Reinforcement Learning

Load balancing using reinforcement learning in a pick and place application.

AALBORG UNIVERSITY
STUDENT REPORT

Anders Flodgaard
Carsten Larsen
Laura Montesdeoca Fenoy
Nicklas Krogh Andersen

Department of Electronics
Aalborg University
Denmark
May 30, 2018

AALBORG UNIVERSITY
STUDENT REPORT

Third year of study
Robotics
Fredrik bajers vej 7
DK-9220 Aalborg
`www.robotics.aau.dk/`

Title:
Reinforcement Learning

Theme:
Load balancing using reinforcement learning in a pick and place application.

Semester:
6th semester of robotics.

Project Period:
Spring 2018

Project Group:
Group 669

Group Members:
Anders Flodgaard
Carsten Larsen
Laura Montesdeoca Fenoy
Nicklas Krogh Andersen

Supervisor:
Rasmus Skovgaard Andersen
rsa@mp.aau.dk

Number of Pages:
59 pages

Submitted:
May 30, 2018

Abstract:

This project investigates how a robotic solution can be implemented in a pick and place application to load balance a system with a reinforcement learning agent. The task was implemented in a simulated environment using the program V-rep. This program is connected to Matlab, which is in charge of the calculations for the decision making in the reinforcement learning. In order to show how the system performs, tests were conducted using two simulations. Once accomplished the task in a simple way, and another utilizes reinforcement learning. This was done to have a baseline, which could be set to compare and contrast the differences between a normal production line and an intelligent system.
This project explores the viability of such system, concluding that while it is possible to load balance a system using reinforcement learning, to do this certain aspects of the production were sacrificed to evenly distribute the workload.

AALBORG UNIVERSITY
STUDENT REPORT

Tredje studieår
Robotics
Fredrik bajers vej 7
DK-9220 Aalborg
`www.robotics.aau.dk/`

Titel:
Reinforcement læring.

Tema:
Arbejdsbalancering ved brug af reinforcement læringen i en pick and place applikation.

Semester:
6. semester robotics.

Projektperiode:
Forår 2018

Projektgruppe:
Gruppe 669

Deltagere:
Anders Flodgaard
Carsten Larsen
Laura Montesdeoca Fenoy
Nicklas Krogh Andersen

Vejledere:
Rasmus Skovgaard Andersen
rsa@mp.aau.dk

Sidetal:
59 sider

Afleveringsdato:
May 30, 2018

Abstract:

Dette projekt undersøger, hvordan en robotopløsning kan implementeres i en pick and place applikation og arbejdsbalanceres ved brug af en reinforcement læringen agent. Opgaven blev implementeret i et simuleret miljø ved hjælp af programmet V-rep. Dette program er knyttet til Matlab, der er ansvarlig for beregningerne for beslutningsprocessen inden for reinforcement læringen. For at vise, hvordan systemet fungere, blev der udført tests, ved hjælp af to simuleringer. Den ene udførte opgaven på en enkel måde, og den anden bruger reinforcement læring. Dette var gjort for at have en basislinje, som kunne bruges til at sammenligne forskellene mellem en normal produktionslinje og et intelligent system.
Dette projekt undersøger virkeliggørelsen af et sådant system, hvor det konkluderes, at selvom det er muligt at arbejdsbalancere et system ved hjælp af reinforcement læring, så vil bestemte aspekter af produktionen blive ofret for at kunne fordele arbejdsbelastningen lige.

Preface

Aalborg University, May 30, 2018

This report is written by four, 6th semester Robotics students from Aalborg University, Denmark.

The report was written over a period of 118 days. The purpose of the project was to create a load-balancing system that was able to learn and improve by itself, based on experience. This was done by use of reinforcement learning. The solution was done in a simulation, the project therefore includes a simulated prototype and test of the solution.

Throughout the paper the authors have used the UNSRT method for referencing e.g.: [1], [2].

[illegible]
Anders Flodgaard

[illegible]
Carsten Larsen

[illegible]
Laura Montesdeoca Fenoy

[illegible]
Nicklas Krogh Andersen

Abbreviations

Acronyms	Description
AAU	Aalborg University
CPS	Cyber Physical System
PaT	Position and Time
VGR	Vision Guided Robotics
IT	Information technology
PLC	Programmable Logic Controller
CPS	Cyber Physical Systems

Contents

Chapter | 1

Introduction

1.1 Introduction

The increasing expansion of industries in nowadays society, means humans are no longer able to keep up with the demands of the market. Therefore, there is a need of new methods capable of being adaptive and flexible while maintaining high productivity and quality.

One of the fields playing a major role helping manufacturers with agile production, is robotics. Since the 1970s when computers and automation were introduced to the industries, robots have experienced great advancements. Initially they were mostly used for repetitive and precise tasks, since they can perform faster, more accurately and over long periods of time. However, as the industry evolves, the incorporation of machines lies on their capability to learn new tasks, which can be a complex, specialized and time consuming process.

Robots provide a wide range of services. However, one of the most predominant applications of robotics is pick and place. The reason for this arises due to the need of reducing investment while increasing productivity. Pick and place robots improve production speed while decreasing downtime, since they can be customized to fit specific production requirements.

In order to have a flexible application, robots can use reinforcement learning. Which is an area of machine learning that focuses on solving a task without explicitly being told how to. Instead, it has to learn how to solve the task at hand on its own, based on the system's interactions with the environment. This project will explore the challenges involved when implementing reinforcement learning to a production line dealing with pick and place.

1.2 Task

The purpose of this project is to use reinforcement learning to aid a robotic system distributing the load equally among several manipulators. The manipulators task is to sort objects appearing at one conveyor onto another. An example of this can be seen in Figure 1.1, where three manipulators sort objects from one location to another. Note that Figure 1.1 is only an example of a setup.

This project was proposed as a research task, therefore restrictions were not specified in the task. This means that every parameter can be a variable. As long as the solution succeeds to implement reinforcement learning. However, the setup should be as realistic as possible and plausible to implement in the real world if it was chosen to do so. Thus, to reduce the level of complexity certain restrictions were set, which will be further described throughout the report.

With that said, the goal is to have a realistic setup that uses reinforcement learning in a pick and place application. Which purpose is to optimize and balance the work load equally between the implemented manipulators. Resulting in them having the same wear and tear time. This means that the system should be able to compensate for variations on different parameters and give the most flexible solution possible.

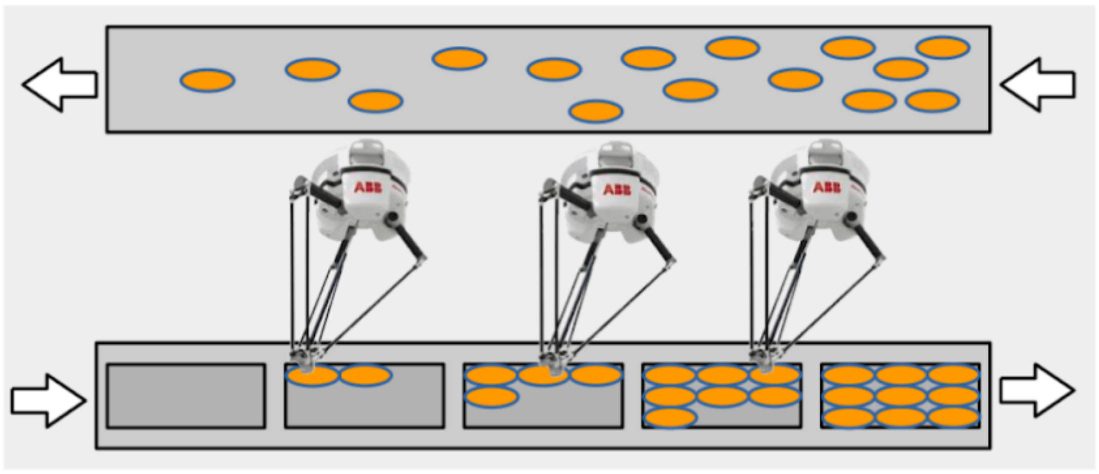


Figure 1.1: Example of a possible setup. Where three manipulators sort objects, distributing the load equally. [1]

1.3 Initial Problem Statement

How can machine learning be used to optimize a robotic solution in modern industries.

Chapter | 2

Problem Analysis

2.1 Industry 4.0.

This project is oriented towards a manufacturing application, therefore it should comply with the current demands industries are facing. This section provides a quick overview of the advances in the industrial environment, as well as an analysis of the current state of manufacturing with Industry 4.0 This section is based on the article "Recommendations for implementing the strategic initiative industrie 4.0" [2]

Throughout the last couple of centuries industrial manufacturing has been wildly implemented in many countries all over the world. This means that the production line and how the products are produced, has changed and/or improved to maximize productivity while reducing waste. These changes in the production system have resulted in four different industrial revolutions. The first one dating back to the late 18th century, where power generation through mechanical automation was introduced, such as the steam engine. About 100 years later the second revolution began, where electricity was used to gain mobility in the production lines. The third revolution started in the early 1970s with the introduction of IT, PLC's, computers and digitalization in general. The present revolution is the fourth industrial revolution, which is an expansion of the third. The fourth industrial revolution is still in its early stages. This revolution is identified by combining different fields of science, such as biotechnology, nanotechnology, robotics, The Internet of Things and Services, etc. In short, embed technology within different aspects of societies.

Since 2006, Germany has been pursuing the Internet of Things and Services under their High-Tech Strategy 2020 action plan. This plan was formulated to ensure that Germany would have a strong competitive position in the manufacturing industry. In 2011 they took the fourth revolution and made it into a concept of their own called "industrie 4.0", which is a name that has been extended to the rest of the world, and is now used as a term to denominate the fourth revolution.

The purpose of this revolution is to connect everything to the internet, making it able to communicate. Thus combining the cyberspace with the physical world creating the Cyber Physical Systems (CPS). With CPS, different information and communication technology are used so manufacturers can easily share information, resources and establish global network. Enabling them to meet the costumers demands. See what Industry 4.0 contains in Figure 2.1.

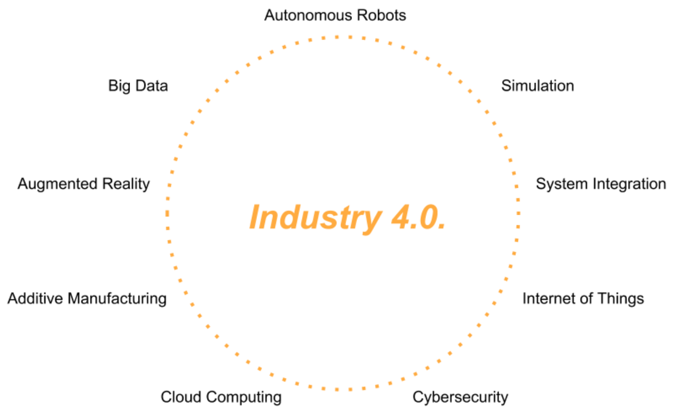


Figure 2.1: The different features of industry 4.0.

In this project Industry 4.0. is desired to be implemented as a proof of concept of a "smart factory". Where reinforcement learning will provide the system with decentralized decision-making, which is the ability to make simple decisions on its own and become as autonomous as possible. In the case of reinforcement learning, the whole system will learn on its own, how to accomplish the task. This is achieved through interoperability, which means that machines, devices and sensors communicate with each other, in order to accomplish the pick and place application.

2.2 Pick and Place

When analyzing the manufacturing work flow of a product, there are several distinguishable stations which it must go through. Each one of them related to a different process, but all of them are relevant for the end product. One of these stations is the pick and place, where, as the name suggests, products are relocated from one place to another. Generally from a conveyor belt to either the next stage of the production line or to the final packaging.

Pick and place applications are easily programmable, as well as one of the easiest ways to add automation into a production line, since they run autonomously with quick precision handling.

For this project, one of the main focuses is to improve the production line of a pick and place application, by using reinforcement learning to control the system.

Thus balancing the load of the manipulators, without overworking them, as well as trying to maintain a constant speed on the conveyor belts, to minimize waste of energy resources.

Most pick and place applications relay on Vision Guided Robotics (VGR), which means that one or more manipulators are connected to a vision system that enables them to locate objects and therefore by guided to a desired position. For the execution of VGR, there are several methods or machine vision tools that can be used. One of the most common is blob analysis/connectivity. Where the aim is to isolate pixels which are connected, also known as blobs, using a binary image. In this project, a pick a place application with a VGR system will be implemented in a simulation. This will be used as a baseline to test the reinforcement leaning for load balancing.[3]

2.3 Load Balancing

In a production line with a pick and place application, manipulators usually pick up objects as they come. So unless there is a specific algorithm they follow, the workload is distributed randomly. Usually the manipulator at the beginning of the line picks up more objects and therefore works at a higher intensity than those further down the production line. Because they have less objects to pick, since they already have been picked by other manipulators. To prevent this from happening, load balancing will be implemented.

Load balancing refers to the process of effectively distributing the work load among the different components. In this project, the work load to distribute are the incoming objects and the components to execute the work, are multiple manipulators, in charge of pick and placing. In order to improve the system, the workload must be evenly distributed among the manipulators. Load balancing will prevent a specific manipulator from overworking, which can potentially lead to a quicker breakdown for that specific manipulator. As well as preventing other manipulators from not being used, to their full potential. This process aims to maximize the throughput, which is the number of processes completed per unit time, as well as equitably distribute the work load. [4]

In order to have an appropriate load balancing, the system should not only keep track of the number of objects picked, but also have a strategy to determine which object should be picked by which manipulator. As well as in which position the object should be placed in order to increase the overall productivity of the system. [5]

The load balancing depends mostly on two factors, the amount of objects and the amount of manipulators available. Suppose a case with a fixed amount of manipulators, where there is an overflow of objects. In this particular scenario, load balancing will not contribute to the system. Since there is too many objects to pick up, all manipulators will be working at 100 % regardless, and objects will still be leaving the workspace before they can be picked. If an extra manipulator is added, so there no longer is an overflow of objects. In this case, load balancing will distribute the objects evenly among all manipulators. From this, it can be assumed that load balancing should be implemented when there is slightly more available resources than objects.

The solution consists of a pick and place application which should implement load balancing. The next step is to explore a method which works in accordance with the ideals of Industry 4.0. Therefore, next section explores the possibilities of machine learning.

2.4 Machine Learning

Machine learning is a combination of algorithms, which gives a computer system the ability to learn. This is done by providing the computer system with training data and letting it make educated guesses for a correct action, without being explicitly programmed what to do. Machine learning is typically classified into tree categories: 'Supervised learning', 'Unsupervised learning' and 'Reinforcement learning'. This chapter is based on the book: "Reinforcement Learning: An Introduction" [6].

Supervised learning is learning from training data. This is used to categorize where the object of interest belongs to. Each example in the training data is a description of an object with the correct action to take. This is done so the program can extract a generalized answer to a problem or situation, that is not in the training data. To be used when, *"I know how to classify this data, I just need you (the classifier/program) to sort it."[7]*

Unsupervised learning, is about finding structure in a collection of data, without the need for training data. Then the algorithm classifies the input, without explicit told how to do it. To be used when, *"I have no idea how to classify this data, can you (the algorithm) create a classifier for me?"[7]*

Reinforcement Learning, is learning by 'trial and error' and 'rewards'. The goal for the algorithm is to optimize a numerical policy signal. The problem is to figure out what action to do, when not told, for either a immediate or long term reward. A learning agent (A part of the program that decides what action(s) to take and when) must be able to sense, react and manipulate the state (The state is a signal describing the current position of the agent in the environment.) of it's surroundings, in order to move towards a goal(s). To be used when, *"I have no idea* *how to classify this data, can you classify this data and I'll give you a reward if its correct or I'll punish you if it's not."[7]*

The agent needs a good balancing between exploration and exploitation. Where the first one means to attempt a new action, that might give a better reward. Whereas in exploitation the agent chooses a known action which gives the best reward. The agent will prefer to take actions it knows from the past ('exploit') that are considered a success but it has to try new actions, in order to look for better solutions ('explore').

Reinforcement learning can consist of: a policy, a reward signal, a value function, and a model of the environment.

- A policy defines the behavior of the agent at any given time. It is mapping the state to an action, for example a look up table. With experience the policy can be changed in order to maximize the immediate or long term reward.
- A reward signal is the goal for the agent, for every action taken the environment sends the agent reward. This means that the action can be either good or bad depending on the reward.
- A value function, is a long term goal for the agent, it is the expected or total reward value over time, starting from the beginning of the state.
- The model of the environment, is used for planning the action needed, by predicting the future state, action and rewards. The state is an input to the policy or value function. As well as an input or output to the environment model.

These are the core ideas of reinforcement learning. The approximation of the value functions from state and action, to find a optimal value and policy function. This is used to describe solution methods for the reinforcement learning problem. The important part of reinforcement learning, is the ability to evaluate on the action taken, and not to instruct the agents with the right action. This is done by exploration, to search for a good action. Depending on the action taken, compared to a total reward expected over an unspecified amount of actions or time spend, an average action value can be calculated as seen in Equation 2.1.

$$Q_t(a) \doteq \frac{sum\ of\ rewards\ when\ action\ taken\ prior\ to\ time}{number\ of\ times\ an\ action\ is\ taken\ prior\ to\ time} = \frac{\sum R_i * I_{A_i=a}}{\sum I_{A_i=a}} \tag{2.1}$$

Where $Q_t(a)$, is the estimated action value, R is the reward, I_A is a variable, true(1) or false(0). For $denominator = 0$ then $Q_t(a)$ is a default value, but for $denominator \to infinity$ then $Q_t(a) \to q_*(a)$, by the law of large numbers. To save on computational power the estimated action values can be calculated, using an old estimate to calculate a new estimate, as seen in Equation 2.2:

$$Q_{n+1} \doteq Q_n + \alpha \left[R_n - Q_n\right] \tag{2.2}$$

Q_{n+1} is the next estimated action value, The α is the step size value. The $\left[R_n - Q_n\right]$ is an error difference. The general form of the Equation 2.2 is:

$$NewEstimate \leftarrow OldEstimate + StepSize \left[Target - OldEstimate\right] \tag{2.3}$$

In case of multiple actions with the same highest reward, the simplest rule is to select between them by random:

$$A_t \doteq argmaxQ_t(a) \tag{2.4}$$

This policy uses the current known best action, also known as a Greedy policy, see Equation 2.4. This policy dose not explorer other option, even though they might be better. It goes step by step and at every step, it chooses the next step, that looks best at the moment, offers the most obvious and immediate reward. To explore other options a small probability "E" can be used, also known as $\varepsilon - greedy$ policy. In a test between the greedy and $\varepsilon - greedy$, the greedy policy would result in better rewards early on, but later on the $\varepsilon - greedy$ would explore better or equally good options, meaning in the long run it would give equally or better rewards.

An example of this can be seen in Figure 2.2. The greedy algorithm will chose the patch which gives the highest value step by step. Meaning it will take the red path. The $\varepsilon - greedy$ algorithm on the other hand forces it to explore other paths. Meaning it will explore every possible path over time and thereby find the green path.

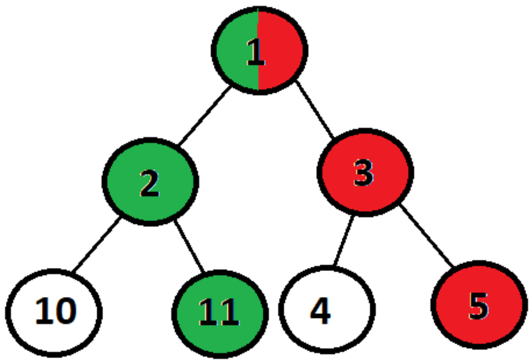


Figure 2.2: Greedy (red) vs $\varepsilon - greedy$ (green)

One way of solving the reinforcement problem is by using Q-learning, as seen in Equation 2.5. Q-learning is a algorithm that aims to reach the state with the highest reward possible, when or if the agent reaches this state, it will try to remain there.

$$Q_{new} = (1 - LR) * Q_{old} + LR * (Reward + discount * max(Q) - Q_{old} + bonus); \quad (2.5)$$

It does this by storing the expected reward for each action in every possible state. It then forms its path by executing those actions, with the highest expected reward. An example of this could be a maze. See Figure 2.3. In this example, the agent has to go from one side of a maze to the other side, indicated by arrows. It does not know the environment and have to learn through experience by taking any route and see if it leads though or not. The quicker it gets though, the better the reward. Each time it hits a dead end, it will get punished. After having attempted several routes, it should know the fastest way out and would chose this route every time, since the reward is highest here.

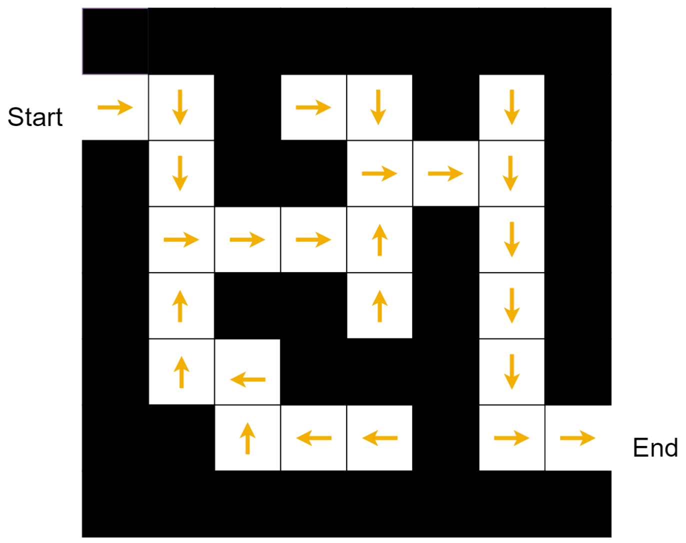


Figure 2.3: Maze for reinforcement learning

So far this section has covered the basics of reinforcement learning. A description of other methods can be found in the appendix 7.1.

This section went through the basic and more advanced methods in reinforcement learning, and in the appendix it can be seen what disadvantages and advantages each method has. This will then be used later in the development chapter to make a solution for the problem stated in the section 1.2. It will be used for the analysis, where the different variables that can be changed or are of interest in the the reinforcement problem state(s).

2.5 Setup

The purpose of this project, is to investigate how reinforcement learning can be used to load balance a system. This project was presented as a research opportunity, therefore, no specifications were given. In order to simplify the problem, certain parameters that compose the system, will remain as constant conditions, while the main project will focus on certain variable parameters and conditions. This approach was chosen due to time and resources limitations, since having a dynamic setup where everything is changeable would be unmanageable.

Furthermore some of the parameters are not sensible to change, such as the amount and placement of the conveyor belts. As seen in Figure 2.4, having more than two rows of conveyors, results in a less efficient setup. Since the manipulators' workspace is increased and they have to reach over other conveyors to pick and/or place the objects. Meaning they would have to cover a larger distance, and therefore spend more time palletizing. A similar issue occurs if the manipulators are not placed in between the conveyors as shown in Figure 2.5.

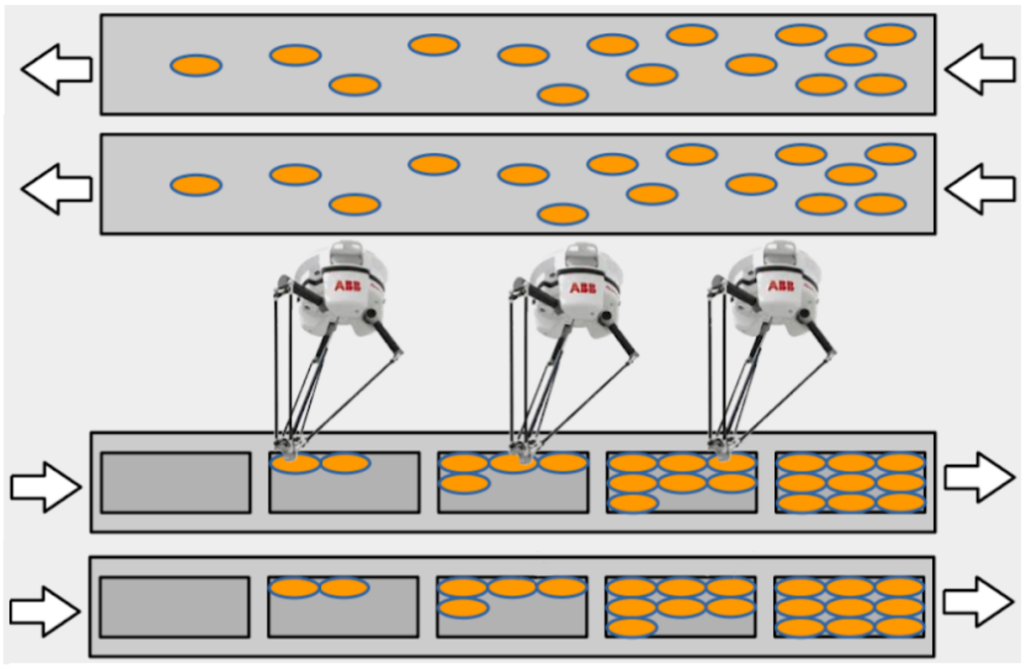


Figure 2.4: An example of a setup using multiple conveyors next to each other. [1]

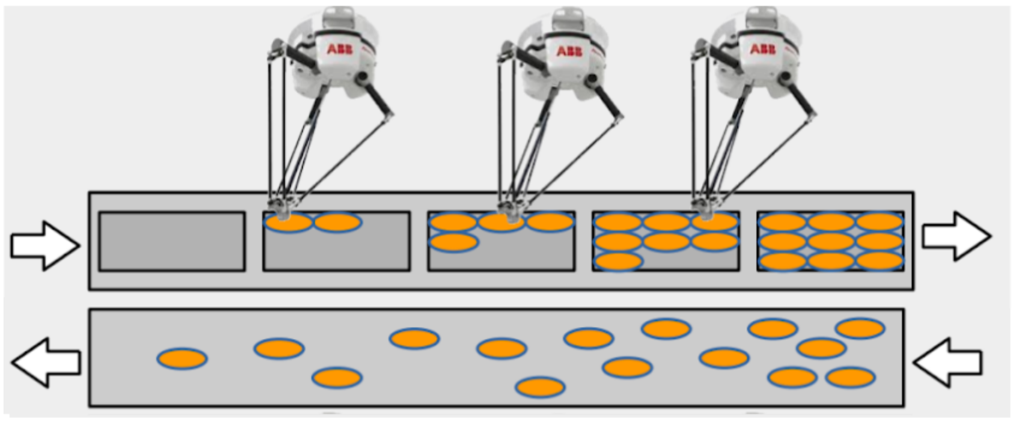


Figure 2.5: An example of a setup illustration the importance of the conveyor placement. [1]

There are a lot of possible adjustable parameters which would have different impacts on the solution. Listed below there is a set of topics which contains parameters which could be interesting to further look into. Since they will determine how the system learns and compensates for the changes in the environment:

- Conveyor belts.
- Manipulators.
- Sorting method.
- Cameras.
- Objects.

For the conveyors, there are two different possible variables in the setup. The first one, is the direction the conveyors are going. The second parameter is changing the speed of one or both the conveyors. Looking at the possible setup in the Figure 1.1 in task description Section 1.2. The conveyors are going the opposite direction. However, it would be possible to have both conveyors going in the same direction, as seen in Figure 2.6, which would affect the sorting method.

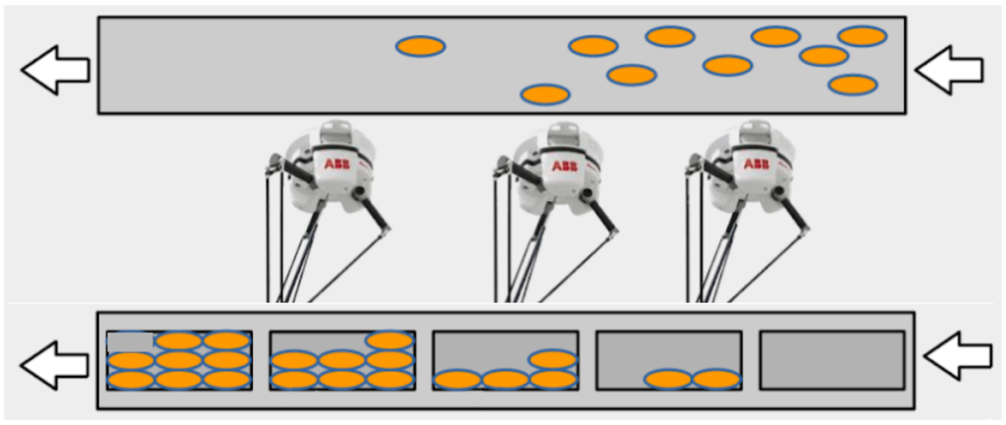


Figure 2.6: Setup where conveyors are going in the same direction. [1]

The second parameter is the speed of the conveyor(s). Changing the conveyor speed, might have an influence on the solution's efficiency and productivity. Since the faster the conveyors move, the more objects will be introduced into the manipulators workspace. Therefore they should be able to compensate for this, so no resources are wasted.

This leads to the next topic: The manipulators, the most interesting parameter to look into, is the speed of the manipulators. Because it allows the production to go faster or slower depending on the flow of objects. It also provides a good base for load balancing, since the speed of a manipulator being overworked, could be diminished so the other manipulators have a chance to catch up to its working pace.

Besides the speed, the number of manipulators would also impact the setup. Though this parameters depends on the demanded productivity. Adding or subtracting manipulators, means the system would have to figure out a new way of properly balancing the workload. It would be interesting to test, what is the most efficient number of manipulators, so the solution is able to determine when it is over or under performing.

In contrast to the above stated, the sorting method will have repercussions on the overall solution. There are lot of different ways the objects can be sorted in. Choosing which object to pick up first, will determine in which sequence the other objects will be picked up. Deciding where to place them, will also affect how the rest of objects can be placed afterwards.

Once it has been decided which object to pick and where to place it, the layout of the objects will impact the overall solution. Since they can be sorted in either boxes or trays. Furthermore, they can be stacked or placed right next to each other one by one. The capacity of each container will also have to be accounted for when adjusting the method to fill them up.

In regards to ensuring that resources are not being wasted, cameras are needed to detect the incoming objects and ensure that they are picked. In order to obtain this, the cameras placement is essential, so they are able to see and detect every object. However, their task is limited to the detection of objects and therefore add little to no value, to the reinforcement learning or load balancing.

The last thing to consider is the objects and boxes in general. Their shape, production time, placement and how many should be produced at the same time. Aspects such as orientation, will affect how the manipulators pick objects and place them in the boxes. The boxes design and capacity have en influence on how the solution works overall. Aspects such as sorting and placement method will be affected by this.

A basic solution will be made, which is a simple solution that solves the task. This simulation is going to be used as a base line, which the final solution will be compared to. The reason behind this is to analyze how reinforcement learning improves the performance of the task. To make the comparison fair, the basic solution and the reinforcement learning solution should have identical parameters. Those parameters which was decided to remain as constants, will be considered conditions. The Table 2.1 showcases the different parameters and conditions.

Simulation variables			
Possible parameter	***Relevant during learning***	***Conditions***	***Implementation***
Conveyor speed	√	X	√
Picking up algorithm	√	X	√
Manipulator speed	√	X	√
Amount of manipulators	X	√	√
Direction of conveyors	X	√	X
Amount of objects	X	√	X
Placement of conveyors	X	√	X
Amount of conveyors	X	√	X
Box layout	X	√	X
Box capacity	X	√	X
Shape of objects	X	√	X
Cameras position	X	√	X
Amount of cameras	X	√	X

Table 2.1: Parameters based on their relevance to the reinforcement learning and load balancing, as well as whether or not they are planned to be implemented

All the parameters stated in the Table 2.1, would introduce a flexible and dynamic environment. However, due to limitations, some of the parameters will not be tested or looked further into. Instead, the report will focus on those parameters which were regarded as "the most relevant" for the reinforcement learning and load balancing. This means that they would have a significant impact in the overall solution in regards to load balancing. The parameters that will be in focus, is the speed of the conveyors, picking up algorithm and the speed and amount of manipulators. Since they were regarded to add most, to the reinforcement learning and load balancing.

The conveyor speed parameter, can refer to multiple variables, it can either refer to the conveyor where the objects are being produced, the box conveyor or both. For the solution, it was decided to only focus on the conveyor carrying the boxes, where the objects will be sorted in. The decision for this is based on a combination of different reasons. From a production point of view, it does not seem efficient to adjust the speed of the conveyor, carrying objects, to be varying over time, since this could alter the other steps in the production. That's why keeping the speed as a constant condition is a more sensible approach.

Secondly, the reinforcement learning is concerned not only on successfully picking up objects and placing them. But also, it has to make sure that the overall solution, is able to fill all the boxes before they leave the production line. Therefore, being able to adjust the speed of the conveyor handling the boxes, gives the manipulators a better chance to finish their task. Furthermore, being able to adjust the speed of the box conveyor accordingly to the flow of objects, is expected to improve the efficiency of production and provide an interesting environment to test the load balancing.

The picking up algorithm refers to the process behind how a certain manipulator, will choose which object is the preferred option to pick up. This is in direct correlation with reinforcement learning and balancing, since in the basic simulation, the manipulator will choose whatever object is the closest, whereas in the solution featuring reinforcement learning, it will decide what actions should provide the highest reward.

Manipulator speed, this parameter allows control over the speed of the manipulators while solving their task. This parameter will directly affect the load balancing. When one manipulator has a higher pick up rate than the others, this manipulator can be slowed down, so the workload can be equally balanced among all manipulators. Likewise, if a manipulator is falling behind, its speed can be increased, so it can catch up to the others.

The last implemented parameter is the amount of manipulators. This was chosen, since it a change in the amount of manipulators, means that the system has to rethink and find a way to re-balance the production.

Apart from the ones mentioned, there are two other parameters which seemed interesting, but are assumed as conditions. The reason for this, is due to time limitations of the project and because they were thought to have the smallest impact on reinforcement learning and load balancing. Therefore, the direction of conveyor and amount of objects will not be looked further into after this section.

The direction of the conveyors, will be assumed to go in opposite direction. If they were to go in the same direction, it could cause a problem. As seen in Figure 2.6, there is not enough objects for the last manipulator to fill the half empty box, resulting in slowing down the production, until there is enough objects to fill the last box. This however, could be used as a possible scenario, where the system would have to learn, how to make up for the possible lack of objects. But as stated above, due to time limitations, this parameter will not be further looked into.

The amount of objects being produced, will have an impact due to the fact that if more objects enter the workspace, the faster the manipulators will have to work. Otherwise, a lot of the objects would not be palletized and go to waste. Being able to change the flow of objects accordingly to current state of the system, would help maximizing the production. However, a similar effect can be obtained from the "conveyor speed" parameter, therefore it would seem redundant to have two different parameters which have similar goals.

The shape and orientation of the boxes and objects is considered conditions. The reason for this, is because these parameters add little to no value to the load balancing. Furthermore keeping them as conditions simplifies the problem, since there are so many variation in both parameters. So it was decided that the objects would be flat and circular, the boxes would be a 3x1 box where only one object can be placed in.

The rest of the parameters listen in Table 2.1, were considered as the least relevant for the project, therefore, even though they might have some impact on the overall solution, they have been disregarded and will be set as constant conditions throughout the simulation.

This section has been a clarification of which different features will be set as conditions and which ones as parameters. All the features will remain the same for both, the basic and reinforcement learning simulation, only varying the method each simulation uses in order to cope with the task.

2.6 Advantages of a simulation

This research project is focused around reinforcement learning and load balancing. Both of them are relevant software areas which should improve the system. The reinforcement learning makes the solution choose independently, the most appropriate course of action, based on its previous experience and interactions with the environment, thereby improve the overall performance. While the load balancing is in charge of properly distributing the workload among the available resources, making the solution exploit its capabilities.

The solution for this project will be regarded as a theoretical application. This means that the emphasis of the project, will be placed on mathematical modeling and tests that are carried out using a simulation, instead of a practical approach.

The reason for this decision lies in that the main focus of this project, revolves around software algorithms, which benefit from a flexible environment. Creating this in a simulation have the advantage, that it allows for a wide range of variables to be changed with no economic impact.

Another advantage of a simulation. Is that it can be used to model the physical world, allowing user interaction with a complex environment, without any financial, spacial and hardware problems. This enables the users to focus on specific aspects they want to learn about, while being able to modify any characteristic they are interested in, so they can analyze the effects of these interactions. Since a simulation mimics the real world, it is possible to investigate and test how objects will behave under certain conditions, even allowing to be experimented within the 4^{th} dimension, time. [8]

Having a complex environment modeled in a computer system, certain computational errors may occur. In order to have an accurate simulation of the physical world, objects must be described in several dimensions, allowing them to have similar properties to the original ones as possible. Certain aspects like physical properties, kinematics and dynamics models, etc. must be implemented to make the simulation realistic. However, the more characteristics and accurate details added to the simulation, the more computational power it will require in order to be executed. A possible solution for this, could be to reduce the complexity of the simulation itself. This is done by generalizing or reducing the mathematical models that define the objects. However, this has the drawback, that the simulation might work, but when trying to implement it into the physical world, some errors might occur. An example of this was illustrated in Donald C. Craig thesis [9], where it showcases the simulation of a circuit. In order to simplify the design and therefore the processing time, it makes the assumption that one wire does not adversely affect current flowing in an adjacent wire. However, even though this might be true in the simulation. When two wires are placed too close to each other in the physical world, electromagnetic interference can occur between the two wires. This ultimately results in that the simulation works, but cannot properly recreate the physical world.

Since this project is a theoretical project, with a lot of parameters that can be looked into. It is decided that the solution for this project, will be made as a simulation in order to have more flexibility for testing the solution.

In this chapter it was decided which parameters and conditions that will be further look into. These features have been chosen in order to comply with the task given and is related to pick and place applications in Industry 4.0, which manages to load balance a system using reinforcement learning.

2.7 Problem Statement

How to design a robotic simulation setup, which facilitates the training and testing of a pick-and-place reinforcement learning agent(s) with multiple robots.

Chapter | 3

Development

3.1 Basic simulation

The basic simulation is, as the name implies, a basic solution to the task at hand. It is able perform and solve the task, without usage of reinforcement learning and load balancing. The setup for the basic simulation can be seen in Figure 3.1. This setup looks similar to the possible setup, Figure 1.1 in section 1.2, with three manipulators and two conveyor belts with objects and boxes. As seen in Figure 3.1

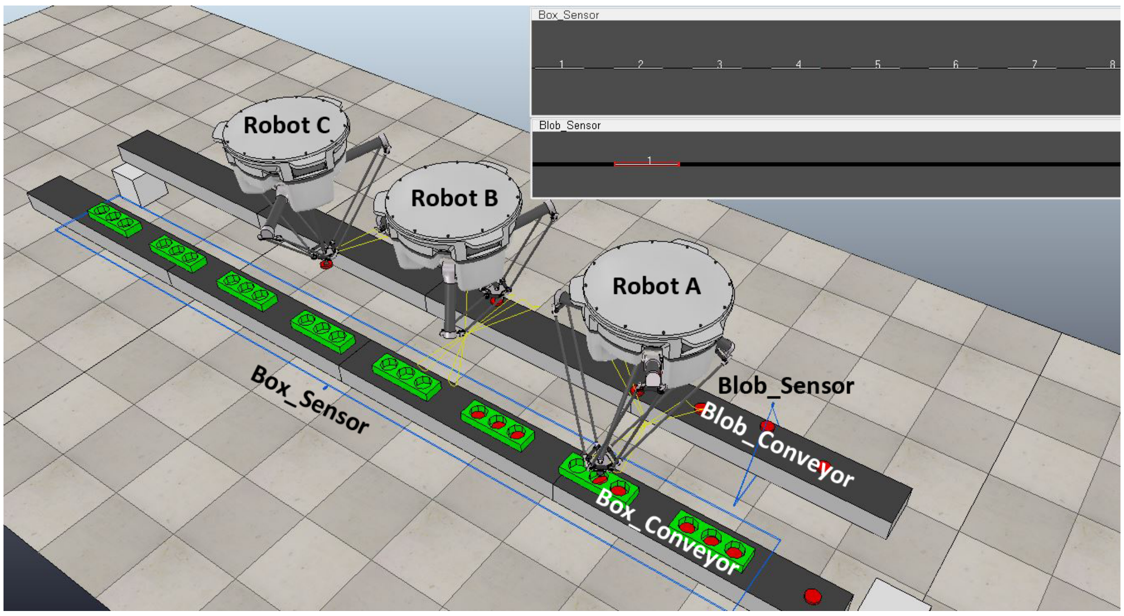


Figure 3.1: An overview of the simulation created in V-rep. Where all the components can be seen and how they are placed in relation to each other.

As stated in Section 2.5, all of the parameters are fixed to be conditions, expect for the manipulator speed and the conveyor handling the boxes. The amount of objects is randomized from 1 - 3, every time they appear. The box conveyor is in charge of controlling its own speed to prevent boxes from leaving without being filled. In order to do that, it checks the status of the three boxes, which are closest to the end of the conveyor. If they are not completely filled, the conveyor will slow down, but not make a full stop, to allow the manipulators to finish their task before speeding up again. This means that while the simulation is running, the conditions are as seen in Table 2.1. When the simulation starts, both conveyors will be running in opposite directions. The first conveyor is carrying the objects. The amount will be randomized between one and three objects, each time interval. The second conveyor is carrying the boxes, where the objects will be relocated. These boxes appear accordingly to the box conveyor speed, one after another and keeps a fixed distance between them.

As seen in Figure 3.1, there are two sensors. One is placed on top of the conveyor carrying the objects and another one is placed perpendicular to the box conveyor. These cameras' task is to detect and register how many objects and boxes there are, and their position relative to the world frame. This type of vision sensor is able to analyze a row of pixels of 1xn dimensions. The cameras will use blob detection in order to detect the objects and boxes. To do this, the input of the camera is converted into a binary image, where the objects are separated from the background. Anything red, will be detected as a blob in the blob sensor, and anything green will be an box for the box sensor. The blobs detected by the cameras will be numbered. This can be seen in Figure 3.2.

The camera saves the time and position when a blob was registered and sends this information to the manipulators. The manipulators will then use this to calculate the position of the blobs, as they move along the conveyor. This information is used to determine when the blobs enter their workspace.

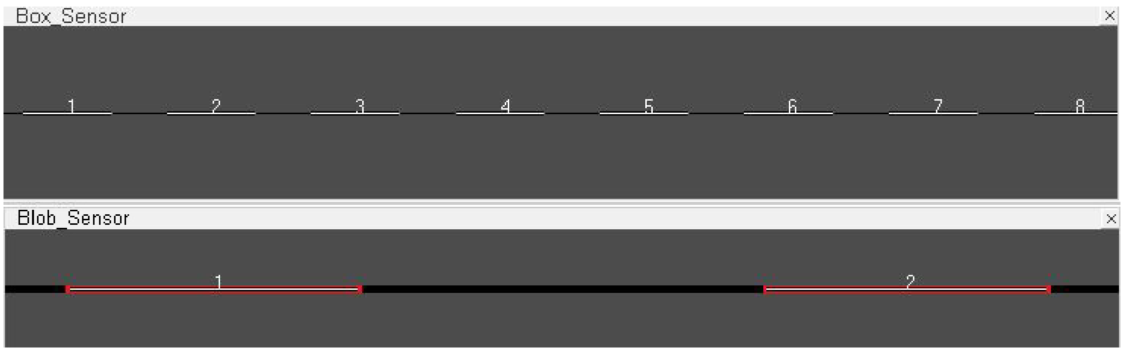


Figure 3.2: Picture of the blob- and box sensor view. Upper one is the box sensor and the lower one is the blob sensor. The number and white lines indicates the detection of an blob or box.

Once a blob has entered a manipulator's workspace, the shortest distance from the manipulator to the blob is calculated. This process is done through Matlab, which will handle all the calculations in the basic simulation. Matlab calculates the distance from each manipulator, to all the blobs on the conveyor. It then chooses the blob, closest to each manipulator. When it has been decided which one to pick up, each manipulator will check if the blob, that was assigned as the closest one, is within its workspace. If it is, the manipulator will pick it up as seen in Figure 3.3. After the manipulator has picked up a blob, it then places it in the box, furthest along the conveyor still within reach and fills it up, in an orderly manner. As seen Figure 3.4.

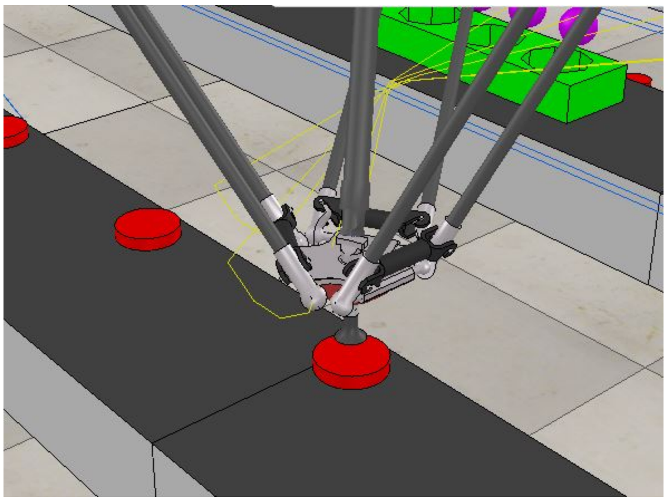


Figure 3.3: Close up of a manipulator picking up a blob.

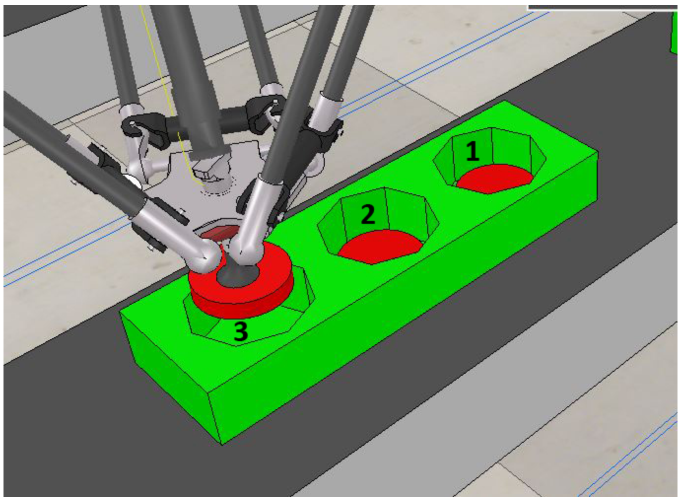


Figure 3.4: Close up of a manipulator placing a blob. The numbers indicate the order the box is filled, starting from one.

The spots where a blob can be placed, are numbered ranging from 1 - 3 in a sequential order. When a manipulator arrives at an empty box, it place the blob in spot number one, two and three in that order. See Figure 3.4. The program will always choose to fill up the empty spot with the lowest value first, which corresponds to the spot further along the conveyor. Therefore, the spots to fill up first, will be those which are more likely to leave the robot's workspace first.

Figure 3.5 is a representation of the communication flow in the basic simulation. Where the "Pick up select" happens in Matlab, while the rest takes place in V-rep. When the simulation starts, the box and blob conveyor will start running, and the blob producer will start generating blobs, while the box producer generates boxes. Each of them placed on their respective conveyors. Meanwhile, the sensors in charge of detecting the blobs on each conveyor. Whenever it detects a blob, the sensor will save the position and time (PaT), which is sent to the Matlab communication, as well as to each of the manipulators. The Matlab communication will send the PaT input to the Pick up select function, which will calculate the distance from the blobs to each manipulator. Then it will choose whichever blob is the closest to each manipulator and then send it back to the manipulators.

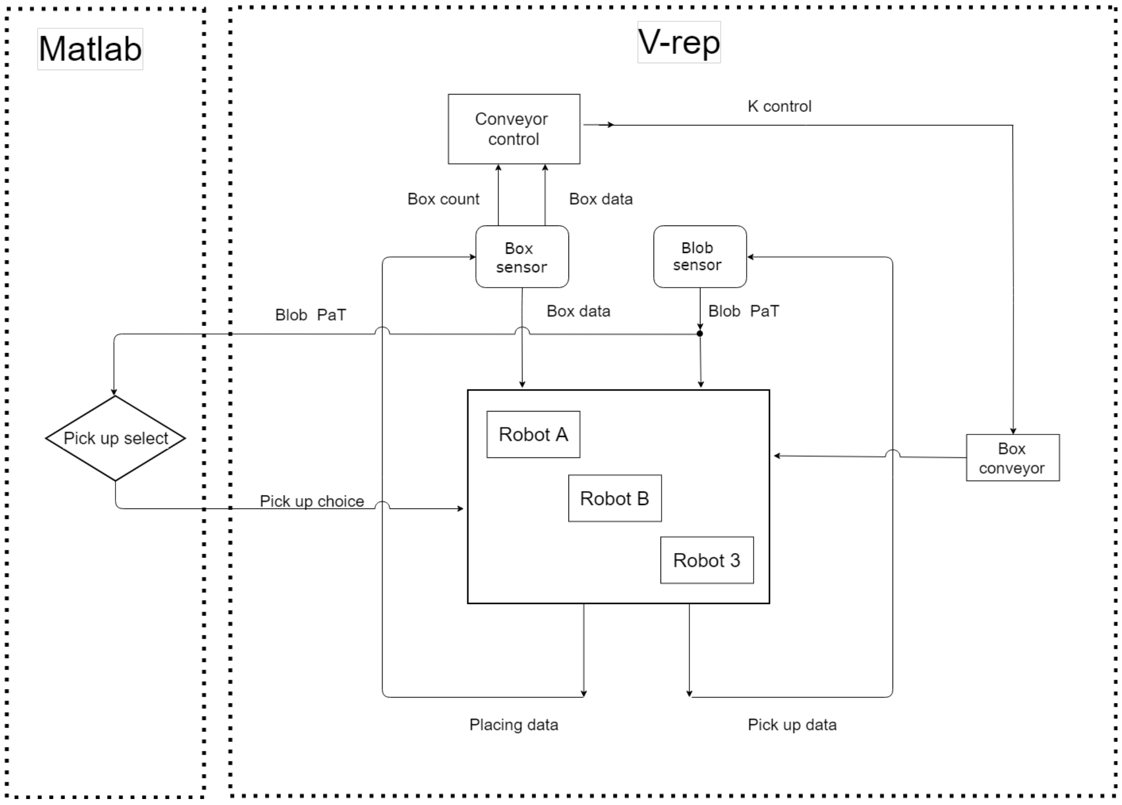


Figure 3.5: Block diagram of the basic simulation.

After a blob has been picked up, the program sends the information back to the blob sensor, which then deletes PaT from its memory. The manipulator gets the box data from the box sensor and then places the blob in the box closest to the end of line, within its workspace. This data will be send to the box sensor, which will save that spot as a filled spot. If there is no empty spot in a reachable box, the manipulator will wait until a box with space in it, enters its workspace. The box sensor, which is connected to a conveyor controller, checks out whether or not, the three boxes closest to the end of the conveyor are filled. If not, the speed of the conveyor will decrease, allowing the manipulators to finish their task. On the other hand, if they are filled, the conveyor speed will increase. Through this process, the conveyor never stops, it just changes velocity.

In this solution, neither reinforcement learning nor load balancing have been implemented. The main reason behind creating a simple solution, capable of solving the task as easy as possible. Is to compare its performance to a "smart" solution, which is able to load balance using reinforcement learning. This makes it able to see, how a system with learning abilities, is able to distribute the load equally, as it learns about its environment. So that the manipulators have the same tear down time, as mentioned in Section 2.3.

It is worth mentioning that the basic simulation, showed that using more than three manipulators might cause V-Rep processing problems. This means that no more than three manipulators will be used throughout the rest of the testing.

Now that the basic solution has been described, the next step is to find a solution featuring reinforcement learning. Both solutions should have the same settings and looks into the same parameters, that were discussed in Section 2.5. The parameters and some conditions will be modified before or during testing to analyze the performance of each solution under different circumstances.

3.2 Solution proposal

This section aims to explain how the solution will be implemented. It describes what parameters will be improved in comparison to the basic simulation, as well as how this is done.

Figure 3.6 shows the communication flow in the simulation with reinforcement learning. It is based on the basic simulation, described in Section 3.1, but with a few alterations. Mainly, the conveyor controller has been substituted with the reinforcement learning agents. The conveyor speed, which is chosen according to the input from the box sensor and the manipulator speed is used to control the load balancing. As it can be seen, the manipulators have an extra output, the pick up count. Each manipulator will send the amount of blobs they have picked up, and based on this data, the reinforcement learning agent will determine whether or not the workload is evenly distributed. In case it is not, it will force a manipulator to slow down, so the others can pick up its pace.

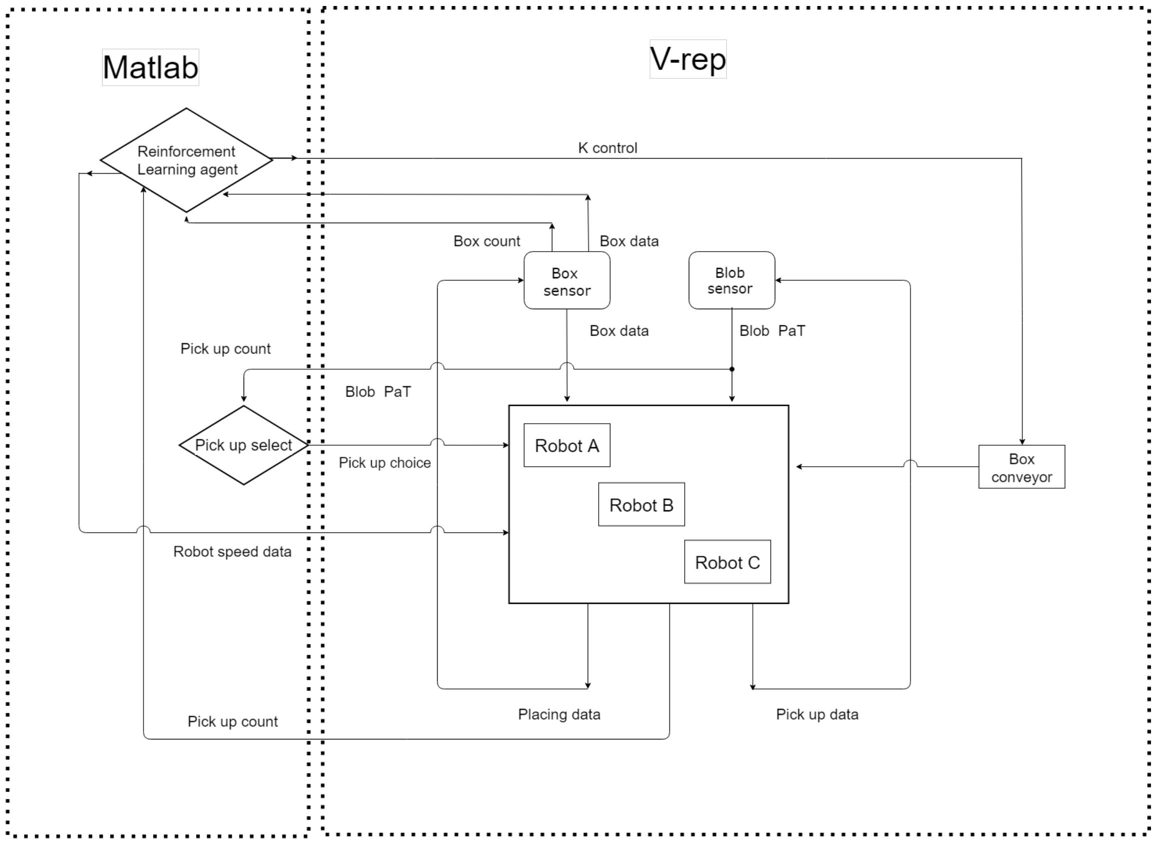


Figure 3.6: Block diagram of the simulation with reinforcement learning.

In this project, the reinforcement learning and load balancing are depended on four distinguishable components: Three manipulators in charge of picking objects and a conveyor with variable speed. These components will be divided into two agents. One controlling the manipulators and the other controlling the conveyor. The agents work independent with respect to each other, but should be able work and communicate together. This is due to several reasons, one of them being that the more dimensions, the harder it is to find features to properly control the system. Therefore, due to the nature of the components, it has been decided to split them into two separate reinforcement learning problems.

3.2.1 Load balancing agent

The following subsection is an explanation of how the system balances the load between the three manipulators. If the total amount of objects to distribute is considered as 100%, each manipulator should pick up approximately $33.3\bar{3}$ % of the objects. However, in the basic simulation, without reinforcement learning, each manipulator will try to pick up as much as they can. This means that the manipulator at the beginning of the line, will have a larger pick up portion and therefore work more. Whereas the manipulator at the opposite end of the line, will have little to no objects to pick up. This, from a production point of view would mean that the first manipulator is being overloaded, while the last is under performing. The purpose is to solve this with load-balancing. However as stated in Section 2.3 this only makes sense if there is an underflow of objects.

Figure 3.7 is a representation of load balancing between the manipulators. The method followed to load balance the system consists of having Robot A and Robot B be controlled with reinforcement learning, while Robot C will remain active picking up all the blobs which are not picked up by the other manipulators. Looking at Figure 3.7, the green diamond represents a random work distribution at a given moment. At this position, the manipulators have an uneven distribution of objects, which should be improved. In this case, it can be seen that Robot A is picking up 40 % of the objects, Robot B picks up 42 % and Robot C will pick up whatever is left, which in this case is 18 %. The goal is then to go from this random distribution, to the red hexagonal goal position. Since this position is where each manipulator will pick up a third of the objects, making the solution evenly balanced.

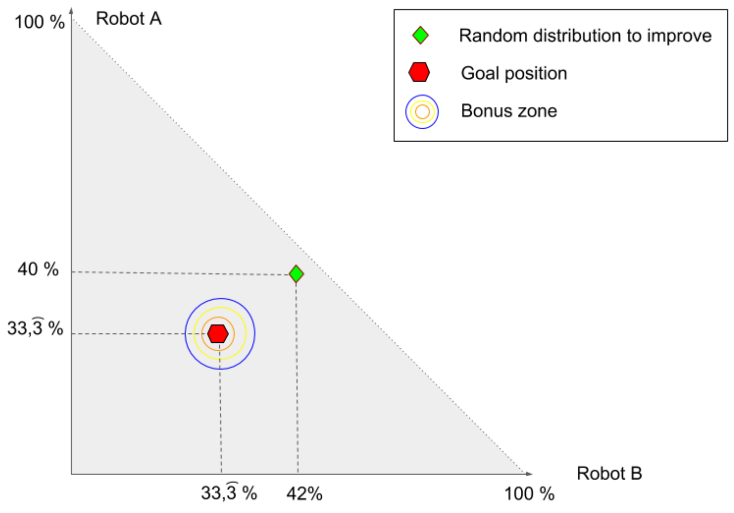


Figure 3.7: 2D representation of the Reinforcement Learning.

As it can be seen in Figure 3.7, there is an area around red hexagon point, which has been categorized as "Bonus zone". This area refers to those points, which are close enough to the target. Meaning this is the desired work zone, so whenever the system operates in this zone, it will receive an increment in reward points. This will make it feel inclined to explore similar positions in the future and eventually learn where the target is located. In order to obtain this, the system follows an exponential equation, seen in Equation 3.1. Meaning the closer it gets to the target, the reward will increase consequently as represented in Figure 3.7 as a circle around the target. The outer layers have colors from the cold spectrum, meaning that the reward is small. Whereas the inner ones are warmer and therefore more points will be obtained. This is illustrated in Figure 3.8, where it can be seen, that the closer it operates to the target position, the higher the reward.

$$f(x) = 2^{-(5x - 6)} \tag{3.1}$$

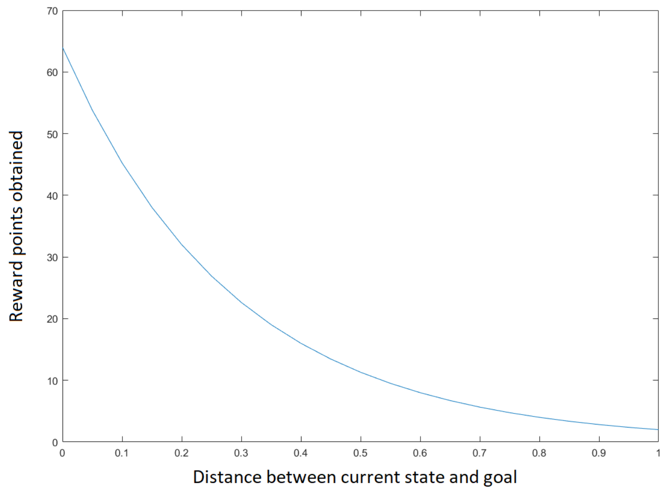


Figure 3.8: Exponential function illustrating the reward gained in sphere, with distance from target hexagon in the X-axis and reward gained in Y-axis.

Now that the load balancing aspect of the reinforcement learning has been explained, it is time to add another dimension. As it can be seen in Figure 3.9, the goal position is no longer a single point in space, but instead a vector which moves along the 3^{rd}-axis, depending on position of the last box on the conveyor. This is refereed to as variable "d" in Figure 3.9. This variable refers to the position of the boxes, which are about to leave without being filled.

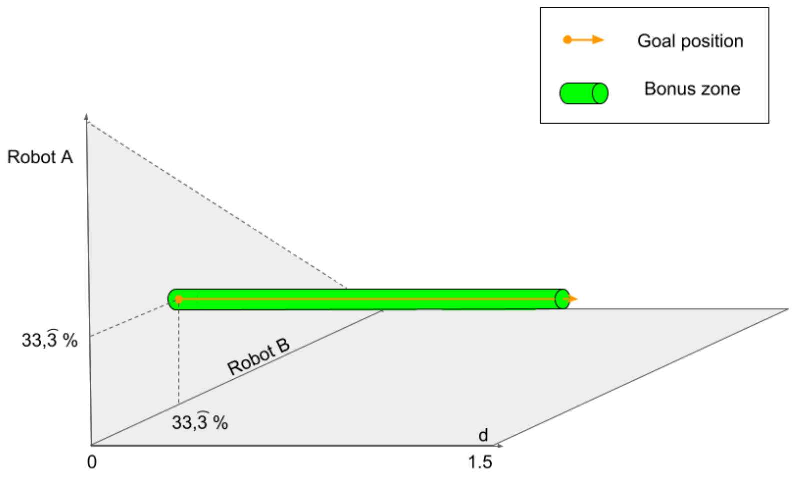


Figure 3.9: 3D representation of the Reinforcement Learning with the 3rd-axis implemented.

As mentioned it will get punished or rewarded in its learning process. The program should know when it can start or stop the manipulators without letting any boxes leave the line half empty. This will be done by setting a "line" on the conveyor, as seen in Figure 3.10. When a non-filled box passes this line, it will get punished and thereby know what it is doing now, is not good enough.

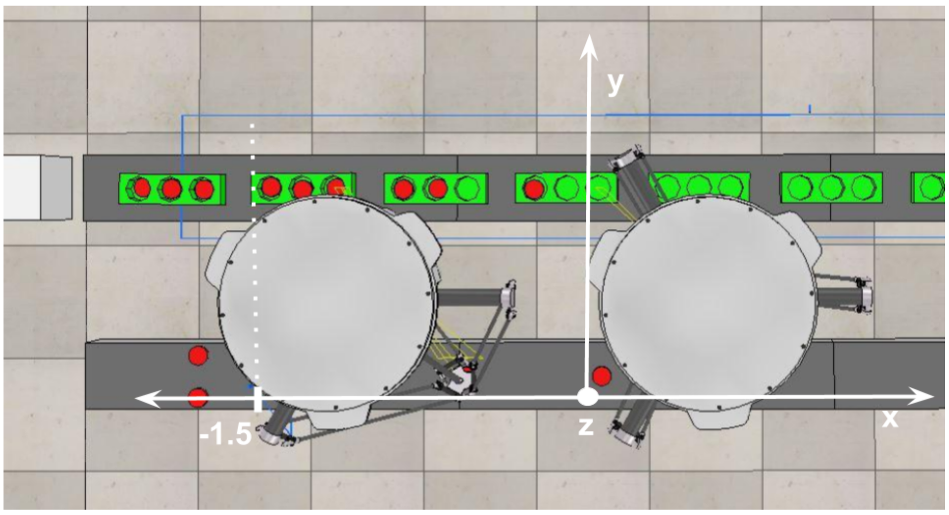


Figure 3.10: Figure showcasing the position of the boxes in relation to the worldframe.

3.2.2 Conveyor Agent

The second agent is related to the speed of the conveyor belt. For this agent, the main goal is to find the optimal speed, at which all boxes are filled, but without slowing down the production time. Therefore, the system aims for efficiency, meaning that all manipulators are being use to their potential and the throughput is maximized. Meanwhile the number of blobs not being picked should preferably be 0 or as close to 0 as possible. To obtain this, the agent uses the explore and exploit policy. Which means, that the system will choose the speed at which it receives the highest reward, but once in a while, it will randomly choose another speed and check whether or not the performance increases. Figure 3.11 illustrates how the relationship between reward and conveyor speed is predicted to be. It follows a parabolic model, where at some point, the system will find the optimal speed, where it gains the most reward. Before that point, the system is not able to cope with the flow of blobs. Since the conveyor is working too slow, the manipulators will have to wait for more boxes to come. Therefore it will get less reward points, since few boxes leave the line. Whereas after this point, it will get negative reward points, since the speed of the conveyor is too fast to fill up the boxes, resulting in making the overall solution worse.

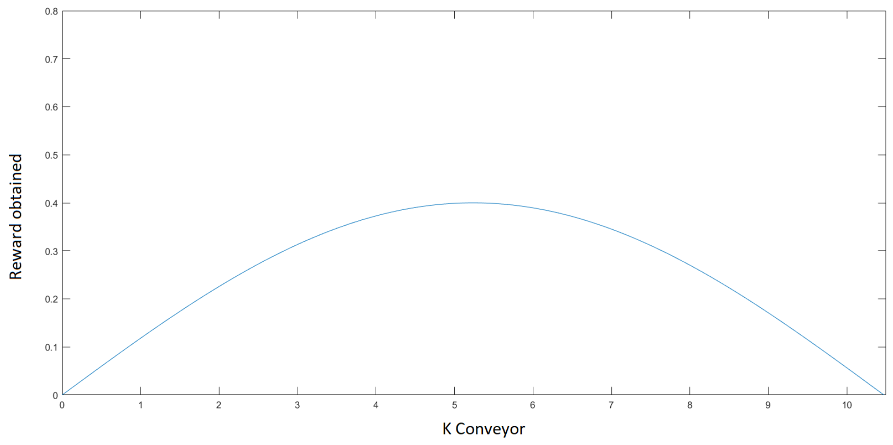


Figure 3.11: Graph illustrating the speed of the conveyor. With the speed in the X-axis and the reward points gained in the Y-axis.

The reinforcement learning agent for the conveyor speed, can be seen in Figure 3.12. This block diagram explains how the speed of the conveyor is chosen based on the performance of the system.

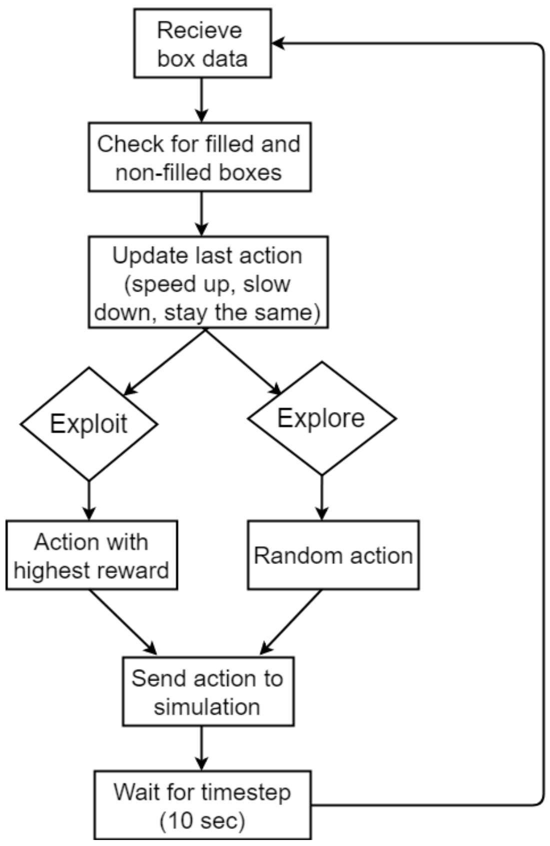

Recieve box data
Check for filled and non-filled boxes
Update last action (speed up, slow down, stay the same)
Exploit
Explore
Action with highest reward
Random action
Send action to simulation
Wait for timestep (10 sec)

Figure 3.12: Graph illustrating the reinforcement learning of the conveyor speed.

This chapter described how the basic simulation and reinforcement learning solution are designed and expected to work. The following chapter, Learning Algorithm, is a more detailed explanation on how the reinforcement learning was implemented in the solution.

Chapter | 4

Learning algorithm

4.1 Solution Implementation

This section aims to describe the solution with reinforcement learning implemented using a simulation. The simulation is composed of two programs V-rep and Matlab. V-rep, which is the simulation environment, where objects interact with each other and provides a visual interface. Matlab is in charge of the calculations for the basic simulation and reinforcement learning agents for the final solution.

It was discovered during this chapter, that the proposed solution in Section 3.2 with two agents, one controlling the manipulators and another one controlling the box conveyor, was not a viable solution. The main goal for the agents was to be independent from each other, but still be able to cooperate. However, during the initial learning stages when both agents are still exploring the environment and trying random actions, they tend to interfere with each other. Meaning that whenever one agent tries to learn, the learning progress of the other will be reset, meaning that neither of them will properly learn. Therefore, from this point onward, the agent to be implemented is the one controlling the load balancing, while the speed of the box conveyor will be discarded.

Figure 4.1 is a block diagram explaining how the reinforcement learning program for the manipulators, calculates the actions in Matlab. This agent is used for the load balancing. When the program starts, it will receive data from V-rep. This data contains the pick up count from each manipulator and calculates the balance in percentage, from the total amount of blob pickups.

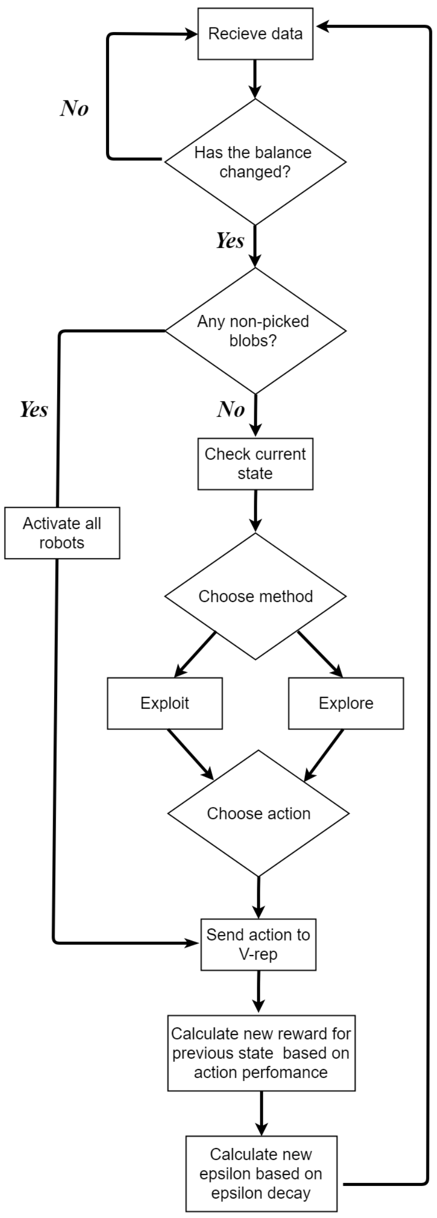


Figure 4.1: Block diagram of the reinforcement learning program for load balancing.

If there is a change in balance. The system will check for the current state, which is the percentage of blobs, picked up by each manipulator ranging from 0% to 100%. When adding the combined percentage count of the manipulators, it should be a total of 100%.

Depending on the state, the system will have four different actions to choose from. These actions can be seen in Table 4.1. As seen robot A and B, will be the only ones changing. While robot C will always remain active. This is done in order to simplify the learning, by having less actions for the agent to choose from.

In order to choose an action, there are two policies to follow, as stated in Section 2.4. The first, is the greedy policy, which will choose the action that provides the highest reward, or if two or more actions provide the same reward, it will choose the first action it encounters. Secondly, the $\epsilon - greedy$ policy, which decides to take a random action, to explore new paths towards the goal. The decision of which policy to choose, is based on epsilon, which is a variable that describes the probability of exploring. Epsilon decreases over time, that way, it encourage the system to explore in the beginning, but will over time exploit for the highest reward.

Action	Robot A	Robot B	Robot C
1	Stop	Stop	Start
2	Stop	Start	Start
3	Start	Start	Start
4	Start	Stop	Start

Table 4.1: Table explaining the different actions

After an action has been chosen, this information will be sent back to V-rep, where the simulation will execute it. This action, will provide the system with a new state, which will be either closer or further away from the goal. Based on the new state, the reward for the previous action will be changed accordingly. Giving a higher reward for those actions which have been tried before, that lead the system closer to the goal. After this, the value of epsilon will change as well, making the program less likely to take a random action, the longer the program has been running.

When the program starts, it will create several matrices, which helps to determine what actions to take, this can be seen in Figure 4.2. First it creates a matrix with ($16 \cdot 100^2$) rows and 3 columns. Where all the possible positions in balance space are created. This matrix is called "State Matrix".

Next, the "State Reward" matrix is created. This column matrix describes the reward points, based on the distance to the goal, which is the difference between the current balance of the system and the balance distribution of the goal. Meaning that the closer the system gets to the goal load balance, the higher the rewards. Finally "Q State" matrix is generated. This matrix is the reward system for all the actions at each state. "Q state" consists of a matrix with four columns (one per action), where each column is a copy of "State Reward" when it is initialized. The first time "Q State" is initialized, all actions for each state will give the same reward, since they are based on the distance to the goal. However, as the program starts exploring actions, it will update the "Q State" actions, based on two parameters: The new states corresponding reward in the reward function, as a result of the previous action. And the reward of the best action in the new state.

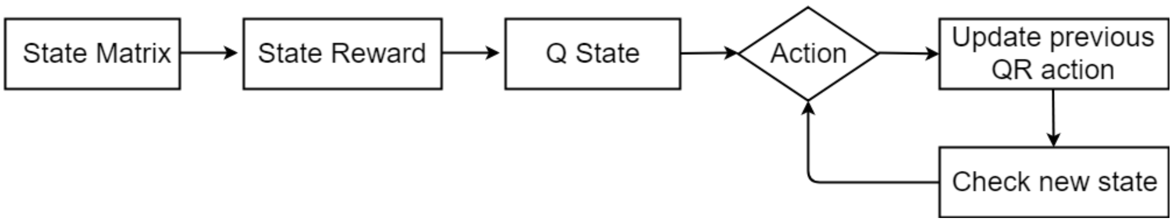


Figure 4.2: Figure showcasing the different components of the reinforcement learning in Matlab.

Table 4.2 is a representation of the position of each manipulator in the state space. The two first columns represents the pick up percentage of two of the manipulators. These two columns have values from 0.00 to 1.00, where 1.00 is 100 %, with a varying step of 0.01. Whereas the third column represents the position of the last box on the conveyor, which ranges from 0.00 to -1.50 meter and with a step of 0.10 meter. For each distribution of the two first positions, there are sixteen possible values in the third column. Therefore, there are sixteen positions for the goal of the load balance, which is $1.3\bar{3}$ for each manipulator, each of them being considered as a viable goal.

Position in balance space (R State)		
Robot A	Robot B	Distance in the conveyor
0.00	0.00	0.00
0.00	0.00	0.10
0.00	0.00	0.20
⋮	⋮	⋮
0.00	0.00	1.50
0.00	0.01	0.00
⋮	⋮	⋮
0.00	0.56	0.00
⋮	⋮	⋮
0.26	0.15	0.90
⋮	⋮	⋮
0.33	***0.33***	***1.10***
⋮	⋮	⋮
0.52	1.07	0.80
⋮	⋮	⋮
1.00	1.00	1.50

Table 4.2: State Matrix Table

Table 4.3 is a column matrix, which calculates the reward based on the distance to the goal. Therefore, rewards will increase as they approach the goal, having an exponential growth when the load balance approaches the goal, as shown in Equation 3.1.

Reward based on distance to goal (State Reward)
SR ($state_1$)
SR ($state_2$)
SR ($state_3$)
SR ($state_4$)
$\vdots$
SR ($state_n$)

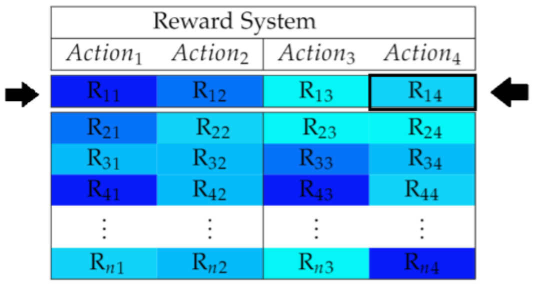
Table 4.3: State Reward Table

Lastly, the "Q State" matrix is generated, as seen in Table 4.4, where there are four actions for each state. Each one with a reward depending on whether they help the system approach the goal or not.

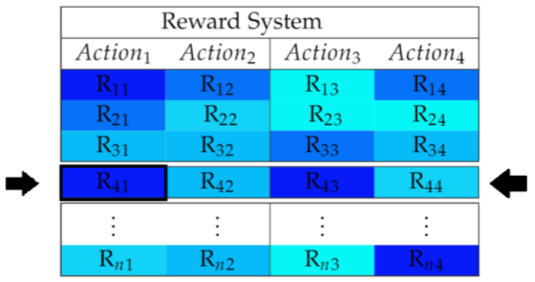
Reward System (Q State)			
$Action_1$	$Action_2$	$Action_3$	$Action_4$
R($state_1 Action_1$)	R($state_1 Action_2$)	R($state_1 Action_3$)	R($state_1 Action_4$)
R($state_2 Action_1$)	R($state_2 Action_2$)	R($state_2 Action_3$	R($state_2 Action_4$)
R($state_3 Action_1$)	R($state_3 Action_2$)	R($state_3 Action_3$)	R($state_3 Action_4$)
$\vdots$	$\vdots$	$\vdots$	$\vdots$
R($state_n Action_1$)	R($state_n Action_2$)	R($state_n Action_3$)	R($state_n Action_4$)

Table 4.4: Matrix of "Q State"

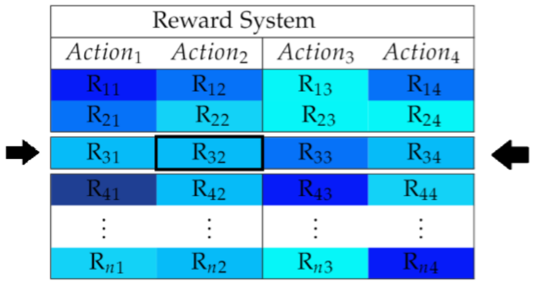
In order to showcase how "Q State" works, Figure 4.3 will give a step-by-step explanation. This figure represents "Q State" once it has been created and in the process of learning. Note, that this figure is used as a simplified proof of concept and is not an actual representation of the system. The rewards are showcased in a colour scheme, where the lighter colours represent low reward, and the darker ones, higher rewards. This Figure explains the decision making behind the program, based on current state and previous action.


Reward System
$Action_1$ $Action_2$ $Action_3$ $Action_4$
R_{11} R_{12} R_{13} R_{14}
R_{21} R_{22} R_{23} R_{24}
R_{31} R_{32} R_{33} R_{34}
R_{41} R_{42} R_{43} R_{44}
R_{n1} R_{n2} R_{n3} R_{n4}

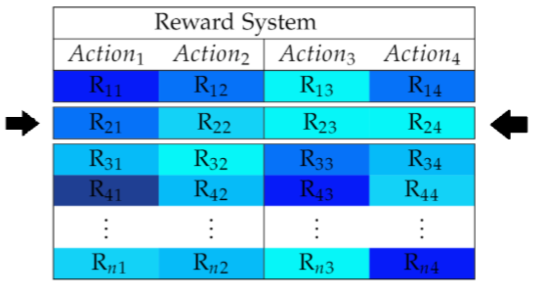
(a) The start position is in the first row. With the $\epsilon - greedy$ policy it randomly chooses $Action_4$


Reward System
$Action_1$ $Action_2$ $Action_3$ $Action_4$
R_{11} R_{12} R_{13} R_{14}
R_{21} R_{22} R_{23} R_{24}
R_{31} R_{32} R_{33} R_{34}
R_{41} R_{42} R_{43} R_{44}
R_{n1} R_{n2} R_{n3} R_{n4}

(b) After the previous action, the new state is the fourth row. In this particular example, it is assumed that R_{14} brought us closer to the goal, therefore, the reward for the previous action increases, as it can be seen due to the darker colour of R_{14} compared to the previous matrix. Now, with the greedy policy, $Action_1$ is chosen.


Reward System
$Action_1$ $Action_2$ $Action_3$ $Action_4$
R_{11} R_{12} R_{13} R_{14}
R_{21} R_{22} R_{23} R_{24}
R_{31} R_{32} R_{33} R_{34}
R_{41} R_{42} R_{43} R_{44}
R_{n1} R_{n2} R_{n3} R_{n4}

(c) The current state is now the third row and the policy chosen is $\epsilon - greedy$. The system randomly chooses $Action_2$. We now calculate the reward for the previous action. As it can be be seen in the previous matrix, this action already had a dark colour compared to the rest, furthermore, it was chosen by the greedy policy, which already knows that this action gives a high reward. And it also helped the system get closer to the goal. Therefore, it's reward increases.


Reward System
$Action_1$ $Action_2$ $Action_3$ $Action_4$
R_{11} R_{12} R_{13} R_{14}
R_{21} R_{22} R_{23} R_{24}
R_{31} R_{32} R_{33} R_{34}
R_{41} R_{42} R_{43} R_{44}
R_{n1} R_{n2} R_{n3} R_{n4}

(d) The current state is the second row. And the previous action caused the system to get further away from the goal. Therefore, the reward from R_{32} decreases.

Figure 4.3: Step by step example of how the Reward system works in the "Q State".

Initializer.
In certain cases, the system will run the program and try to reach the goal. But for some reason, never be able to find an appropriate path towards it and end up in a different direction, as seen in Figure 4.4. In order to avoid this problem during the load balancing. The system will be using an initializer, which tries to calculate a path within specified amount of steps. Otherwise the system will reset and start finding a new path from the starting point.

The initializer is a simplified programmed version of the simulation, that only works with numbers and therefore can run much faster. Once it successfully finds a path multiple times, the system will continue to work with the actual simulation and learn from there. This is done so when the simulation starts, the program will have a general idea of which direction to go, instead of trying random actions while getting negative rewards. This will speed up the learning process, since less trial and error is required during simulation. However the initializer is not perfect, even if it helps the solution in the right direction. This is caused by the position of the boxes not being implemented, meaning it will not receive any punishment if an empty box leaves the line. This then has to be learned in the actual simulation. Furthermore the initializer pickup count of blobs, is a calculated version of the simulation and it is therefore not working entirely the same, as the simulation. The initializer is only used as a tool to help reduce the learning time.

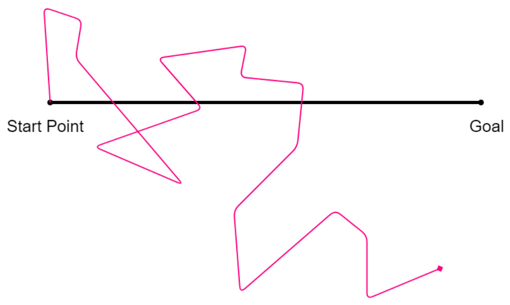


Figure 4.4: Figure illustrating the system working in the wrong direction.

The problem presented for the load balancing, can be described in a simple graphical way. Where a start point and a goal is given, the task is to find the shortest path between them. It is an axiomatic assumption, that the shortest path between two points is a straight line. Therefore, the closer the path selected resembles this optimal straight line, the higher the reward obtained. Let's assume that after running for some time, the program has found a path from the start position to the goal. This can be seen in Figure 4.5. As shown, this path is not optimal, but since it provides a successful way of solving the task, the program will feel inclined to repeat this path in the future.

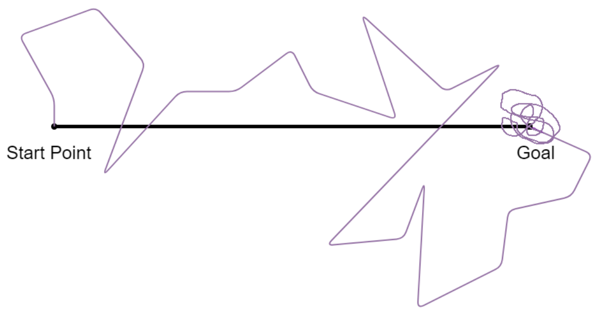


Figure 4.5: Figure illustrating the path towards the goal.

However, with the greedy policy, the program will get stuck in the first path it finds able to solve the task at hand, even when there are better solutions, witch have not been found yet. Therefore, with the $\epsilon - greedy$ policy, every once in a while, it will try a random action to see if the method can be improved. This kind of behaviour, can be observed in Figure 4.6, where the system tries a random action, which happens to be a better solution. When this occurs, the system will remember this new path and choose it over the old one, since this one provides a higher reward.

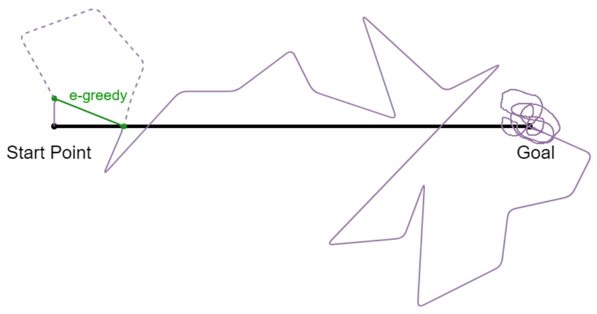


Figure 4.6: Figure illustrating the advantages of the $\epsilon - greedy$ method.

Over time, the system's initiative to try a random action will decrease. This is due to the fact, that after several tries, the program will have a general overview of what actions are better to compensate for imbalance in the system, whereas taking a random action, will have a lower likelihood of helping the program explore for better options, but instead will take it further away from the goal. Therefore, over time the system's path will look similar to Figure 4.7, where the system is able to have a path as close to a straight line as possible.

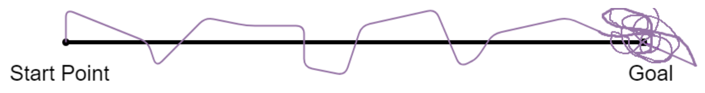


Figure 4.7: Figure showcasing the path towards the goal over time.

As it can be seen through Figures 4.5 to 4.7, the system does not reach a unique point in space as a "goal" and then resets, but instead it is enclosed around certain values. Thus it will be constantly trying to find balance, changing every time a new object is picked, in order to stay around the goal.

Once the solution has been implemented, the next step is to proceed with testing. The next chapter aims to determine whether or not the solution is viable.

Chapter | 5

Test

5.1 Test

To find out if the how viable the solution is, this section will test the proposed solution and show the results along with some calculations. The discussion of the results will be in Section 6.1.

For the solution to be considered successful, there are different aspects, the solution should be able to fulfill. The best outcome is if the solution is able to balance the workload between the manipulators, fill all the boxes and pick up all blobs. However the solution can still be considered successful, even though it did not fulfill all of them. Therefore they have been given a priority on how important they are.

First priority is that all boxes leave the production line filled up. From a production point of view, if some boxes left half empty, a worker would have to check every box and see if it is full or not. Which counteracts the whole purpose of the task and would thereby not making a viable solution.

The second priority is distributing the load equally between the manipulators. Even if this was a part of the task description, this is found as second priority, since the solution would be pointless, if it were not able to fulfill what is written in the first priority. Furthermore it only makes sense to do balancing if there is a underflow of blobs as stated in Section 2.3.

The third priority is that all blobs are being picked up. This is not as important as the boxes. From a manufacturing point of view, it would be desired that all blobs would be picked, so all products would be sold. A solution to the leftover blobs could be to make them go around in a circle and come back into the system.

Tests were preformed, to see how well the solution fulfills the priorities. The testing was done by running the simulations for 10 simulation minutes. Each solution was tested three times to see if the results were consistent. The first test was for the basic simulation as a control test. This is what the reinforcement learning solution is compared to in Section 6.1. The second test is for the reinforcement learning using three manipulators. This is to test how well the reinforcement solution worked. The different aspects within it was tested to see, how each aspect preformed. To test and show that the initializer actually works and if the reinforcement learning solution learns something from running. Two more tests were performed, where the first one only had the initializer running and the other without any form of learning at all. Lastly both the basic and reinforcement learning solution were test using only two manipulators. This is done, as mentioned in Section 2.5, to see if and how the system would compensate for this and if it is able to think and find a way to balance the production in a new environment.

Table 5.1, showcases the results obtained from running the basic simulation. This test will be used a guideline in order to compare and contrast the rest of the tests.

Test	*Full boxes*	*Non-full boxes*	*Robot A*	*Robot B*	*Robot C*	*Lost blobs*	*Total blobs*
1	110	3	65	150	122	33	370
2	102	1	67	140	104	20	331
3	102	3	68	144	105	20	337

Table 5.1: Results of test of basic simulation with three manipulators.

Table 5.2 contains the results from the simulation with reinforcement learning. It has been both, initialized and learned.

Test	*Learn time*	*Full boxes*	*Non-full boxes*	*Robot A*	*Robot B*	*Robot C*	*Lost blobs*	*Total blobs*
1	$\approx$ 2h	98	5	107	105	91	21	324
2	$\approx$ 2h	96	7	103	111	89	24	327
3	$\approx$ 2h	96	7	107	101	98	43	349

Table 5.2: Results of test with reinforcement learning with three manipulators.

Table 5.3 shows the results of how the program preformed after it had been initialized using three manipulators.

Test	*Learn time*	*Full boxes*	*Non-full boxes*	*Robot A*	*Robot B*	*Robot C*	*Lost blobs*	*Total blobs*
1	$\approx$ 1h	99	5	64	114	112	30	320
2	$\approx$ 1h	94	5	68	112	119	34	333
3	$\approx$ 1h	88	6	68	108	110	49	335

Table 5.3: Results of test with only initialization with three manipulators.

The next Table 5.4 shows how the program performed using three manipulators without any form of training.

Test	*Learn time*	*Full boxes*	*Non-full boxes*	*Robot A*	*Robot B*	*Robot C*	*Lost blobs*	*Total blobs*
1	0 min	93	4	69	135	96	28	328
2	0 min	92	4	53	112	125	33	323
3	0 min	93	5	63	104	132	53	352

Table 5.4: Results of test with no learning with three manipulators.

Table 5.5 is the test for the basic simulation using only two manipulators and Table 5.6 is same but for the reinforcement learning simulation.

Test	*Full boxes*	*Non-fullboxes*	*Robot A*	*Robot B*	*Lost blobs*	*Total blobs*
1	76	2	149	87	11	247
2	89	0	160	99	17	276
3	77	1	144	85	10	239

Table 5.5: Results of test of basic simulation with two manipulators.

Test	*Learn time*	*Full boxes*	*Non-full boxes*	*Robot A*	*Robot B*	*Lost blobs*	*Total blobs*
1	$\approx$ 6h	68	8	114	104	29	247
2	$\approx$ 6h	67	7	104	109	28	241
3	$\approx$ 6h	51	18	75	103	63	241

Table 5.6: Results of test with reinforcement learning with two manipulators.

To see how the if the agent was able to learn, Figure 5.1 was made to showcase the learning curve over time.

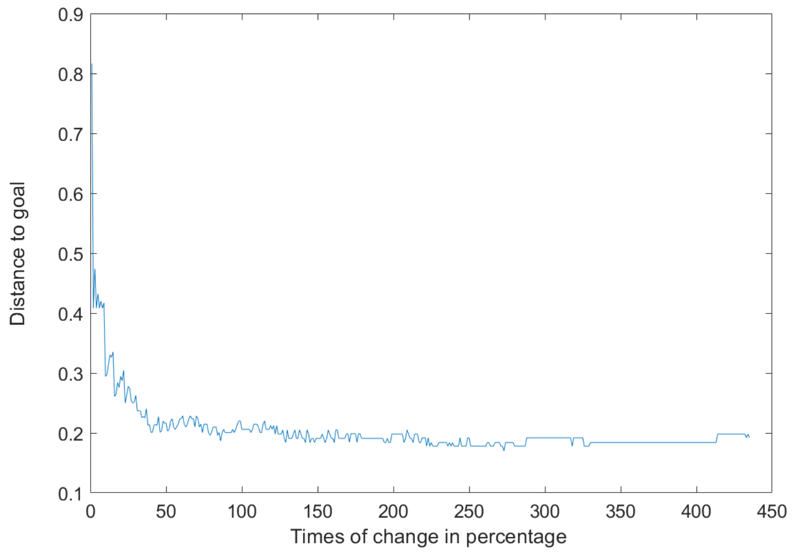


Figure 5.1: Figure showcasing the learning curve of the reinforcement learning program running for 30 mins without initialization.

Processing of test results:
In order to withdraw conclusions and see if the solution was consistent. The different aspects in the solution was tested, to see how well each aspect preformed. The tables from Section 5.1 will be calculated, so they represent the results percentage-wise. This provides a better overview of the simulations performance.

The percentage-wise results from Section 5.1 can be seen in Table 5.7, 5.8, 5.10, 5.9, 5.11 and 5.12.

Test	*Non-full boxes*	*Balance Robot A*	*Balance Robot B*	*Balance Robot C*	*Lost blobs*
1	2.65%	19.29%	44.51%	36.20%	8.92%
2	0.97%	21.54%	45.02%	33.44%	6.04%
3	2.86%	21.45%	45.43%	33.12%	5.93%
Average	2.16%	20.76%	44.99%	34.25%	6.96%

Table 5.7: Percentage-wise analysis of the basic simulation using three manipulators.

Test	*Non-full boxes*	*Balance Robot A*	*Balance Robot B*	*Balance Robot C*	*Lost blobs*
1	4.85%	35.31%	34.65%	30.03%	6.48%
2	6.80%	33.99%	36.63%	32.34%	7.34%
3	6.80%	34.97%	33.00%	32.02%	12.32%
Average	6.15 %	34.75%	34.76%	31.46%	8.71%

Table 5.8: Percentage-wise analysis of the reinforcement learning simulation using three manipulators.

Test	*Non-full boxes*	*Balance Robot A*	*Balance Robot B*	*Balance Robot C*	*Lost blobs*
1	4.81%	22.07%	39.31%	38.62%	9.38%
2	$5.0\bar{5}\%$	22.74%	37.46%	39.80%	10.21%
3	6.38%	23.78%	37.76%	38.46%	14.63%
Average	5.41 %	22.86%	38.18%	38.96%	11.41 %

Table 5.9: Percentage-wise analysis of the initialization test using using three manipulators.

Test	*Non-full boxes*	*Balance Robot A*	*Balance Robot B*	*Balance Robot C*	*Lost blobs*
1	4.12%	23%	45%	32%	8.54%
2	$4.1\bar{6}\%$	18.28%	38.62%	43.10%	10.22%
3	5.10%	21.07%	34.78%	44.15%	15.06%
Average	4.46 %	20.78 %	39.47 %	39.75 %	11.27 %

Table 5.10: Percentage-wise analysis of the no-learning test using three manipulators.

Test	*Non-full boxes*	*Balance Robot A*	*Balance Robot B*	*Lost blobs*
1	2.56%	63.14%	36.86%	4.45%
2	0%	61.78%	38.22%	6.16%
3	1.28%	62.88%	37.12%	4.18%
Average	1.28 %	62.60 %	37.40%	4.93 %

Table 5.11: Percentage-wise analysis of the basic simulation using two manipulators.

Test	*Non-full boxes*	*Balance Robot A*	*Balance Robot B*	*Lost blobs*
1	10.53%	52.29%	47.71%	11.74%
2	9.46%	48.83%	51.17%	11.62%
3	26.09%	42.13%	57.87%	26.14%
Average	15.36 %	47.75 %	52.25 %	16.50 %

Table 5.12: Percentage-wise analysis of the reinforcement learning using two manipulators.

Chapter | 6

Evaluation

This chapter focuses on evaluating how the solution fulfilled the task and answered the problem statement. Moreover, in further development, it is described how the solution could be improved, given more time and resources.

6.1 Discussion

This section will discuss the testing, provide an overview of the results and discuss how the solution performed, from the results in Chapter 5.1.

Several test were made with different focus, to see the consistency of the solution. As it can be seen in all tests, some boxes left the production line while not being completely filled. It was discovered that the reason why this happened, was due to a programming error. The manipulators sometimes think they pick up a blob, without actually doing it. They then proceed to "place" it in a box, which makes the program think that the spot is now taken. However, when the box leaves the line, it is recognized that the box is not full. Unfortunately due to limited amount of time this couldn't be fixed.

Looking at the basic simulation percent-wise results, Table 5.7, it can be seen that it performed, as described in Section 2.3. The workload is distributed randomly and the manipulators just pick up blobs when they were able to. Looking at the average pick up rate from each manipulator, it can be observed that this simulation, only focus was on filling the boxes and ignore balance. That is why Robot A picked up significantly less blobs than the others. Robot A's purpose in this simulation, was to fill up the missing boxes but the other two were able to fill up most of them before they left their workspaces.

The solution featuring reinforcement learning, turned out to improve the load balancing aspect of the task descried in 1.2, compared to the basic simulation. As it can be seen in table 5.8, each manipulator picks up around one third of the total amount. Where the highest difference in percentage between manipulators was 3.3%, when subtracting the lowest percentage pickup from the highest using the average results. On the other hand, when doing the same calculation with the results obtained from the basic simulation, in Table 5.7, the difference in load balance was as high as 24.23 %. Likewise, in the basic simulation with two robots, the average difference percentage wise was 25.2%, whereas when reinforcement learning was implemented, this value decreased to 4.5%. Therefore, it can be seen that the load balance is greatly improved in both cases, regardless of the amount of manipulators.

It was discovered that the solution using reinforcement learning, lost more boxes and blobs due to the need for load balancing. The basic simulation lost on average $2.\bar{1}6$ % boxes and the reinforcement learning lost 6.15%, as seen in the last row, in table 5.7 and 5.8. The amount of boxes lost is worse in the reinforcement learning compared to the basic simulation. This is probably due to the programming error, in addition to the reinforcement agent is still exploring random actions and therefore, may have chosen to stop the last robot, when a non-full box was about to leave.

Moreover the basic simulation lost in in average 6.96% blobs, whereas the reinforcement lost in average 8.71% blobs, this can be seen in the last row in table 5.7 and 5.8. A reason to the higher blob loss in the reinforcement is due to the manipulators having to frequently stop in order to keep up the balance of the system. By doing this, it can happen that Robot A has to stop when a box is about to leave the conveyor in order for the other robots to keep up the balance. When this happens, the conveyor slows down while robot B and C continues to fill up the boxes. At the time Robot A starts again and fills up the box, Robot B and C have been waiting for new boxes to arrive and therefore had no place to put the new blobs, thus failing to pick up some blobs.

As it can be seen in Tables 5.10 and 5.9, the difference in percentage between the robot which picks up the most and the one which picks up the least, improves by 2.85 % after the initialization. If we compare this to the solution featuring the reinforcement learning, it can be seen that there is a difference of 15.67 % between Table 5.8 and the one without learning in Table 5.9. These results, together with Figure 5.1 shows, that the systems can learn over time and the initializer does help the agent figure out the way towards the goal.

6.2 Further development

Because of the limited time and resources to work with the project proposal, some features were left out. This applies not only to the conditions specified in Section 2.5, but also to certain features which were intended to be implemented, but were not. Therefore, if the project were to continue, certain aspects, mentioned below, could be modified or implemented.

- **Agents co-operation** - As stated in 3.2 originally the reinforcement learning was supposed to have two distinct agents. The first one focuses around the distribution of the workload, whereas the second one aims to control the speed of the conveyor. However, it was found out during Section 4.1, that both agents can not co-operate. This is due to the fact that when the agents are on their learning phase, they tend to explore, meaning they take random actions. So when one of the agents explores, the second one will have to compensate for this new action, and take another action, to which the previous agent will have to rectify upon. Therefore, both agents will interfere with each other, making the learning process a complex process. In order to solve this problem, either the agents will have to fully learned independently and once they have learned, they might be able to be implemented together. Or the problem could also be approached from a completely new perspective.
- **Maximize box fill in reinforcement learning** - As it can be seen from Section 5.1, the basic simulation fills up more boxes than the one with reinforcement learning. While the main focus for the project is to load balance the system, from a manufacturing point of view, the intelligent system should not only evenly distribute the workload, but it should be able to produce as many, or more, filled boxes as the one without reinforcement learning. Therefore, for future development, the solution should be programmed to focus more on the boxes and make sure that they are all filled up when leaving the conveyor.
- **Manipulator speed** - Currently, for the load balancing, when a manipulator has a higher pickup count than the others, it will be paused, so the other manipulators can pick up more objects and restore the balance. However, stopping and starting the manipulators is not ideal. If the project were to continue, the manipulators would be controlled based on their speed, slowing down when they have picked up too many objects, and speeding up when they are falling behind.
- **Pickup Algorithm** - As seen in Section 2.5, it was stated that it would be interesting to look further into the pickup algorithm for the manipulators. However, even if this parameter was decided as relevant, it was not possible to include it in the final solution withing the time frame of this project.

- **Fixing errors in the basic simulation** - When comparing the basic simulation to the reinforcement learning simulation, it can be difficult to tell simulation errors apart from errors the agent does. Therefore, in order to have more accurate test results, the simulations should be debugged so errors do not occur.

6.3 Conclusion

This project demonstrates that the load balancing of a robotic system, in a pick and place application, can be achieved through a simulation with the use of reinforcement learning.

This chapter provides an overview of the process of the solution development, how the solution performed while accomplishing the task and concludes on the overall solution. Furthermore, the results of the testing in Section 5.1, will be used to determine whether or not the problem statement was answered.

The aim of this project was to implement a system which learns how to evenly distribute the workload among several manipulators. The task was to fill boxes, on one conveyor with objects from another conveyor. In order to have an unbiased result, two different simulations were created. One of them called the "Basic simulation", which focuses on picking and placing the objects as they come, regardless of the load balance. Whereas the second simulation, aims to fulfill the same task while also accomplishing load balance using reinforcement learning.

The intention of the project, as previously stated, is to use reinforcement learning in a pick and place application. This system is inspired by the modern manufacturing lines. Therefore, it is inspired by Industry 4.0. Aiming to have a system make decisions on its own based on the input received from its environment. In order to achieve this, it uses reinforcement learning, which allows computers to learn without human intervention, thus making the program autonomous. This project is focused around Q-learning and learns based on trial and error, while obtaining feedback based on a reward system. The main goal is to implement load balancing in the setup. This means that in the pick and place application, the work load is supposed to be evenly distributed among the manipulators. This project aims to explore the implementation of reinforcement learning in such a system. Therefore for flexibility purposes, it was decided to develop the solution in a simulated environment. Where V-rep provides the simulated setup, and Matlab is in charge of the reinforcement learning algorithms.

The flexibility of the choice of solution allowed for the discussion of multiple possible approaches. As mentioned in Section 2.5, initially, all parameters and conditions were variables. Since the main focus is the load balance with reinforcement learning, certain parameters and conditions were chosen over others and given a priority based on the authors' criteria. However, had other parameters and conditions been chosen, the solution would have been completely different. These were chosen due to the fact that they were thought to be most interesting in regards to the load balance.

From the testing Section 5.1 and what is mentioned in Section 6.1, it was discussed whether or not the reinforcement learning did perform better than the basic simulation. Certain aspects, such as the number of filled boxes or the lost blobs, obtained better results in the basic simulation. However, the difference in percentage was lower than 4% in both cases. Meaning that even though it did not perform as well, it was not far behind. On the other hand, the load balance when using the reinforcement learning, proved to considerably improve compared to the basic simulation, where the difference on average between the percentage pick up in the reinforcement simulation was 3.3%, whereas in the basic simulation it went up to 24.23% for the tests with three manipulators, as mentioned in Section 6.1. The load balance worked similarly when implementing just two manipulators, proving that the reinforcement learning is consistent regardless of the amount of manipulators.

According to Section 5.1, the solution follows a priority list in order to determine which requirements are more significant to properly fulfill. The first priority is that all the boxes leaving the production line must be full. This was not accomplished since, as mentioned in 5.1, the average of non filled boxes for the reinforcement learning simulation was 6.15% of the total production. However, when comparing it to the basic simulation, it can be seen that this did not accomplished the task of filling up all the boxes either. The reason behind this, is thought to be a bug in the simulation, as mentioned in 6.2, for the basic simulation. While for the reinforcement learning simulation it is thought that this priority is not met due to a combination of the bug and a lower blob-handling rate which occurs from the load balancing. However, since the percentage difference in box loss, between the basic simulation and the reinforcement learning, is less than 4%, it is concluded that even though this requirement was not accomplished, it was a partial success.

The second highest priority, performed significantly better in the reinforcement solution. Making the system balance the workload. This requirement improved so much in comparison to the basic simulation, that it could be said to outweigh the small difference in the first priority.

Lastly the third priority is to have all the blobs in the production line be picked up. And it can be seen in 5.1, that this was not accomplished in either of the simulation. Instead, both them performed very similarly, having a difference of 1.75% of the total amount of blobs. Since this is the third priority and both simulations performed equally well, this requirement is considered accomplished.

Besides analyzing the performance of the system while testing if it is able to fulfill these priorities, it is also evaluated whether or not the solution managed to learn over the course of the training period. In Section 1.2, it is described that this project aims to create a solution able to load balance a pick and place application while using reinforcement learning to do so. As stated in Section 6.1 it seems as the solution is able to accomplish this task, concluding that the learning aspect of the project is fulfilled.

The problem statement described in 2.7, is answered through the contents of this paper and through the tests in Section 5.1. It showcases how to design a system, that enables a robotic simulation setup, to do load-balancing through reinforcement learning, in a pick and place application.

Overall the project did not entirely succeed in all aspects as initially intended. However, the load balancing, which was the main focus of this project, was considered a success. Likewise, the problem statement was answered throughout the report. Leading to the conclusion that this project was a success, getting as close as possible to fulfill all aspects with the time and resources available.

Chapter | 7

Appendix

7.1 Reinforcement learning

Finite Markov Decision Processes In this method the agent must evaluate the immediate and delayed rewards(the return), to choose different action for the right task. So for this Reinforcement learning problem the agent will interact with its environment, this is the state. Using the state for making choices and rewards are for evaluating the choices. The optimal value function is for each state that returns the best expected reward for a policy.
Dynamic Programming
Dynamic Programming(DP) is used to compute the value function and find the optimal policy:

$$v_*(s) = maxE[R_{t+1} + \gamma u_*(S_{t+1})|S_t = s, A_t = a] = max \sum p(s', r|s, a)[r + \gamma u_*(s')] \tag{7.1}$$

With the use of policy evaluation and policy improvement, the agent can compute an optimal/improved policy or value function, this is known as generalized policy iteration (GPI). When a DP methods goes through a state set updating the value of one state based on the probability of a successor in all states, this is known as bootstrapping. Where the agent will perform policy evaluation to change to value function and policy improvement changing the policy for the optimal one. Both helping the agent to obtain a optimal policy.
Monte Carlo Methods
This method those not need a knowledge of the complete environment, but gathers experience from known states, actions and rewards and its based on the returns from each state. Meaning that all state are divided into episodes, and all episodes terminated at some point, this is known as episodic task. This method is better than DP, because it can learn from the environment. It can learn from a simulation or real world task, it can be used on subset of the states and it those not bootstrap. One problem with this is the lack of exploring for action that might be better than the current estimated best action, to solve this a random action should be selected

when starting a new state.
Temporal-Difference Learning
This methods is a combination of DP and Monte Carlo, the methods will bootstrap and can be used without a complete model of the environment. The TD, DP and Monte Carlo, can be combined and used together in different ways. A reinforcement problem can be divided into two problems: A prediction problem (Value function for predicting the return of the policy) and a control problem (Policy to improve the value function), in this case the TD is used for solving a for long-term prediction problem, where DP is used to solve the control problem.
n-step Bootstrapping
The problem with the TD methods is the interval which bootstrapping is done per time-step and when the action can be changed. With the n-step bootstrapping, bootstrapping can be done over multiple steps, and is not depending on a time-step, this will add a delay to the methods when updating it and will require more computational power and memory to store the the states, actions, rewards.
Planning and Learning with Tabular Methods
There are two types of reinforcement learning methods: model-based(like DP and heuristic search) where they rely on planning in their environment ex. A sample model and distribution model is made with probability for the future state(s) and reward(s) for all possible action, this is used in DP. Model-free(Monte Carlo and TD) where they rely on learning their environment. But all methods will look ahead to future events in order to approximate the value function.
All the methods above will do three things: estimate a value function, backing up values from state, generalizing a policy iteration (GPI) and approximate a value function and policy. But one problem for all these methods is large amount of states or data, then these methods is not variable methods of solving the reinforcement problem. So to do this a generalization(function approximation coming from supervised learning) methods is needed to generalize states from previous states to make sensible decisions to the problem(s). One way is to estimate the state-value function, by approximating from a known policy(parameterized function approximation). In these methods the policy is parameterized by a weight vector w, the vector is found by stochastic gradient descent(SGD).
Off/On policy Methods with Approximation
On-policy: The parameterized function approximation and semi-gradient descent can also be used in control part(episodic case) of the reinforcement problems, but not in continuing case, because policies cannot be a value function.
Off-policy Is about learning the value function for a policy from another policy's data. In the prediction case it seeks the state value and in control case action values are learned. The problem here is the target of the update and distribution of the updates.
Eligibility Traces

Is an updated version of the n-step method, where the n-step methods needed to store n number of feature vectors, learned with a delay and had to catch up in the end of an episode. This method only need to store a single trace vector and can learn continually. The reason to use the eligibility traces is that it will converge towards the Monte Carlo methods, a good idea is to use this methods in an online application when data cannot be generated, since for offline application data can be generated meaning Eligibility Traces is not needed.
Policy Gradient Methods
All methods so far used values of actions to select actions from estimated actions values. This methods will then learn a parameterized policy from a value function(policy-gradient methods), so the agent can select an action without the need of an value function. The reason to use policy-based methods, would be for continuous state spaces, since it would be impossible for action-values methods in general.
Sub Conclusion This chapter went through the basic and more advanced methods in reinforcement learning, what problem and advantages each methods had. This will then be used later in the development chapter to make a solution for the problem stated in the section 1.2. It will be used in the analysis, when the different variables that can be changed or are of interest in the the reinforcement problem state, is considered. From that it should be known which of the reinforcement learning methods is of interest in order to solved the task from this chapter.

For tracking a Non-stationary Problem, where the reward probabilities do change over time and where the agent should give more weight to recent rewards and less to old rewards.

This can be calculated by:

$$Q_{n-1} = (1-\alpha)^n Q_1 + \sum \alpha(1-\alpha)^{n-i} R_i = 1 \tag{7.2}$$

This is called the weighted average and using the step-size conditions below this, insures that it will convergence and overcome initial conditions.

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty | \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty \tag{7.3}$$

These methods are biased because of the initial action-value estimates $Q_1(a)$, until all the actions is taken. The initial action-value is used to encourage exploration, so in a 10-arm bandit test example, as seen in Figure 7.1, with 0 for the realistic $\varepsilon - \mathit{greedy}$ function and +5 for the optimistic, greedy function, as initial action-values.

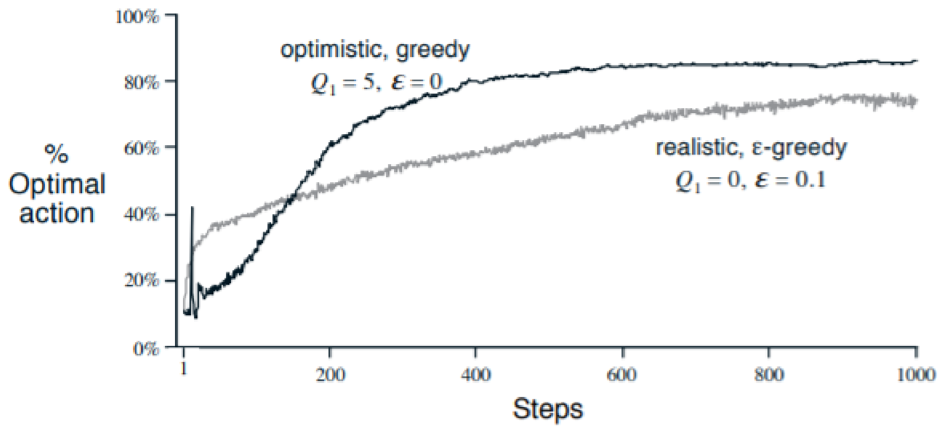


Figure 7.1: 10-armed bandit test optimistic greedy vs realistic $\varepsilon - greedy$ [6]

The optimistic, greedy is worse in the beginning but would perform better in the long run, and the realistic, $\varepsilon - greedy$ will perform better in the beginning but worse in the long run.

But with the problem of not exploring the different non-greedy options for a potential non-greedy action being more optimal (their estimates and uncertainties), the Upper Confidence Bound (UCB) function below is used to select the actions:

$$A_t \doteq argmax[Q_t(a) + c\sqrt{\frac{ln(t)}{N_t(a)}}], \tag{7.4}$$

Where the: $ln(t)$, is the natural logarithm of t, $N_t(a)$, is the number of times an action is used at time t and $c > 0$, controls the exploration. In a 10-armed bandit test the UCB vs $\varepsilon - greedy$, the UCB is better at selecting the action.

In a 10-armed bandit test. The UCB is setup up anaginst the $\varepsilon - greedy$,

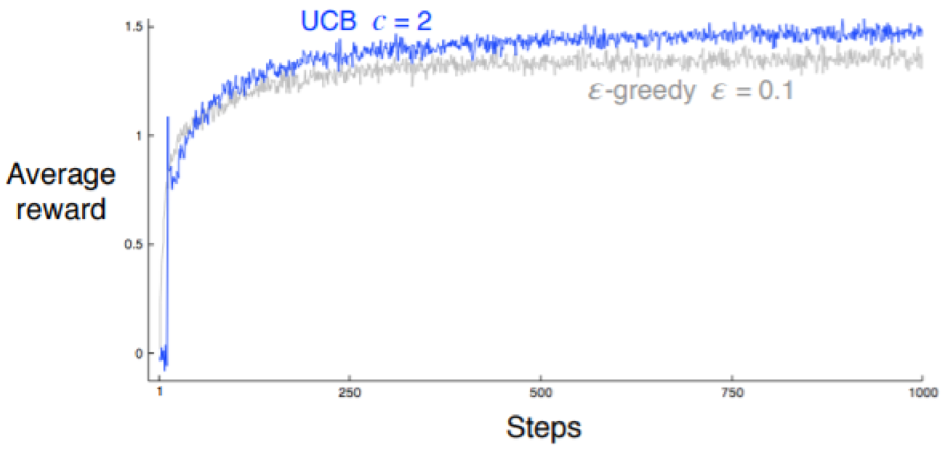


Figure 7.2: 10-armed bandit test with UCB vs $\varepsilon - greedy$ [6]

In Figure 7.2 it can be seen that the UCB performed better than $\varepsilon - greedy$ in the long run. This method is often used in simpler non-stationary problems, it will not work or is less useful for advanced non-stationary problems. For all these methods the estimates are used to select the action. However, another way of doing this is to use a numerical preference for all actions, $H_t(a)$. The larger the number, the more often the action is selected.

$$Pr(A_t = a) \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a) \tag{7.5}$$

$\pi_t(a)$ is the probability of taking action a, at time t. The algorithm of 7.5 will work like seen in 7.6 and 7.7 , when selecting actions and receiving the reward.

$$H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \dot{R}_t)(1 - \pi(A_t)) \tag{7.6}$$

$$H_{t+1}(a) \doteq H_t(a) - \alpha(R_t - \dot{R}_t)\pi_t(a) \tag{7.7}$$

The Equation 7.7 for all $a \neq A_t$ and the R_t is a baseline. The selection is based on the expected rewards with a normal distribution and a mean of +4. Then if the reward for action a is higher than the baseline. Then the probability of taken action a, is higher and vise versa. This will in a 10-armed bandit test give a better overall selection of the optimal action as seen in Figure 7.3.

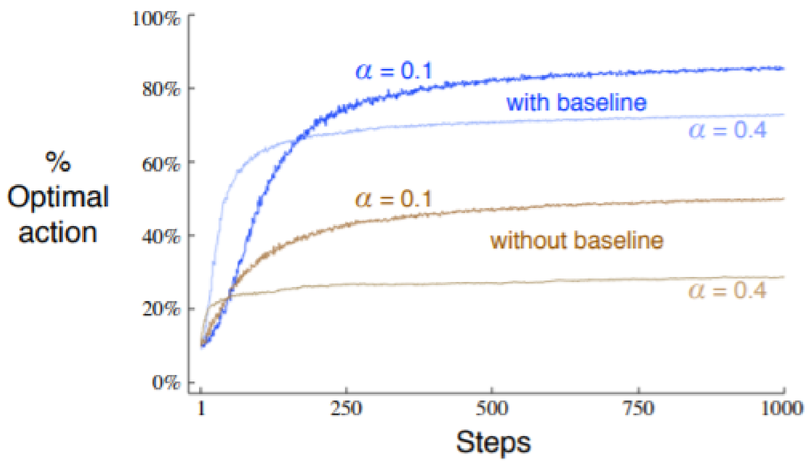


Figure 7.3: 10-armed bandit test, with base line for reward for selecting the optimal action. [6]

In Figure 7.3 it can be seen that using a baseline for selections of actions, performs better than not having a baseline. This should be used in combination with a one of the other methods of calculating the optimal action value or policy.

All these methods are for a k-armed bandit tasks, with one task and one optimal action for that task. Then if the task and the optimal action can change over time, this is called associative search task or contextual bandits. To solve these k-armed bandit tasks a policy that can map the states to action and then action to rewards, must be used. This section went through ways of balancing exploration and exploitation: The $\varepsilon - greedy$ and UCB methods for choosing an action and exploration, with the use of action preference, were covered. To compare these methods, a 10-armed bandit test is used in Figure 7.4:

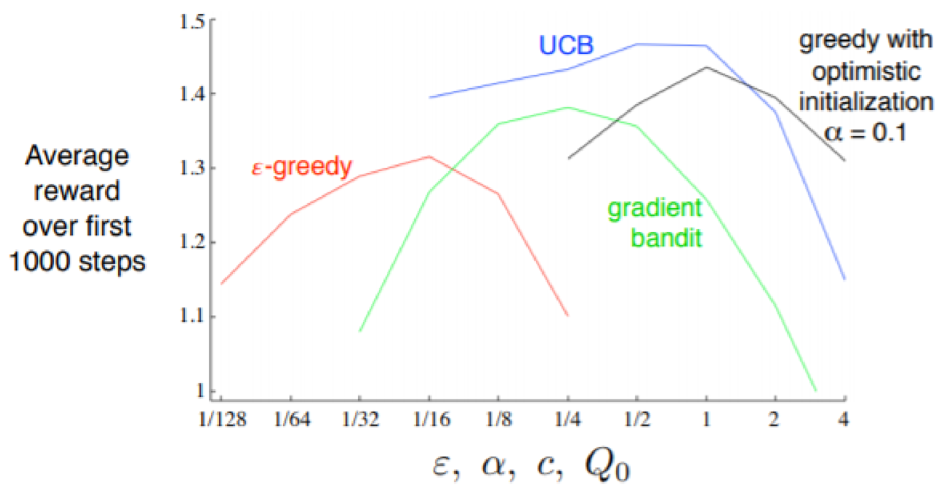


Figure 7.4: Methods compared from this section in a 10-armed bandit test [6]

In Figure 7.4 a comparison between the $\varepsilon - greedy$, optimistic-greedy, UCB, gradient bandit, in a 10-armed bandit test, was done. Here it can be seen that the best performing with an average reward over 1000 steps, is the UCB.

An algorithm that can be used to help the agent to choose the actions that gives the highest reward in the long run, at any given state is Q-learning. Q-learning is a algorithm that aims to reach the state with the highest reward possible, when or if the agent reaches this state, it will try to remain there. The Q-learning algorithm can be seen in Equation 7.8.

$$QR_{new} = (1 - learnRate) * QR_{old} + learnRate * (RR_{reward} + discount * max(QR)); \tag{7.8}$$

It does this by storing the expected reward for each action in every possible state. And then forms its path by executing those actions with the highest expected reward and saves the reward it gets in the state the agent is in. An example of this could be a maze. See Figure 2.3. In this example the agent has to go from one side of a maze to the other side and indicated by arrows. It does not know the environment and have to learn through experience by taking any route and see if it leads through or not. The quicker it gets though the better the reward. Each time it hits a dead end, it will get punished. After having attempted several routes it should know the fastest way out and would chose this route every time, since the reward is highest here.

Bibliography

[1] Rasmus Skovgaard Andersen and Simon Bøgh. Reinforcement learning approach for load-balancing a multi-robot pick-and-place line, 2018. 29-05-2018.

[2] Prof. Dr. Wolfgang Wahlster Prof. Dr. Henning Kagermann and Dr. Johannes Helbig. Recommendations for implementing the strategic initiative indusrtie 4.0. *Forschungsunion and acatech*, 82:page from still to, 2013. 15.12.2016.

[3] David Dechow. Vision guided robotics: Techniques for pick and place applications. Webinar. 26-03-2018.

[4] Emilio Frazzoli Marco Pavone, Stephen L. Smith and Daniela Rus. Load balancing for mobility-on-demand systems. *MIT Press*, pages 249 – 256, 2012. 11-04-2018.

[5] Amir Masoud Rahmania Einollah Jafarnejad Ghomia and Nooruldeen Nasih Qaderb. Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*, 88:50–71, 2017. 06-04-2018.

[6] Richard S. Sutton Andrew G. Barto. Reinforcement learning: An introduction. PDF. 27-03-2018.

[7] Karl Morrison. Supervised learning, unsupervised learning and reinforcement learning: Workflow basics. Webinar. 27-03-2018.

[8] Leon Zlajpah. Simulation in robotics. *Mathematics and Computers in Simulation*, 79, 2008. 17-04.2018.

[9] Donald C. Craig. Extensible hierarchical ob ject-oriented logic simulation with an adaptable graphical user interface. *School of Graduate Studies*, 1996. 25-04-2018.