving targets decreases. Additionally, the difficulty of correlated input data is overcome by performing the Q-network training on randomly selected mini-batches from the replay memory. Moreover, the replay memory stores only the last N experiences. On the one hand, this ensures that experiences from previous policies can be replaced by new learned behaviour. On the other hand, data can be used more often and thus more efficiently.

Algorithm 2 [26] demonstrates how the Q- and target-network are trained during the interaction with the environment. The training starts by generating an experience (line 7-9 in Algorithm 2). At first, an action is chosen by means of the ε-greedy approach. The chosen action A_t is executed and the environment returns the next state S_{t+1} and a reward R_{t+1}. Then, this gained experience is stored in the replay memory. Further, out

Algorithm 2 Deep Q-Learning algorithm [26]

1: Initialise replay memory D of length N
2: Initialise Q-Network with random weights θ
3: Initialise target network with weights $\theta^{-} \leftarrow \theta$
4: **for** each episode **do**
5: Initialise state of environment $S_t \leftarrow S_0$
6: **while** S_t is not terminal **do**
7: Choose action A_t according to $\varepsilon - greedy$ approach
8: Execute action A_t and observe reward R_{t+1} and next state S_{t+1}
9: Store experience in replay memory D
10: Sample random mini-batch from D
11: **for** each $e \in mini - batch$ **do**
12: Perform forward pass through Q-network and receive $Q(S_t, A_t; \theta)$
13: **if** S_{t+1} terminal **then**
14: Approximate target Q-value $Q(S_t, A_t; \theta) \leftarrow R_{t+1}$
15: Compute loss as $L(\theta) = (Q(S_t, A_t; \theta) - R_{t+1})^2$
16: **else**
17: Approximate target Q-value
$Q(S_t, A_t; \theta) \leftarrow R_{t+1} + \gamma \cdot \underset{A_{t+1}}{max} Q(S_{t+1}, A_{t+1}; \theta^{-})$
18: Compute loss as $L(\theta) = (Q(S_t, A_t; \theta) - R_{t+1} + \gamma \cdot \underset{A_{t+1}}{max} Q(S_{t+1}, A_{t+1}; \theta^{-}))^2$
19: **end if**
20: **end for**
21: Backpropagate loss of entire mini-batch with respect to network parameter θ
22: Every C steps set $\theta^{-} \leftarrow \theta$
23: $S_t \leftarrow S_{t+1}$
24: **end while**
25: **end for**

of this memory a random mini-batch is provided. For each experience of this batch, a forward pass through the Q-network (line 12 in Algorithm 2) is performed. Next, the target Q-value is calculated according to the Bellman-optimality-equation (5.4.4) applied in Q-Learning. Furthermore, for the calculation of the target value, a distinction is made whether the subsequent state S_{t+1} is terminal or not (line 13-17 in Algorithm 2). If the state S_{t+1} is considered terminal, then only the received reward R_{t+1} determines the target Q-value. On the other hand, if the state S_{t+1} does not reach a terminal state, the received reward R_{t+1} and a discounted approximation of the next optimal Q-value determined by the target-network yield the target Q-value. Based on a prediction of the Q-network and an approximation of the target value the loss e.g. as squared error is determined (Equations (5.4.5) and (5.4.6)).

$$L(\theta) = E[(Q(S_t, A_t; \theta) - Q(S_t, A_t; \theta^-))^2] \tag{5.4.5}$$

$$L(\theta) = E[(Q(S_t, A_t, \theta) - (R_{t+1} + \gamma \cdot \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}, \theta^-)))^2] \tag{5.4.6}$$

As a next step in line 20 in Algorithm 2, with respect to the network parameters the computed gradient loss is passed backwards through the Q-network. A so-called backpropagation is performed. This leads to an update of the Q-networks' weights. Furthermore, every C steps the target-network is replaced by the Q-network. Since both networks underly the same architecture, this means that the weights of the target-network θ^- are updated by those of the Q-network θ as indicated in line 22 in Algorithm 2.

The training of the Q-network runs over a predefined number of episodes. In each episode, the agent interacts with the environment until a terminal state is reached. Like in Q-Learning, this loop-condition applies especially for episodic tasks. Furthermore, in every iteration, the previously updated networks are used.

All in all, Sutton and Barto summarise in [39, pp. 436] that the DQN-algorithm overcomes many difficulties of deep reinforcement learning by stabilising the training. Moreover, this approach outperforms most of previous used methods.

Running Example The running example has four different action options. These actions are *Move to input conveyor*, *Move to station A*, *Move to station B* and *Move to output conveyor*. We still consider the input conveyor as current state of the environment. The Q-value of the state *input conveyor S* is determined by the Q-network as visualised in Figure 5.4. The current state in shape of the location of the loader is input to the neural network. Then, this input is passed through various hidden layers. The results of these hidden layers are put into the output layer, where one output is provided for every possible action of the action space. This corresponds to four different nodes for the running example. The Q-network outputs a vector of four different Q-values, one for each action. Based on this output, the action with the highest Q-value can be selected.

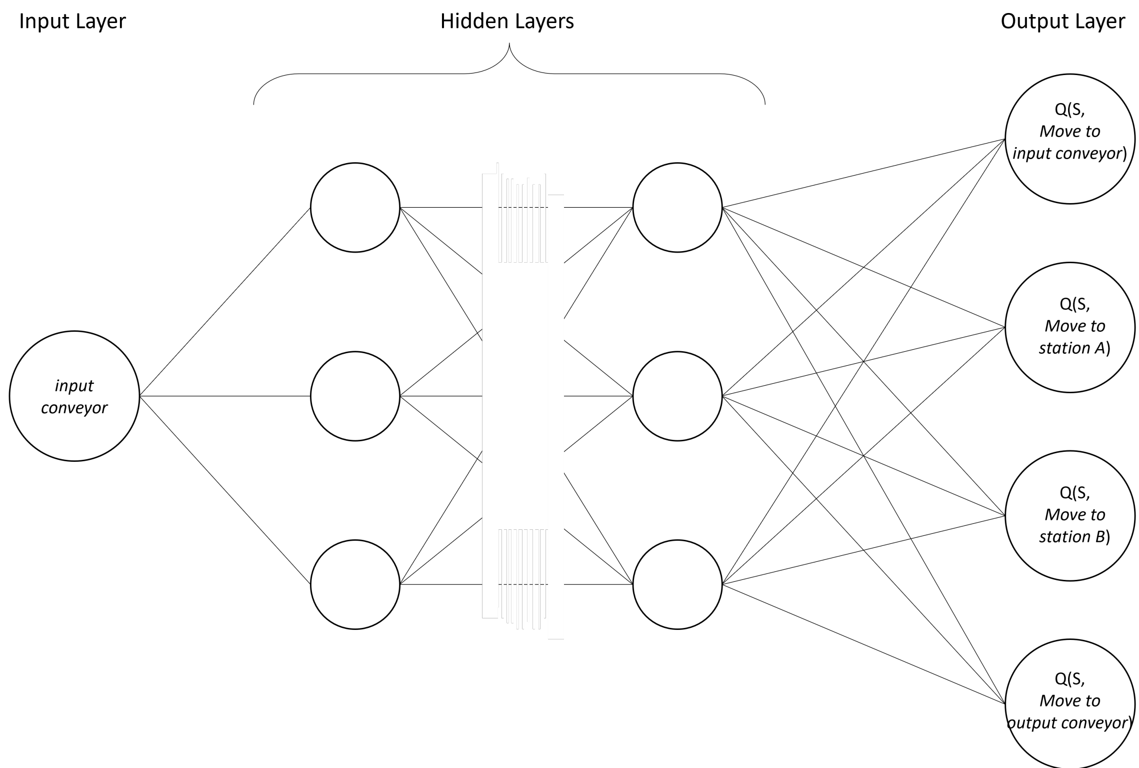


Figure 5.4: Q-network for the running example with S as *input conveyor*

5.5 Reward Function Design

As the description of the core elements of reinforcement learning already implies, the rewards received by an agent constitute the key element. Both the state-value function in Equation (5.3.3) and the action-value function in Equation (5.3.6) are based on returns and thus also on rewards. Furthermore, the optimal policy derived from the value functions is strongly affected by the choice of rewards.

Rewards frame the general problem an agent should learn to solve. Therefore, the reward function design represents a key factor to the success of an agent. However, defining a successful reward function does not need any detailled knowledge about how to act. Sutton and Barto clarify this problem of reward function design in [39, pp. 53] using the game of chess as an example. An agent playing chess is only rewarded for the final result of the game. This might be receiving a reward of +1 for winning, -1 for losing and 0 for draw. No further specifications for allowed moves need to be specified. Yet, the agent has to receive negative rewards as penalties for executing illegal moves.

As relevant as designing a reward function is, finding a good definition can be difficult and time-consuming. Especially when the complexity of a reinforcement learning problem

increases, according to Abbeel et al. [1] specifying a reward function manually becomes a great challenge.

Furthermore, in [39, pp. 53, 469], Sutton and Barto present different challenges that need to be overcome when designing reward functions. On the one hand, defining rewards manually often leads not only to translating the goal into rewards but also to facilitating how to achieve that goal. This is on the same page as rewarding subgoals which are not mandatory for reaching the overall goal. Both aspects can result in an agent which is restricted to a given behaviour. On the other hand, another difficulty occurs when indeed the agent attains the defined goal but the learning process takes a long time. This problem can arise for a number of reasons. For example, the agent may receive a reward only when completing the whole task. These sparse rewards induce the agent to explore the environment without a clear aim before experiencing a reward. This affects the learning speed in a negative way. Moreover, delayed rewards make it more difficult to quantify the impact of previous actions.

The following chapter explains the concept of reward machines as a further approach of reward function design or more specifically on exploiting the process of reward function design. However, this concept goes beyond the basic idea of reward function design and describes the core on which this master thesis is build.

6 Reward Machines

Reward machines are a relatively new way to address the challenge of reward function design. In 2018, Icarte et al. [17] introduced the concept of reward machines to speed up the search for an optimal policy.

The general idea of reward machines is based on the fact that the design of rewards appears to be hidden from the agent. However, as explained already earlier, reward design is a manual process. Therefore, there is no reason why the agent should not know the structure of the reward function. Reward machines provide a way to make this structure and high-level idea of the reward function accessible. While sparse rewards lead to slow training, providing insights into the overall structure of rewards in turn speeds up the learning process.

In the following, reward machines will be formally defined and an overview of approaches developed to take advantage of reward machines will be provided.

6.1 Definition

According to Icarte et al. [17], reward machines can be defined as finite state machines, strictly speaking Mealy-machines, which were introduced earlier in Chapter 3. A reward machine aims to represent the structure of rewards. On the one hand, the structure can be revealed from the environmental states by capturing reward-relevant aspects of the environment in the reward machine. On the other hand, a reward machine can be defined by relevant high-level events of the history. In the latter case, the reward machine serves as an external memory for aspects that are in fact relevant for the calculation of rewards but are not associated with the states of the environment.

Reward machines [17] can be formalised as a tuple of six elements

$$R = (U, 2^P, \delta_u, \delta_r, u_0, F).$$

As in the definition of automata, U represents the finite set of states and u_0 as an element of U indicates the initial state. The states of the reward machine do not directly reflect the environmental states of a reinforcement learning problem. Instead, they aggregate the states of the environment. In doing so, these environmental states can be aggregated based on a high-level description. Furthermore, if a reward machine provides additional information from high-level events of the history of a state, states with the same past high-level events are grouped together. In that case, the states of the reward machine extend the states of the environment by additional information. A particular state in the

reward machine includes all states in the environment that satisfy the same particular aspect or have the same particular events in their history. Therefore, states in the reward machine compress the states in the environment with regard to certain aspects. However, the more details the reward machine contains, the less compression is possible.

P stands for propositional symbols and relates to a high-level description of states in the environment. These propositional symbols are used to define the alphabet 2^P which contains logical formulas formulated over P.

The state-transition function δ_u determines the next state depending on the current non-terminating state and the propositional symbols, $\delta_u : U \setminus F \times 2^P \rightarrow U$. In contrast to the transition function from automata theory, a logical connection between different propositional symbols can be used as input and is considered as a single input symbol. Since propositional symbols represent specific events that have taken place in the history of an environmental state, it is understandable that several events could have happened and therefore several propositional symbols must be linked together.

The state-reward function $\delta_r : U \setminus F \times 2^P \rightarrow \mathbb{R}$ corresponds to the output-function of a Mealy-automaton and defines which reward to return. This function depends not only on the state of the reward machine but also on the propositional symbols. Icarte et al. [17] point out that the reward machine approach aims to provide different reward functions depending on the state of the reward machine.

F represents the finite set of terminal states which correponds to accepting states in an automaton.

This formal definition of reward machines differs slightly from the definition in [17] and is more aligned with the definition of automata. Nevertheless, this does not affect the theoretical concept of reward machines.

Running Example In the running example presented, the state of the environment consists of the current position and the status of the loader, while the reward machine states aggregate these states based on the status of the loader. Details on the reward machine of this example will be presented later. Figure 6.1 illustrates how states of the environment can be compressed into states of a reward machine. In the case of the given running example, a particular state of the reward machine is that the loader is empty. This state represents many different states in the environment, all of which have at least in common that the loader is not filled with an item. So, for example, four different states of the environment are represented by only one state of the reward machine, because the position of the loader is not taken into account in the states of the reward machine. If the environment were to contain further information on e.g. whether a station is busy or not because an element was unloaded there in the past, then the compression rate would increase. That means that the state *Loader empty* would bundle further states of the environment.

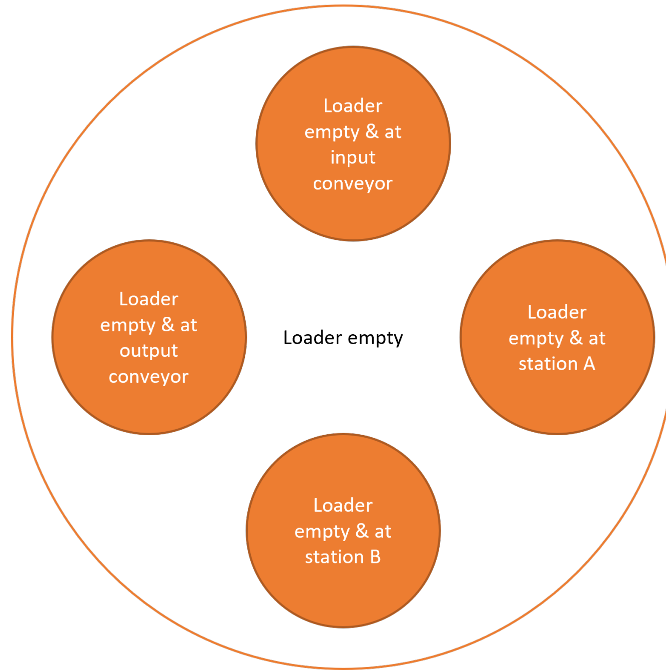


Figure 6.1: Compression of the environment to the state *Loader empty* of the reward machine of the running example

6.2 MDP with Reward Machine

Reward machines have been developed for application to reinforcement learning problems formally defined as MDP. Therefore, an additional definition that combines MDP and reward machines to MDP with reward machine (MDPRM) was introduced by Icarte et al. [17]. A MDPRM is defined as a tuple

$$(S, A, R, p, U, 2^P, \delta_u, \delta_r, u_0, F, L).$$

This definition includes the elements of the underlying MDP (S, A, R, p) which was explained in Chapter 5, the elements of a reward machine $(U, 2^P, \delta_u, \delta_r, u_0, F)$ and a labelling function L. This labelling function is defined as $L : S \times A \times S \rightarrow 2^P$ and assigns truth values to the propositional symbols P. These truth values are input to the reward machine. However, this labelling function is rather myoptic, since only events that have occurred in the current state-action-state transition of the environment can be taken into account for the assignment of truth values. Therefore, we extend the labelling function by the state of the reward machine as input. This leads to the labelling function $L : U \times [S \times A \times S] \rightarrow 2^P$. Hence, a longer history of relevant events can be considered. In terms of automata, this labelling function generates the current letter σ_t of the word that is input to the automaton.

Figure 6.2 illustrates the interplay between agent, environment and reward machine - as an extension of the general agent-environment interaction shown in Figure 5.1. The

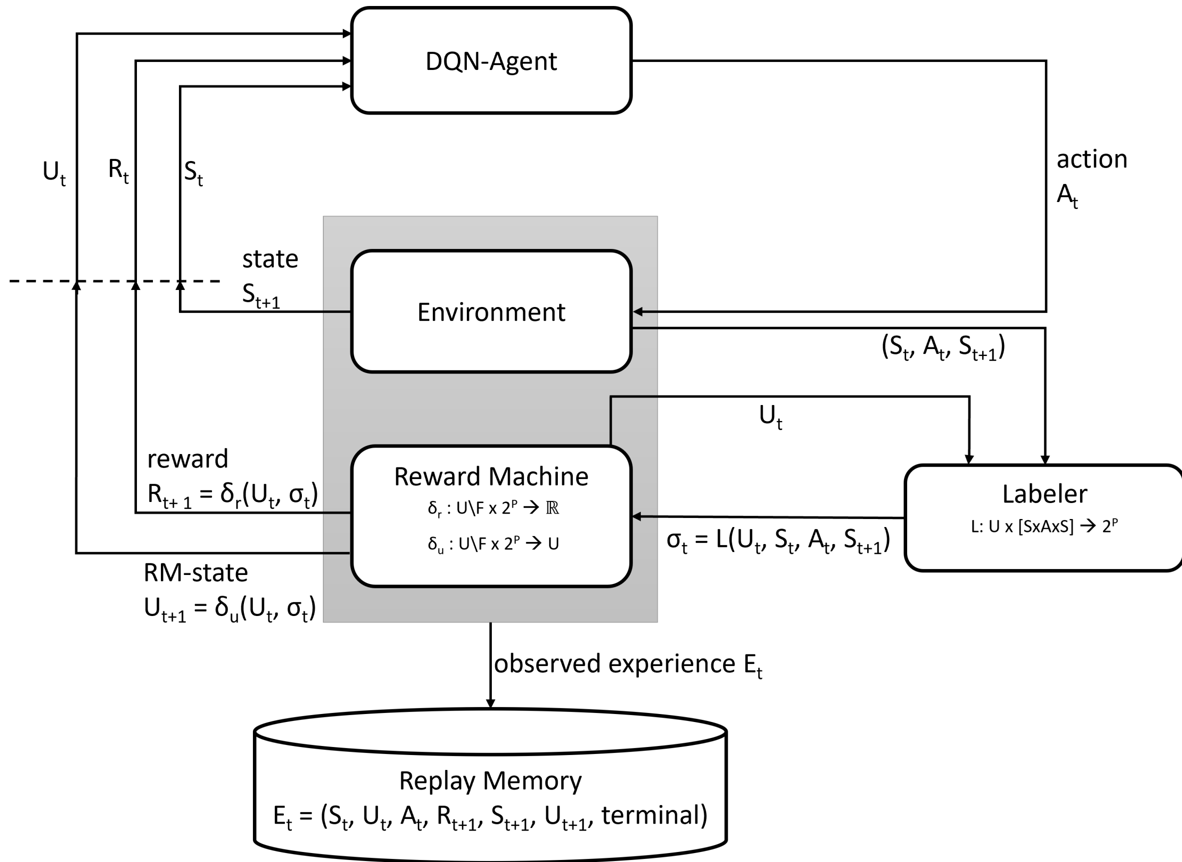


Figure 6.2: The agent-environment-reward machine interface

relationship shown in Figure 6.2 begins with the selection of an action A_t by the agent based on a given state of the environment S_t and the current state of the reward machine U_t. Thereupon, the environment executes the selected action and transitions to a next state S_{t+1} according to its state-transition function p. At each step t, the agent moves not only from one state to another in the environment, but transitions also from state to state in the reward machine. However, before the next state of the reward machine can be determined, the current input symbol σ_t must be calculated by means of a labeler. This truth assignment σ_t is computed from the observed state-action-state sequence and the current reward machine state U_t. Thus, the environment produces the input for the automaton. At each iteration of the interaction, an input symbol σ_t is delivered to the reward machine. This extracted input symbol indicates which propositions in P are currently satisfied. The values of these propositional symbols P form a logical formula. This logical formula represents what is regarded as letter or word in automata theory.

Hence, after T iterations, a word of length T has been formed. This word is represented by the current state of the reward machine. The input letter σ_t as element of 2^P leads to a transition of states in the reward machine according to its state-transition function δ_u. Furthermore, the current reward machine state U_t and the logic formula of the truth values σ_t calculate the reward R_{t+1} which is passed on to the agent.

When applying the DQN learning approach, an experience is formed from the state transitions in the environment as well as in the reward machine. This experience is added to the replay memory used for training the DQN agent.

Icarte et al. [17] found that reward is non-Markovian when the reward depends on both the current state of the environment and the reward machine while the states of the reward machine add information to the environment states. This means that the amount of reward results from the history of states and actions.

Furthermore, [17] states that an MDPRM can itself be considered an MDP. This is especially relevant, if the states of the reward machine contain information that is not part of the environmental states, as it is the case when the reward is non-Markovian. The set of states of the new MDP is formed by a cartesian product $S \times U$ of the MDPRM. This means that the states of the environment of the new MDP also contain information about the history displayed by U. The set of actions remains the same. However, the state-transition function p and the reward function R are composed of δ_u and δ_r, respectively. In contrast to MDPRM, an MDP of the cartesian product only defines Markovian rewards, since the cartesian product of states of the environment and states of the reward machines build the new state space of the environment. Thus, history of states and actions is implicitly taken into account.

Running Example A reward machine of the running example is illustrated in Figure 6.3. For simplicity an episodic production scheduling task is assumed. The states of the environment are defined by the position of the loader, the status of the loader and the states of the machines inside the work cell. Since, the loader's state is considered as reward-relevant aspect in the environment, the states of the reward machine reflect only the loader's status. The reward machine accepts states that are empty, filled with an unprocessed part, carrying a processed product, marking the occurence of an error or the completion of the task. This set of states of the reward machine is summarised in (6.2.1).

$$U = \{Loader\ empty,\ Loader\ filled\ (unprocessed),\ Loader\ filled\ (processed),\ Finished,\ Error\} \tag{6.2.1}$$

Furthermore, the loader is considered to be initially *Loader empty*. This reward machine is defined over the alphabet 2^P with the propositional symbols indicated in (6.2.2).

$$P = \{empty,\ filled_{unprocessed},\ filled_{processed},\ finished,\ error\} \tag{6.2.2}$$

The five different states U (6.2.1) are shown as orange circles. The final states F of the reward machine are highlighted by double framed circles. In this given example, the states

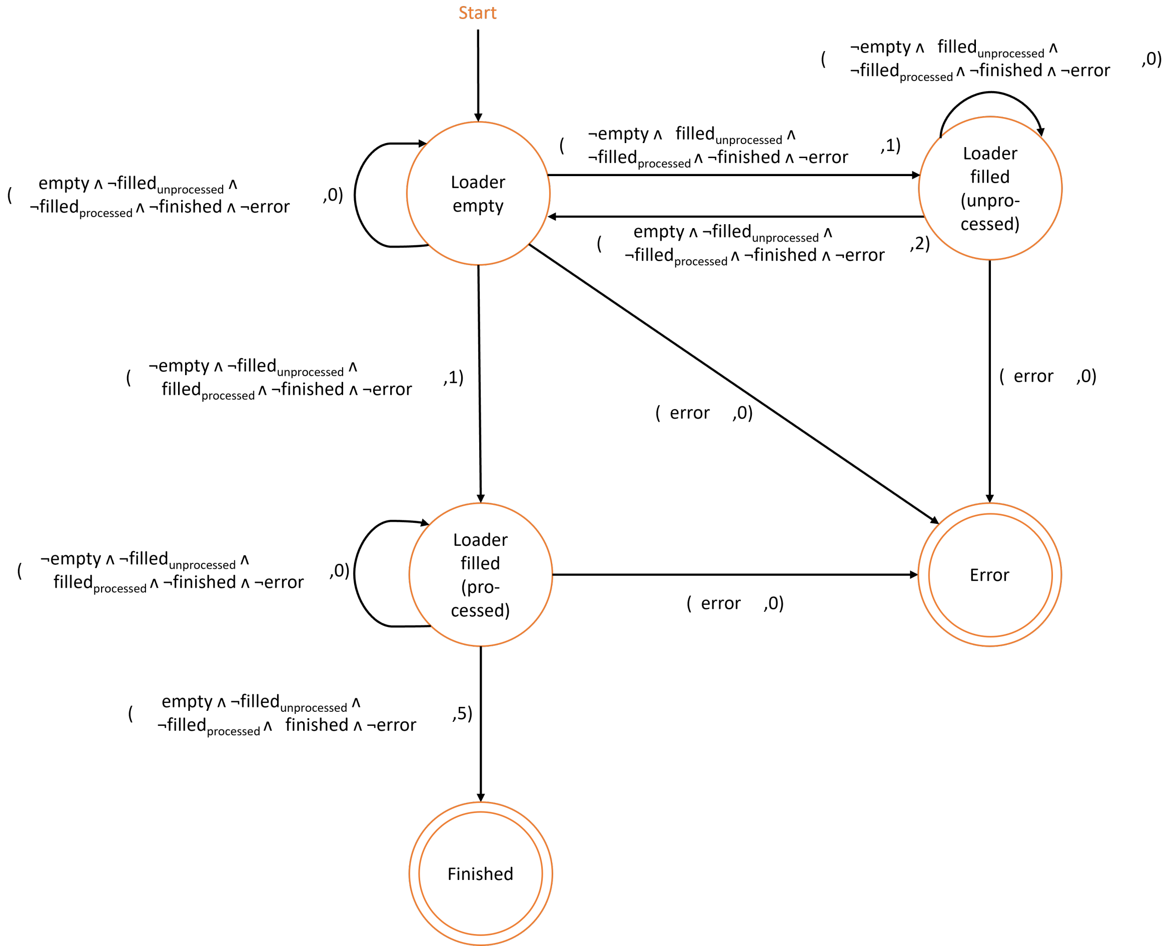


Figure 6.3: Reward machine of the running example

Finished and *Error* mark a terminal state. The edges are described by a tuple of a logical formula and a numerical value. The edges indicate which logical formula - or in terms of automata theory which letter - causes which next state in the reward machine. The numerical value specifies the reward if the logical formula is true in a particular reward machine state.

When the current state of the reward machine is $Loader\ filled(processed)$ and the logical formula $empty \wedge \neg filled_{unprocessed} \wedge \neg filled_{processed} \wedge finished \wedge \neg error$ is satisfied, a transition to the terminal state *Finished* takes place, the task is completed and the production system terminates with the return of a non-zero reward. Moreover, an error can occur in any state of the reward machine if an impossible action is performed, for

example if an unprocessed item is unloaded at the output conveyor. In that case, a letter is entered into the automaton that leads to the terminating state *Error*. Logical formulas at edges to the state *Error* contain only the propositional symbol *error*, because the truth value of any other propositional symbol does not matter for the transition. An input of *error* would result in an input sequence that is accepted by the automaton, however, it would end the task prematurely and without completion.

To reduce the complexity of the visualisation of the reward machines, logical formulas are simplified to only non-negated propositional symbols. The logical formula $empty \wedge \neg filled_{unprocessed} \wedge \neg filled_{processed} \wedge finished \wedge \neg error$ would be shortened to $empty \wedge finished$. Figure 6.4 illustrates the reward machine of the given example simplified regarding the logical formulas.

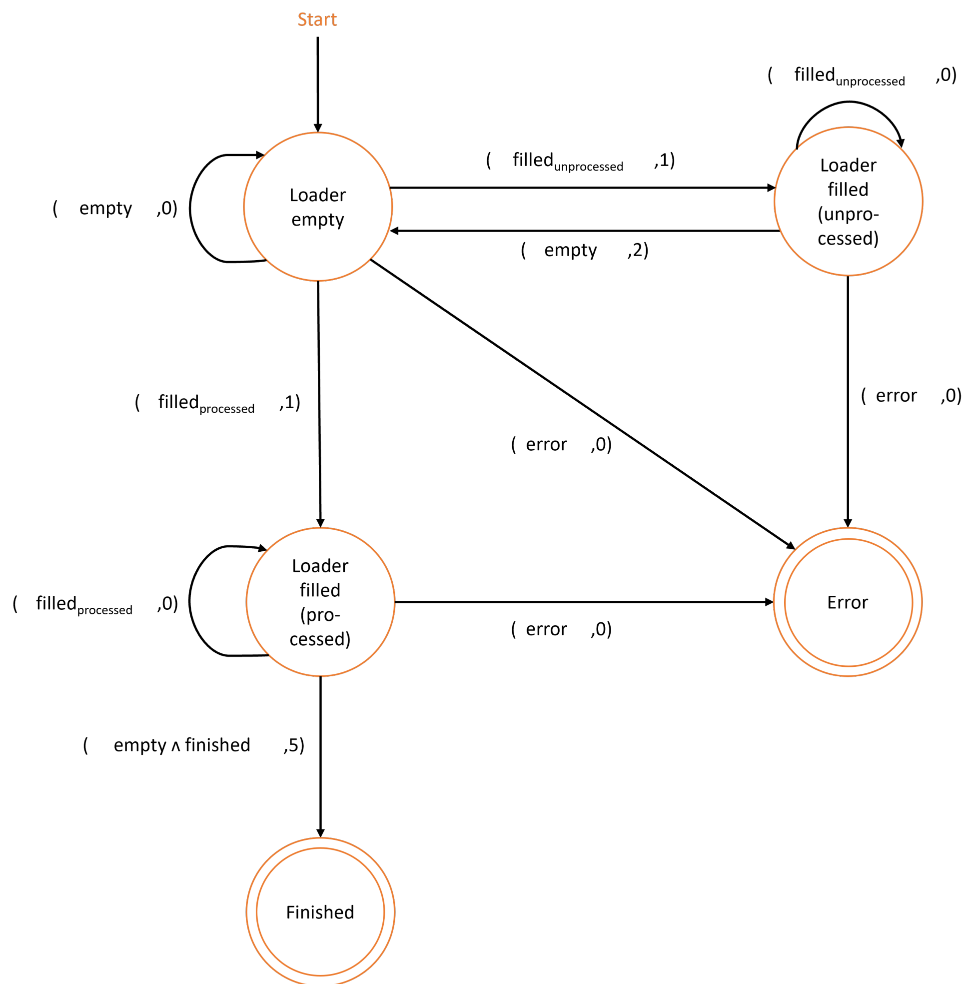


Figure 6.4: Simplified visualisation of the reward machine of the running example

6.3 Approaches for Reward Machines

Since MDPRMs can be considered as regular MDPs, the application of state-of-the-art learning approaches already leads to optimal policies. However, to reach an optimal policy faster than these standard learning approaches and to exploit the structure of a reward machine, the learning approaches need to be adjusted. Icarte et al. [17] introduced three different ways to do so - Counterfactual Experiences for Reward Machines (CRM), Hierarchical Reinforcement Learning for Reward Machines (HRM) and Automated Reward Shaping (RS). These three methods work in combination with state-of-the-art learning approaches such as Q-Learning and DQNs. A successful application only requires considering the reward machine state when computing the Q-value. This can be realised by extending $Q(S_t, A_t)$ to $Q(S_t, U_t, A_t)$ and by considering the reward machine state U_t in the experience for the replay memory in deep learning approaches.

6.3.1 Counterfactual Experiences for Reward Machines

CRM is applied to use information from a reward machine. Counterfactual experiences describe observations that have not actually occurred. Since the general concept of rewarding is revealed by reward machines, it is possible to analyse for each actual observation how the state of a reward machine would have changed if the given state had been different, and what reward would have been returned. This would not be possible if the agent had no way of knowing about a reward without actually performing an action in a given state.

The general idea of the CRM approach is to force the generation of observations with different reward-relevant aspects of states in the environment than those actually observed. The actual experience comes from the observed sequence of agent-environment interactions. This takes into account the current state of the reward machine and the current state-action-state transition from the environment. The counterfactual observations differ from the actual observation in that for the observed state-action-state transition in the environment, the other states of the reward machine are also taken into account. Thus, it is examined how the current state-action-state transition would behave in relation to the reward if another history - in the form of a different state of the reward machine - had been taken as a basis. Furthermore, these counterfactual observations are only synthetically generated.

Thus, for every actual observed experience, a synthetically generated experience is added for every other non-terminating state of the reward machine. Only the non-terminating states of the reward machine are considered for counterfactual reasoning, while the states of the environment remain unchanged. The execution of the same action is assumed for each of these non-terminating states in the reward machine. Since actions have different effects depending on the respective state of the reward machine, it is necessary to first generate the next letter for each counterfactual state of the reward machine, in order to determine the subsequent state according to δ_u.

In terms of automata, counterfactual experiencing means generating an input letter and analysing the effect of this input letter for each possible structure of a prefix represented by a non-terminating state in that automaton. Hence, in addition to the actual transition, it is examined what input letters and transitions would occur for a certain action in any other non-terminating state. These non-terminating states can be formalised as the set $U \setminus F$.

Within DQNs CRM is realised by adding these counterfactual experiences to the replay memory as well. This indicates an extension of the continuously developed illustration of the agent-environment interaction in Figure 6.5. In contrast to Figure 5.3, the agent-

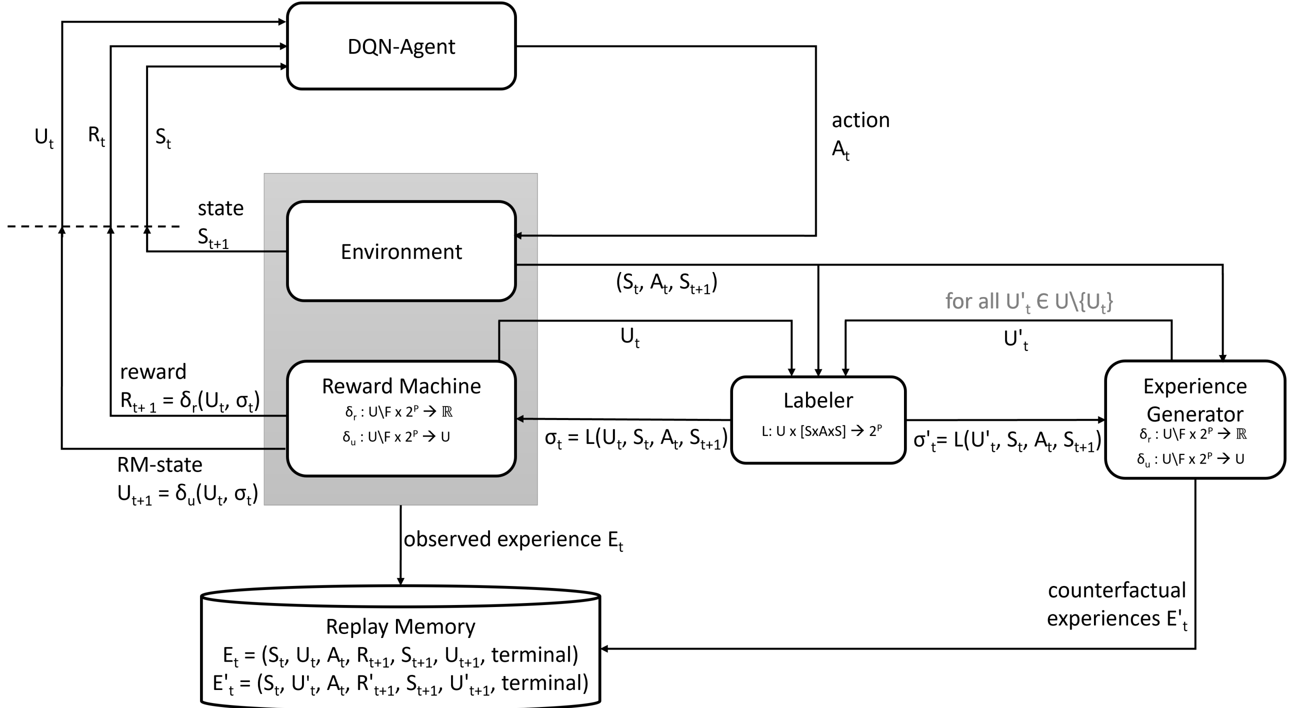


Figure 6.5: DQN-agent-environment interaction combined with the CRM approach

environment interaction combined with the CRM approach holds an experience generator. This experience generator contains a replica of the reward machine and creates one experience for each state of the reward machine - except the observed one. The experience generator inputs a state of the reward machine U_t' into the labeler which receives the current state-action-state transition from the environment and returns an input symbol σ_t'. This input symbol is returned to the experience generator which determines the subsequent state of the reward machine U_{t+1}' and the resulting reward R_{t+1}'. This is repeated for each state of the reward machine except the currently visited one. The experience generator receives the current state-action-state transition as input from the environment, in order to be able to form a propriate synthetical experience E_t' consisting of the actual current state-action-state transition of the environment and the theoretical transition in the reward

machine with the corresponding reward. These counterfactual experiences E_t' are added to the replay memory.

At the same time, the actual observed experience is generated in a similar way, but without using an experience generator mechanism. The actual reward machine inputs its current state into the labeler. Together with the received state-action-state transition from the environment, the labeler function calculates the current truth assignment σ_t. Furthermore, based on the truth assignment σ_t, the actual reward machine transitions to the next state of the reward machine and adds the actually observed experience to the replay memory as well. This corresponds to the process of generating experience without using reward machine-specific learning approaches, as shown in Figure 6.2.

In addition to the experiences that are usually added to a replay memory, the experiences generated by the CRM approach contain information about the current and subsequent state of the reward machine. Since both observed and counterfactual experiences are added to the replay memory at each iteration of the agent-environment interaction, the replay memory is filled much faster than without the CRM approach.

This agent-environment interaction in combination with the CRM approach was algorithmically formalised by Icarte et al. [17] in Algorithm 3. If a terminal state is reached in the actual observation, then the entire episode has to be cancelled, as the condition of the while-loop in line 7 of the Algorithm 3 indicates. Unlike Algorithm 2, the reward cannot be already calculated when the action is performed and the subsequent state S_{t+1} is observed in line 9. Due to the consideration of reward machines, in line 10 the input symbol must first be determined before the reward of the current agent-environment interaction can be returned in line 12. Moreover, this input symbol σ_t is also necessary to observe the next state of the reward machine U_{t+1} in line 11. The central difference due to the CRM approach arises in line 14-20 where the generation of synthetic experiences takes place. The rest of the algorithm (line 21 and the following) corresponds to line 10 and following in Algorithm 2. Despite the same flow, the approximated Q-values now also depend on the state of the reward machine.

All in all, the additional experiences support earlier learning of the correct intermediate steps of an optimal policy. Since it takes many simulation runs to observe a particular state of a reward machine and to be able to analyse a certain action in this state, CRM provides this experience much earlier.

Running Example To continue on the exemplary reward machine presented in Figure 6.4, the environment performing an unload, when filled with a processed element at the output conveyor, would transfer the state-action-state transition to the labeler. For the actual observation, the labeler would receive the current reward machine state *Loader filled (processed)*. Then, the logical formula $empty \land finished$ would be determined and returned by the labeler. Based on this letter, a transition from *Loader filled (processed)* to *Finished* is observed. However, due to the CRM approach, the letter is also computed for the reward machine states *Loader empty* and *Loader filled (unprocessed)*. In both cases,

Algorithm 3 Deep Q-Learning algorithm with CRM [17]

1: Initialise replay memory D of length N
2: Initialise Q-Network with random weights θ
3: Initialise target network with weights $\theta^{-} \leftarrow \theta$
4: **for** each episode **do**
5: Initialise state of environment $S_t \leftarrow S_0$
6: Initialise state of reward machine $U_t \leftarrow U_0$
7: **while** S_t is not terminal and $U_t \notin F$ **do**
8: Choose action A_t according to $\varepsilon - greedy$ approach
9: Execute action A_t and observe next state S_{t+1}
10: Compute input symbol $\sigma_t \leftarrow L(U_t, (S_t, A_t, S_{t+1}))$
11: Compute next state of reward machine $U_{t+1} \leftarrow \delta_u(U_t, \sigma_t))$
12: Compute reward $R_{t+1} \leftarrow \delta_r(U_t, \sigma_t)$
13: Store experience E_t in replay memory D
14: **for** each $U'_t \in U \setminus F$ **do**
15: Compute input symbol $\sigma'_t \leftarrow L(U'_t, (S_t, A_t, S_{t+1}))$
16: Compute next state of reward machine $U'_{t+1} \leftarrow \delta_u(U'_t, \sigma'_t))$
17: Compute reward $R'_{t+1} \leftarrow \delta_r(U'_t, \sigma'_t)$
18: Set experience $E'_t \leftarrow (S_t, S'_t, A_t, R'_{t+1}, S_{t+1}, U'_{t+1})$
19: Store experience E'_t in replay memory D
20: **end for**
21: Sample random mini-batch from D
22: **for** each $E^* \in mini-batch$ **do**
23: Perform forward pass through Q-network and receive $Q(S^*_t, U^*_t, A^*_t, \theta)$
24: **if** S^*_{t+1} terminal or $U^*_{t+1} \in F$ **then**
25: Approximate target Q-value $Q(S^*_t, U^*_t, A^*_t, \theta^{-}) \leftarrow R^*_{t+1}$
26: **else**
27: Approximate target Q-value
$Q(S^*_t, U^*_t, A^*_t, \theta^{-}) \leftarrow R^*_{t+1} + \gamma \cdot \max_{A^*_{t+1}} Q(S^*_{t+1}, U^*_{t+1}, A^*_{t+1}, \theta^{-})$
28: **end if**
29: Compute loss as $(Q(S^*_t, U^*_t, A^*_t, \theta) - Q(S^*_t, U^*_t, A^*_t, \theta^{-}))^2$
30: **end for**
31: Backpropagate error of entire mini-batch with respect to network parameter θ
32: Every C steps set $\theta^{-} \leftarrow \theta$
33: $S_t \leftarrow S_{t+1}$
34: $U_t \leftarrow U_{t+1}$
35: **end while**
36: **end for**

the resulting letter is *error*, which is why both transition to the terminal state *Error*. The

run would terminate early because an empty loader cannot unload anything and a loader filled with an unprocessed part cannot unload at the output conveyor. Both transitions would lead to an early end of the run and therefore return a reward of zero, while the actual transition does not terminate the run as a correct state-action-state transition can be observed.

Due to the CRM approach, three experiences instead of only one would be generated - an actual observation and two synthetic experiences. This allows the agent to identify both favourable and unfavourable decisions earlier. For example, in the given state, the agent would learn that a performed unloading yields positive rewards if located at the output conveyor and if the loader is filled with a processed part. Moreover, the agent is also trained to avoid unloading if the loader is empty and if it is filled with an unprocessed item because this action leads to an early termination without any chance to gain more positive rewards.

6.3.2 Hierarchical Reinforcement Learning for Reward Machines

Besides CRM, Icarte et al. [17] developed another approach which aims to facilitate the structure of the reward machine to discover an optimal policy faster. However, in contrast to CRM, HRM decomposes the global problem into subproblems which are called options. Each option describes the transition from one state of the reward machine to another one. The general idea is to learn a policy for each option and an additional global policy that chooses between different options. Hereby, all option policies are learned in parallel by falling back to the idea of counterfactual experience generation. As soon as the current option terminates or a terminal state in the environment is reached, the high-level policy is trained.

However, the HRM approach suffers from the greedy approach of finding the best local solution for each subproblem. Since a best local solution does not necessarily lead to an optimal global solution, this approach risks converging to suboptimal solutions.

6.3.3 Automated Reward Shaping

The RS approach differs from CRM and HRM by reshaping the actual rewards, since the choice of reward function affects the learning speed. This method [17] extends the approach of potential-based reward shaping developed by Ng in 1999 [27]. The basic reward shaping idea applies to MDPs and uses a potential function ϕ. As shown in Equation (6.3.1), the adjusted reward R' builds on the previous rewards R of the MDP and adds the difference between the value of a potential function for two successive states.

$$R'(S_t, A_t, S_{t+1}) = R(S_t, A_t, S_{t+1}) + \gamma \cdot \phi(S_{t+1}) - \phi(S_t) \tag{6.3.1}$$

The automated reward shaping approach determines potentials considering the reward machine as its own MDP. The states of the reward machine correspond to the states in the MDP. Furthermore, the action space results from the transitions between reward

machine states. In addition, the reward function of the MDP is defined based on the reward-transition function of the reward machine. The transitions are considered deterministic and not stochastic. Since reward machines are viewed as MDP and transition probabilities are known, value iteration is used as basic reinforcement learning approach to calculate an optimal state-value function. These state-values are deployed for the potential function. Thus, each state of the reward machine receives an additional reward based on its state-value. Thus, rewards are provided already before the task is completed.

All in all, these three methods for reward machines have different benefits and disadvantages. They all surpass state-of-the-art algorithms applied on an MDPRM defined as MDP. Moreover, CRM finds optimal solutions, because it is able to learn simultaneously how to behave optimally for different states of the reward machine. On the other hand, HRM policies tend to be sub-optimal. This arises from the fact that all options are learned independently. Thus, policies for each option converge to a local optimum, but may not find a glocal optimum. Moreover, Icarte et al. [17] found in experiments that RS is less helpful for continuous domain problems. Besides, RS is not able to distinguish between final states that terminate a task and those that do not. Furthermore, RS represents an approach to exploiting the internal structure of rewards that can be combined with CRM and HRM.

In the following, CRM is used for exploiting the internal reward structure of continuous reinforcement learning problems because, unlike HRM, it promises to find optimal policies.

7 Related Work

In the past, much attention has been paid to job shop scheduling problems. While research initially focused on creating an optimal static schedule beforehand, the rise of reinforcement learning opens up a new way to react to uncertain events online. In the following, we first give an overview of reinforcement learning on scheduling tasks in general, and in production systems in particular. A next step brings work considering transportation in production systems into focus since this master thesis concentrates on gantry robot scheduling in a production context. The summary of related work concludes with a differentiation between previous work and the gap that this master thesis aims to bridge.

To the best of our knowledge, Zhang et al. [47] were the first who introduced a reinforcement learning approach for job shop scheduling problems. They revealed the great potential reinforcement learning has for job shop scheduling. In contrast to later work, this job shop scheduling use case did not refer to a production context. Instead, scheduling of NASA space missions was considered. The approach starts with a critical-path schedule as starting state which contains constraint violations e.g. overallocation of resources. Each schedule that still contains conflicts is penalised by a negative reinforcement. Furthermore, the final conflict-free schedule is rewarded with the negative value of the schedule length. This encourages short final schedules. A TD-Learning approach is used which employs a neural network to approximate the value function. Results show that this approach finds conflict-free schedules faster and is able to generalise to larger problems better than an iterative repair method. However, it is assumed that the jobs are known when creating a schedule. Furthermore, only a fixed number of jobs is considered which corresponds to an episodic task.

During the last decades, efforts have been made to apply reinforcement learning in production systems. However, production systems offer a wide range of reinforcement learning problems of varying complexity.

To the best of our knowledge, the first paper employing reinforcement learning in production systems was published in 1997. Mahadevan et al. [22] applied reinforcement learning to a continuous-time MDP problem to find an optimal policy. They used Q-Learning and a feed-forward network for learning. Their paper was concerned with a maintenance scheduling problem. In such a problem, the objective is to find a trade-off between maintenance cost and downtime cost. Their approach called SMART is able to schedule maintenance with a nonlinear dependency to maintenance costs. This has shown that SMART can adjust more easily to varying systems. However, maintenance scheduling is considered as a less complex problem than job shop scheduling which is the

focus of this master thesis.

While in [22] an environment consisting of a single machine, multiple products and multiple buffers is analysed, in [23] the SMART algorithm is applied on a more complex environment. They investigate a transfer line environment that adds multiple machines to the existing environment. In addition to a maintenance policy as in [22], a production policy is learned. Experiments show that a better policy than standard heuristics can be achieved. [23]

As a contrast, Brauer and Weiß [4] developed a multi-agent learning approach for production scheduling and focused on a multi-machine scheduling problem. This kind of problem can be classified as a flexible job shop problem. They borrowed the concept of Q-Learning and adapted it to their goal of achieving short cycle times. A task is decomposed into local tasks where each machine decides which machine is addressed next based on the estimated remaining manufacturing time. In this process, each machine represents an agent that learns independently of other machines. This leads to a decentralised learning process. In contrast to other scheduling approaches, this work focuses on the optimisation of a production path. Therefore, the problem of choosing which job is processed next is only implicitly addressed.

Similar to [4], Riedmiller and Riedmiller [36] focus on multiple locally trained agents. A job shop scheduling problem is also studied in [36], where the problem is simplified by assuming one machine per operation. Furthermore, a fixed number of jobs which is known in advance is assumed. Moreover, in contrast to Brauer and Weiß, local decisions are optimised with respect to minimising the sum of tardiness over all jobs. Since job shop scheduling problems are considered as NP-hard, Riedmiller and Riedmiller aim at efficiently computing a near-optimal solution. A neural network is used to approximate the value function. The local policy determined in this way focuses on a global goal. Since each local agent concentrates on this global goal, this approach corresponds to building a global decision by all local decisions. Experiments show that this approach performs at least as well as heuristics, and even better depending on the input. Riedmiller and Riedmiller investigated that especially the representation of a state is crucial for more advanced policies. Moreover, good performance is also obtained for unknown states which illustrates the generalisability of this approach.

In 2008, Gabel and Riedmiller [10] published another approach to deal with job shop scheduling problems. This approach takes into account the global perspective by combining local dispatching decisions with a global optimisation goal, as in [36]. However, in contrast to Riedmiller and Riedmiller [36], an optimistic assumption is made that all other agents act optimally. Therefore, this approach is applicable for production systems without central control or communication between different working centers. Gabel and Riedmiller use a neural fitted Q-Iteration algorithm. This method collects training data before updating Q-values. Batches of the training data are input to the neural network to approximate the value function. This batch-oriented training distinguishes it from other multi-agent reinforcement learning approaches. Another difference is in the policy screening. After each iteration a policy screening is executed which evaluates the

currently generated policy. Experiments show that this approach is able to outperform random and local as well as global dispatching rules. Especially for unseen more complex problems, heuristics with a global view achieve better performance. In addition, the discovered dispatching rules allow generalisation to unknown scheduling problems of identical size.

While previous mentioned work developed policies independent of any heuristics, Wang and Usher [42] investigated whether reinforcement learning in particular Q-Learning can be applied to the selection of dispatching rules. They provide three different dispatching rules to solve a single machine scheduling problem. The agent has to choose between these three heuristics depending on the current state to find an optimal policy for a continuous production scheduling problem. This paper shows that a reinforcement learning agent is able to learn which dispatching rule best fits a given scenario. However, a single machine problem and its rather small state space is less complex than most real-world systems. Furthermore, this approach remains limited to the capabilities of the underlying dispatching rules. The given use case is less complex than many other approaches in research, e.g. [36], where a policy is identified independent of any heuristics. However, finding a policy is equivalent to discovering a wide variety of heuristics that have not been defined.

Especially with the development of DQNs in 2013, progress was also made in applying deep reinforcement learning in the context of production systems. Thus, the relevance of deep reinforcement learning in this field increased. An analysis of papers published between 2010 and 2021 on deep reinforcement learning in production systems [31] illustrates its rising importance.

A highly regarded paper using DQNs was published in 2018 by Waschneck et al. [43]. They apply deep reinforcement learning to production scheduling in complex job shops. For that purpose, cooperative DQN-agents are used. Each work center possesses a DQN-agent which is trained separately. However, they can monitor and consider the other agents' actions. This is contrary to multi-agent approaches like [4], [36] and [10] which considered only agents with a local view but a global goal. Waschneck et al.'s approach consists of two phases. At first, for each work center a DQN-agent is trained while the other work centers are controlled by a heuristic. The next step is to train the independently trained DQN-agents in cycles. As a contrast to the first phase, now all work centers are controlled by the previously trained DQN-agents. Due to this two-phase approach including a basic training and an improvement step the training can be completed faster. Nevertheless, the results show that policies discovered remain at the level of heuristics without requiring human knowledge.

So far, most research on reinforcement learning in production systems has been conducted in the context of job shop scheduling, as shown by the approaches described earlier. However, they only answer the question of which job to process next, but ignore the availability of transportation resources required to deliver that job to a particular machine. As explained in Chapter 2, this assumption simplifies the job shop scheduling problem, reduces the complexity and limits the applicability to real-world problems. In

the following, we concentrate on approaches that take material handling into account when applying reinforcement learning to production scheduling problems.

According to Heger and Voß [12], material handling can be viewed through two different lenses. On the one hand, the handling system can be responsible for which job to process next. On the other hand, delivery scheduling can be a selection of which vehicle has to carry the next job from one machine to another. While a lot of work has been spent on optimising production scheduling using reinforcement learning, often vehicle selection was still done heuristically according to priority rules, e.g. shortest travel time (SST), longest idle vehicle (LIV) or least utilised vehicles (LUV).

However, it should be noted that the issue of vehicle selection is only relevant in a multi-vehicle environment.

Arviv et al. [2] applied reinforcement learning on a flow shop problem with job transfer robots in a setting where jobs are known previously. They deal with environments with multiple transportation robots. Therefore, they decompose the robot system into subsystems of equal size that contain only a single robot. These different subsystems are distinct and each robot is responsible for transportation between a sequence of specific machines. Thus, the entire scheduling problem is solved by tackling each single-robot scheduling problem and combining the results. The reinforcement learning problem is defined with the objective of minimising the makespan. A comparison with an optimal solution indicates a low error rate.

However, the decomposition of a system into several distinct subsystems can also cause difficulties. In particular, since a flow shop production line is considered, the performance of one robot directly influences the quality of the following subsystems.

In 2018, another approach was published by Xue et al. [46] which enables each automated guided vehicle (AGV) to decide for itself which task to transfer next. In contrast to [2], all vehicles are defined over the same action space and are not limited to particular machines. The AGV scheduling policy is trained using Q-Learning with focusing on minimising the total makespan. Since both AGV waiting time and job waiting time affect the makespan, AGVs are rewarded minimising either AGV waiting time or job waiting time. Each vehicle that finishes a transfer task has to decide which job to transfer next. Thus, an agent is trained for each AGV independently. However, AGVs applying the same reward function are able to share their learned knowledge through a common Q-table to speed up training and assure convergence. Therefore, those AGVs make use of the same policy. Furthermore, the state space considered when choosing an action contains information about the entire plant. This enables for a more cooperative behaviour.

While Xue et al. [46] realise cooperation between different AGVs through a global common state, Popper et al. [34] use a central coordination layer to organise collaboration between agents and the absence of conflicts in their decisions. However, they developed a multi-agent reinforcement learning system where an agent is assigned to each machine or transport unit. In contrast to previous approaches, they considered AGVs in production scheduling problems by tackling two scheduling problems simultaneously - AGV schedul-

ing and production scheduling. A deep reinforcement learning method called proximal policy optimisation is applied. At first, each agent is trained independently. After that, all pre-trained agents are trained cooperatively. Experiments show that reinforcement learning applied on both machines and transportation yields better results than combining one of them with heuristics. On the one hand, this approach enables control of automated guided vehicles, but on the other hand, two separate scheduling problems have to be solved.

Another way to deal with AGVs in a multi-agent reinforcement learning problem is presented by a simulation-based deep reinforcement learning method [9]. They focus on flexible job shops where material and semi-products are delivered by AGVs. Unlike previous multi-agent problems, in this work a policy is learned for each agent without any knowledge about other agents. Thus, no cooperation is required but no global goal is pursued either. Each AGV agent is trained using double DQNs. Since they face a continuous task problem, the training takes place within a defined simulation time. While other approaches only allow valid decisions, the proposed method does not restrict the action space for a given environment state. Instead, a learning episode is terminated before simulation time expires if an invalid action is chosen. This results in a faster learning progress. Allowing invalid actions and cancelling a learning episode early has some similarities to the counterfactual reasoning approach for reward machines introduced in Chapter 6. However, the CRM algorithm goes beyond the idea presented by Feldkamp et al. [9].

In 2020, a multi-agent reinforcement learning approach [25] was developed to cope with dispatching of autonomous mobile robots (AMR). In contrast to AGVs, AMRs are able to move around without any predefined paths. A learning agent is assigned to each robot and each develops its own policy independent from other robots. However, the agent cannot decide on its own which job to take on next, instead it has to bid on a transportation order. Then, the environment chooses the agent with the highest bid from all agents with less than two orders. Thus, the environment controls the dispatching process from a central point of view. Although agents have to compete against each other, cooperation is supported through an equal reward design for all agents. Due to a large state space and a continuous action space resulting from bidding, an extension of the deep deterministic policy gradient approach approximating the Q-function is chosen. This approach outperforms existing dispatching heuristics such as a nearest vehicle first rule.

The complexity of the problem increases if a vehicle is able to carry more than one part at a time. In 2015, such a multiple-load carrier scheduling problem was tackled using reinforcement learning. In contrast to [2], [46], [34], [9] and [25], the considered production environment employed conveyors to deliver semi-products from one machine to another, however line-side buffers were handled by robots. Thus, this scheduling approach [5] focuses on replenishing a buffer in time and scheduling an appropriate drop-off sequence. The reinforcement learning technique is applied to determine the best action in a given state according to a combination of maximal throughput and minimal transportation distance. These actions correspond to doing nothing and applying a heuristic dispatching

algorithm with different base rules. Overall, this reinforcement learning problem is trained to request and deliver parts optimally. Experiments indicate that reinforcement learning is a suitable approach to deal with a multiple-load carrier scheduling problem. However, since this approach employs heuristic dispatching algorithms, it remains on the level of heuristics and is not able to create an entire policy.

Although gantry robots can be considered as automated guided vehicles, the reverse is not true. Movement of both types of robots is fixed to a predefined lane, however the constraints are much stronger for gantry robots since they are connected to a gantry. Such a fixed path restricts the reinforcement learning problem to a discrete action space. On the other hand, multiple gantry robots in a work station do not have any opportunity to sidestep evasion which makes gantry robot scheduling a complex problem. There are only a few research papers that deal with gantry robots and their limitations. This master thesis focuses on this particular type of robot as a special case of general transportation scheduling. Therefore, the work on gantry robots is examined in more detail below.

In 2018, a research group evaluated the application of reinforcement learning for this particular use case [28]. They considered the impact of disruption events on the productivity. In addition to random disturbances such as machine failure, they also looked at a lack of material due to delayed gantry operation. It is precisely these disruptions that lead to idle machines and thus have a negative impact on productivity. Therefore, the reward function of the reinforcement learning problem takes these potential losses on the productivity into account. Since a stoppage of the slowest machine M^* leads to a permanent production loss, a ratio of stop duration D^{M^*} and speed $T^{M^*}_{Processing} + T^{M^*}_{Transportation}$ of the slowest machine M^* quantifies the immediate loss (7.0.1).

$$PL(T) = \frac{D^{M^*}}{T^{M^*}_{Processing} + T^{M^*}_{Transportation}} \tag{7.0.1}$$

Furthermore, future system performance is affected by the present state, hence a potential production loss is also considered. This is computed as the expected value of the production loss (Equation (7.0.2)) assuming a disruption event e would happen at the current moment t'.

$$PLR(T) = E[PL_{e^{t'}} | t' = T] \tag{7.0.2}$$

The risk of production loss takes the vulnerability of a system into account. Based on the immediate production loss $PL(T)$ and the production loss risk $PLR(T)$, a reward function r is designed.

$$r = -PL - \eta \cdot PLR \tag{7.0.3}$$

The factor η balances the trade-off between immediate and potential production loss. Based on this reward function, a gantry scheduling policy is learned using the Q-Learning approach. A comparison with a first-come-first-serve heuristic shows that the Q-Learning approach leads to less production loss.

However, Ou et al. [28] consider only a small state space of 48 different states in their experiment. Therefore, a tabular method already provides good results. Moreover, they

analyse a flow shop production system which is considered as less complex than a job shop production environment. Therefore, this approach reaches its limits with increasing complexity as observed in real-world production systems.

Besides this Q-Learning approach, another method for gantry robot scheduling was introduced in [30] a few years later. Ou et al. build on the Q-Learning approach explained in [28]. As in [28], they use a trade-off between immediate and potential production loss as reward function. Their goal is to obtain better policies for gantry robot scheduling and converge faster. For this purpose, a model-based approximate dynamic programming (ADP) approach is integrated in the Q-Learning algorithm. Ou et al. point out that the Q-Learning approach ignores relevant aspects of the production system that could impact the policy exploration. On the other hand, the exclusive use of an ADP approach may lead to biases depending on the design of the transition probabilities. To overcome these disadvantages, the Q-ADP-Learning method combines the advantages of both approaches. First, the Q-values are updated by the Q-Learning update rule. In a next step, the planning method is performed. As in dynamic programming approaches, it is iterated over a predefined number of paths for a certain number of time steps. This generates short sample paths. At each time step, the Q-value is adjusted considering the underlying transition model. The whole process runs until a defined number of simulations is reached or the Q-values converge. Experiments show that this approach performs better than the previously described standard Q-Learning for the same training duration. In particular, the repeated updating of Q-values due to the extension by ADP accelerates the learning process.

Both, [28] and [30] use a combination of immediate and potential production loss as a reward function. At the same time, both aim to maximise the production output. Their choice of this reward function is based on a comparison of five different reward functions under various assumptions in [29]. They analysed five reward functions which based on different levels of understanding of production systems. The first reward function prioritises the last machine, as the output of this machine determines the output of the production line. This machine is given a high priority by penalising waiting time. The second reward design takes the production efficiency of machines into account. This is realised by the proportion of failure time towards a sum of failure and repair time. Such a reward function leads to higher priority for machines of lower efficiency. This is implemented by penalising the waiting time of this machine. In order to not only look at individual machines, the entire waiting time is considered in a third design approach. Thus, a punishing reward for each machine is included proportionately. In contrast to these three reward function ideas, the production loss or a combination of production loss and production loss risk is formulated as the fourth and fifth reward design. All of these five reward functions are employed with Q-Learning. Furthermore, a first-come-first-serve heuristic is used as baseline for comparison. Experiments show that different reward functions lead to different policies and thus outcomes. Reward function five achieves the highest production output, while reward function four is only slightly lower. Reward functions one and three, however, remain at the level of the basic heuristic. [29] This

observation can be justified by the different reward function designs. While the first three reward designs operate more as a fixed priority rule and thus rather heuristically, the last two take into account the dynamics of the environment. The first three reward functions therefore run the risk of prescribing a path instead of leading to the goal as described in Section 5.5. Furthermore, all five reward functions make use of negative rewards and try to minimise any punishment. Most research regarding reinforcement learning does not even reveal which reward function was applied. This paper demonstrates the relevance and difficulty of designing reward functions.

All in all, this summary of previous related work shows that much research has been done in the area of reinforcement learning on job shop scheduling problems. Furthermore, these applications often made use of state-of-the-art deep reinforcement learning approaches. However, most research simplified the job shop scheduling problem by considering a fixed number of jobs, specified in advance. This corresponds to an episodic task and limits its real-world applicability as well as the complexity arising from a continuous scheduling problem. Another simplification is that it is often not taken into account how parts can be transported from one machine to another. Few research groups have considered this constraint in the form of automated guided vehicles, and even fewer have investigated material handling by gantry robots which is the focus of this master thesis.

Moreover, most papers concentrate on the algorithm to solve the MDP and not on the computation or fine-tuning of a reward function. Only [29] analysed different reward functions and their effects on learning. However, in [9] and [30], the goal was not only to find an optimal policy, but also to obtain this policy faster. To our knowledge, techniques to better exploit the design of reward functions such as reward machines have not yet been employed in the production context in general and in the scheduling of gantry robots in particular. This research gap is addressed in the following.

8 Exploitation of Reward Machines for Continuous Tasks

Continuous tasks represent a more complex setting than episodic tasks. Therefore, in continuous tasks, it is much more important to exploit the internal structure of reward machines in order to realise more efficient training. First of all, the differences in the complexity of these two types of tasks and especially in the design of their reward machines are explained. Furthermore, due to these differences in complexity, it needs to be analysed whether the previous definition of reward machines and the existing learning approach CRM can be applied to continuous tasks as well. Possible challenges are discussed and modifications are proposed to cope with these challenges. However, these adjustments itself can make further changes necessary.

8.1 Reward Machine Design

The different characteristics of continuous and episodic tasks also influence the design of a reward machine. Some aspects in the design of episodic tasks contradict the general definition of continuous tasks. Therefore, these aspects need to be identified and the design criteria for continuous tasks need to be made clear.

Difficulties

A reward machine of an episodic task has a well-defined end and reaches an accepting state when a particular task is completed. Hence, the last x-steps that need to be executed to reach a terminal state are well known. In terms of automata theory, the accepting words of an episodic task have a specific suffix. Such a suffix corresponds to a finite word that is concatenated with another finite word. Moreover, reward machines of episodic tasks need not contain cycles. In fact, however, these reward machines often include cycles of length one, which means that the word can remain in the same state and does not change to any other state. In episodic tasks, however, these cyclic states are usually not accepting states. Unlike an episodic reinforcement learning problem, a cycle-free automaton can never represent a continuous task, because the repetition of subtasks requires loops.

In general, one can say that when extending an episodic task to a continuous task, the episodic task represents a part of the continuous one. Therefore, the continuous task can be considered as generalisation.

Adjustments

As explained above, the design of a reward machine for a continuous task has to be structurally different from that for an episodic task. From the above problem statement, different requirements can be derived.

On the one hand, the reward machine for a continuous task has to enable infinite repetitions of several working steps. However, these different states as part of the repetition are not considered to be visited periodically. Furthermore, not each state of the reward machine has to be part of these infinitely visited states. For example, a few states may constitute a prefix of the input and do not have to occur later on. A prefix conforms to a finite word that can be concatenated with an infinite word as explained in Chapter 3. On the other hand, the words represented by the reward machine for infinite tasks must not contain a suffix. This results from the fact that an infinite task never ends and therefore, cannot reach a point from which on always the same sequence is executed that leads to a terminal state. As mentioned in Chapter 3, an infinite word cannot be concatenated with a finite word where the infinite word builds the first part. It becomes clear that the direction of concatenation of finite and infinite words is significant, since prefixes are allowed while suffixes cannot occur.

Besides, the reward machine has to contain at least one cycle that includes at least two distinct states. This goes along with a reachability criteria. The reward machine for a continuous task has to contain at least two distinct states u_i and u_j where u_i is reachable from u_j and vice versa.

Running Example The reward machine of the running example introduced in Figure 6.4 considers an episodic task problem. Therefore, the successful delivery of a processed element at the output conveyor completes the task and thus leads to a terminal state. The same problem, extended to a continuous task, would require the completion of not only one element, but infinitely many. This extension from an episodic to a continuous task is evident by the change of the transition which was labeled $(empty \land finished, 5)$ in Figure 6.4. This logical formula is not allowed to mark an end of an episode anymore, instead it has to lead to a transition to the state *Loader empty*. Hence, a loop from that state to a previous state is required, from which all steps necessary for completion of a product can be repeated which is illustrated in Figure 8.1 (a detailled illustration of the reward machine with logical formulas can be found in the Appendix in Figure 11.1). Furthermore, the terminal states reached because of wrong behaviour are still part of this reward machine. This example points out that the episodic task is just a restricted case of a continuous task.

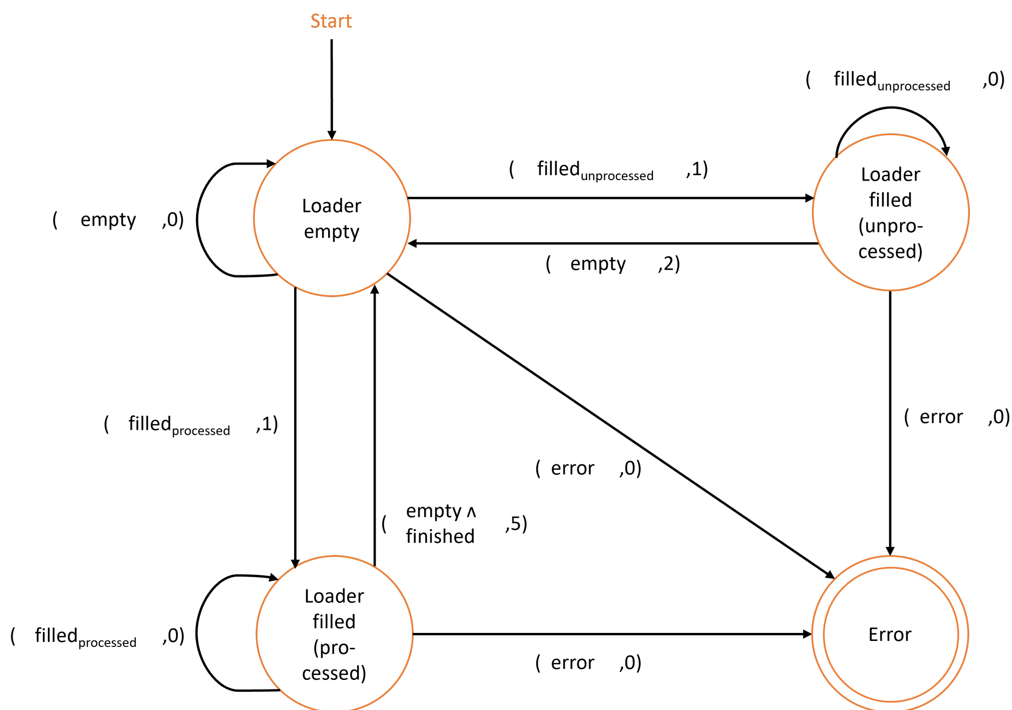


Figure 8.1: Adjusted design of the reward machine for a continuous task

8.2 Reward Machine Definition

Since the general concept of reward machines has been geared towards episodic tasks, it has to be analysed whether the general formal concept is also applicable to continuous tasks.

Difficulties

As explained in Chapter 6, reward machines are Mealy-automata and thus belong to the group of finite state machines. However, finite state machines are restricted to the use of finite input sequences since infinite inputs can never reach a terminal state, which was mentioned in Chapter 3. Because continuous reinforcement learning problems can be considered as infinite inputs to the reward machine, the original definition with its terminal states is not applicable to non-episodic tasks. Therefore, adaptions have to be made to the definition of the reward machine and in particular to the set of acceptance states F in order to work for continuous reinforcement learning problems as well.

In Chapter 3, different acceptance conditions for infinite input sequences have been introduced. However, these acceptance conditions are restricted to infinite words and in turn are not applicable to finite words. Although we focus on the applicability of reward

machines for continuous tasks, a run that terminates the task prematurely, e.g. due to an incorrect behaviour, must also be considered in order to work for real-world examples. Therefore, the reward machine definition has to be extended in a way that both finite and infinite words can be accepted.

Adjustments

In order to deal with both finite and infinite words, we propose a combination of both criteria, the terminal states concept of an episodic reward machine and the acceptance criteria for infinite input sequences. As long as an input sequence did not reach a terminal state, the acceptance conditions for infinite inputs are applied. However, when a state is reached from which no transition to any other state is possible, the set of terminal states F is used to evaluate whether the word is accepted.

Since a set of terminal states is already part of the reward machine definition, only acceptance criteria need to be added. Further, all of the introduced criteria of automata accepting infinite languages are interchangeable, which is why any of them can be chosen. Therefore, we apply the Muller acceptance condition. This condition defines the set of accepting states as set of sets $\mathcal{F}$. A word w is only accepted if states that occur infinitely often in that word, defined as $inf(w)$, are an element in $\mathcal{F}$. To combine the requirements formulated in Section 8.1, the states describing the subtasks that are repeated over and over again are visited infinitely often and must, therefore, be part of the accepting condition. This condition ensures that infinite words can be accepted as well.

All in all, two different conditions for accepting words are infered. On the one hand, words are accepted if they reach a terminal state in F. On the other hand, a given word is accepted if the set of states visited infinitely often during this run is an element in $\mathcal{F}$. These two different conditions can be combined as follows:

$$\delta_u(u_0, w) \in F \lor inf(w) \in \mathcal{F} \quad \forall w \in (2^P)^*$$

It has to be mentioned that a word w that satisfies the first condition has to have an empty set $inf(w)$ since this word is considered finite. Besides, states of F cannot be elements of sets in $\mathcal{F}$. Otherwise, the nature of F would be inconsistent. Furthermore, due to this strict distinction between elements of F and $\mathcal{F}$, the extension to infinity has no impact on the state-transition function δ_u since the states that can be visited infinitely often are elements of $U \setminus F$.

To be consistent with the general definition of automata and as a consequence reward machines, F and $\mathcal{F}$ do not both become part of the reward machine definition. Instead, the combination of these two acceptance conditions is formulated as $Acc = F \cup \mathcal{F}$. As a consequence, a reward machine for continuous tasks is defined as a tuple $R = (U, 2^P, \delta_u, \delta_r, u_0, Acc)$.

Running Example These changes in the definition of the reward machine also necessitate adjustments in the reward machine of the running example. The set of terminal states

$F = \{Error\}$ highlighted by double-framed circles in Figure 8.1 remains unchanged, while $\mathcal{F}$ has to be defined. Since the given exemplary task corresponds to a repetition of the two distinct tasks - delivering an unprocessed item to a machine and delivering a processed element to the output conveyor -, states that are part of these tasks have to build an element in $\mathcal{F}$. We define $\mathcal{F}$ as follows:

$$\mathcal{F} = \{ \{Loader\ empty, Loader\ filled(unprocessed), Loader\ filled(processed)\} \}$$

A sequence that, for example, visits the states *Loader empty* and *Loader filled(processed)* infinitely often, but the state *Loader filled(unprocessed)* only finitely often, cannot be accepted by the reward machine. This results from the fact that the set $\{Loader\ empty, Loader\ filled(processed)\}$ is not an element in $\mathcal{F}$. The newly introduced acceptance condition *Acc* would result in

$$Acc = \delta_u(Loader\ empty, w) \in \{Error\} \lor$$
$$inf(w) \in \{ \{Loader\ empty, Loader\ filled(unprocessed), Loader\ filled(processed)\} \}.$$

8.3 CRM Learning Approach

As pointed out above, the general design of a reward machine as used for episodic tasks as well as the definition of reward machines need to be extended for the specific properties associated with continuous tasks. Therefore, the question arises whether the CRM learning approach to exploit the internal structure of the reward machine is still applicable to a reward machine of a continuous task.

Difficulties

Icarte et al. [17] demonstrated a successful use of the CRM algorithm for continuous tasks under the assumption that the set of terminal states is empty. However, the reward machine used in the experiment did not take into account the specific aspects that come with continuous tasks as explained above. Moreover, we argued in Section 8.2 that a reward machine representing a continuous task may have terminal states as well. Therefore, difficulties arise with reward machines that set these restrictions of an empty set of terminal states aside and allow a more complex environmental setting.

Furthermore, due to the requirements for the design of reward machines and the introduction of acceptance criteria that were justified above, the role of the states that fulfil the acceptance criteria must also be examined.

First, the condition of the while-loop in line 7 in Algorithm 3, which interrupts an episode as soon as a terminal state of the reward machine is reached, stands in the way of a more efficient learning process as promised by the concept of reward machines. On the one hand, the first part of the condition requires that no terminal state of the environment is reached. This aspect is always given in a continuous task since there is no clear end.

This is true no matter whether F is empty or not. Thus, only the second part of the while-condition is relevant. This part of the condition checks whether a terminal state F of the reward machine is reached. If the limitation to $F = \emptyset$ as suggested by Icarte et al. [17] would be valid and there would be no terminal states in the reward machine, then the while-loop goes on for ever and is never interrupted. From an algorithmic point of view, this leads to one infinite episode and thus to a never-ending training. However, in practice this might be avoided through a predefined simulation time.

More relevant, however, is the aspect, when F is not considered empty. In Section 8.2 we argued that for continuous tasks terminal states could be part of the reward machine as well which contradicts the assumption $F = \emptyset$. However, the frequency these terminal states can be achieved is crucial. If at any step of an episode there is a chance that such a terminal state will be reached, the episode will terminate without an advanced state being observed. An early end of an episode would initiate a new episode which starts exactly at the same beginning s_0 and u_0 as all previous and all subsequent episodes. Especially, at the beginning of the trial-and-error process, the exploration success would be very limited. It is worth to mention that this difficulty of frequent early ends is not only valid for continuous tasks in particular but for complex tasks in general. Lower frequencies of terminals states could arise from restrictions of the action-space. However, reward machines should be able to learn these kinds of behaviour as well.

These difficulties stand in the way of a more efficient learning process as promised by reward machines. It can be deduced that the CRM learning approach is not as beneficial for complex continuous tasks as for (simple) episodic tasks, whereby the limitation results more from the complexity of the task and less from its infinity.

Moreover, an analysis of the CRM algorithm makes it clear that states that are elements of sets in $\mathcal{F}$ do not need to be treated specially, since they do not mark an end of the current episode, which would have resulted in a consideration of $\mathcal{F}$ in the if-statements in line 24 - 28.

Nevertheless, adjustments need to be made in order to benefit from the successes of the general CRM approach also in complex continuous tasks as they occur in practice.

Adjustments

As argued above, in a complex continuous task, it does not seem reasonable to cancel an episode when a terminal state is visited. This results from the fact that error states, which should also be reproducible by reward machines, can lead to an abortion of the current episode in each iteration and thus preclude the collection of further useful information. To prevent this, the while-condition is considered to be always true, or in practice limited by a reasonable simulation time.

However, this raises the question how the reward machine should react if such terminal states are reached due to actions that are indeed incorrect but can be managed by the environment. This is a relevant question because in that case a reward machine already has a terminal state as its current state. Therefore, the handling of these states requires a

transition from a terminal state back to an element of $U \setminus F$. This is necessary to proceed the current episode. However, as mentioned earlier, this handling must work in such a way that a terminal state cannot occur infinitely often in the set of states of an infinite input sequence. To achieve this, on the one hand, the terminal state still has to be considered as terminating. On the other hand, a forwarding must take place from this terminal state, so that the terminal state is passed but never visited. The agent then never has to decide what action to execute next in that state.

However, a forwarding realised as a transition from a terminal state F to a state of the reward machine $U \setminus F$ is not covered by the state-transition function δ_u as one can see in the definition of $\delta_u : U \setminus F \times 2^P \to U$. Therefore, an additional function $\delta_f : F \times 2^P \to U \setminus F$ must be introduced. This function δ_f forces transitions from a terminal state of F to a state of the reward machine $U \setminus F$. Hereby, the next non-terminal state of the reward machine depends on the current state-action-state triple as observed in the environment. This triple is used since the environment ensures a transition from one valid to another valid state although an incorrect or impossible action was executed.

Furthermore, handling an incorrect action does not mean to redirect to the latest state before the reward machine reached a terminal state F. This would not work since the environment processes the chosen action and transitions to a valid state. More importantly, the time would go on and therefore a roll back of time in a simulation environment is impractical. Instead, the general idea is to get back to a state in the reward machine that is consistent with the state transition in the environment.

This is achieved by applying the labelling function L on the terminal state of F under consideration of the current state-action-state triple of the environment. As usual, these inputs map on a logical formula of the alphabet 2^P. The labeler outputs an additional letter σ_t^+, so that it appears to the agent that σ_t and σ_t^+ are concatenated and considered as one input letter.

The terminal-state function δ_f could be simplified to $\delta_f : F \times [S \times A \times S] \to U \setminus F$ without application of the labelling function since the transition shall only align with the last transition in the environment. However, we choose to use the first definition in order to be consistent with the state-transition function δ_u. Furthermore, since there is only one transition function in a reward machine, both functions δ_u and δ_f need to be combined. This transition function would be defined as follows:

$$\delta : \begin{cases} \delta_u, & \text{if } U_t \in U \setminus F \\ \delta_f, & \text{if } U_t \in F \end{cases}$$

Nevertheless, the distinction between δ_u and δ_f is necessary to make clear that these terminal states are considered as terminating. Therefore, experiences that represent the transition to such a terminal state are added to the replay memory as terminal. Therefore, line 24 - 30 of Algorithm 3 remains unchanged in the adjusted Algorithm 4.

In contrast to regular transitions in the reward machine, a transition from an element in F to a state in $U \setminus F$ is forced. This is implemented in line 33 - 36 in Algorithm 4.

Furthermore, it is worth to mention that the transition by δ_f does not lead to the generation of additional counterfactual experiences. Moreover, no experience covering this transition is added to the replay memory.

All in all, the introduction of the additional transition function δ_f impacts the reward machine, since δ_u and δ_f are now combined in a transition function δ. This combined transition function is necessary to satisfy the general formal definition of finite state machines. However, in the following, the specific transition functions will be used.

Since reward machines are Mealy-automata and therefore make use of an output-function, the reward-function δ_r has to be extended as well. However, transitions of δ_f should not impact the behaviour of the agent and furthermore, cannot even contribute to the return since no experiences of these transitions are added to the replay memory. With that in mind, the reward-function is extended similar to the transition function δ. Therefore, δ_r is adjusted as follows:

$$\delta_r : \begin{cases} U \times 2^P \rightarrow \mathbb{R} & \text{if } U_t \in U \setminus F \\ 0 & \text{if } U_t \in F \end{cases}$$

Although this reward has no relevance in the agent-environment interaction, for completeness it is specified in line 36 of the algorithm.

Furthermore, as the general idea of the CRM approach is not affected by the proposed changes, it is assumed that the advantages of this learning approach, such as convergence to optimal policies, are still valid.

Algorithm 4 Adjustment of Deep Q-Learning algorithm with CRM

1: Initialise replay memory D of length N
2: Initialise Q-Network with random weights θ
3: Initialise target network with weights $\theta^- \leftarrow \theta$
4: **for** each episode **do**
5: Initialise state of environment $S_t \leftarrow S_0$
6: Initialise state of reward machine $U_t \leftarrow U_0$
7: **while** True **do**
8: Choose action A_t according to $\varepsilon - greedy$ approach
9: Execute action A_t and observe next state S_{t+1}
10: Compute input symbol $\sigma_t \leftarrow L(U_t, (S_t, A_t, S_{t+1}))$
11: Compute next state of reward machine $U_{t+1} \leftarrow \delta_u(U_t, \sigma_t))$
12: Compute reward $R_{t+1} \leftarrow \delta_r(U_t, \sigma_t)$
13: Store experience in replay memory D
14: **for** each $U_t' \in U \setminus F$ **do**
15: Compute input symbol $\sigma_t' \leftarrow L(U_t', (S_t, A_t, S_{t+1}))$
16: Compute next state of reward machine $U_{t+1}' \leftarrow \delta_u(U_t', \sigma_t'))$
17: Compute reward $R_{t+1}' \leftarrow \delta_r(U_t', \sigma_t')$
18: Set experience $E_t' \leftarrow (S_t, U_t', A_t, R_{t+1}', S_{t+1}, U_{t+1}')$
19: Store experience E_t' in replay memory D
20: **end for**
21: Sample random mini-batch from D
22: **for** each $E^* \in mini - batch$ **do**
23: Perform forward pass through Q-network and receive $Q(S_t^*, U_t^*, A_t^*, \theta)$
24: **if** S_{t+1}^* terminal or $U_{t+1}^* \in F$ **then**
25: Approximate target Q-value $Q(S_t^*, U_t^*, A_t^*, \theta^-) \leftarrow R_{t+1}^*$
26: **else**
27: Approximate target Q-value
$Q(S_t^*, U_t^*, A_t^*, \theta^-) \leftarrow R_{t+1}^* + \gamma \cdot \max_{A_{t+1}^*} Q(S_{t+1}^*, U_{t+1}^*, A_{t+1}^*, \theta^-)$
28: **end if**
29: Compute loss as $(Q(S_t^*, U_t^*, A_t^*, \theta) - Q(S_t^*, U_t^*, A_t^*, \theta^-))^2$
30: **end for**
31: Backpropagate error of entire mini-batch with respect to network parameter θ
32: Every C steps set $\theta^- \leftarrow \theta$
33: **if** $U_{t+1} \in F$ for observed experience **then**
34: Compute input symbol $\sigma_t^+ \leftarrow L(U_{t+1}, (S_t, A_t, S_{t+1}))$
35: Adjust $U_{t+1} \leftarrow \delta_f(U_{t+1}, \sigma_t^+)$
36: Adjust $R_{t+1} \leftarrow 0$
37: **end if**
38: $S_t \leftarrow S_{t+1}$
39: $U_t \leftarrow U_{t+1}$
40: **end while**
41: **end for**

8.4 Reward Machine Visualisation

These proposed modifications in the CRM learning approach also bear on how the concept of a reward machine is visualised. This is examined in the following.

Difficulties

So far, the illustration of reward machines does not consider transitions from terminal states F to states of the reward machine $U \setminus F$. However, in order to incorporate the existence of the added transition function δ_f, changes need to be made in the visualisation.

Adjustments

Transitions executed by state-transition function δ_u will be illustrated as usual, since the effect of these terminal states does not change. However, it must be made clear that the terminal states of F continue the episode although every further input symbol would usually leave the state space of the reward machine. Since this should be prevented by a specific transition function δ_f, a transition has to take place that is forced if a terminal state of F is reached in the reward machine. These transitions of δ_f are illustrated as dashed lines. Furthermore, since transitions initiated by δ_f carry only a dummy reward value according to δ_r, these edges are specified by a numerical value of zero.

Running Example Figure 11.2 represents the reward machine of the given example considering all proposed adjustments. The difference towards Figure 8.1 consists in the three additional transitions, from the terminal state to each state of the reward machine that is not terminating, illustrated as dashed lines.

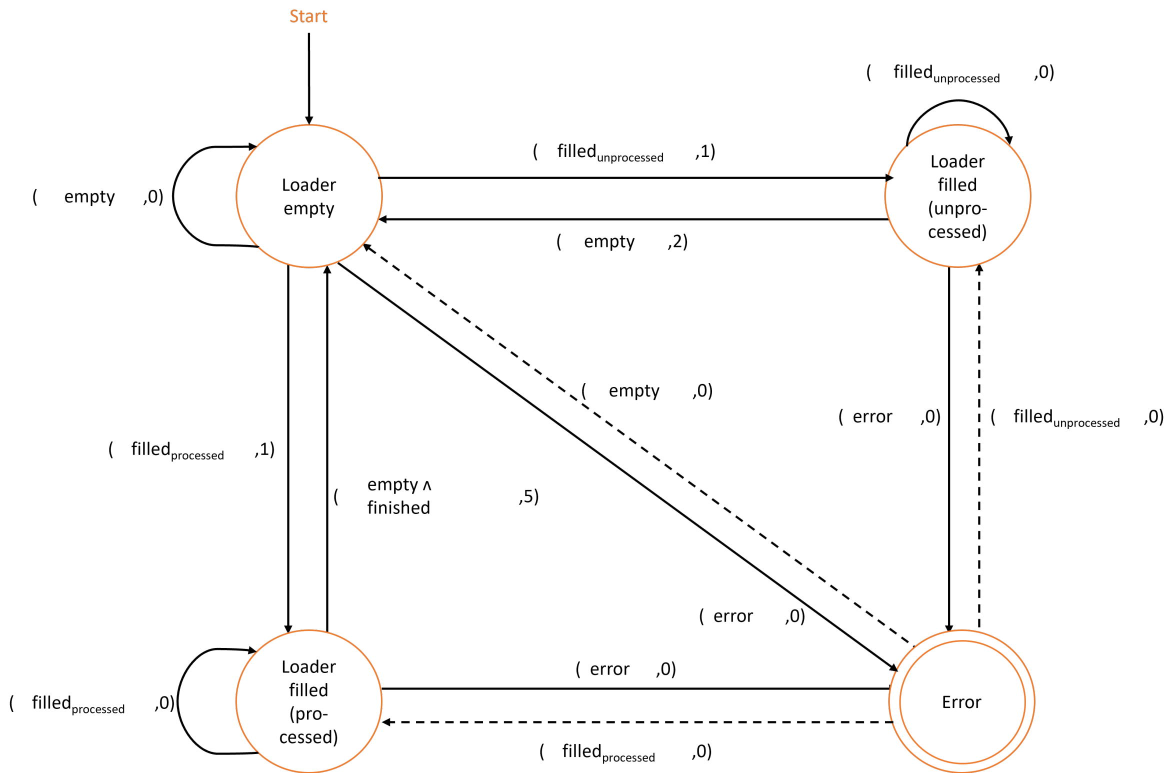


Figure 8.2: Adjusted reward machine for a continuous task

8.5 Summary of Adjustments

Several adaptations for continuous tasks were derived from different perspectives. Some of these changes led to further modifications at different levels. In the following, a brief summary of the adaptations mentioned is provided.

The most fundamental modification arises from the infinite repetitions of subtasks, which require the existence of at least two mutually reachable states at any point in time. In addition, the design of the reward machine prohibits a unique suffix for a continuous task.

In addition to the question how to design a reward machine for continuous tasks, aspects regarding the reward machine definition were analysed. On the one hand, we replaced the set of terminal states F by an acceptance condition Acc that can deal with both finite and infinite input sequences. However, this new acceptance condition takes the original set of terminal states F into account and combines it with a Muller acceptance condition.

Moreover, the transition function in a reward machine was extended by a terminal-state-transition function δ_f which was merged with the state-transition function δ_u to a general transition function δ. This change became necessary to prevent myoptic learning and to enable greater collection of useful information for complex tasks which would be rather limited if each episode terminated as soon as a terminal state was reached. At the same time, the reward function δ_r was extended for terminal states by assigning a dummy reward of zero to each transition from a terminal state to a non-terminal state.

These modifications in the definition lead to a reward machine formalised as $R = (U, 2^P, \delta, \delta_r, u_0, Acc)$.

Moreover, the introduction of the terminal-state-transition function led to a minor change in the visualisation of a reward machine. To distinguish the properties of δ_u and δ_f, different types of edges were introduced. A relevant difference consists in the fact that δ_f is forced to be executed when a terminal state is reached. Nevertheless, transitions triggered by δ_f do not lead to any additional experiences in the replay memory, nor do they play any role in the calculation of the return. Instead, these transitions merely bridge back to a non-terminating state of the reward machine.

9 Experiment

This chapter aims to demonstrate the capabilities of the developed approach. Since this approach addresses the exploitation of reward machines for continuous tasks in general, the experiments deal with the gantry robot scheduling task in particular, which is the main application field of this thesis. As the overall goal of this thesis is to find an optimal production schedule for a continuous problem faster, the outcome of this learning approach will be compared with standard approaches. First, section 9.1 provides an insight into the general experimental setup and the used example production environment. Next, the reward machine of the given production environment is deployed. Then, both reward machine based approaches (the adjusted reward machine approach and the original reward machine approach) and a basic learning approach without reward machines are applied on the given production environment. Moreover, general observations for each model will be summarised and in Section 9.3 the results of the experiments will be presented.

9.1 Experimental Setup

This section on the experimental setup contains both technical details about the implementation and information about the environment on which experiments of this master thesis are based.

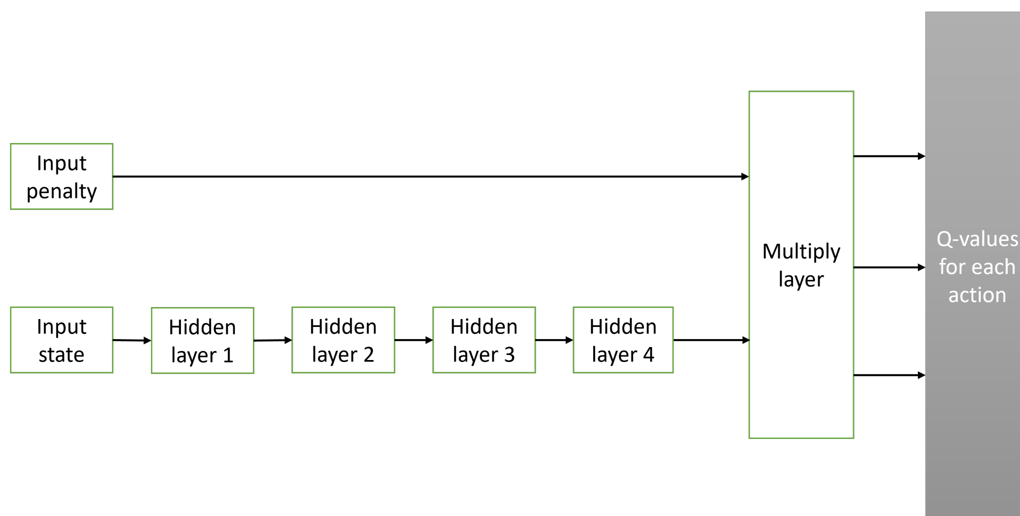
9.1.1 Experimental Design and Implementation

General Implementation Settings Performing interactions between an agent and a physical environment is too expensive and impractical just to collect information about how a system acts in a particular state. Therefore, a digital twin is necessary to train a reinforcement learning agent. Such a digital twin as explained by Makarov et al. [24] facilitates obtaining data from a real system (physical twin) by interacting with a virtual model (digital twin). Further, a digital twin is able to lead to changes in the system during interaction. This allows an agent to choose an action and to observe how the environment reacts to that action.

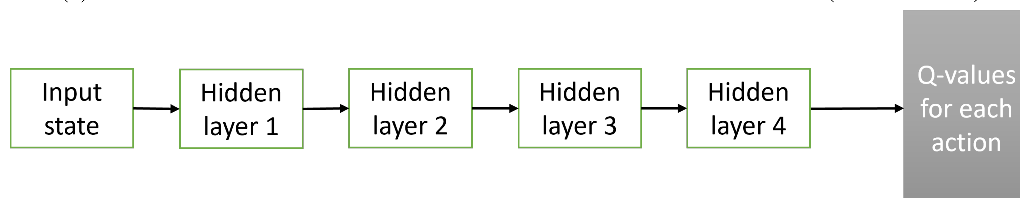
In this experiment, a digital twin of a production environment implemented via AnyLogic 8.7, a java-based simulation engine, is provided.

Implementation of the Reinforcement Learning Agent The different reinforcement learning approaches are conducted in Python 3.9. The DQN agent relies on the

software library tensorflow (version 2.9) which provides neural network specific classes and simplifies building a neural network model. The DQN implementation slightly differs between the basic learning approach and those employing reward machines. Figure 9.1a illustrates that a basic model without reward machine applies an additional layer (Multiply layer) and an additional input (Input penalty) in order to correct output values by penalties. This additional layer multiplies each output values with zero or one. Actions that are not possible or allowed for the current input state are suppressed by multiplication with zero. The output value of allowed actions is not affected, since a multiplication with the one takes place. As Figure 9.1b visualises, this additional input is not necessary for models with reward machine. This results from the fact that a reward machine does not require to restrict the action space depending on the current input. Instead, it is applied in order to learn directly which behaviour is allowed and which is not. However, both types of



(a) Neural network architecture of the models without reward machines (basic models)



(b) Neural network architecture of models with reward machine

Figure 9.1: DQN architecture of models with and without reward machines

models receive an input of ten data points containing information about the current state of the environment. Furthermore, both output a Q-value for each action in the action space. Each hidden layer facilitates 32 neurons whereby the activation function varies depending on the parameter settings.

For all models, training takes place not only at the end, but also during an episode. During training, the mean squared error is used as loss function. This loss function was mentioned in the explanation of DQNs in Chapter 5. Moreover, the target Q-network is overwritten by the current Q-Network, after a defined number of episodes.

The discount γ, which differentiates according to when a reward is returned, amounts 0.99 for all experiments. However, in the following not only a weighting based on the discount γ takes place, but the duration of an action is taken into account. Usually, the Q-value is calculated as introduced in Chapter 5 via $Q(S_t, A_t) = R_{t+1} + \gamma \cdot \underset{A_{t+1}}{max} Q(S_{t+1}, A_{t+1})$. However, the time can be considered by computing the Q-value as $Q(S_t, A_t) = R_{t+1} + \gamma^{duration} \cdot \underset{A_{t+1}}{max} Q(S_{t+1}, A_{t+1})$. This enables a more time-oriented view.

Furthermore, a replay memory has to contain at least 10,000 experiences before training can start. In addition to that, an ε-decay is set which defines the ratio between random and greedy behaviour according to the $\varepsilon-$greedy approach. Thus, training starts with a preference for random decisions. However, as soon as the replay memory is filled completely for the first time, the ε-decay comes into play. An example how the relation between random and greedy decisions change over iterations is presented in Figure 9.2. At the beginning, the agent makes decisions randomly. However, due to the ε-decay the relation reverses to the advantage of the trained behaviour after a few iterations. That means, as training progresses, more and more decisions are made greedily according to the Q-value. A greedy chosen action corresponds to the selection of the greatest Q-value for the current state S_t. Figure 9.2 assumes that the replay memory is filled in iteration 0. However, the illustration would start with a constant line of zero when the replay memory must be filled during the first episodes.

All episodes simulate the production process in a factory for 20 minutes. Restricting the simulation time corresponds to what is often used in practice in continuous tasks for the while condition (line 7 in Algorithm 4) in order to terminate an episode.

Since many further aspects such as the degree of ε-decay vary among implementations, the details will be presented separately.

Application Model The trained model to be used for gantry robot scheduling results from the trained model of the Q-network and is called the application model. However, this application model does not necessarily correspond to the latest Q-network. Instead, only at the same time the target model is updated, it is checked whether an update of the application model should also be performed. This review happens based on the mean throughput of the runs since the last review compared with the mean throughput of the runs based on which the application model was updated the last time. Then, if the review indicates a higher value for the mean throughputs of the last runs since the last review, the application model is updated to the latest model of the Q-network. Thus, an improvement of the application model is ensured.

The results presented in the following experiments that relate to the application of a trained model originate from such an application model, unless noted otherwise.

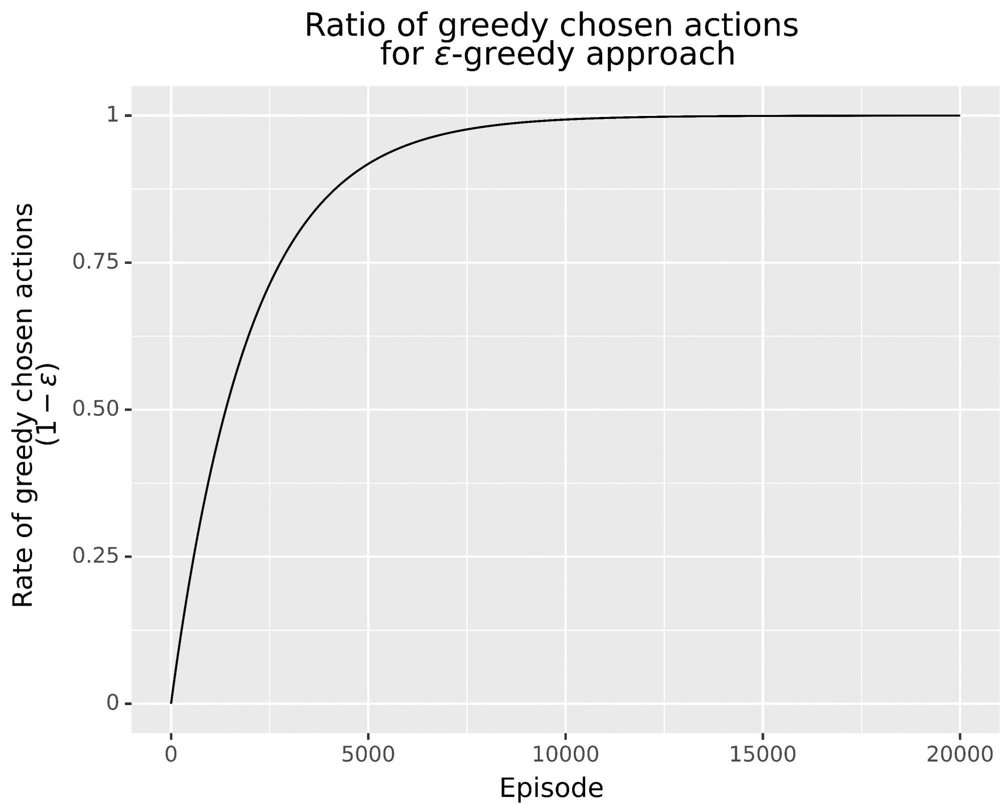


Figure 9.2: Ratio of greedy chosen actions with an ε-decay of 0.9995 (ε-greedy approach)

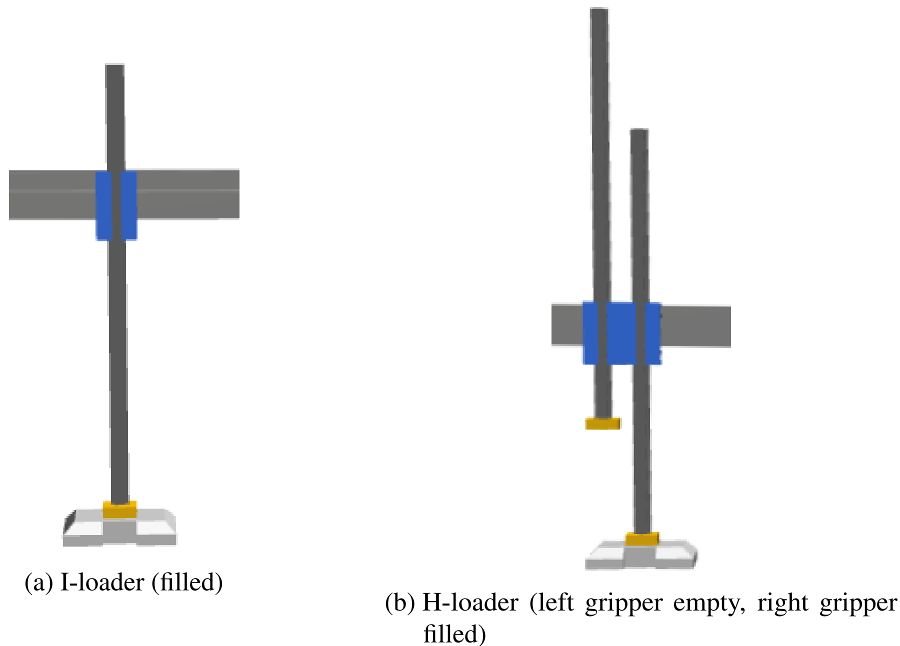
9.1.2 Production Environment

The gantry robot scheduling task of this experimental production system corresponds to a continuous task. Therefore, the modified concept of reward machines can be tested for this task.

The setup of the production environment in the following experiments is similar to the running example. The production environment consists of an input conveyor, two machines and an output conveyor, similar to Figure 1.1. In addition, this production environment does not use buffers where items could wait to be moved without blocking the current machine. Therefore, any machine waiting for the gantry robot to pick up a processed part or to deliver an unprocessed part is blocked. As a result, the gantry robot is the limiting factor in the given production environment.

In addition to the running example, the gantry robot of this production environment is also characterised by the fact that it can transport two different items at the same time. This is possible because the robot has an H-loader which means there are two independent gripper. Figure 9.3 demonstrates the differences between the I-loader as used in the

running example and the H-loader that is employed in this production environment.



(a) I-loader (filled)

(b) H-loader (left gripper empty, right gripper filled)

Figure 9.3: Comparison of I-loader and H-loader

Furthermore, in contrast to the running example, this environment contains a few additional stochastic aspects. On the one hand, the availability of each machine is stochastic, since machine failure is part of this production environment. On the other hand, the arrival of elements at the input conveyor is also not deterministic.

In contrast, the time taken to travel from one place to another is deterministic. The duration of different actions has a huge impact on the optimal throughput of the production system. These fixed but different times are summarised in Table 9.1. All rides are symmetric which means that ride time between station A and station B equals ride time between station B and station A. Furthermore, since all four parts of this production environment are arranged in a row, ride times passing the same parts add up to the same value. For example, a ride from the input conveyor to the output conveyor (6.25 sec) takes as long as from the input conveyor to station A (1.95 sec), from station A to station B (2.775 sec) and from station B to the output conveyor (1.525 sec). Besides, the time required for loading measures 5.8 seconds for gripper 1 and 6.5 seconds for gripper 2 at each position. The unloading process requires 4.8 seconds for gripper 1 and 5 seconds for gripper 2. The difference between the two grippers results from the need to readjust the position of the gantry when gripper 2 is required.

The reinforcement learning agent to be trained aims to maximise the throughput per hour by finding an optimal policy for scheduling of the gantry robot. The four different

Start	Destination	Ride time (in seconds)
input conveyor	station A	1.95
input conveyor	station B	4.725
input conveyor	output conveyor	6.25
station A	station B	2.775
station A	output conveyor	4.3
station B	output conveyor	1.525

Table 9.1: Ride time between different positions in seconds.

elements (S,A,R,p) of an MDP instance of this production environment are stated below.

9.1.2.1 State Space

The state of the gantry robot is factored in the current environmental state S. Hereby, the current position of the loader, the information whether the loader is currently carrying an element as well as the status of the machines are taken into account. Since the loader incorporates two different gripper, the information on the loading status has to be provided for each gripper. The loading status indicates information about the processing status of the transported item. Furthermore, the state of each station is used to represent the current state of the production environment. This includes information whether the station is busy or finished. Moreover, the state space contains whether a machine has a malfunction. In total, the status of a station can take four different values: empty, in progress, processing completed and failed. Besides, the availability of an element at the input conveyor is also part of the environment.

The initial state S_0 of the environment describes the state of the production system at the beginning of each episode. The loader always starts at the input conveyor without transporting an element. Futhermore, in the initial state, there are no elements in the entire system - neither at the input conveyor nor at a station. Table 9.2 summarises relevant information a state in the state space contains and defines it exemplarily for the initial state.

Location	Element at input conveyor	Status of gripper 1	Status of gripper 2	Status of station A	Status of station B
input conveyor	false	empty	empty	empty	empty

Table 9.2: Overview of the information considered in a state, using the example of the initial state

Due to the continuous characteristic of the underlying task, the production system runs forever and therefore, no terminal state is defined.

9.1.2.2 Action Space

The action space A is defined over nine distinct actions. On the one hand, a motion can be performed. This means that the agent can move to the input conveyor, station A, station B or the output conveyor. This clearly defined space of moving actions prevents the gantry robot from moving to any position in between. Furthermore, the agent can facilitate a loading or unloading action. However, this action has to be specified for each gripper. Since the loader contains two gripper, a loading and an unloading action have to be defined twice - one for each gripper. Additionally, the agent can wait for a trigger from the simulator. Such a trigger is sent when the system state changes, e.g. because a new element has arrived at the input conveyor and the queue is therefore no longer empty or because a machine finished processing a part. The agent does not do anything as long as it is waiting for a change caused by the environment. However, this waiting action has to receive a trigger within 45 seconds, since most changes made by the system itself, such as the arrival of an element or processing of an element, should take less than 45 seconds.

The state of the environment determines whether an action can be performed or results in an error. For example, a loading action can only be executed if the loader is empty and there is an element at its current position that can be picked-up. Although a waiting action can be chosen at every environment state, sometimes there is nothing to wait for. For example, this is the case if at least one element could be picked-up at the input conveyor or any machine.

9.1.2.3 Rewards

The combination of environment state and action determine how to reward a behaviour. Positive rewards are returned when an unprocessed element is successfully delivered to a machine in order to be further processed. Moreover, carrying a processed element to the output conveyor and thus hand-in a finished object, obtains positive reward as well. Conditions that cause these positive rewards are summarised in Table 9.3. Any other behaviour that is either incorrect or contributes to completing an element is neither rewarded nor punished, instead they trigger a reward of zero. Actions that cannot be executed successfully at the current environment state are considered as such incorrect behaviour.

Condition	Reward
load unprocessed element from an input conveyor	1
unload unprocessed element at station A or station B	2
load processed element from station A or station B	1
unload processed element at an output conveyor	5

Table 9.3: Conditions or subtasks that need to be fulfilled to gain positive rewards

9.1.2.4 State-Transition Probabilities

The state-transitions are probabilistic due to stochasticity of arrival time at the input conveyor and machine failures. However, most aspects of the state are deterministic. For example, performing the action *move to station A* leads to a deterministic change in the loader's position. But, during this ride, an element could arrive at the input conveyor. Thus, the same state could transition to different subsequent states, one in which an element has arrived and one in which it has not.

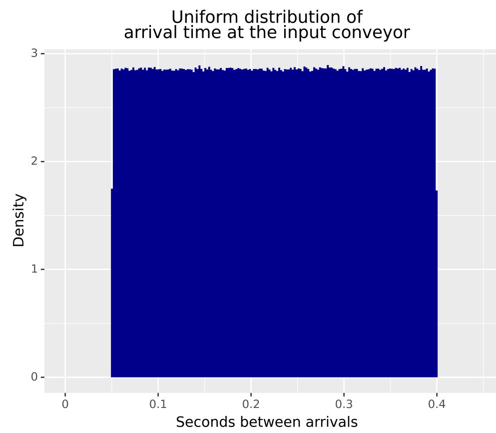
Further stochastic transitions occur due to machine failures. Such a machine failure leads to two different possible transitions in the environment as well. On the one hand, if a machine is down at the moment, the repair of the machine could end while any action is executed. On the other hand, if a machine is working, any action can entail a machine failure.

Arrival of elements The elements arrive at the input conveyor with equal probability at an interval of five to 40 seconds after the previous element. This distribution is visualised in Figure 9.4a.

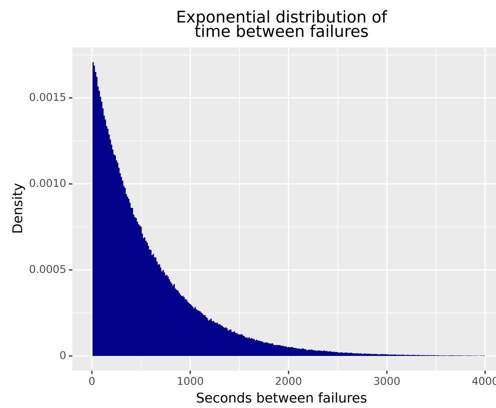
Machine Failure A machine failure can only occur when the machine has been occupied by an element. Furthermore, such a malfunction prevents the loader from loading or unloading elements at that machine. Besides, elements that have just been processed on the machine are considered to disappear when a broken machine is repaired. Therefore, the machine is empty as soon as it is available again. In the given production environment, both machines fail independently of each other, i.e. theoretically a fault can also occur on both machines simultaneously.

The mean time between failures (*mtbf*) amounts to 570 seconds. This time follows an exponential distribution with λ of $\frac{1}{mtbf}$ which yields a density distribution as illustrated in Figure 9.4b. Hereby, the time is calculated based on time a machine was working. For example, if a random number of 500 seconds is drawn from the described exponential distribution, it means that a machine failure will occur after 500 seconds of operation at this machine. Moreover, the repair of a machine failure takes on average 30 seconds (mean time to repair, *mttr*). This parameter follows an Erlang distribution with β of $\frac{mttr}{2}$ and m of two. This distribution is visualised in Figure 9.4c. These two parameters *mtbf* and *mttr* result in a machine availability of 95 %. Since the machines are independent and identically distributed (iid), the probability of both machines failing at the same time amounts to 0.25 %.

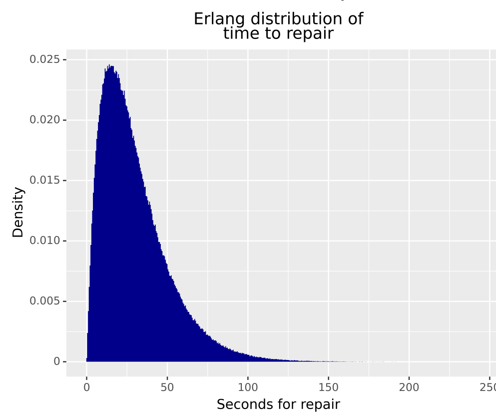
Theoretical details of the exponential distribution and Erlang distribution are given in the Appendix 11.



(a) Uniform distribution of arrival of elements at the intervall [5, 40) seconds



(b) Exponential distribution of time between failures with parameter $\lambda = \frac{1}{mtbf}$ (mtbf = 570)



(c) Erlang distribution of time to repair with parameters $\beta = \frac{mttr}{2}$ and $m = 2$ (mttr = 30)

Figure 9.4: Distributions of incoming elements and machine failures (histograms)

9.2 Reward Machine

In order to be able to evaluate the proposed adaptions of the reward machine concept and algorithm for continuous reinforcement learning problems, a reward machine is designed and trained for the given production environment. First, the design of the reward machine is presented. Then, the capabilities of the proposed modifications are examined in more detail by training an agent with optimised settings regarding the use of reward machines. Additionally, these training settings are performed with the original CRM algorithm, as described in Chapter 6, to demonstrate the shortcomings of this algorithm when applied to continuous problems.

The reward machine of the given production environment is defined over the status of the loader. The status indicates whether the gantry robot is filled and if so, with what type of element. However, this reward machine refines the states by including information about the status of each individual gripper. Therefore, it differentiates which of the two grippers holds which item. This reward machine distinguishes whether, for example, gripper 1 is empty and gripper 2 is filled with an unprocessed item or vice versa. An illustration of this second reward machine can be seen in Figure 9.5. However, due to its complexity, this illustration is split into two pieces with more details, in the Figures 9.6 and 9.7.

In the following, the different elements by which a reward machine is defined are specified for the reward machine of the given production environment.

9.2.1 States

The reward machine consists of ten different states. These states arise from the loader status. The reward machine state is *G1 & G2: empty* if both gripper are not filled with any element. Further, the states distinguish between filled with an unprocessed element and a processed element. The difference exists in the consequence of the subsequent actions. The reward machine state *G1 & G2: filled (unprocessed)* indicates that the agent has to deliver this piece to a machine, while a reward machine state of *G1 & G2: filled (processed)* makes an unloading at the output conveyor necessary. However, states can also represent a combination of different status, like *G1: empty & G2: filled (processed)*, which stands for a loader with the first gripper empty and the second one carrying a processed element. Hereby, which gripper satisfies which condition does matter. The set

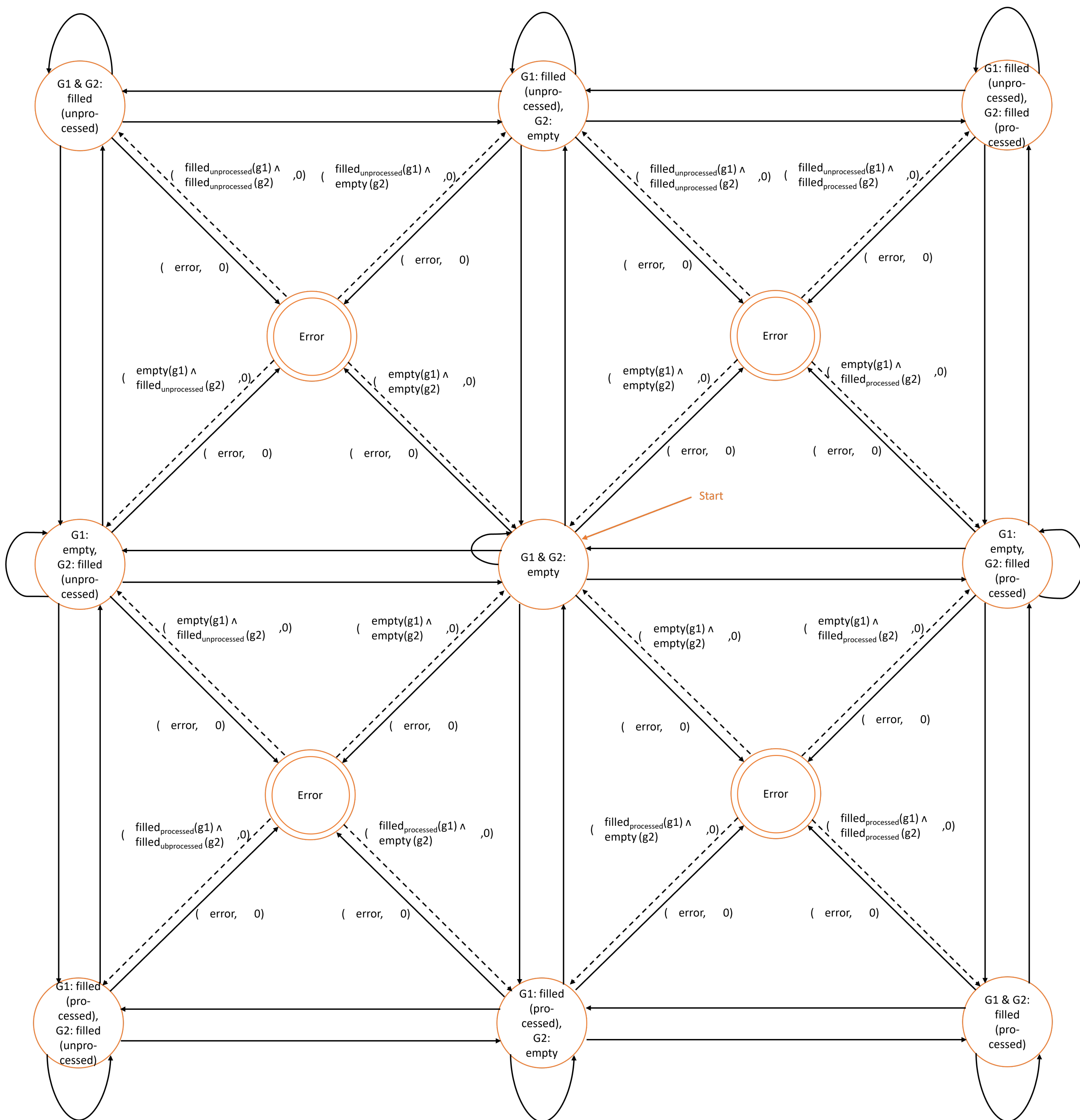


Figure 9.5: Reward machine with loader status for each individual gripper, G1 indicates gripper 1 and G2 specifies gripper 2

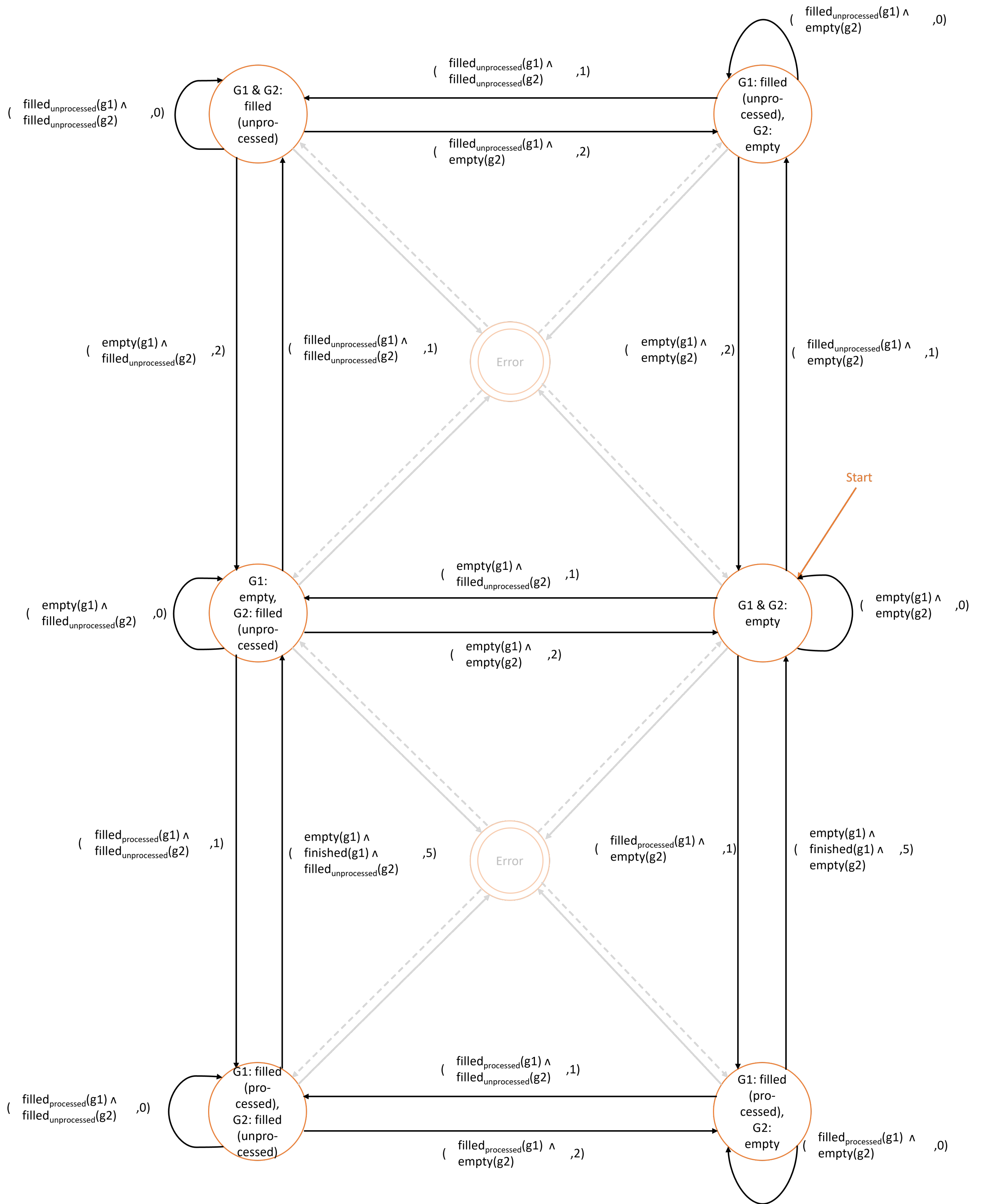


Figure 9.6: Reward machine with loader status for each individual gripper, left part of Figure 9.5

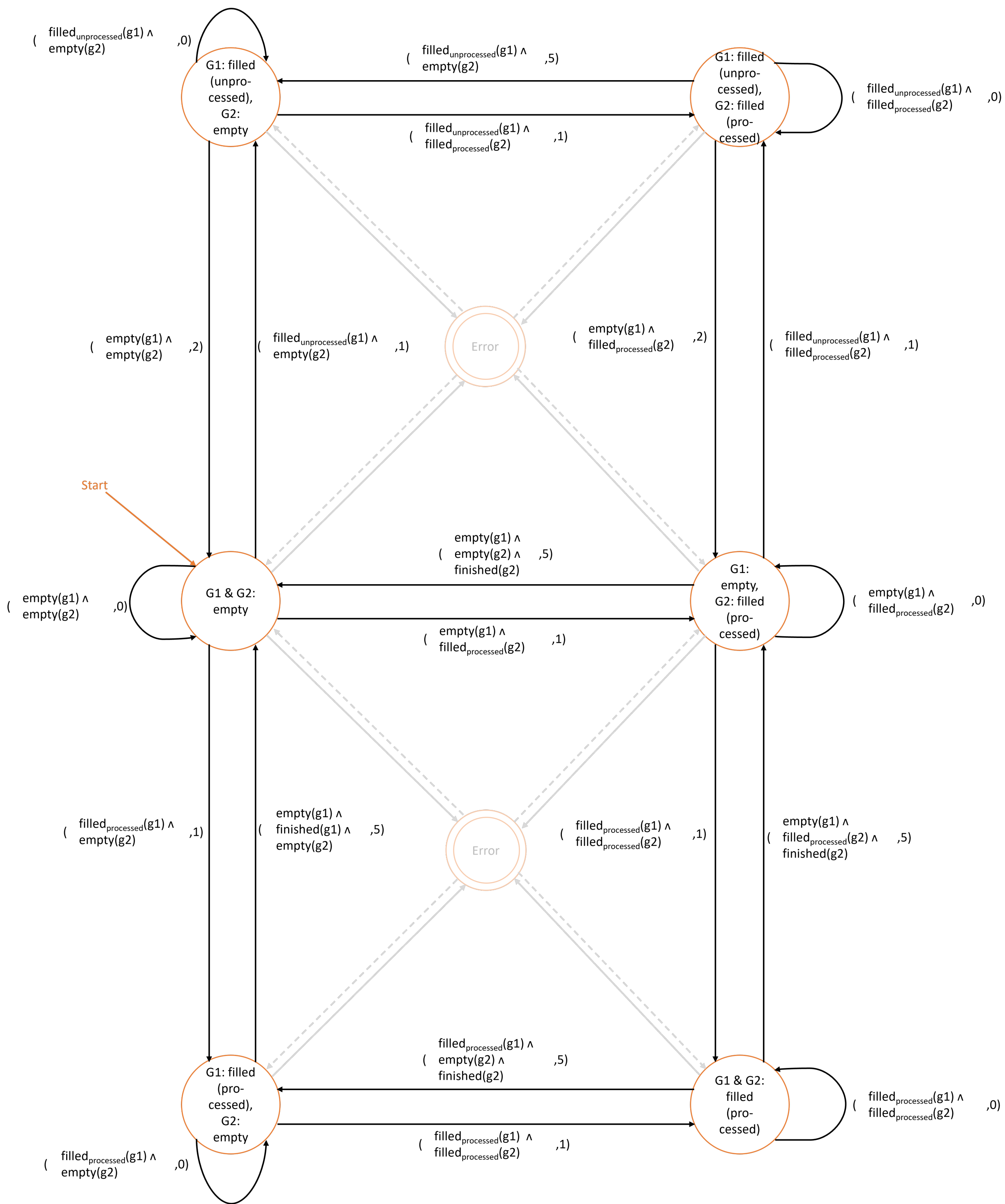


Figure 9.7: Reward machine with loader status for each individual gripper, right part of Figure 9.5

of all states is summarised in (9.2.1).

$$\begin{aligned} U = \{ & G1\ \&\ G2: \ empty \\ & G1: \ empty\ \&\ G2: \ filled\ (unprocessed), \\ & G1: \ empty\ \&\ G2: \ filled\ (processed), \\ & G1: \ filled\ (unprocessed)\ \&\ G2: \ empty, \\ & G1\ \&\ G2: \ filled\ (unprocessed), \\ & G1: \ filled\ (unprocessed)\ \&\ G2: \ filled\ (processed), \\ & G1: \ filled\ (processed)\ \&\ G2: \ empty, \\ & G1: \ filled\ (processed)\ \&\ G2: \ filled\ (unprocessed), \\ & G1\ \&\ G2: \ filled\ (processed), \\ & Error\} \end{aligned} \tag{9.2.1}$$

The initial state u_0 of this reward machine marks the state *G1& G2: empty*, since this is the only possible state at the start of the production system.

9.2.2 Acceptance Criteria

In terms of automata theory, an input sequence is accepted if the terminal state *Error* is reached. Actions that cause an error in the production environment lead to this terminal state. Such an action could be an unload even though the loader is not yet filled with an element, an unload in the wrong place, or an impossible load. Furthermore, a terminal state is reached when the agent waits even though there are elements that need to be carried. Pointless actions such as moving to a position where the loader is already located will also cause a stop. It follows the set F of terminating states:

$$F = \{Error\} \tag{9.2.2}$$

Furthermore, an input sequence is accepted if states that occur infinitely often during a run are an element of $\mathcal{F}$. The character of the given task requires all non-terminating states to occur infinitely often, since both the processing of unprocessed elements as well as the delivery to the output conveyor of processed items need to take place. Therefore, a set of all non-terminating states has to be an element of $\mathcal{F}$. It follows the set $\mathcal{F}$ for the

acceptance criteria:

$$
\begin{aligned}
\mathcal{F} = \{\{ & G1 \,\&\, G2: \ empty, \\
& G1: \ empty \,\&\, G2: \ filled \ (unprocessed), \\
& G1: \ empty \,\&\, G2: \ filled \ (processed), \\
& G1: \ filled \ (unprocessed) \,\&\, G2: \ empty, \\
& G1 \,\&\, G2: \ filled \ (unprocessed), \\
& G1: \ filled \ (unprocessed) \,\&\, G2: \ filled \ (processed), \\
& G1: \ filled \ (processed) \,\&\, G2: \ empty, \\
& G1: \ filled \ (processed) \,\&\, G2: \ filled \ (unprocessed), \\
& G1 \,\&\, G2: \ filled \ (processed)\}\}
\end{aligned}
\tag{9.2.3}
$$

If the agent learns a policy where the gantry robot moves only from one location to another without performing a loading or unloading action, on the one hand, the terminal state *Error* would not be reached. However, it would also not satisfy the acceptance condition of $\mathcal{F}$, since the status of the loader would not change and thus only one state of the reward machine would be visited infinitely often, which is no element in $\mathcal{F}$. It can be deduced that such a run would not solve the continuous task.

9.2.3 Input Alphabet

The definition of a reward machine requires an alphabet 2^P containing logical formulas that are formulated over a set of propositional symbols P. Due to the consideration of the status of each individual gripper, the set of propositional symbols P needs to take into account this aspect. The following propositional symbols can be derived from the provided production environment:

empty(g1) This propositional symbol describes that gripper 1 is empty. It is negated when a loading action is performed so that this gripper is filled.

empty(g2) Similar to the propositional symbol $empty(g1)$, this high-level event denotes an empty gripper 2.

$filled_{unprocessed}(g1)$ A high-level event where an unprocessed element is picked up from the input conveyor by gripper 1 is denoted by this propositional symbol. This symbol becomes false when gripper 1 is no longer carrying such an item.

$filled_{unprocessed}(g2)$ Similar to $filled_{unprocessed}(g1)$, this propositional symbol describes that gripper 2 is filled with an unprocessed item.

$filled_{processed}(g1)$ This propositional symbol indicates that gripper 1 is carrying a processed element. It is negated as soon as the gripper is not filled with an processed item anymore.

$filled_{processed}(g2)$ Following the example of $filled_{processed}(g1)$, this propositional symbol denotes that gripper 2 is filled with a processed element.

$finished(g1)$ When a processed element is successfully delivered to the output conveyor by gripper 1, this propositional symbol becomes true. This high-level event can only occur together with the propositional symbol $empty(g1)$.

$finished(g2)$ Similar to $finished(g1)$, this symbol marks a successful delivery of a processed item to the output conveyor by gripper 2. It can also occur only in combination with $empty(g2)$.

error This propositional symbol becomes true when an action is executed that transitions to the terminal state *Error*. It could occur for an incorrect loading action, e.g. if the loader attempts to pick up an element that is not present. Furthermore, as a complement to an incorrect loading action, this propositional symbol also denotes an impossible unload execution. This can be the case when an unload is performed on the input conveyor or on an occupied station. Furthermore, this propositional symbol becomes true when an unprocessed item is delivered to the output conveyor or when a processed element is discharged at a station. In addition to these incorrectly executed actions, some actions do not lead to harm but are just useless. On the one hand, an action that requires to drive to the current location, is considered as useless since it has no impact on the state of the environment. On the other hand, the waiting action is only helpful in a few settings. For example, at the beginning of the production run, the gantry robot has to wait until the first item arrives at the input conveyor and can be picked up. However, in most of the cases, the production system itself is waiting for an action of the gantry robot which is why waiting would slow down the process. All these events lead to a terminal state of the reward machine.

The alphabet 2^P can be formulated as a logical conjunction of these different propositional symbols. For example, loading an unprocessed item by gripper 1, while gripper 2 is empty, satisfies the logical formula $filled_{unprocessed}(g1) \wedge empty(g2)$ [1].

This logical formula triggers a transition from *G1 & G2: empty* to *G1: filled (unprocessed) & G2: empty*.

The propositional symbol *error* is the only one that occurs without any other symbols, as their status does not make a difference.

[1] extended: $\neg empty(g1) \wedge filled_{unprocessed}(g1) \wedge \neg filled_{processed}(g1) \wedge \neg finished(g1) \wedge empty(g2) \wedge \neg filled_{unprocessed}(g2) \wedge \neg filled_{processed}(g2) \wedge \neg finished(g2) \wedge \neg error$

9.2.4 State-Transition Function

As Figure 9.5 illustrates, this reward machine incorporates 65 different transitions. At each state of the reward machine a transition leads to a terminal state. Further, edges exist only between the reward machine states that are reachable from each other.

As long as the loader does not carry an element, the reward machine remains in the state *G1 & G2: empty*. However, when an element is picked up, a transition is triggered. This transition depends on the status of the element and which gripper performed the loading action. If the element is a processed item, then the logical formula $empty(g1) \wedge filled_{processed}(g2)$ or $filled_{processed}(g1) \wedge empty(g2)$ is valid for gripper 1 or gripper 2, respectively, which performs the loading action. This leads to a transition from *G1 & G2: empty* to state *G1: empty & G2: filled (processed)* if gripper 2 is the executing loader. This can be seen in Figure 9.7. As long as the gripper carries such a processed element, this logical formula is true and thus the reward machine remains at this state. Furthermore, if the loader would load any further element by the empty gripper 1, the state of the reward machine would change, either to state *G1 & G2: filled (processed)* or to state *G1: filled (unprocessed) & G2: filled (processed)*, since the propositional symbol $empty(g1)$ becomes false and instead $filled_{unprocessed}(g1)$ and $filled_{processed}(g1)$ become true, respectively. As a consequence, the transition to the respective state is enabled. These two possible transitions can be found in Figure 9.7 as well.

However, for example, the states *G1 & G2: filled (unprocessed)*, *G1: filled (unprocessed) & G2: filled (processed)*, *G1 & G2: filled (processed)* and *G1: filled (processed) & G2: filled (unprocessed)* cannot be reached from the state *G1 & G2: empty*. This results from the fact that these states require both grippers to be filled. However, a loading action can be only performed by one gripper at a time which would first necessitate a transition from the state *G1 & G2: empty* to a reachable state.

Furthermore, transitions resulting from the proposed terminal-state-transition function δ_f, that is included in δ, come from the terminal state *Error*. Depending on the value of the propositional symbols influenced by the last state-action-state transition of the environment leading to a terminal state in the reward machine, the next non-terminating state of the reward machine is determined.

In Table 11.2 in the Appendix, all elements of the state-transition function are presented.

9.2.5 State-Reward Function

Every transition between the non-terminal states of the reward machine gains a positive reward of one, two or five. In contrast, self-loops of a state and transitions to terminal states are rewarded with zero. The detailed rewards resulting from $\delta_r : U \times 2^P \rightarrow \mathbb{R}$ are outlined in Table 11.2 in the Appendix. As explained in Chapter 8, transitions triggered by the terminal-state-transition function δ_f hold a reward of zero.

9.3 Application of Learning Approaches

In the following, an agent scheduling a gantry robot is trained by means of the adjusted reward machine approach proposed in Chapter 8. For later comparison, agents are trained by further learning approaches. On the one hand, the original CRM algorithm is applied to the given production environment. On the other hand, a model is trained by a basic learning approach that does not make use of reward machines. These last two approaches are trained with both a parameter set that is adjusted to the respective approach and the training parameters that lead to an optimal model for the adjusted reward machine approach. The last model is relevant to create a solid basis for a later comparison.

9.3.1 Adjusted Reward Machine Approach

A gantry robot is trained by means of the presented reward machine and the CRM approach with adjustments proposed in Chapter 8. Only actions that mark an incorrect behaviour are considered as terminating and as a consequence learned to be disadvantageous. In addition, the agent is trained with an ε-decay of 0.998. The development of the ratio of greedy action choices can be seen in Figure 9.8.

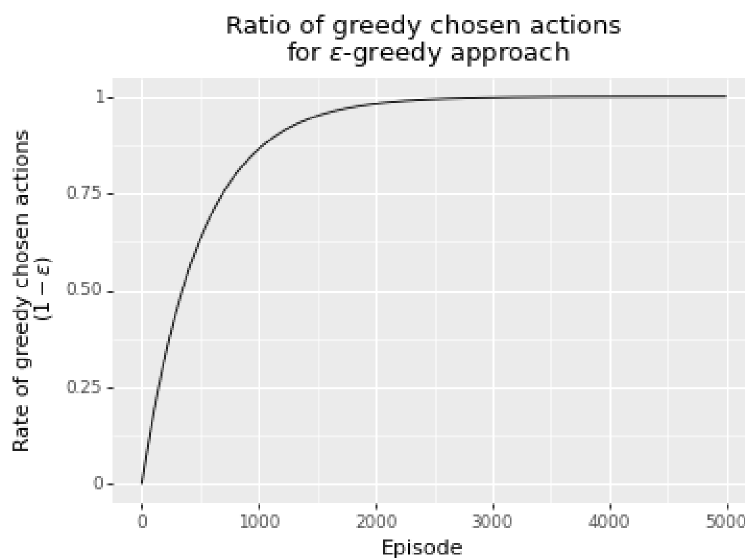


Figure 9.8: Development of the ratio of random/greedy chosen actions according to the ε-greedy approach

The replay memory has to be initially filled with 450,000 experiences before the ε-value is allowed to shrink. However, training starts already after 10,000 agent-environment interactions have been executed. The training takes place on average after execution of two actions out of three (train probability of 0.66). Moreover, a batch of size 288 is drawn

randomly from the replay memory. All relevant training parameters are summarised in Table 9.4.

Parameter	Value
replay memory size	450,000
min. number of elements in replay memory for training	10,000
min. number of elements before ε decreases	450,000
batch size	288
ε-decay	0.998
time discount	true
update of target network	every 7 episodes
activation function of hidden layer	rectifier linear unit

Table 9.4: Training parameters for the adjusted reward machine approach

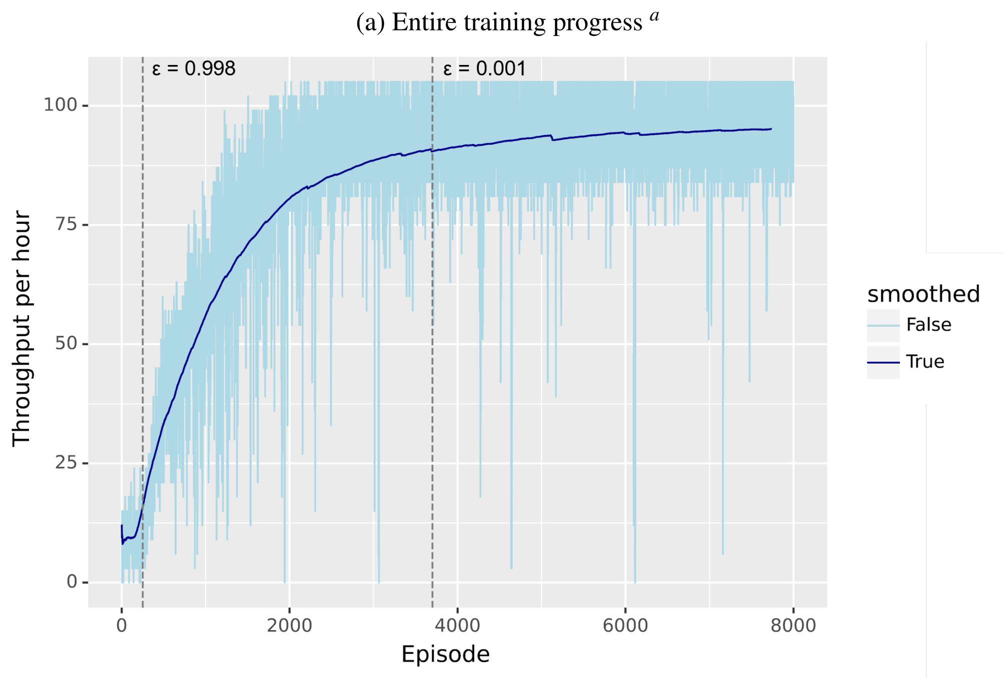
Besides, for training of the gantry robot a neural network of four hidden layers with each facilitating 32 neurons is used. Hereby, three of these hidden layers apply a rectified linear activation function[2] which belongs to the group of non-linear functions.

Observations The model was trained for more than 8,700 iterations. After six episodes, the model starts training the Q-network, as the minimum required number of experiences in the replay memory is then reached. Figure 9.9 illustrates how the amount of throughput items changes over training. While Figure 9.9a gives an impression of the entire training progress, Figure 9.9b zoomes into the first 350 episodes of the training. Around 250 episodes are necessary until the replay memory is full for the first time. Up to this point, all decisions were made randomly. But as soon as the replay memory is filled for the first time, the percentage of random decisions decreases continuously. In Figure 9.9b, this point is marked by a dashed line. All episodes left of this dashed line hold an ε-value of 1.0, while all episodes on the right side are trained with an ε-value less than 1.0.

In Figure 9.9b, we see that it still takes about 50 episodes (or reaching an ε of approximately 0.9) before the previously learned behaviour becomes noticeable in the throughput. During the first 250 episodes, the throughput is approximately normally distributed with a mean of 9.3 and a standard deviation of 4.5. However, for the second 250 episodes ranging from 250 to 500, a mean of 23.3 and a standard deviation of 11 can be observed. One can see, that after the periode of constant throughput (up to epsiode 250) resulting from completely random actions (ε of 1.0), the number of pieces passed increased and the training progress started to slow down after 2,500 episodes with an average throughput of about 80. Up to this point, an additional processed piece was delivered to the output

[2] rectifier linear unit (relu): $f(x) = max(0,x)$

(a) Entire training progress [a]



[a]smoothed as explained in the Appendix in section Smoothing

(b) Focus on first 350 episodes, with end of random decisions at episode 250

Figure 9.9: Development of throughput per hour during training

conveyor approximately every 30 episodes. Furthermore, Figure 9.9a visualises that the throughput per hour changes only slightly for more than 6,500 training episodes.

Beyond that, a relevant point is reaching the minimum required ε-value of 0.001. This is achieved after 3,700 episodes. From then on, most iterations contain only one randomly chosen action. All other actions follow the policy learned so far. The low percentage of random actions is reflected in the unsmoothed illustration of the throughput, in Figure 9.9a. One can see that after 2,200 training episodes, the throughput achieved ranges between 80 and 108 elements. Further, Figure 9.8 indicates that then the ε-value amounts to less than 0.01.

Moreover, some states, especially machine failures, can only be observed as training progresses. The model with reward machines experiences the first machine failure after almost 500 episodes and thus an ε-value of 0.61. Hence, the agent is still able to explore how to deal with such machine failures.

The initial filled replay memory contains already more than 450 different states S_t. Since each experience consists of the environmental states S_t and S_{t+1}, the states of the reward machine U_t and U_{t+1}, the executed action A_t, the received reward R_{t+1} and a terminal flag, the 450 different states S_t occur in several shapes. Due to the CRM approach it is guaranteed that each state of the reward machine occurs equally likely. Furthermore, since the first filling of the replay memory happens only randomly, all actions can be observed to the same extent.

The initial state of the environment with both grippers empty represents around 0.25 % of the states in the replay memory. The most frequent one is a state where the gantry is located at the output conveyor while an element lays at the input conveyor and is ready to be loaded. Due to the concept of the CRM algorithm, this state can be observed with all nine different non-terminal states of the reward machine. In Table 9.5, an overview of the five most visited states is given. The statuses of the two grippers are not specified, since all environmental states always occur with all non-terminating states of the reward machine. Therefore, two different frequency values are provided. On the one hand, the *frequency of each state* indicates how often a specific state with a certain reward machine state occurs. On the other hand, the *frequency of all reward machine states in the replay memory* specifies how often the given state of the environment appears with all possible states of the reward machine in the replay memory.

Considering the first state in Table 9.5, the replay memory contains experiences of this state combined with all possible forms of the non-terminating reward machine states in equal quantity. Due to the CRM algorithm, each of these different occurrences is observed always together and as a consequence equally often. Therefore, the second value (43,623) corresponds to nine times the first value (4,847). Thus, Table 9.5 actually represents the 45 most frequent states in the replay memory. It is interesting to see that the most visited state occurs in almost equal numbers for all possible locations. Moreover, states that hold a processed element that is ready to be loaded rank on place five to ten of the most frequent states. Beyond that, states that do not hold an element at the input conveyor occur all in all less than 13,000 times which is less than 3 %. Thus, in more than 97 % of

Location	Element at input conveyor	Status of station A	Status of station B	Frequency of each state	Frequency for all reward machine states
output conveyor	true	empty	empty	4,847 (1.08 %)	43,623 (9.69 %)
input conveyor	true	empty	empty	4,767 (1.06 %)	42,903 (9.53 %)
station A	true	empty	empty	4,518 (1.00 %)	40,662 (9.04 %)
station B	true	empty	empty	4,488 (1.00 %)	40,392 (8.98 %)
output conveyor	true	empty	processing completed	2,176 (0.48 %)	19,584 (4.35 %)

Table 9.5: Overview of the five most frequent states of the environment in the first full replay memory (trained by the adjusted reward machine approach)

the experiences, an element at the input conveyor is ready to be picked up by a gripper.

At the end of the training, the replay memory consists of different environmental states than after its initial formation. The replay memory contains 864 different states, which is almost twice as many as at the beginning. Table 9.6 shows the five most frequently visited states of the environment at the end of the training. While the first four states could be only distinguished by the location of the gantry at the beginning of the training, a greater variance in the status of the machines can be seen at the end of the training. This highlights the difference between a randomly and a structurally filled replay memory. While the initial replay memory contained the four most frequently visited states about equally often and the fifth state already only half as often, the most recent replay memory indicates a greater difference in the frequency of all five most frequently occuring states. Moreover, states without an element at the input conveyor occur almost 23,000 times and thus considerably more often than in the randomly filled replay memory at the beginning. Beyond that, at the end of the training the replay memory includes the initial state of the environment around 430 times which corresponds to less than 0.1 %.

Location	Element at input conveyor	Status of station A	Status of station B	Frequency of each state	Frequency for all reward machine states
output conveyor	true	in progress	empty	4,414 (0.98 %)	39,726 (8.83 %)
station B	true	in progress	empty	3,202 (0.71 %)	28,818 (6.4 %)
input conveyor	true	processing completed	empty	3,048 (0.68 %)	27,432 (6.1 %)
station A	true	processing completed	empty	2,258 (0.50 %)	20,322 (4.52 %)
station B	true	in progress	processing completed	2,136 (0.47 %)	19,224 (4.27 %)

Table 9.6: Overview of the five most frequent states of the environment in the replay memory at the end of the training (trained by the adjusted reward machine approach)

To assess how the Q-values change over training and how incorrect actions are learned to avoid, we also look at the evolution of the Q-values for certain states. To do this, we examine Figure 9.10, 9.11 and 9.12 which map the predicted Q-values of a particular state of the environment over time for each possible action. A greedy decision always chooses the action with the greatest Q-value for the given state. All three figures show a rather unsteady curve which is due to the fact that only the Q-values of every 25th episode are taken into account.

In Figure 9.10, we consider the initial state of the environment, as we investigated that this state occurs very often in the initial replay memory. Initially, it is learned which actions were not allowed to be performed. Furthermore, at the same time, the Q-value of location-related actions such as moving to a station or to the output conveyor rises. A few batches later, the Q-value of the waiting action, which is the optimal action choice in the given state, increases very quickly and later outperforms other actions. This results from the fact that a waiting action holds on until the system has triggered a change, while moving to other locations waits only passively. The gantry can move between locations until the required change happens. Both actions lead to the same result, however, since the time until a change occurs is random, the waiting action can watch for a change more precisely.

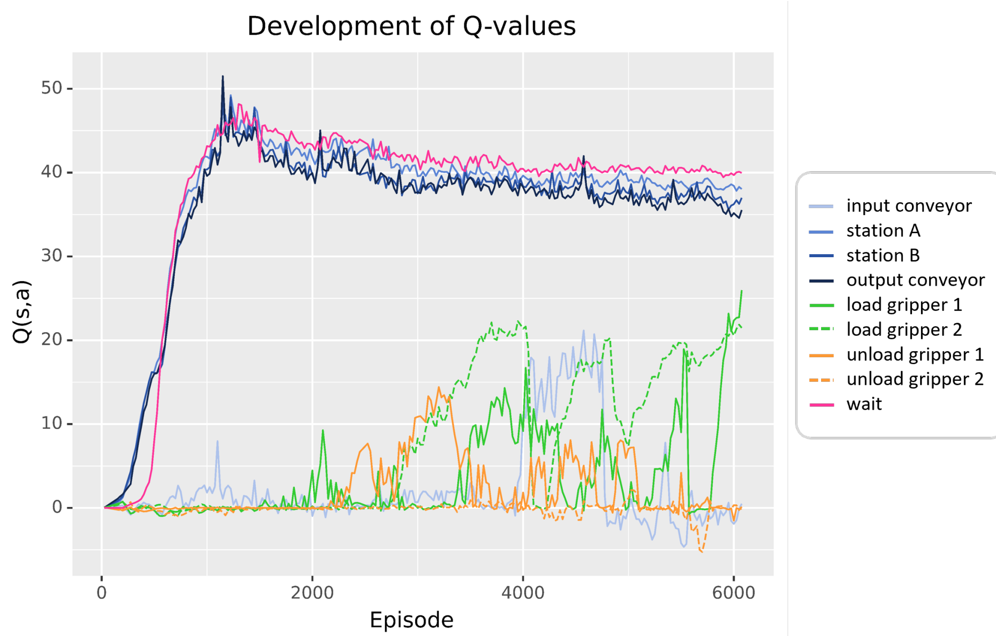


Figure 9.10: Development of the Q-values for the initial state and all actions

Additionally, some states are more difficult to learn than others. For example, loading an item from the input conveyor is not always the best way. Especially if the loader is already filled with an unprocessed item and both machines are busy, loading a second unprocessed item from the input conveyor would lead to an error in the near future. This results from the fact that a full loader has no possibility to load processed elements from the stations, which would be necessary to unload an element from any gripper. The agent has learned to avoid collecting further unprocessed items in such a state, even though the consequences of such a loading action only occur a few steps in the future. Figure 9.11 illustrates the learning curve of this state. At the beginning, the agent chose the loading action due to a highest Q-value. However, a few iterations later, the Q-value of this action decreased. This results from the fact that an incorrect action, like unloading an unprocessed item at the output conveyor or loading even though both grippers are already filled, has to follow if the loading is performed. Further, the waiting action increased and location-based actions were identified as beneficial. In detail, at the end of training, driving to station A is considered particularly worthwhile.

Furthermore, it is also difficult for an agent to learn how to behave in situations where a machine fails and thus cannot be controlled by the robot. An agent without advanced training would drive to different locations and bide his time until the machine is available again. In this case, no more items would be loaded or unloaded, not even at the machine that can be controlled. However, after training, the agent is able to choose actions that

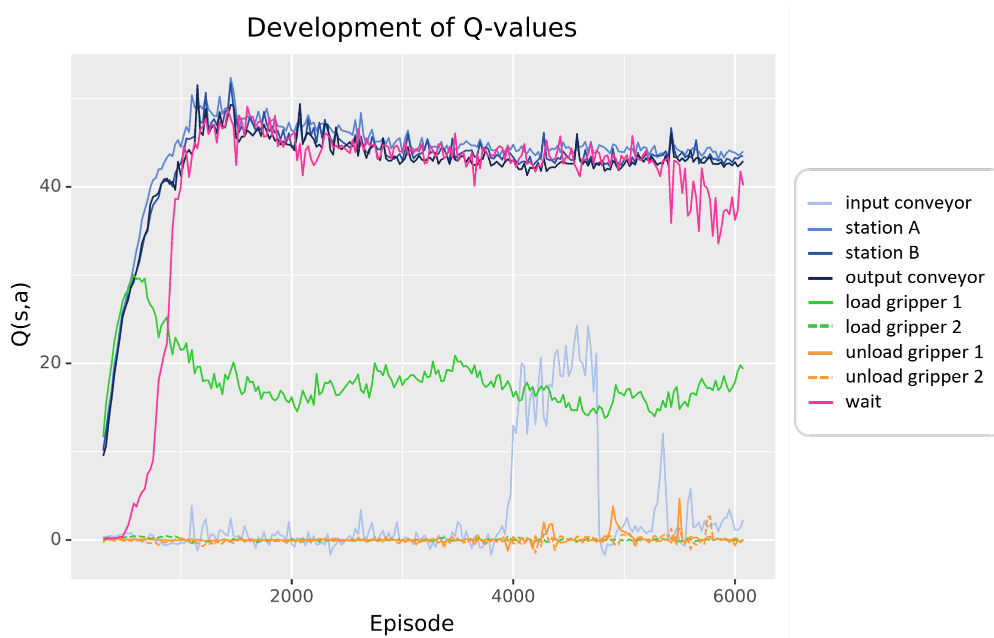


Figure 9.11: Development of the Q-values for the state where one gripper is empty, located at the input conveyor and could theoretically be loaded

allow further processing and keep the gantry robot productive. For example, if station A has failed, the agent will focus on working with station B and will interact with station A once it is up and running again. Figure 9.12 demonstrates that loading at the input conveyor is the best choice if station A is failed and station B is still processing. It is interesting to see that the same state as before leads to an entire different action if only the aspect of failed station A is considered. More importantly, the agent is able to distinguish between these two different states accordingly.

All three figures show only Q-values up to episode 6,100, since the last application model was extracted at this point. Furthermore, the x-axis of these figures indicate in which episode the neural network was trained with this experience for the first time. Hereby, it becomes clear that the initial state is part of the training batch from early on. However, the state where loading would be the wrong action choice is used after around 750 episodes. Thereupon, the neural network is trained with states that contain a failure at station A in episode 1,300 for the first time. Moreover, one can see that a distinction between different actions is made from the beginning. Nevertheless, the Q-values predicted by the network adjust over time and converge towards a certain value, even if they continue to fluctuate.

The last application model was extracted as a replica of the Q-network after ca. 6,100 training episodes. The remaining 2,600 episodes in which the Q-network was also trained

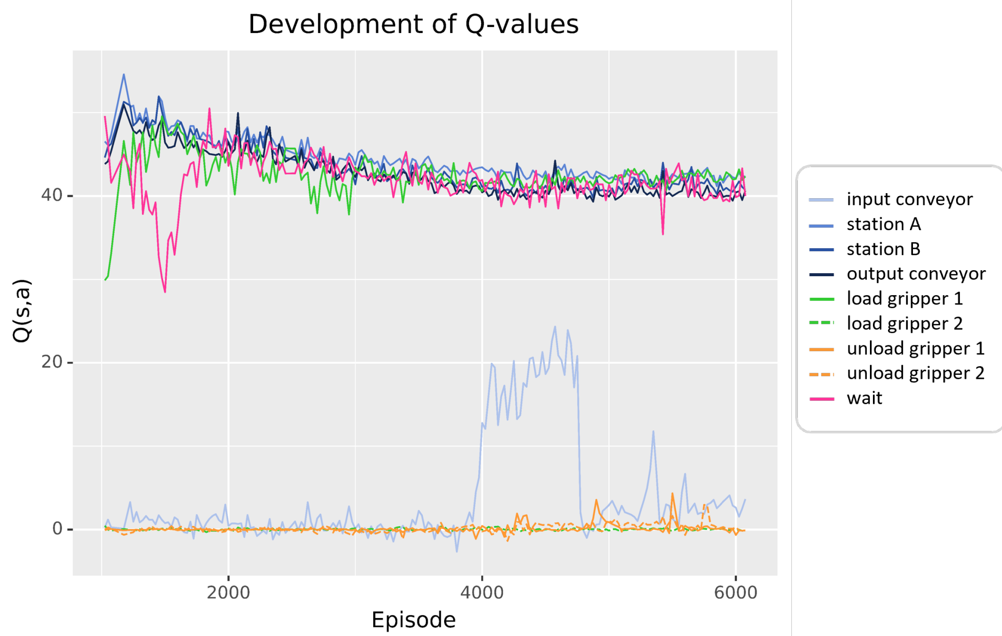


Figure 9.12: Development of the Q-values for the state where station A failed, station B is processing, one gripper is empty and the loader is located at the input conveyor

had no impact on the final model. This results from the fact that the update condition (the last average throughput value must be higher than that of the last update of the application model) was not fulfilled for the 2,600 subsequent training episodes. Up to the extraction of the application model, 1.6 million actions have been executed and due to the adjusted CRM approach more than 15 million experiences have been collected. This corresponds to almost 34 charges of the replay memory.

The trained gantry robot achieves an average throughput of 112.24 pieces per hour with a standard deviation of about 11. The learned policy works as follows: First the agent waits until an element occurs at the input conveyor, then this element is picked up by gripper 2. The element is delivered to station A and the gantry returns to the input conveyor after a short waiting time. Then, gripper 2 again picks up an item and moves to station A. In the meantime, station A had processed the input element which can be picked up by gripper 1. Shortly after, gripper 2 unloads the unprocessed item at this station. Then, the loader again returns to the input conveyor to pick up an item by gripper 2 and now moves to station B to empty this gripper. Hereupon, the gantry moves further to the output conveyor, in order to unload the processed element previously loaded from station A. Then, again a piece is picked up from the input conveyor and delivered to station A where a processed part is also loaded. This policy is executed until the system

is stopped manually.

Due to the continuous character of the task, the policy is repeated over and over again. Beginning from the reward machine state *G1 & G2: empty*, it takes twelve actions until the same state is reached again. However, 26 actions are necessary to achieve not only the same state in the reward machine but also the same state in the environment. Thus, the cycle length in the environment amounts to 26, while a cycle in the reward machine takes twelve steps. The cycle in the environment corresponds to two cycles in the reward machine. This results from the fact, that every second cycle a different sequence of actions has to be executed. In the first cycle, an element is loaded from station B, while in the second cycle an element is delivered to station B. However, since the cycle starts with loading an element from station A, in each cycle station A is filled and unfilled. This results from the fact that it takes 9 to 10 actions until an item can be picked up again from a station.

A validation run proved that this model is able to handle if both machines are failed at the same time. This is a state that occurs only rarely. In that case, the agent chooses to go to the input conveyor and load items. After that, it waits until the first station is up again.

For better distinction, the model trained with the adjusted reward machine approach is referenced as *adjustedRM*.

9.3.2 Original Reward Machine Approach

Since Icarte et al. [17] indicated that their reward machine approach, especially their CRM algorithm, is also applicable for continuous reinforcement learning problems, the gantry robot is also trained with the same underlying reward machine as used by model *adjustedRM*. Furthermore, we want to examine whether the assumptions made in Chapter 8 regarding the application of this algorithm to continuous tasks are sound.

9.3.2.1 Model

The model was trained for 41,400 episodes with an ε-decay of 0.999. The remaining parameters are quite similar to the model trained by the adjusted reward machine approach. Furthermore, the same neural network architecture is used. An overview of the applied training parameters can be found in Table 9.7.

Parameter	Value
replay memory size	450,000
min. number of elements in replay memory for training	10,000
min. number of elements before ε decreases	450,000
batch size	288
ε-decay	0.999
time discount	true
update of target network	every 7 episodes
activation function of hidden layer	rectifier linear unit

Table 9.7: Training parameters for the original reward machine approach

Observations The first 630 iterations were necessary to achieve the minimum required number of experiences in the replay memory required for training. Furthermore, Figure 9.13 illustrates the number of executed actions during the phase of random action selection (27,800 episodes). One can see that mostly only one to five actions were executed in each episode. Therefore, in total, it took 27,800 episodes until the replay memory was filled for the first time and the ratio of random decisions started to decrease. After further 7,000 iterations, the minimum possible ε-value of 0.001 was reached.

Figure 9.14 shows that no throughput could be achieved for over 30,000 episodes. When the first item was successfully delivered to the output conveyor, the ε-value amounts already to 0.10 and more than 0.6 million experiences have been collected. In comparison, the *adjustedRM* model achieved a throughput already in the first training episode with less than 1,800 collected experiences. Besides, while the agent is still exploring the state

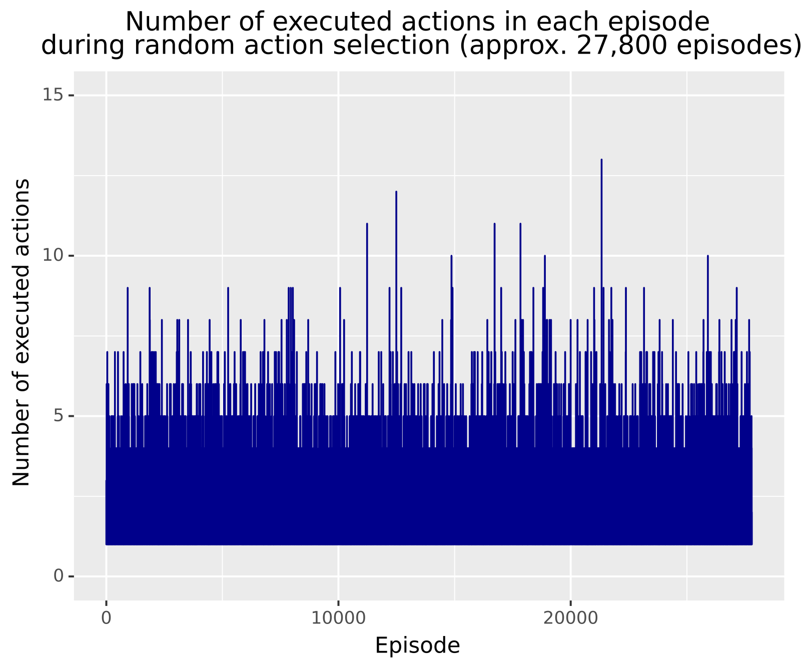


Figure 9.13: Number of executed actions when the replay memory is only filled randomly

space and randomly selecting actions, in the *adjustedRM* model, an item is successfully delivered to the output conveyor with an observed probability of 98 %. Moreover, Figure 9.14 illustrates that there are still many episodes achieving no throughput at the end of the training. This demonstrates a big difference between the original and the adjusted reward machine approach, where the throughput varies less at the end of the training.

Relevant states that depend on the so far learned behaviour such as machine failures were observed approximately 3,600 episodes (in episode 31,432) after pure random decision making ended. At this point, the ε-value amounts to 0.026. Accordingly, if 200 actions could be performed in one iteration, calculative 5.2 actions would be chosen randomly. When the machine failure is experienced for the first time, around 1.0 million experiences have been observed so far. This means it has approximately 100,000 more experiences than when model *adjustedRM* first observed a machine failure.

Moreover, a throughput of 108 elements was observed in episode 33,011 (ε of 0.00545 and 3.0 million collected experiences) for the first time. This is shown in Figure 9.14. Furthermore, one can see, that from this point on, 90 % of the episodes achieved throughputs in the range between 75 and 108. In contrast, the *adjustedRM* model needed around 5.3 million experiences before observing a throughput of 108.

Besides, Figure 9.15 visualises how the simulation duration evolves over time. Since the original reward machine algorithm cancels an episode when a terminal state is reached, it is not possible to train each episode for 20 minutes (1,200 seconds). It is shown that

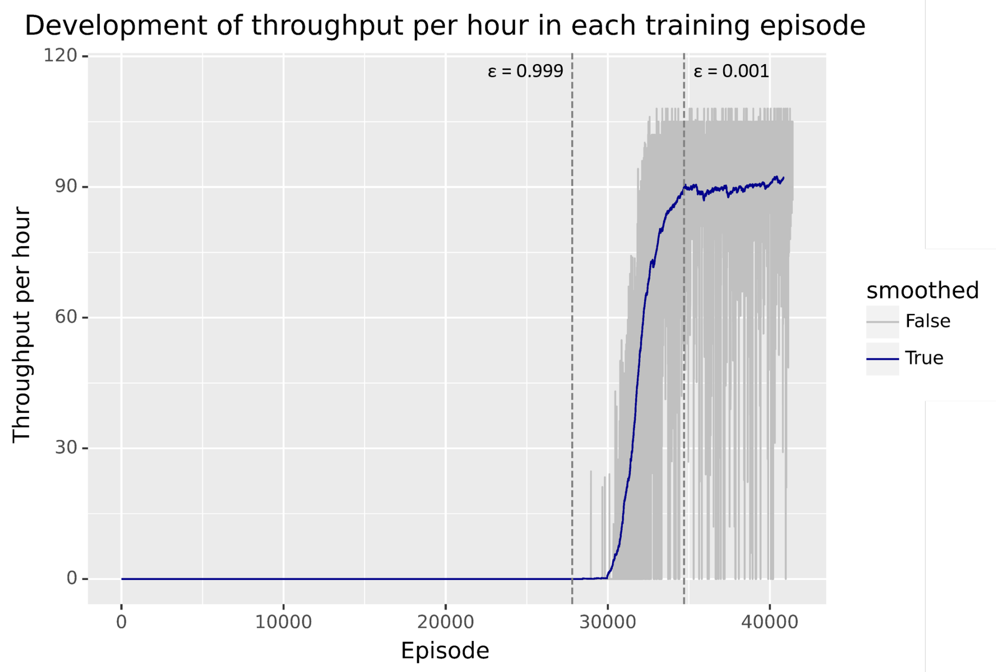


Figure 9.14: Training progress of the model trained with the original reward machine algorithm [a]

[a]smoothed as explained in the Appendix in section Smoothing

the simulation terminates very early for the first 28,000 iterations. From this point on, the agent starts to learn, which affects the duration of the episodes. After around 30,000 training iterations, the first episode is trained for the entire 20 minutes. This means that no terminating behaviour has been executed. It is interesting to see that even at the end of the training there are still episodes that terminate very quickly. However, most episodes can simulate for the entire predefined time. In addition, a clear correlation between the development of the throughput (Figure 9.14) and the evolution of simulation duration (Figure 9.15) can be seen. In contrary, the model trained with the adjusted reward machine approach always executed an episode until the simulation duration had expired.

Furthermore, although the replay memory was filled with 450,000 experiences after 27,800 episodes, the variance of these experiences is very limited. All in all, around 225 different states can be identified, whereby the initial state S_0 occured more than 6 % of the time. In general, the initial state of the environment occurs most frequently, as one can see in Table 9.8 in the first row. Besides, around 85 % of all states do not hold an element at the input conveyor. This ratio is also reflected in the overview of the five most frequent states, which cover already 92 % of the entire replay memory (considering all forms of reward machine states). The *adjustedRM* model holds twice as much distinct

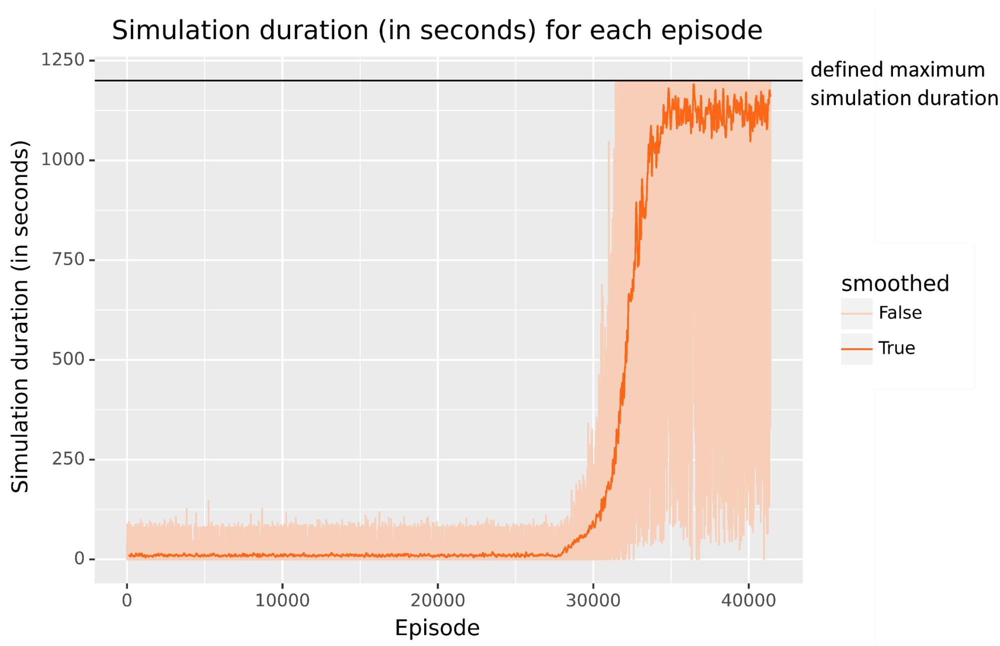


Figure 9.15: Simulation duration for the original CRM algorithm for each episode [a]

[a] smoothed over 100 episodes by a triangular window centred around the corresponding episode

Location	Element at input conveyor	Status of station A	Status of station B	Frequency of each state	Frequency for all reward machine states
input conveyor	false	empty	empty	30,013 (6.67 %)	270,117 (60.0 %)
output conveyor	false	empty	empty	4,250 (0.94 %)	38,250 (8.50 %)
station A	false	empty	empty	4,155 (0.92 %)	37,395 (8.31 %)
station B	false	empty	empty	4,114 (0.91 %)	37,026 (8.23 %)
input conveyor	true	empty	empty	3,855 (0.86 %)	34,695 (7.71 %)

Table 9.8: Overview of the five most frequent states of the environment in the replay memory at the beginning of the training (trained by the original reward machine approach)

states in its initial replay memory. Furthermore, this model indicated a greater variance, since the top five states cover around 41 % of the entire replay memory.

At the end of the training, the replay memory contained 855 distinct states and thus, nine states less than model *adjustedRM*. In Table 9.9, the five most occuring states are summarised. In comparison to the five most frequent states in the initial filled replay memory, one can see that the variance of states increased over training. All five most frequent states of the environment included empty machines, machines that were currently processing an element and machines with a status of a completed processing. The formation of the final replay memory is quite similar to the one of model *adjustedRM*.

Location	Element at input conveyor	Status of station A	Status of station B	Frequency of each state	Frequency for all reward machine states
output conveyor	true	in progress	empty	4,267 (0.95 %)	38,403 (8.53 %)
station B	true	in progress	empty	3,042 (0.68 %)	27,378 (6.08 %)
input conveyor	true	processing completed	empty	2,530 (0.56 %)	22,770 (5.06 %)
station B	true	in progress	processing completed	2,355 (0.52 %)	21,195 (4.71 %)
station A	true	empty	in progress	2,291 (0.51 %)	20,619 (4.58 %)

Table 9.9: Overview of the five most frequent states of the environment in the replay memory at the end of the training (trained by the original reward machine approach)

The last application model was extracted from the trained Q-network after episode number 40,739, which corresponds to around 13,000 episodes after the initial filling of the replay memory and the begin of a decreasing ε. The model was trained for further 600 episodes but with no affect on the application model. The application model results from a collection of 20.9 million experiences (ca. 45 replay memories) which were gathered from 2.3 million observed state-action-state transitions.

The trained gantry robot achieves an average throughput of 110.96 elements per hour. The resulting policy is similar to the one learned by the model *adjustedRM*. Furthermore, the given policy is able to respond accordingly to machine failures. If either station A or station B fails, the robot is still able to achieve throughputs by interacting with the machine that is still running. However, when both machines are not available, then an incorrect decision is made. Instead of waiting or moving between different machines, the agent tries to unload an element at an unavailable station. The policy differs from that of the *adjustedRM* model by the behaviour when both machines suffer a failure.

In the following, the trained model is referenced as *origRM1*.

9.3.2.2 Model based on Training Parameter from Adjusted Reward Machine Approach

To provide a solid comparability between the original and the adjusted approach, the same training parameters as used for training with the adjusted algorithm are applied.

Observations Since the model *adjustedRM* trained by the adjusted reward machine approach and the optimised model *origRM1* trained by the original CRM algorithm differ only in the value of the ε-decay, the development of the training is equal for both models trained by the original approach. However, the difference in the ε-decay value comes into play after the replay memory has been filled initially. This is after around 27,800 episodes. Up to this point, Figure 9.13 also represents the number of actions performed in each episode for this model. Differences between the *origRM1* model and the current model only arise from this point on. In total, the current model was trained for 35,450 episodes.

Due to the smaller ε-decay, the ratio of random actions decreases faster for the current model than for the first model trained by the original approach. Therefore, the minimum possible ε-value of 0.001 was already reached 3,500 iterations (in episode 31,253) after the initial replay memory was filled. This corresponds to a collection of 3.5 million experiences which is one third of the number of experiences necessary for model *adjustedRM* reaching the minimum ε-value.

Furthermore, the first throughput is already achieved after 28,500 episodes and an ε of 0.23. At this point, around 475,000 experiences have been collected. Moreover, the first machine failure can be observed in episode 29,798 (ca. 2,000 episodes after the begin of random decisions) when the agent holds an ε-value of 0.018. This means that approximately 3.6 actions were randomly selected in this episode. By the time the first machine failure was observed, almost 0.9 million experiences had been gained. Beyond that, a throughput of 108 elements was initially achieved in episode 30,767 (ε of 0.0026), by which time approximately 2.4 million experiences had been collected. From this point on, 90 % of the episodes achieve a throughput of at least 70 items per hour. The training progress as described can be seen in Figure 9.16. A comparison between the model *adjustedRM* and the currently trained model shows that relevant states are observed in fewer episodes by model *adjustedRM*, but fewer actions had to be performed to be observed by the currently trained model.

Due to the similar training behaviour for the first 27,800 episodes of the two models trained with the original reward machine approach, the initial replay memory varies only slightly. However, at the end of training, the replay memory contains nine more different states. Moreover, the five most occuring states vary not only in frequency but also in the states itself. The replay memory of this model contains either empty machines or machines that were currently processing an element. However, no machine with a status

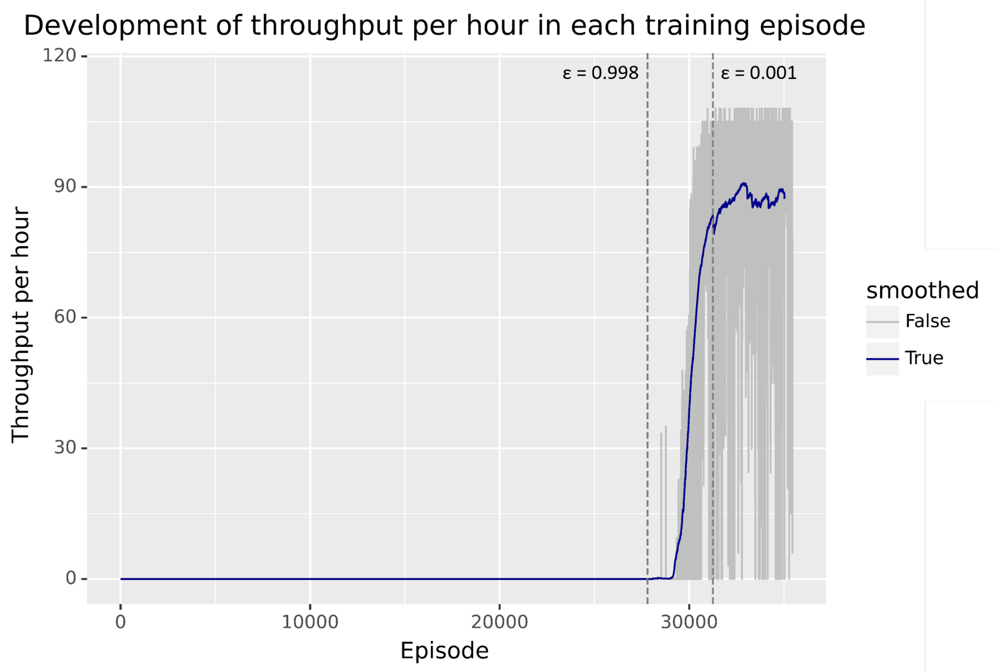


Figure 9.16: Training progress of the model trained with the original reward machine algorithm, training parameters from model *adjustedRM* [a]

[a]smoothed as explained in the Appendix in section Smoothing

of a completed processing occurs in the top five list (Table 9.10). This states also a difference compared to the final replay memory of the model *adjustedRM*.

The last application model originates from 34,248 training episodes which corresponds to 6,445 episodes after the initial replay memory was filled. So, the model from the adjusted approach (*adjustedRM*) and this model iterated almost the same amount of training episodes based on the number of episodes with a decreasing ratio of random decisions ($\varepsilon < 1.0$). In fact, the training went on for another 1,200 episodes. However, this further training had no effect on the application model. The resulting application model used 10.8 million experiences (approx. 24 replay memories), derived from 1.2 million actual observed transitions in the environment. This corresponds to half as much as model *origRM1* and two third of model *adjustedRM*.

The trained gantry robot achieves an average throughput of 109.17 elements per hour. The resulting policy is similar to the one learned by means of the adjusted algorithm. However, when station A fails, the gantry robot moves between station B and the output conveyor. If station B is busy processing an item, the robot moves between station B and the output conveyor until the element is processed and ready to be picked up. Then this element is successfully delivered to the output conveyor, but the agent continues driving

Location	Element at input conveyor	Status of station A	Status of station B	Frequency of each state	Frequency for all reward machine states
output conveyor	true	in progress	empty	3,744 (0.83 %)	33,696 (7.49 %)
station B	true	in progress	empty	3,010 (0.69 %)	27,090 (6.02 %)
input conveyor	true	empty	empty	2,581 (0.57 %)	23,229 (5.16 %)
input conveyor	true	empty	in progress	2,552 (0.57 %)	22,968 (5.10 %)
station A	true	empty	empty	2,341 (0.52 %)	21,069 (4.68 %)

Table 9.10: Overview of the five most frequent states of the environment in the replay memory at the end of the training (trained by the original reward machine approach with training parameters from model *adjustedRM*)

from station B to the output conveyor and back again. Yet, a breakdown at station B does not lead to an inefficient behaviour of the gantry robot, instead the loader focuses on loading and unloading at station A. Thus, in this case, the robot is still able to achieve throughputs. Compared to the learned policies of model *adjustedRM* and *origRM1*, the currently trained model can handle machine failures less efficiently.

In the following, the model presented is called *origRM2*.

9.3.3 Basic Learning Approach without Reward Machine

Usually, reinforcement learning agents are trained without a reward machine. In order to evaluate the proposed reward machine approach in terms of training performance and policy quality, such a model must also be trained without a reward machine. This model facilitates a later comparison between reward machines and default reinforcement learning.

While reward machines have the opportunity to observe terminating behaviour and consider their terminating character, this kind of behaviour is usually prohibited when learning without reward machines. Therefore, the action space must be restricted to only actions that are allowed in a certain environment state. All actions that would lead to an error in a certain state of the production environment are suppressed. This limits the action space for each situation to the permitted operations. In addition, the allowed actions must be determined in advance based on certain conditions. For example, a loading action can only be chosen if the corresponding gripper is empty and an object to be picked up is present.

We disallowed all actions that lead to a terminal state in the reward machine in order to be congruent in both learning approaches.

Moreover, since the information of the reward machine states are still part of the environment, the basic learning approach without reward machine is trained on a MDP that was formed as a cross-product of MDP and reward machine. However, as it is an MDP, the reward structure revealed cannot be exploited.

9.3.3.1 Model

As mentioned earlier, training without reward machines requires a restriction layer (multiply layer) in the neural network to penalise restricted actions and thus suppress actions that are not allowed.

In addition, a training setting is used to train the DQN-network for the basic model, which is summarised in Table 9.11. For the first 10,000 agent-environment interactions, actions are drawn randomly from the restricted action space. Furthermore, to prevent training on correlated data, an initially filled replay memory is required before training can begin. In total, the gantry robot agent is trained with a batch of 64 elements from the replay memory containing the last 50,000 experiences. This batch is randomly drawn during each training iteration. Furthermore, the probability that the model is trained by such a batch after an action is executed amounts to 0.66. An ε-decay of 0.9995 results in a slow decrease in the exploration rate ε, providing a longer phase of random behaviour.

Besides, the training facilitates for time-based discounting, as explained in Section 9.1.

Parameter	Value
replay memory size	50,000
min. number of elements in replay memory for training	10,000
min. number of elements before ε decreases	50,000
batch size	64
ε-decay	0.9995
time discount	true
update of target network	every 10 episodes
activation function of hidden layer	rectifier linear unit

Table 9.11: Training parameters for a standard learning approach without reward machine

Observations The model is trained for more than 16,000 episodes. After about 150 episodes, the replay memory of size 50,000 was filled for the first time. This corresponds to approximately 300 executed actions per episode. As with the adjusted reward machine approach, the model can observe machine processing and throughput from the first iteration on. During these 150 episodes, in which actions are only randomly selected, at least one throughput is achieved 86 % of the time and consequently a loop of the continuous task is performed. This is visualised in Figure 9.17. Furthermore, one can see that the achieved throughput increases for approximately 9,500 episodes and remains on almost the same level for further 6,500 episodes. The first machine failure as a relevant but stochastic aspect in the environment can be observed in episode 568 with an ε-value of 0.81. Hence, the machine failure occurs at a rather similar point in time as in the *adjustedRM* model.

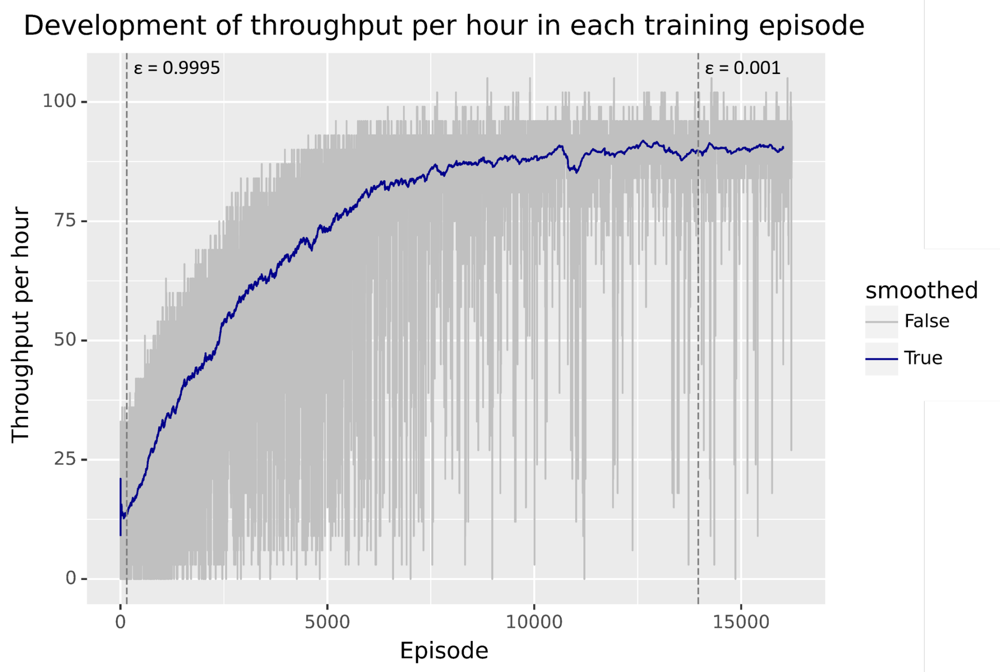


Figure 9.17: Development of the observed throughput during training (approach without reward machines) [a]

[a]smoothed as explained in the Appendix in section Smoothing

The initial replay memory contains around 350 different states, which ranges between the number of distinct states of the adjusted (459) and the original reward machine approach (225). Table 9.12 lists the five most frequent states in the replay memory. The first four states differ only in location, yet their frequency varies. In contrast to the two reward machine approaches, the frequency of the states varies for different statuses of the grippers. Therefore, Table 9.12 contains the statuses of the grippers which was not relevant for reward machine approaches due to the CRM algorithm. Furthermore, the column *Frequency for all reward machine states* makes no sense for a learning approach that does not utilise reward machines. In Table 9.12, one can see that both machines are in use for the most frequent states in the replay memory. Moreover, a sharp decrease in frequency can be seen between the first four states compared to the fifth most frequent state.

The shape of the replay memory at the end of the training paints a rather similar picture. At the end of the training, the replay memory contains ten states less than at the beginning. Four out of five states in the replay memory hold the same status of the grippers. Furthermore, the frequency of the top five states is more evenly spreaded than at the beginning which one can see in Table 9.13.

Location	Element at input conveyor	Status of gripper 1	Status of gripper 2	Status of station A	Status of station B	Frequency
input conveyor	true	filled (unprocessed)	filled (unprocessed)	processing completed	processing completed	5,951 (11.9 %)
output conveyor	true	filled (unprocessed)	filled (unprocessed)	processing completed	processing completed	5,936 (11.87 %)
station A	true	filled (unprocessed)	filled (unprocessed)	processing completed	processing completed	5,837 (11.67 %)
station B	true	filled (unprocessed)	filled (unprocessed)	processing completed	processing completed	5,755 (11.51 %)
input conveyor	true	empty	empty	empty	empty	701 (1.4 %)

Table 9.12: Overview of the five most frequent states of the environment in the first full replay memory (trained by the approach without reward machine)

Location	Element at input conveyor	Status of gripper 1	Status of gripper 2	Status of station A	Status of station B	Frequency
station B	true	empty	empty	processing completed	in progress	3,015 (6.03 %)
input conveyor	true	empty	empty	empty	in progress	2,735 (5.47 %)
output conveyor	true	empty	empty	empty	in progress	2,716 (5.43 %)
station B	true	filled (unprocessed)	empty	processing completed	in progress	2,699 (5.4 %)
station A	true	empty	empty	empty	in progress	2,068 (4.14 %)

Table 9.13: Overview of the five most frequent states of the environment in the replay memory at the end of training (trained by the approach without reward machine)

The application model resulted from a training over 14,500 episodes, with no further change in the application model with further training of 2,000 episodes. A validation run of the final model achieved an average throughput of 105.45. Hence, the agent learns to solve the gantry robot scheduling problem. In general, the learned policy corresponds to the one of model *adjustedRM*, however, these policies differ when a new element is to be loaded from the input conveyor. In this case, a waiting action is selected.

Although the reward machine state does not play a role when training without reward machine, the information on the gripper status are still part of the environmental state. Furthermore, the validation run shows that only seven different combinations of the gripper statuses has been used. That corresponds to seven different states of the reward machine.

In the following, this model trained without reward machines is referenced as *basicModel1*.

9.3.3.2 Model based on Training Parameter from Adjusted Reward Machine Approach

When a model is trained without reward machine but using the same neural network architecture and training parameters as the model trained with reward machine, there is no multiply-layer in the neural network that ensures that only valid actions are executed. While decisions are met randomly based on a restricted action space, this has no impact on which action is executed next. However, as soon as the randomness decreases, an action is chosen based on the highest Q-value the Q-network predicts for a given state. Since there is no layer restricting the action space, each action - also incorrect once - could be selected.

Observations The model was trained for more than 8,300 episodes. It takes around 30 episodes until the actual training of the neural network can start. Furthermore, after about 1,300 episodes enough experiences have been collected so that the replay memory is filled for the first time. Moreover, the minimum achievable ε-value of 0.001 is reached in episode 4,809. In addition to that, the first machine failure can be observed after around 1,500 episodes. At this point, the ε-value amounts to 0.75 which lays more than 0.1 above the value of the *adjustedRM* model. This training progress can be seen in Figure 9.18. Furthermore, this plot shows that the throughput per hour reaches a plateau after around 4,500 episodes. Nevertheless, a throughput of 108 is never observed.

Figure 9.19 compares the development of the two models trained without reward machine. One can see that the current model (training parameters from the adjusted reward machine model) has a longer initial phase with constant throughput than the model *basicModel1* (own training parameters). However, after this constant phase, the achieved throughput of the current model increases faster than that of the other one. Moreover, this graph visualises the difference in the number of episodes used for training.

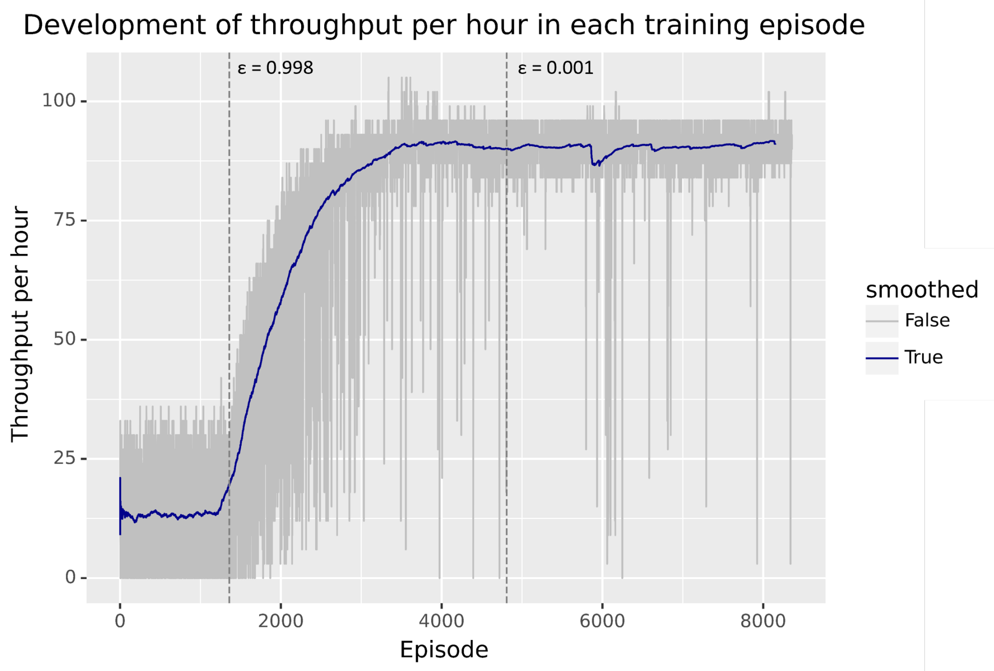


Figure 9.18: Training progress of model trained by the approach without reward machine, training parameters from adjusted reward machine approach [a]

[a] smoothed as explained in the Appendix in section Smoothing

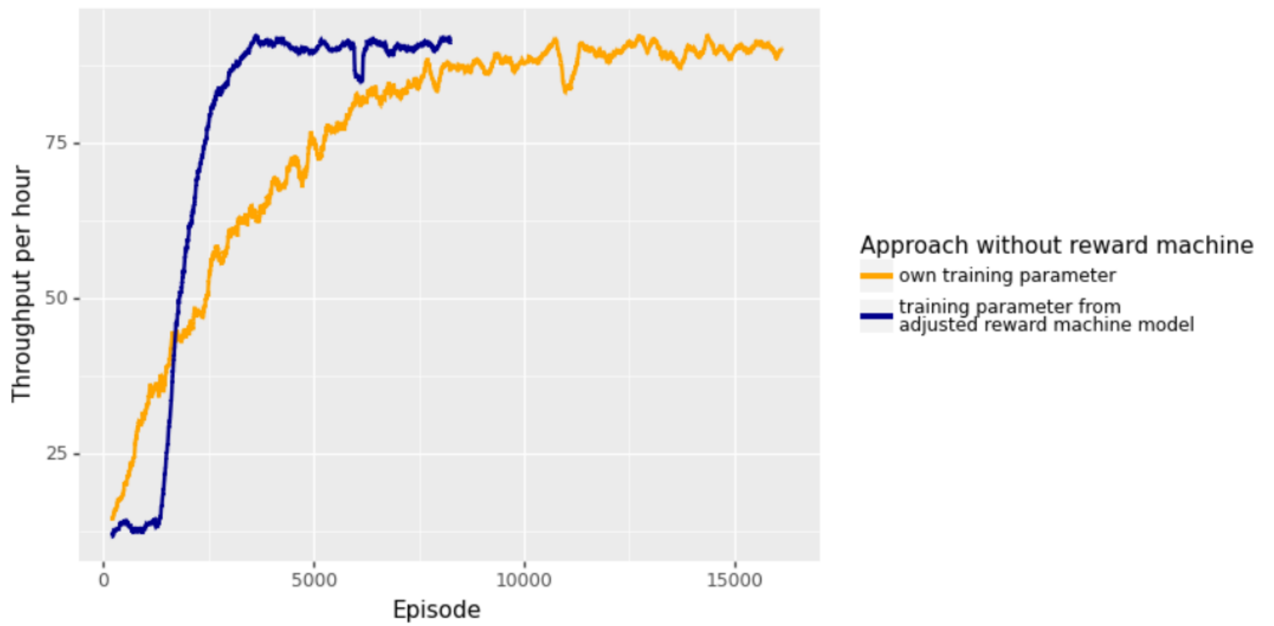


Figure 9.19: Training progress of both models trained by the approach without reward machine

The first filled replay memory contains 430 distinct states, of which the five most frequent correspond to the states listed in Table 9.12. However, due to the larger replay memory, the absolute frequency differs, even if the percentage remains approximately the same. Furthermore, unlike model *basicModel1*, the current model includes 80 more states in the initial replay memory.

While the replay memory at the beginning contains only around 430 distinct states, at the end of the training, it contains 508 different states. Hereby, almost one third are allotted to the five most frequent states. These five states are listed in Table 9.14. Moreover, as explained above, unlike the approaches with a reward machine, the statuses of the grippers are described explicitly, since they occur independently of any other state. Nevertheless, Table 9.14 demonstrates that the three most occuring states represent the same status of gripper 1 and gripper 2.

Location	Element at input conveyor	Status of gripper 1	Status of gripper 2	Status of station A	Status of station B	Frequency
input conveyor	true	empty	empty	empty	in progress	35,671 (7.93 %)
output conveyor	true	empty	empty	empty	in progress	34,965 (7.77 %)
station A	true	empty	empty	empty	in progress	23,551 (5.23 %)
output conveyor	true	empty	filled (processed)	empty	in progress	20,335 (4.52 %)
station B	true	filled (unprocessed)	filled (processed)	empty	empty	20,321 (4.52 %)

Table 9.14: Overview of the five most frequent states of the environment in the replay memory at the end of training (trained by the approach without reward machine with parameters from model *adjustedRM*)

During the entire training no incorrect action could be observed, although the trained neural network does not contain a specific layer restricting the action space.

The final model originated from the training of the first 3,500 episodes, since training of further 4,500 episodes did not impact the application model. A validation run of 100 iterations returned an average throughput per hour of approximately 107.5 which is below the achieved processed items of models trained by the approaches with reward machine. In contrast to the policy of model *adjustedRM*, the trained model selects a waiting action whenever both machines are busy and one gripper is filled although this was not forced by the restricted action space. In addition to that, this model is capable of handling failures on both machines simultaneously.

In the following, this model trained without a reward machine is referenced as *basic-*

Model2.

9.3.4 General Observations over all Learning Approaches

Since we used the same training parameters in the models *adjustedRM*, *origRM2* and *basicModel2*, observations can be made that contrast all three learning approaches. The same training parameters make it possible to attribute differences in observations to the underlying learning approach. Furthermore, as the research question focuses on the acceleration of reinforcement learning using reward machines for continuous tasks, the differences between the approaches are considered based on various comparison criteria such as the number of episodes, the filled replay memories and the executed actions.

Figure 9.20 illustrates how the throughput changes across the training episodes. It can be seen that the model trained with the adjusted reward machine approach (*adjustedRM*) and the model trained without a reward machine (*basicModel2*) are quite similar, while the model resulting from the original reward machine approach (*origRM2*) is shifted by almost 30,000 episodes. Moreover, a difference in the throughput at the beginning of the training can be recognised. The model *basicModel2* has the highest throughput at the beginning. In contrast, the model *origRM2* does not achieve any throughput in the first 27,000 episodes, as was already found in the analysis of this model.

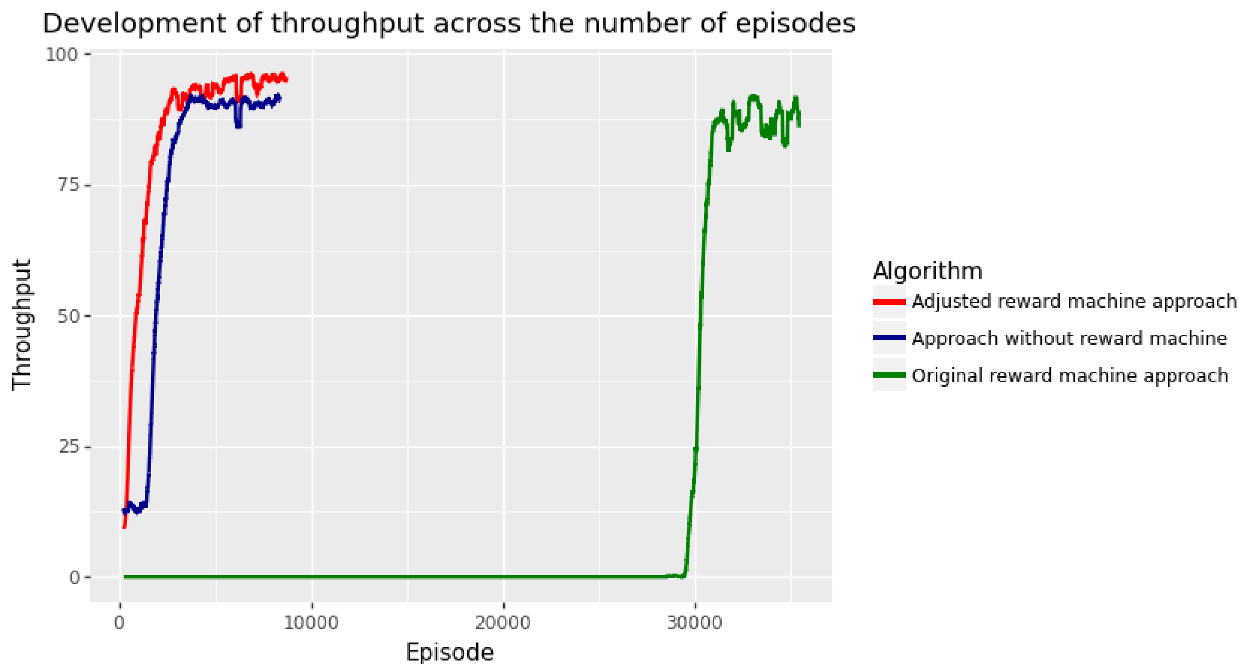


Figure 9.20: Development of throughput across the number of training episodes [a]

[a] smoothed over 250 episodes by a triangular window

In addition to this comparison criteria, the number of filled replay memories has been

used as a basis. This number results from the amount of all experiences during the training. These experiences can be observed or synthetically generated as in the CRM algorithm. Hence, it describes the amount of data available for training. Figure 9.21 illustrates the development of the throughput in relation to the number of filled replay memories. In contrast to Figure 9.20, all three approaches show a similar development. The ranking of the achieved throughput at the beginning of the training remains the same as it was determined in the comparison based on the episodes. Nevertheless, one can see that all three approaches vary in their progress. The model trained without reward machine recordes the largest increase in throughput. This is followed by the model trained with the original reward machine approach, and in third place by the model trained with the adjusted approach. On the one hand, the model from the original reward machine approach grows faster than that of the adjusted approach. On the other hand, both approaches cross between 15 and 20 filled replay memories and reach a similar level of throughput. Besides, all three models show a different final number of filled replay memories, in contrast to Figure 9.20, where all models were trained for approximately the same number of episodes starting with the one with a decreasing ε-value.

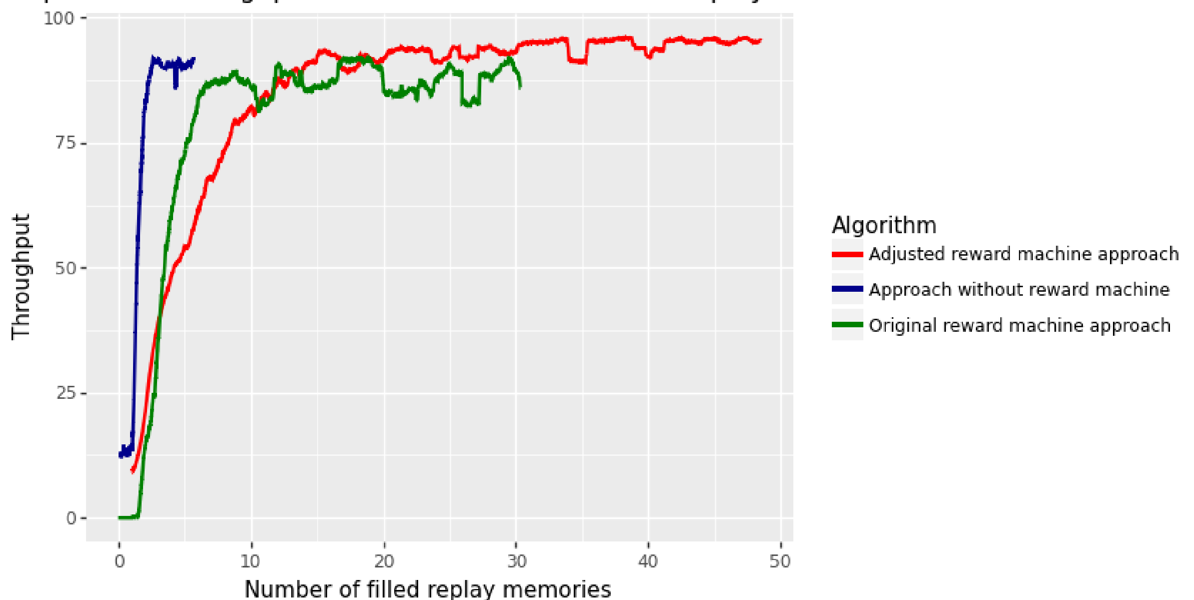


Figure 9.21: Development of throughput across the number of filled replay memories (observed and counterfactual experiences) [a]

[a] smoothed over 250 episodes by a triangular window

A third possibility for comparison results from the number of actions performed. Here, only observed, but not synthetically generated experiences are taken into account. Figure

9.22 visualises how the three approaches develop depending on the number of actions observed so far. One can see that both reward machine approaches increase earlier than the model trained without a reward machine. Furthermore, the model from the original reward machine approach indicates a stronger increase in throughput than the model trained with the adjusted reward machine approach. In addition to that, the adjusted reward machine approach shows a similar development as the model without reward machine, however shifted to the left by 500,000 observed actions.

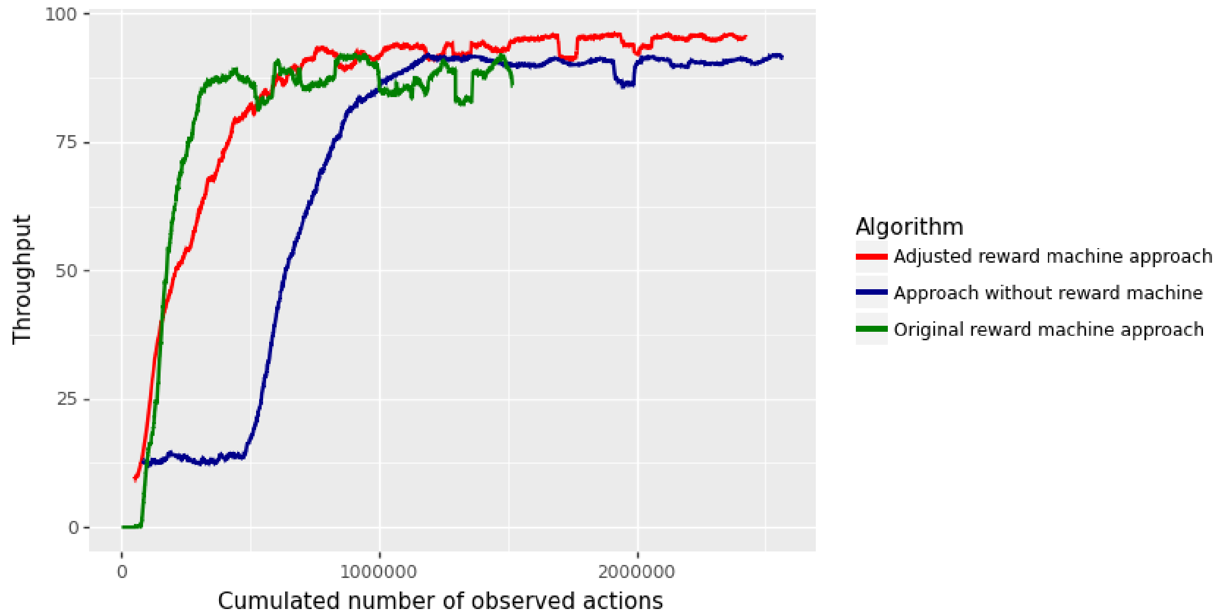


Figure 9.22: Development of throughput across the number of executed actions (without counterfactual experiences) [a]

[a]smoothed over 250 episodes by a triangular window

These three figures reflect very different training progresses depending on the comparison criteria. For each criterion, another approach increases the strongest.

10 Discussion

Reward machines are able to find an optimal policy for episodic reinforcement learning problems faster than standard approaches without reward machines. However, since continuous problems are considered more complex, one might wonder whether the advantages of the reward machine approach also apply to continuous reinforcement learning problems. Furthermore, while basic models tend to use a restriction of the action space, reward machines allow erroneous behaviour and react to it by terminating an episode. Therefore, the question arises whether the reward machine approach is capable of learning the restriction of its action space itself in a state-dependent manner.

The experiments revealed that both the adjusted and the original reward machine approaches were suitable for continuous reinforcement learning problems. Furthermore, the reward machine approaches were able to learn the difference between correct and incorrect actions.

This discussion is structured as follows: First, the results from the experiments of the adjusted reward machine approach are discussed. Then the same is done for the original reward machine approach. In a next step, both approaches are compared with each other. Furthermore, the results of the learning approach without reward machines are discussed. Finally, all three approaches are compared with each other, leading to an implicit comparison of reward machine approaches and approaches without reward machine.

Adjusted reward machine approach The experiments demonstrated that the adjusted reward machine approach is able to learn an optimal policy for a continuous reinforcement learning problem. Enabling an agent to continue with the current episode, even though it has performed an incorrect action, has a positive effect on the variety of states that enter the initial replay memory. This results from the different number of distinct states of the three approaches. While the adjusted approach holds 459 distinct states, the original reward machine approach has 225 and the approach without reward machine 430 different states. By adapting the CRM algorithm, the DQN can be trained earlier with relevant data. Furthermore, since relevant events in the environment can be observed earlier, the number of episodes needed to observe such states also impacts the ratio of random behaviour. An early observation supports a trial-and-error training process for these states. This could be recognised by the different ε-value when observing a machine failure. Moreover, the evolution of Q-values shows that the agent first learns which actions are wrong, as the Q-values of incorrect actions are shrunk to zero, while advantageous actions need more training iterations to reach a plateau. The last aspect results also from the

fact that it takes many episodes to reach the maximum possible reward. Since each successful throughput increases the return, this also affects the Q-values of a given state. In particular, Figures 9.10 and 9.11 illustrated that the Q-value for favourable actions increases in the first 1,500 episodes. Occassionally, however, the Q-value also starts to increase for incorrect actions. This could be due to the increasing proportion of correct behaviour. In addition, backpropagation affects weights in the DQN that also have an impact on the prediction of the Q-values for incorrect actions. Although the proportion of correct actions is increasing, there is still a non-neglectable ratio of incorrect observed actions resulting from the synthetically generated experiences of the CRM algorithm.

Original reward machine approach However, not only the adjusted reward machine approach is able to learn how to behave for continuous tasks, but also the original reward machine approach developed a policy to solve the gantry robot scheduling problem. The original reward machine approach terminates an episode early if an incorrect action is executed. This affects the duration of an episode and as a consequence leads to only few performed actions, especially at the beginning of the training which is implied by Figure 9.13. Furthermore, Figure 9.15 visualised how the early termination of the episode also influences the simulation duration. Since in each episode not the entire simulation time is utilised, less actions can be executed and as a consequence more episodes are necessary to fill the replay memory initially. However, as the training progresses, one gets closer and closer to the intended simulation duration. Moreover, a limited number of executed actions in each episode leads to a less varied replay memory at the beginning. In particular, the initial state of the environment occurs more frequently than average, which also results from the fact that mostly only up to five actions are executed, one of which corresponds to the initial state. Through the CRM algorithm, the observed experiences, adapted to the nine different states of the reward machine, are added to the replay memory in each episode. Thus, up to 45 experiences are added in one iteration. At least one of these 45 experiences contains the initial state.

Since some actions are randomly selected at the end of the training, a faulty action can be performed at any time during the simulation run. As a consequence, the episode terminates. This explains why the variance of throughput at the end of training is quite large compared to the other two approaches.

Due to the early termination, it was expected that the original reward machine approach is not capable to learn solving a continuous task. It was assumed that the early termination limits the variance of the environment states since not all aspects of the environment are covered by the reward machine states. Especially, at the beginning of the training, variations in the status of a machine are rather rare. Furthermore, since less than five actions were executed most of the time at the beginning of the training, no throughput could be observed. On the other hand, the expectation was not satisfied since the original reward machine approach (models *origRM1* and *origRM2*) was able to solve the gantry robot scheduling problem. Moreover, the observed throughput missing at the beginning

was compensated by the CRM algorithm, which synthetically generates such an observation under certain circumstances for a chosen loading action. In addition to that, the abort of an episode due to an incorrect executed action supports learning which actions are erroneous. Further, early termination limits the relevance of the ε-value, as shown by the occurence of machine failures in the experiments. When a machine failure was observed for the first time, the ε-value and thus the degree of exploration (ratio of random behaviour) was rather limited. However, with a high degree of exploration it is rather unlikely that a machine failure could be observed. This results from the fact that the agent must execute many actions without error in order to observe such a rare state. Since there are fewer opportunities to explore how to behave in a rare state, e.g. machine failure, the agent is very likely to be forced to choose the action with the highest Q-value. Since the network has not been trained by such a state, the greedy output is only influenced by related states. If an incorrect action is selected, the current episode is terminated and the observed experience is recorded in the replay memory with a note of termination. As a consequence, the original reward machine approach would learn what kind of behaviour is erroneous in order to prevent it in the future. Hence, the agent learns which actions not to execute in certain states. This corresponds to the policy of model *origRM2* where the agent was able to perform only correct actions when a machine was unavailable. However, these actions were not optimal. Moreover, in model *origRM1*, further training might be necessary to eliminate the incorrect behaviour when both machines fail.

These observations and implications match with the assumption in Chapter 8 that the exploration success is rather limited at the beginning. However, the experiments showed as well that the original reward machine approach is nevertheless able to learn the general behaviour to solve a continuous task. Only the selection of the correct action for aspects in the environment, that are not covered by the states of the reward machine and occur stochastically, is rather difficult.

Comparison between different reward machine approaches Both reward machine approaches were able to learn how to behave for continuous tasks. Furthermore, both approaches figured out which actions are erroneous in certain states. Beyond that, the adjusted and the original reward machine approach have in common that they first learn which actions are incorrect before exploring the action space for the best action. This is especially true when the level of exploration has begun to decrease. Besides, both the adjusted as well as the original reward machine approaches generate synthetical experiences for each action performed, regardless of whether that action is wrong or not.

Despite these similarities, both approaches differ in the way they learn these behaviours, even though both apply the concept of counterfactual learning. However, the main difference in these two approaches consists in the treatment of incorrect executed actions. While the original CRM algorithm cancels each episode as soon as an incorrect action was executed and a terminal state was reached, the adjusted algorithm considers an experience with an incorrect action as terminal, but is able to continue on the current

episode. This explains the great difference in the number of episodes necessary for training. Furthermore, the original approach requires more episodes because only few actions can be executed in each and thus, the simulation terminates before the maximum duration expires. As mentioned before, the difference in both approaches effects the number of episodes necessary to observe infrequent states. Since the adjusted approach is able to simulate an episode until the simulation duration expires, many interesting observations can be made from early on. Already at the beginning of the training process, the agent trained by the adjusted approach observes more than one iteration of the continuous task. This is expressed by the achieved throughput and constitutes a major advantage of the adjusted reward machine approach over the original reward machine approach, since the adjusted reward machine approach allows for observation of states that go beyond the first few actions.

As the adjusted approach takes note of an incorrect and theoretically terminating action, the agent gets the opportunity to try out different actions in one run. Especially, since the state of machine failure lasts longer and can thus be observed for a longer sequence, it can be experienced how different actions react to such a state. In contrast, the original reward machine approach is only able to observe machine failures, if the ε-value reached a certain level of exploitation. Thus, the scope of action is rather small. However, a different parameter set would not lead to an earlier observation of the machine failure. This is reflected by the ε-value of around 0.02 for the models *origRM1* and *origRM2*.

Furthermore, the initial replay memories of the two reward machine approaches differ due to the different treatment when incorrect actions are executed. The advantage of generating synthetical experiences turns out smaller for the original approach, since these synthetical experiences relate only to states that are covered by the underlying reward machine. However, some aspects in the gantry robot scheduling task that are not covered by the reward machine could impact its decisions, e.g. machine failures. If a machine is unavailable, an element cannot be loaded or unloaded at this station. However, this behaviour is not represented by the reward machine. Since the adjusted reward machine approach does not terminate an episode early, the initial replay memory contains a larger variance of states.

Furthermore, machine failure is not only difficult to learn because it is not one of the states of the reward machine, but also because it only occurs stochastically and requires that some processing time has passed on average. This makes it quite difficult for the original reward machine approach to observe this type of state. This can be justified by the experiments. Measured by episodes, the adjusted reward machine approach experiences the first machine failure much earlier which is indicated by the ε-value. Measured by the number of collected experiences, the model *origRM2* needs fewer actions than *adjustedRM* which on the other hand needs fewer actions than *origRM1*. Therefore, it can be followed that the adjusted approach requires more actions but less episodes to observe a machine failure assuming the same training parameters (model *adjustedRM* and model *origRM2*). Nevertheless, the adjusted reward machine approach is able to handle machine failure optimally, while the original reward machine approach to some extent only learns

avoiding incorrect actions. This results from the fact that the original approach has less freedom to explore the best action in such a state, as it is observed when the training has already progressed.

In addition to that, Figures 9.20, 9.21 and 9.22 showed a direct comparison of the development of achieved throughput. Based on the number of episodes, the adjusted reward machine approach is able to learn the final policy much faster. However, taking the number of filled replay memories or the number of observed actions into account, one can see that the original reward machine approach reaches a higher throughput for less replay memories and actions, respectively. Hence, it can be derived that the original approach is able to learn how to behave faster based on less data.

Learning approach without reward machine As the adjusted and the original reward machine approaches, the learning approach that is not using reward machines is also capable of learning how to solve a gantry robot scheduling task. Due to the restricted action space, the approach is able to achieve a throughput from the first episode on. Since this approach contains only experiences in the replay memory that have been truly observed, the shape of the replay memory differs from the reward machine based approaches in states and in frequency. The most important difference is visible by the frequency of the same environmental state with varying statuses of the gripper (state of the reward machine). In contrast to the reward machine approaches, the frequency of these states is independent since the CRM algorithm is not used.

Furthermore, this approach is able to learn which actions not to execute, although it never recognised any incorrect action. Hence, restricting the action space for the different states was successful albeit it requires implementation effort.

The validation runs gave the impression that the trained models fall short of the models trained with reward machines with regard to throughput. Therefore, the achieved policy cannot be optimal. However, this might result from differences in the implementation of terminal and restricted actions.

Comparison between approaches with and without reward machine All three different approaches were able to solve the gantry robot scheduling problem. Although different ways were used, they all learned to avoid incorrect actions. While the approach without reward machines only explores correct actions and hence never experiences an incorrect action choice, the approaches with reward machine do not use a restricted action space. Therefore, the reward machine approaches first explore all actions - correct and incorrect. This way, they learn which actions are incorrect before they start searching for the optimal action choice.

However, the learned policies differ in quality. Based on the validation runs, one can see that in general reward machine approaches were able to achieve policies with a higher average throughput than approaches without reward machines. It has to be further analysed whether this difference originates in different implementations or whether it

can be attributed to the learning approach itself. In addition to that, the adjusted reward machine approach achieved a higher average throughput than the original reward machine approach which can be explained by the different behaviour when machine failures occur. The adjusted reward machine approach was objectively able to deal with such machine failures more efficiently than the original approach.

Furthermore, the implementation of reward machines makes the structure of reward allocation more transparent and easier to understand. The visual representation makes it easier to understand from a human perspective which actions are incorrect and should be avoided. Moreover, the structured representation of the logic behind reward allocation also makes the implementation easier and less error-prone. In addition to that, the formation of reward machines allows for an automated and thus more easily scalable implementation. Since the approach without reward machine aims to translate the terminal states of the reward machine into a restriction rule for certain states, the structure of reward machines could be used to make this process of restriction even simpler and more structured.

However, the advantages of the reward machine approach result not only from its structure, ease of understanding and scalability, but also from its faster learning process. But it is rather difficult to make a statement about the speed of learning. The Figures 9.20, 9.21 and 9.22 demonstrated that each learning approach performed better than the other two on a particular comparison criterion. A comparison based on the number of episodes favored the adjusted reward machine approach over the other two. However, the deviation from the approach without reward machine is rather small compared to the original reward machine approach. On the one hand, the criterion of the number of episodes is suitable since it has an impact on the ε-value and as a consequence on the exploration-exploitation ratio. On the other hand, this criterion comes with some difficulties. For the adjusted reward machine approach and for the approach without reward machines, an episode has the same meaning. Both terminate an episode when the simulation duration had expired. However, despite the same simulation time different actions can be performed and hence, not the same amount of actions is executed within an episode. Furthermore, the original reward machine defines an episode also as end of the simulation time. Yet, this approach terminates an episode early when an incorrect action has been executed. As a consequence, neither simulation duration time nor the number of performed actions are comparable to the other two approaches.

The second criterion for comparison was the number of filled replay memories. For this criterion, the approach without reward machine is the first to achieve a high throughput. Since the reward machine based approaches generate additional synthetical experiences, the replay memory is filled significantly faster than in the approach without reward machine. In contrast to the previous criterion, the number of filled replay memories suffers from the fact that the evolution of the ε-value is not taken into account. Since the ε-value is updated after each episode and each episode corresponds to a certain number of executed actions, the exploration level increases faster with fewer executed actions. However, due to the CRM algorithm, the ratio of experiences per executed action is nine times higher in the reward machine based approaches. This explains why the approach

without reward machine needs the least amount of data to be successfully trained.

The third criterion uses the number of actual observed actions for the comparison. This aspect favors the two reward machine approaches, especially the original approach, over the approach without reward machine. This results from the additional synthetical generated experiences which are not counted as observed actions. However, this criterion does not reflect the data efficiency aspect and obscures how much data had to be collected.

Limitations The production environment used in the experiments can be considered quite complex, not only but also because of the stochastic aspects, e.g. machine failure. Nevertheless, the underlying gantry robot scheduling problem is also limited, since only two machines run in parallel in one work cell. Moreover, the produced elements require only one (and the same) processing step. Thus, no variance exists between the produced items. Interesting insights are expected when extending the existing production environment to a flexible job shop problem with more than one work cell. Beyond that, one cylce in the continuous problem requires twelve actions until the agent returns to one infinitely often visited reward machine state. This cycle span is rather small.

In addition to that, the gantry robot scheduling problem is only one specific application field. Therefore, the results are limited to this use case, although general statements can be derived from the experiments. Nevertheless, to confirm results with reference to the application of reward machines on continuous tasks in general, further experiments also from different tasks need to be conducted.

Besides the limitations arising from the underlying production environment, the DQN learning approach in general has some weaknesses. On the one hand, DQNs are able to solve some problems when applying neural networks to reinforcement learning problems, as mentioned in Chapter 5. However, other approaches, e.g. Double-DQN, have been developed which are able to cope with further challenges. Since all experiments used DQNs, the effect of the critics regarding this learning approach can be considered to have only minor effect on the comparability between the different models.

Furthermore, the models in the experiments section were trained based on recommendations for the original reward machine approach in terms of replay memory size and batch size. However, due to the different nature of the adjusted and the original approach, a parameter set with a smaller replay memory size and batch size could be beneficial for the adjusted approach.

11 Conclusions

An analysis of the original reward machine definition as introduced by Icarte et al. [17] revealed that it is not applicable for continuous tasks due to restrictions of the allowed input sequence for finite state machines. Therefore, adjustments in the definition of reward machines were proposed. However, these changes are only relevant for a sound definition and have little impact on the actual training. Furthermore, the original CRM algorithm was extended for use in continuous tasks. The experiments made it clear that this algorithm would work for continuous tasks even without the proposed modifications. Nevertheless, the adjustments in the CRM algorithm support its application for continuous reinforcement learning problems, as fewer episodes are needed for training and early occurrence of stochastic aspects can be facilitated. The experiments reveal that the adjusted reward machine approach is able to learn a better policy than the original reward machine approach as well as the approach without reward machines. The higher quality of the adjusted approach compared to the original approach results from the treatment of stochastic aspects such as machine failures. Besides, all three approaches were able to learn which actions should be avoided since they lead to an error. Furthermore, a comparison of all three approaches under different criteria showed that each approach has certain advantages. The adjusted reward machine approach needs fewer episodes than the other approaches to learn the correct behaviour. On the other hand, the original reward machine approach requires less observed actions for training. The approach without reward machine turns out to be the most data-efficient one, since it needs the least filled replay memories.

The aim of finding an optimal policy for a continuous gantry robot scheduling task faster than standard learning approaches was achieved by means of reward machines when considering speed as number of episodes (adjusted reward machine approach) or number of observed actions (original reward machine approach). However, the experiments demonstrated that the question regarding speed cannot be easily answered, since each approach favors for a certain aspect. In addition to that, both reward machine based approaches as well as the approach without reward machine were able to identify incorrect behaviour despite different strategies.

All in all, the results of this master thesis tie in with the concept of reward machines and extend it for continuous tasks not only from a theoretical point of view, but also from an algorithmic perspective.

Future Work In this master thesis, the applicability of reward machines to continuous problems was investigated using the example of gantry robot scheduling. General

implications and statements can be derived from the experiments. However, further fields of application and more complex tasks should be researched in order to verify these statements for continuous tasks in general.

Beyond that, since the underlying environment contained stochastic aspects, we got the impression that the original CRM algorithm has difficulties in learning such behaviour. Therefore, it should be analysed how the original reward machine approach works in reinforcement learning with stochastic aspects or less frequent but relevant observations. This investigation should be conducted not only for continuous tasks but also for episodic reinforcement learning problems.

Furthermore, the adjusted approach has been developed for application to continuous tasks. However, future work should validate its applicability to episodic tasks as well.

In the experiments, the models were trained with parameters recommended for the original reward machine approach. Therefore, a detailed analysis needs to be carried out to see if these parameters are the best choice for the adjusted approach. Additionally, the aspect of data efficiency should be further investigated.

In addition, the proposed adjustments related to the reward machine approach also concerned the definition of acceptance criteria for infinite input sequences for automata. Validation runs in the experiment section gave the impression that a policy is only solving the gantry robot scheduling task if the acceptance criteria is satisfied. Hence, the correlation between an optimal policy and the acceptance criteria needs to be analysed in more detail.

Appendix

Theory

Exponential Distribution

As explained in [38, pp.356] the exponential distribution is used to model idle times. As input only non-negative but continuous values are allowed. An important characteristic of this distribution is that it is memoryless. This means that the probability of waiting, for example, five minutes at t is the same as at $t+x$. In addition to that, λ describes the main parameter of this probability distribution. This parameter represents how often an event occurs during a given time. The mean value can be calculated as the inverse of λ and thus describes the mean idle time.

The exponential distribution holds the following density (11.0.1) and distribution function (11.0.2).

$$f(x) = \begin{cases} \lambda \exp(-\lambda x) & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{11.0.1}$$

$$F(x) = 1 - \exp(-\lambda x) \tag{11.0.2}$$

Erlang Distribution

As explained in [32, pp. 33] the Erlang distribution represents a convolution of multiple exponential distributions with the same parameter λ. This is formalised in Equation (11.0.3).

$$\begin{aligned} X_i &\sim exp(\lambda) \\ Y &\sim erl(\lambda, m) \\ Y &= X_1 + X_2 + \ldots + X_m \end{aligned} \tag{11.0.3}$$

Usually, an Erlang distribution is used to map e.g. the service time in queueing theory. This is similar to the repair time application used in this thesis. Due to the convolution of multiple exponential distributions the Erlang distribution describes a continuous but non-negative random variable. Furthermore, this probability distribution is defined by two parameters - λ and m. The parameter λ represents the parameter of the underlying exponential distribution. Moreover, the parameter m indicates how many random variables of the same exponential distribution are added. Hence, the expected value arises from m-times the expected value of the underlying exponential distribution.

Besides, the Erlang distribution holds the following density (11.0.4) and distribution function (11.0.5).

$$f(x) = \begin{cases} \frac{(\lambda-x)^{m-1}}{(m-1)!}\lambda \exp(-\lambda x) & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{11.0.4}$$

$$F(x) = 1 - \exp(-\lambda x) \sum_{i=0}^{m-1} \frac{(\lambda x)^i}{i!} \tag{11.0.5}$$

Smoothing

The training progress is smoothed by an exponentially weighted moving average with a smoothing factor α of 0.005. This means that all values are considered with different weights.

$$\begin{aligned} y_t =& \alpha \cdot x_t + (1-\alpha) \cdot y_{t-1} \\ y_0 =& x_0 \end{aligned} \tag{11.0.6}$$

This smoothing variant is provided by the *pandas* Python library.

Reward Machines in Detail

Running Example as Continuous Task

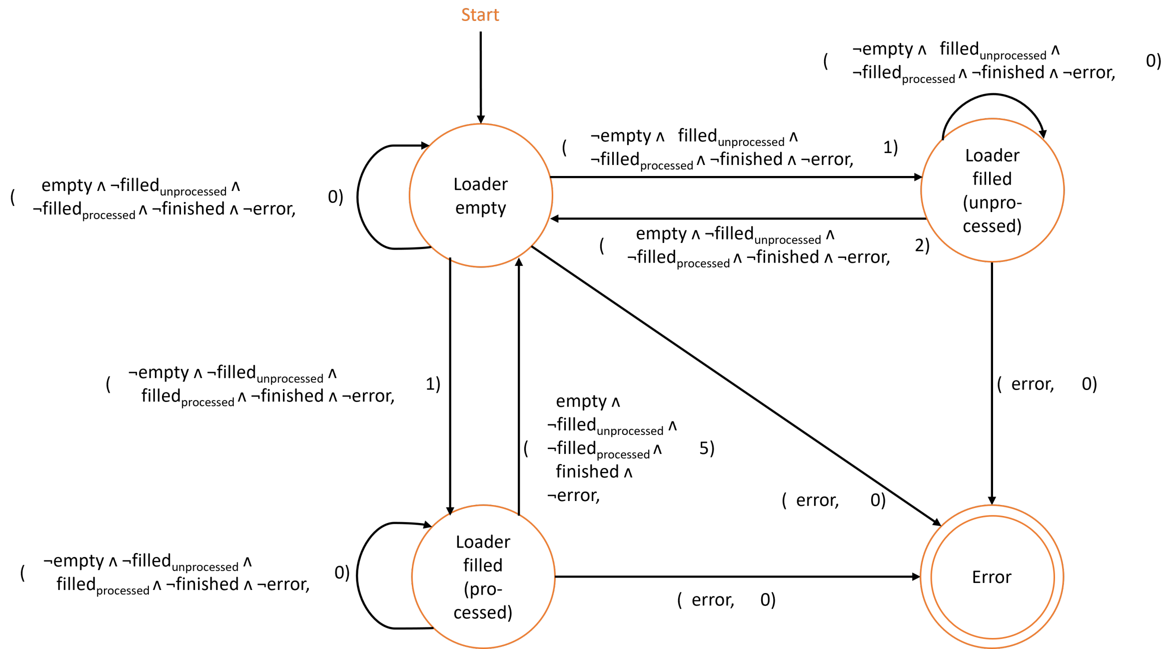


Figure 11.1: Adjusted design of the reward machine for a continuous task in detail

Running Example with adjusted Reward Machine according to proposed Approach

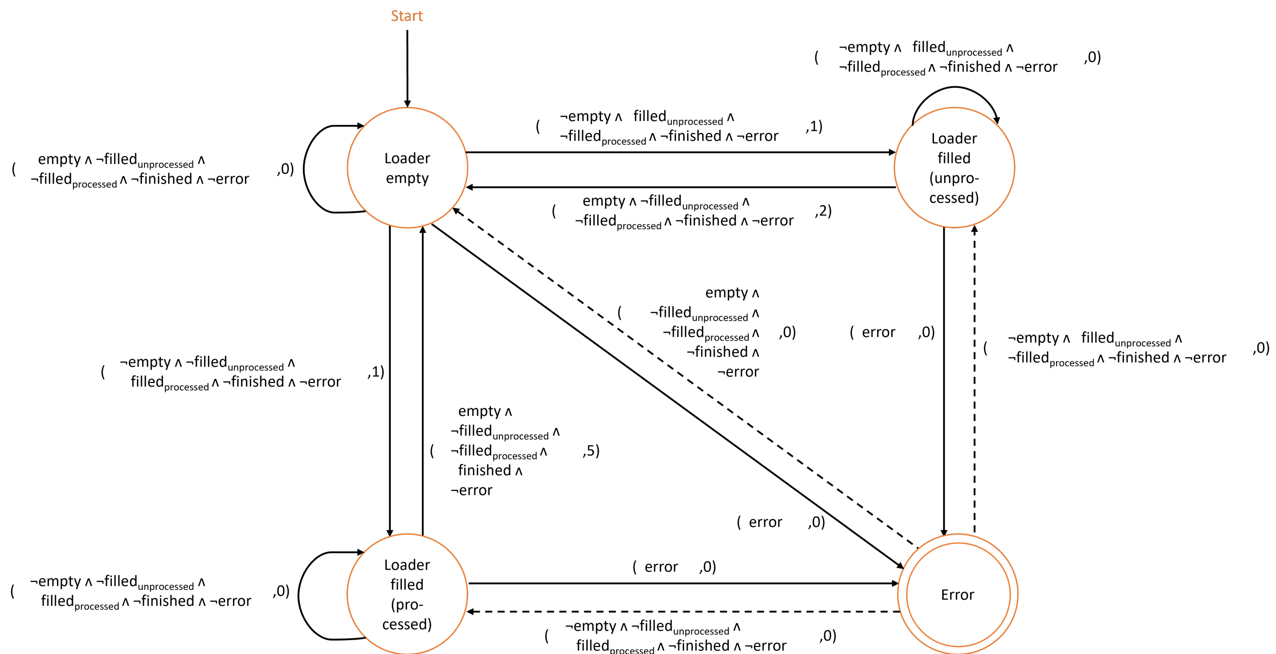


Figure 11.2: Adjusted reward machine for a continuous task in detail

Reward of Transitions in Reward Machines

Reward Machine of the Running Example

Current state of the reward machine (U)	**Logical formula (2^P)**	**Next state of the reward machine (δ_u & δ_f)**	**Reward (δ_r)**
Loader empty	$empty$	Loader empty	0
	$filled_{unprocessed}$	Loader filled (unprocessed)	1
	$filled_{processed}$	Loader filled (processed)	1
Loader filled (unprocessed)	$empty$	Loader empty	2
	$filled_{unprocessed}$	Loader filled (unprocessed)	0
Loader filled (processed)	$empty \wedge finished$	Loader empty	5
	$filled_{processed}$	Loader filled (processed)	0
Error	$empty$	Loader empty	0
	$filled_{unprocessed}$	Loader filled (unprocessed)	0
	$filled_{processed}$	Loader filled (processed)	0
any	$error$	Error	0

Table 11.1: State-transition function and state-reward function of the reward machine of the running example (simplified logical formula)

Reward Machine of the Experiment

Current state of the reward machine (U)	Logical formula (2^P)	Next state of the reward machine (δ_u & δ_f)	Reward (δ_r)
G1 & G2: empty	$empty(g1) \wedge empty(g2)$	G1 & G2: empty	0
	$empty(g2) \wedge filled_{unprocessed}(g2)$	G1: empty & G2: filled (unprocessed)	1
	$filled_{unprocessed}(g1) \wedge empty(g2)$	G1: filled (unprocessed) & G2: empty	1
	$empty(g1) \wedge filled_{processed}(g2)$	G1: empty & G2: filled (processed)	1
	$filled_{processed}(g1) \wedge empty(g2)$	G1: filled (processed) & G2: empty	1
G1: empty & G2: filled (unprocessed)	$empty(g1) \wedge empty(g2)$	G1 & G2: empty	2
	$filled_{processed}(g1) \wedge filled_{unprocessed}(g2)$	G1: filled (processed) & G2: filled (unprocessed)	1
	$filled_{unprocessed}(g1) \wedge filled_{unprocessed}(g2)$	G1 & G2: filled (unprocessed)	1
G1 & G2: filled (unprocessed)	$empty(g1) \wedge filled_{unprocessed}(g2)$	G1: empty & G2: filled (unprocessed)	2
	$filled_{unprocessed}(g1) \wedge filled_{unprocessed}(g2)$	G1 & G2: filled (unprocessed)	0
	$filled_{unprocessed}(g1) \wedge empty(g2)$	G1: filled (unprocessed) & G2: empty	2
G1: filled (unprocessed) & G2: empty	$filled_{unprocessed}(g1) \wedge filled_{unprocessed}(g2)$	G1 & G2: filled (unprocessed)	1
	$filled_{unprocessed}(g1) \wedge empty(g2)$	G1: filled (unprocessed) & G2: empty	0
	$empty(g1) \wedge empty(g2)$	G1 & G2: empty	2
G1: filled (unprocessed) & G2: filled (processed)	$filled_{unprocessed}(g1) \wedge empty(g2)$	G1: filled (unprocessed) & G2: empty	5

	$filled_{unprocessed}(g1) \wedge filled_{processed}(g2)$	G1: filled (unprocessed) & G2: filled (processed)	0
	$empty(g1) \wedge filled_{processed}(g2)$	G1: empty & G2: filled (processed)	2
G1 empty & G2: filled (processed)	$empty(g1) \wedge empty(g2)$	G1 & G2: empty	5
	$filled_{unprocessed}(g1) \wedge filled_{processed}(g2)$	G1: filled (unprocessed) & G2: filled (processed)	1
	$empty(g1) \wedge filled_{processed}(g2)$	G1: empty & G2: filled (processed)	0
	$filled_{processed}(g1) \wedge filled_{processed}(g2)$	G1 & G2: filled (processed)	1
G1 & G2: filled (processed)	$empty(g1) \wedge filled_{processed}(g2) \wedge finished(g2)$	G1: empty & G2: filled (processed)	5
	$filled_{processed}(g1) \wedge filled_{processed}(g2)$	G1 & G2: filled (processed)	0
	$filled_{processed}(g1) \wedge empty(g2) \wedge finished(g2)$	G1: filled (processed) & G2: empty	5
G1: filled (processed) & G2: empty	$empty(g1) \wedge finished(g1) \wedge empty(g2)$	G1 & G2: empty	5
	$filled_{processed}(g1) \wedge filled_{processed}(g2)$	G1 & G2: filled (processed)	1
	$filled_{processed}(g1) \wedge empty(g2)$	G1: filled (processed) & G2: empty	0
	$filled_{processed}(g1) \wedge filled_{unprocessed}(g2)$	G1: filled (processed) & G2: filled (unprocessed)	1
G1: filled (processed) & G2: filled (unprocessed)	$filled_{processed}(g1) \wedge filled_{unprocessed}(g2)$	G1: filled (processed) & G2: filled (unprocessed)	0
	$empty(g1) \wedge finished(g1) \wedge filled_{unprocessed}(g2)$	G1: empty & G2: filled (unprocessed)	1
	$filled_{processed}(g1) \wedge empty(g2)$	G1: filled (processed) & G2: empty	2

Error	$empty(g1) \wedge empty(g2)$	G1 & G2: empty	0
	$empty(g1) \wedge filled_{unprocessed}(g2)$	G1: empty & G2: filled (unprocessed)	0
	$filled_{unprocessed}(g1) \wedge filled_{unprocessed}(g2)$	G1 & G2: filled (unprocessed)	0
	$filled_{unprocessed}(g1) \wedge empty(g2)$	G1: filled (unprocessed) & G2: empty	0
	$filled_{unprocessed}(g1) \wedge filled_{processed}(g2)$	G1: filled (unprocessed) & G2: filled (processed)	0
	$empty(g1) \wedge filled_{processed}(g2)$	G1: empty & G2: filled (processed)	0
	$filled_{processed}(g1) \wedge filled_{processed}(g2)$	G1 & G2: filled (processed)	0
	$filled_{processed}(g1) \wedge empty(g2)$	G1: filled (processed) & G2: empty	0
	$filled_{processed}(g1) \wedge filled_{unprocessed}(g2)$	G1: filled (processed) & G2: filled (unprocessed)	0
any	*error*	Error	0

Table 11.2: State-transition function and state-reward function of the reward machine of the experiment (simplified logical formula)

Program Code

The source code containing the DQN and reward machine implementation is available at `https://code.fbi.h-da.de/hda10404/kispo.git` in the branch *Code_Masterarbeit*. This code is part of the KISPo repository, which contains all implementations of this research project. Access to this repository must be granted by the owner Robert Miltenberger.

List of Figures

1.1 Example of a gantry robot in a production context for material transportation. 2

5.1 The agent-environment interface [39] 14
5.2 Training of a DQN-agent 20
5.3 DQN-agent-environment interaction 21
5.4 Q-network for the running example with S as *input conveyor* 24

6.1 Compression of the environment to the state *Loader empty* of the reward machine of the running example 28
6.2 The agent-environment-reward machine interface 29
6.3 Reward machine of the running example 31
6.4 Simplified visualisation of the reward machine of the running example . 32
6.5 DQN-agent-environment interaction combined with the CRM approach 34

8.1 Adjusted design of the reward machine for a continuous task 49
8.2 Adjusted reward machine for a continuous task 57

9.1 DQN architecture of models with and without reward machines 60
9.2 Ratio of greedy chosen actions with an ε-decay of 0.9995 (ε-greedy approach) 62
9.3 Comparison of I-loader and H-loader 63
9.4 Distributions of incoming elements and machine failures (histograms) . 67
9.5 Reward machine with loader status for each individual gripper, G1 indicates gripper 1 and G2 specifies gripper 2 69
9.6 Reward machine with loader status for each individual gripper, left part of Figure 9.5 70
9.7 Reward machine with loader status for each individual gripper, right part of Figure 9.5 71
9.8 Development of the ratio of random/greedy chosen actions according to the ε-greedy approach 76
9.9 Development of throughput per hour during training 78
9.10 Development of the Q-values for the initial state and all actions 82
9.11 Development of the Q-values for the state where one gripper is empty, located at the input conveyor and could theoretically be loaded 83

9.12 Development of the Q-values for the state where station A failed, station B is processing, one gripper is empty and the loader is located at the input conveyor . . . 84
9.13 Number of executed actions when the replay memory is only filled randomly 87
9.14 Training progress of the model trained with the original reward machine algorithm . . . 88
9.15 Simulation duration for the original CRM algorithm for each episode . . 89
9.16 Training progress of the model trained with the original reward machine algorithm, training parameters from model *adjustedRM* . . . 92
9.17 Development of the observed throughput during training (approach without reward machines) . . . 96
9.18 Training progress of model trained by the approach without reward machine, training parameters from adjusted reward machine approach . . . 99
9.19 Training progress of both models trained by the approach without reward machine . . . 100
9.20 Development of throughput across the number of training episodes . . . 103
9.21 Development of throughput across the number of filled replay memories (observed and counterfactual experiences) . . . 104
9.22 Development of throughput across the number of executed actions (without counterfactual experiences) . . . 105

11.1 Adjusted design of the reward machine for a continuous task in detail . 118
11.2 Adjusted reward machine for a continuous task in detail . . . 119

List of Tables

9.1 Ride time between different positions in seconds. 64
9.2 Overview of the information considered in a state, using the example of the initial state 64
9.3 Conditions or subtasks that need to be fulfilled to gain positive rewards . 65
9.4 Training parameters for the adjusted reward machine approach 77
9.5 Overview of the five most frequent states of the environment in the first full replay memory (trained by the adjusted reward machine approach) . 80
9.6 Overview of the five most frequent states of the environment in the replay memory at the end of the training (trained by the adjusted reward machine approach) 81
9.7 Training parameters for the original reward machine approach 86
9.8 Overview of the five most frequent states of the environment in the replay memory at the beginning of the training (trained by the original reward machine approach) 89
9.9 Overview of the five most frequent states of the environment in the replay memory at the end of the training (trained by the original reward machine approach) 90
9.10 Overview of the five most frequent states of the environment in the replay memory at the end of the training (trained by the original reward machine approach with training parameters from model *adjustedRM*) 93
9.11 Training parameters for a standard learning approach without reward machine 95
9.12 Overview of the five most frequent states of the environment in the first full replay memory (trained by the approach without reward machine) . 97
9.13 Overview of the five most frequent states of the environment in the replay memory at the end of training (trained by the approach without reward machine) 97
9.14 Overview of the five most frequent states of the environment in the replay memory at the end of training (trained by the approach without reward machine with parameters from model *adjustedRM*) 101

11.1 State-transition function and state-reward function of the reward machine of the running example (simplified logical formula) 120

11.2 State-transition function and state-reward function of the reward machine of the experiment (simplified logical formula) 123

Bibliography

[1] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

[2] Kfir Arviv, Helman Stern, and Yael Edan. Flow-shop problem with job transfer robots using reinforcement learning. In *annual meeting ICPR20 International Conference on Production Research, Shanghai, China, August*, pages 2–6, 2009.

[3] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Springer, 2006.

[4] Wilfried Brauer and Gerhard Weiß. Multi-machine scheduling-a multi-agent learning approach. In *Proceedings International Conference on Multi Agent Systems (Cat. No. 98EX160)*, pages 42–48. IEEE, 1998.

[5] Ci Chen, Beixin Xia, Bing-hai Zhou, and Lifeng Xi. A reinforcement learning based approach for a multiple-load carrier scheduling problem. *Journal of Intelligent Manufacturing*, 26(6):1233–1245, 2015.

[6] Rina S. Cohen and Arie Y. Gold. Theory of ω-languages: Characterizations of ω-context-free languages. *Journal of Computer and System Sciences*, 15(2):169–184, 1977.

[7] Ke-Lin Du and Madisetti NS Swamy. *Neural networks and statistical learning*. Springer Science & Business Media, 2013.

[8] Berndt Farwer. ω-automata. In *Automata logics, and infinite games*, pages 3–21. Springer, 2002.

[9] Niclas Feldkamp, Soeren Bergmann, and Steffen Strassburger. Simulation-based deep reinforcement learning for modular production systems. In *2020 Winter Simulation Conference (WSC)*, pages 1596–1607. IEEE, 2020.

[10] Thomas Gabel and Martin Riedmiller. Adaptive reactive job-shop scheduling with reinforcement learning agents. *International Journal of Information Technology and Intelligent Computing*, 24(4):14–18, 2008.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] Jens Heger and Thomas Voß. Dynamic priority based dispatching of agvs in flexible job shops. *Procedia CIRP*, 79:445–449, 2019. 12th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 18-20 July 2018, Gulf of Naples, Italy.

[13] Jens Heger and Thomas Voss. Optimal scheduling of agvs in a reentrant blocking job-shop. *Procedia Cirp*, 67:41–45, 2018.

[14] Martin Hofmann and Martin Lange. *Automatentheorie und Logik*. Springer Berlin, Heidelberg, 1st ed. 2011 edition, 2011.

[15] John E Hopcroft, Jeffrey D Ullman, and Rajeev Motwani. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*, volume 2. Pearson Studium Deutschland, München, 2002.

[16] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[17] Rodrigo Toro Icarte, Toryn Q Klassen, Richard Valenzano, and Sheila A McIlraith. Reward machines: Exploiting reward function structure in reinforcement learning. *Journal of Artificial Intelligence Research*, 73:173–208, 2022.

[18] Chen Lei. Deep reinforcement learning. In *Deep Learning and Practice with MindSpore*, pages 217–243. Springer, 2021.

[19] Yuxi Li. Deep reinforcement learning. *CoRR*, abs/1810.06339, 2018.

[20] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321, 1992.

[21] Bart L Maccarthy and Jiyin Liu. Addressing the gap in scheduling research: a review of optimization and heuristic methods in production scheduling. *The International Journal of Production Research*, 31(1):59–79, 1993.

[22] Sridhar Mahadevan, Nicholas Marchalleck, Tapas K Das, and Abhijit Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *Proc. of 14th international conference on machine learning*, pages 202–210, 1997.

[23] Sridhar Mahadevan and Georgios Theocharous. Optimizing production manufacturing using reinforcement learning. In *FLAIRS conference*, volume 372, page 377, 1998.

[24] VV Makarov, Ye B Frolov, IS Parshina, and MV Ushakova. The design concept of digital twin. In *2019 Twelfth International Conference "Management of large-scale system development"(MLSD)*, pages 1–4. IEEE, 2019.

[25] Andreja Malus, Dominik Kozjek, et al. Real-time order dispatching for a fleet of autonomous mobile robots using multi-agent reinforcement learning. *CIRP annals*, 69(1):397–400, 2020.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[27] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287, 1999.

[28] Xinyan Ou, Qing Chang, Jorge Arinez, and Jing Zou. Gantry work cell scheduling through reinforcement learning with knowledge-guided reward setting. *IEEE Access*, 6, 2018.

[29] Xinyan Ou, Qing Chang, and Nilanjan Chakraborty. Simulation study on reward function of reinforcement learning in gantry work cell scheduling. *Journal of manufacturing systems*, 50:1–8, 2019.

[30] Xinyan Ou, Qing Chang, and Nilanjan Chakraborty. A method integrating q-learning with approximate dynamic programming for gantry work cell scheduling. *IEEE Transactions on Automation Science and Engineering*, 18(1):85–93, 2020.

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

[32] HT Papadopolous, Cathal Heavey, and Jimmie Browne. *Queueing theory in manufacturing systems analysis and design*. Springer Science & Business Media, 1993.

[33] Michael L Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 2016.

[34] Jens Popper, Vassilios Yfantis, and Martin Ruskowski. Simultaneous production and agv scheduling using multi-agent deep reinforcement learning. *Procedia CIRP*, 104:1523–1528, 2021.

[35] Andreas Pott and Thomas Dietz. *Industrielle Robotersysteme*. Springer, 2019.

[36] Simone Riedmiller and Martin Riedmiller. A neural reinforcement learning approach to learn local dispatching policies in production scheduling. In *IJCAI*, volume 2, pages 764–771. Citeseer, 1999.

[37] H.J. Siegert and S. Bocionek. *Robotik: Programmierung intelligenter Roboter: Programmierung intelligenter Roboter*. Springer-Lehrbuch. Springer Berlin Heidelberg, 2013.

[38] Toni C Stocker and Ingo Steinke. *Statistik: Grundlagen und Methodik*. Walter de Gruyter GmbH & Co KG, 2016.

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

[40] Wolfgang Thomas. Automata on infinite objects. In *Formal Models and Semantics*, pages 133–191. Elsevier, 1990.

[41] Gottfried Vossen and Kurt-Ulrich Witt. *Grundkurs Theoretische Informatik*. Springer, 2016.

[42] Yi-Chi Wang and John M Usher. Application of reinforcement learning for agent-based production scheduling. *Engineering Applications of Artificial Intelligence*, 18(1):73–82, 2005.

[43] 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.

[44] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[45] Phil Winder. *Reinforcement learning*. O'Reilly Media, 2020.

[46] Tianfang Xue, Peng Zeng, and Haibin Yu. A reinforcement learning method for multi-agv scheduling in manufacturing. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1557–1561. IEEE, 2018.

[47] Wei Zhang and Thomas G Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI*, volume 95, pages 1114–1120. Citeseer, 1995.