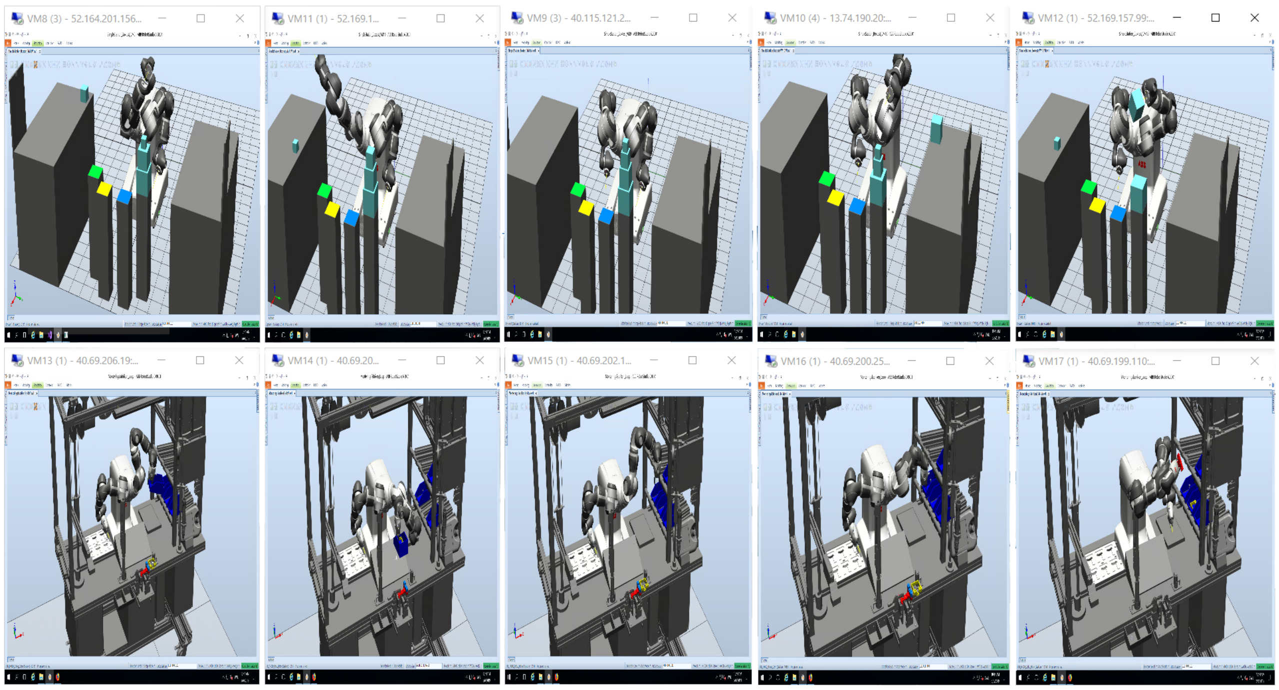
CHALMERS
UNIVERSITY OF TECHNOLOGY



Sequence optimization using reinforcement learning in RobotStudio

Finding the optimal assembly sequence based on behavioral adaptation and production lead time

Master's thesis in Systems, Control & Mechatronics

Oliver Lindberg
Arsam Shokrian

Co-Supervisor: Constantin Cronrath
Supervisor: Kristofer Bengtsson
Examiner: Bengt Lennartson

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019: EENX30

Sequence optimization using reinforcement learning in RobotStudio

Finding the optimal assembly sequence based on behavioral adaptation and production lead time

Oliver Lindberg
Arsam Shokrian

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2019

Sequence optimization using reinforcement learning in RobotStudio
Finding the optimal assembly sequence based on behavioral adaptation and production lead time
OLIVER LINDBERG
ARSAM SHOKRIAN

Co-Supervisor: Constantin Cronrath, Department of Electrical Engineering
Supervisor: Kristofer Bengtsson, Department of Electrical Engineering
Examiner: Bengt Lennartson, Department of Electrical Engineering

Master's Thesis 2019:EENX30
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Multiple instances of RobotStudio simulations run in parallel for rapid environment exploration and learning.

Typeset in $\LaTeX$
Gothenburg, Sweden 2019

Sequence optimization using reinforcement learning in RobotStudio
Finding the optimal assembly sequence based on behavioral adaptation and production lead time
OLIVER LINDBERG
ARSAM SHOKRIAN
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Collaborative robots of today are distinguished from traditional industrial robots by the fact that they can work safely hand in hand with humans. They can slow down or stop when people come too close, collide without causing injuries and be guided to move in directions directly imposed by a person. To augment the collaboration, this thesis proves that RL can be utilized to make a robot observe patterns in the simulated behavior of a human operator and learn how to adapt its own motions in order to optimize the assembly process. This can be combined with learning different optimized assembly sequences depending on the learned preferences of the human operator. To achieve this, tabular Q-learning, linear- and nonlinear function approximation have been evaluated. Furthermore, challenges and possibilities of shorten training process through parallelization have been investigated.

The results suggest that tabular Q-learning finds the global optimum faster than both function approximation methods. However, Q-learning with nonlinear function approximation has the ability to generalize to an unlimited number of human behaviour profiles, which is unreasonable with both linear function approximation and tabular Q-learning. Furthermore, different parallelization strategies such as centralized/distributed learning coupled with synchronized/asynchronous actors have successfully been implemented and compared. Although some results remain inconclusive, it is clear that all strategies have the ability to speed up learning and increase model accuracy while they compare differently depending on the problem complexity and the number of parallel training instances. It has been found that faster convergence using parallelization correlates with larger error, which distinguishes the distributed synchronized learning strategy that explores more intelligently than the asynchronous counterpart. The choice of strategy becomes increasingly important for more complex problems and higher number of instances.

The learning algorithms have been applied to a simulated environment in RobotStudio. However, identical communication tools between the learning agent and the robot controller has been found possible to use for both the virtual robot and the real robot which simplifies transferability. The results combined are promising and motivate continued research to advance the development of the next level of intelligent robots.

Keywords: Artificial Intelligence, Machine Learning, Reinforcement learning, Collaborative robots, Closed loop manufacturing

Acknowledgements

At first, we would like to express gratitude to our supervisor Kristofer Bengtsson, PhD in the Automation research group at Chalmers University of technology, for his guidance and deep insights in the area when we created this thesis work. He contributed greatly to defining applications that motivated the use of reinforcement learning and showed interest and passion throughout the work. For that, we would also like to thank Constantin Cronrath, PhD student in the Automation research group at Chalmers with focus on Reinforcement Learning in industrial contexts who also supported us with similar research that has been valuable to this thesis. Furthermore, we would like to thank Johan Bengtsson and Johannes Persson, project managers at Smarta Fabriker and our supervisors, who provided us with tools crucial for this thesis to become what it is today. They have also put great effort in getting the thesis acknowledged externally, both at the Hannover industrial fair and at the Ericsson office at Lindholmen, Gothenburg. For that we are sincerely grateful. Finally, we want to express great appreciation to our supervisors at ABB, Magnus Seger and Oskar Henriksson, Global Application Technology Manager Automotive Products and Software Engineer respectively, who provided an office, taught us RobotStudio and were always taking their time to assist us in debugging. They ensured that our work proceeded smoothly. It has been a huge privilege to work on this project and we thank you all for showing such great enthusiasm for our work.

Oliver Lindberg, Arsam Shokrian, Gothenburg, May 2019

Contents

List of Figures

List of Tables

Glossary of Acronyms and Terms

ANN	Artificial Neural Network
Azure	Cloud computing service developed by Microsoft
ML	Machine Learning
MSE	Mean Square Error. In this project, when referring to MSE, the error is always the TD-error
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RMSprop	Optimization algorithm commonly used in the machine learning field
RobotStudio	Simulation software for robot simulation and off-line programming developed by ABB
SGD	Stochastic Gradient Descent
TD	Temporal Difference
VM	Virtual Machine/s

1
Introduction

Companies like ABB, GE, Siemens, Intel, Funac, Kuka, Bosch, NVIDIA and Microsoft are all investing substantially in *machine learning* (ML) approaches to make their manufacturing processes more efficient [1], [2]. Simultaneously, a similar investment is seen in collaborative assembly stations where companies like ABB, Kuka and Festo see this as another way to improve their manufacturing systems [3]. In those collaborative stations, the conventional way of an operator learning how to optimally assemble a product together with a robot is expensive since there usually are many possible actions and finding the sequence of actions that efficiently produces a high quality product takes time and effort. Additionally, if the operator finds an optimal solution, it will still only have been tailored for that specific operator and other operators would have to find their own optimal strategies. However, one can imagine a corresponding computational approach where an algorithm teaches the collaborative robot to find optimal strategies for different kinds of operators and products by testing actions in a simulated environment, and in effect creating an intelligent collaborative robot. In this thesis, the use of *reinforcement learning* (RL) is evaluated to achieve this, where an algorithm can learn which action to take given a certain situation in order to maximize or minimize a numerical reward signal [4]. Reward signals in production applications may be production lead time, which should be minimized, or some quality measure to be maximized by choosing different actions such as options among feasible assembly operations.

Sutton, Barto and Bach [4] write that an RL method is suitable for problems with the following three aspects; sensation, action and goal. They explain that a learning agent, also called a learner, must sense the state of its environment, be able to take actions that affect the state and it must have a goal related to the state. This relates closely to what O'Hare [5] describes as key aspects of productivity and *closed loop manufacturing*. He writes that in manufacturing systems, there are three factors with vast impact on productivity; "sensing, thinking and acting". By integrating manufacturing intelligence tools in a strategic way, the tools assist in collecting and understanding data, implementing changes to a manufacturing facility and effectively achieving a closed loop manufacturing system. The term closed loop manufacturing itself carries certain ambiguity as it is widely used and different definitions can be found. In this project the term was used when data collected from a manufacturing facility is processed by a machine learning algorithm which produces a result that is implemented in the factory. While the aforementioned aspects are defined for the physical world, this project realizes the same closed loop process with RL in a simulated environment and builds a solid foundation for possible expansion into real world closed loop manufacturing. The reason why the closed loop is realized in the simulated environment instead of directly to the real one, is because of the fact that implementing new ideas and methods in physical processes and stations frequently takes more time and money and has a high probability of equipment errors. Furthermore, developing the solutions in a simulated environment allows expansion and flexibility that is not possible in the real world, e.g. running multiple simulation instances at the same time is considerably easier than building multiple assembly stations and doing the same. It is therefore more valuable to initially develop the simulated closed loop, experiment and explore using it, and then implement those findings in the real world.

1.1 Project purpose and overview

In accordance to the previous section, the purpose of this project is to investigate the possibility of using RL to improve manufacturing processes by automatizing the identification of optimal assembly sequences in collaborative stations and enabling simulated closed loop manufacturing. Since efficiency is a key aspect of this project, the ability to identify the optimal sequences as fast as possible is also desired. The most promising way to do this is to use parallel simulation instances, because those can essentially be infinitely duplicated.

To encapsulate the purpose of this project, the following three research questions have been developed that this thesis will answer:

1. Can RL be used on a simulation model of a real assembly station to create a simulated closed loop manufacturing system? What is required so that the simulated closed loop can be transferred to the real system and a true closed loop manufacturing system be achieved?

2. Is it possible to teach a collaborative robot both an assembly sequence for a product and to recognize patterns in the behavior of a human operator and adapt to them, finding the optimal sequence of actions to assemble a product for different kinds of operators? Are there different strategies that are better suited for certain scenarios? How do these strategies compare to traditional optimization methods?

3. How much can the data gathering, learning and exploration of the RL algorithm be improved by using parallel simulation instances? How should the parallel instances communicate in order to optimize efficiency and results?

In order to answer these questions, the thesis has been divided into three intrinsically distinct parts. In Section 3, a simulation model in RobotStudio is presented with the purpose of evaluating product assembly optimization. In consonance with the aspects mentioned by Sutton et al, the simulation model allowed the agent to sense the current state of the ongoing assembly, choose an action or an assembly operation based on the state and its experience affecting the environment and being able to reach the goal state which was a complete and correct assembly of the product. The result of the RL algorithm is an optimal operation sequence that determined the best action for each state. The second part, in Section 4, investigates parallelization using different orchestration strategies to merge and control the parallel simulation instances. The same model as in Section 3 was used in order to have easier modification and implementation possibilities. Convergence time and error was compared between 1, 5 and 10 parallel instances using the different orchestration strategies. Section 5 is the third part of the thesis and it concerns the implementation of different RL algorithms to teach a collaborative robot to adapt to varying operator profiles. For this part, a model was created based on a physical collaborative assembly station developed through a strategic alliance between Smarta Fabriker and ABB. This model was used for investigating both closed loop manufacturing systems with increased transferability to the physical assembly station, and which RL algorithms were suitable for human behavioral adaptation.

1.2 Scope and implementation

To provide more detail to the overview of the project described in the previous section, a basic explanation of the implementation of the project's different parts is presented below. These refer to sections 3, 4 and 5 respectively.

Learning optimal product assembly sequence A model consisting of a YuMi robot with the ability to pick four boxes from different locations and place them at one of four eligible locations was developed. The boxes are hollow and open at the bottom so that a smaller box placed on top of a larger box builds a tower but not vice versa. In the model there are sensors at four different heights on solely one location and the goal is to activate all sensors. Due to the construction of the boxes and the placement of the sensors, this can only be achieved by picking and placing the boxes in the correct sequence on the correct location. With no information about the sensors and the geometry of the boxes an RL algorithm will learn how to achieve this goal.

Parallelization in reinforcement learning applications Since the simulation of actions is the main bottleneck in the learning process, issues and possibilities concerning multiple parallel training instances has been investigated. For this, the model described above was used together with virtual machines (VM) on Azure's cloud service, making it easy to simulate many parallel instances of RobotStudio and observe the effect of different learning strategies for different problem complexities.

Training adaptable robot for collaborative assembly A simulation model representing the assembly station developed and built by Smarta Fabriker was created. In the RL algorithm for this model, states carried information about the assembly progression along with observations of the human operator that cooperated with the robot in the assembly. These observations served as a basis for the robot to choose actions best suited to the particular operator. In the station a YuMi robot and a human operator assemble a product consisting of three product parts, two springs and three screws. The product has no practical value but was constructed to demonstrate a collaborative process between a human and a robot.

The algorithms used in the models mentioned above have been trained on specific products and in specific environments. They can therefore not be directly used in other applications. However, the underlying ideas and the conclusions reached from these specific applications may be generalized to other scenarios. Simulation speed, number of parallel instances, data storage and processing are limited to the capacity of the VM available in Azure within the budget given by Smarta Fabriker. Furthermore, the assembly sequences that are generated by the algorithms will be based on factors such as maximizing the number of activated sensors, minimizing production lead time and synchronizing simultaneous operations. Note that anything measurable could be a possible factor to affect the outcome of the algorithms.

2

Background

The theoretical background for this thesis is primarily rooted in the field of RL, however it also relates to general ML theories used in the different implementations of the RL algorithm. Three of the most basic methods for implementing RL is dynamic programming, Monte Carlo methods and temporal difference learning [4]. Dynamic programming requires full knowledge of the environment which makes it impractical for the applications in this project, specially because it is difficult or even impossible to define a complete and accurate model of how an operator behaves in an assembly station. Further, Silver compares Monte Carlo and temporal difference methods and points out that TD-methods are a bit biased and sensitive to the initial values but usually more efficient than Monte Carlo [6]. TD is also fully incremental which is advantageous in this project with tedious actions. For a more exhaustive exposition of the models, the reader is advised to read the book *Reinforcement learning - An Introduction* by Sutton, Barto and Bach [4].

More specifically, Q-learning will be explained and implemented in this project, which is the most common TD method to implement RL. In order to evaluate the advantages of this approach, there is a need to cover theories behind traditional optimization and planning and place this project in the context of the field today.

2.1 Tabular Q-learning

One can implement RL in multiple ways, with one of the most established methods being Q-learning in which each action relates to a quality value that reveals how good it is to do that action in a certain state. To understand the algorithm, a number of terms will be explained: *reward*, *policy*, *value-function* and *action-value* or *Q-value*. This exposition can be compared to [4, Chapter 1.3 and 6.5]. The idea behind the reward function is that the algorithm should get positive reinforcement for doing things well as defined by the programmer. The value function describes how good a particular state is, which in turn is decided by how much reward is attainable from that state. The policy of the algorithm is what determines with what probability certain actions will be conducted at certain states. One common policy is the ϵ-greedy policy. The idea is that at any state with probability ϵ choose an action randomly, and therefore with probability $1 - \epsilon$ choose an action that maximizes the

Q-values. One uses this policy so as to avoid local optima and to combine both exploration and exploitation. Finally, the Q-value function is to actions what the value function is to states, i.e. it tells how much reward can be attained from doing a certain action.

Q-learning is an off-policy *temporal difference* (TD) control algorithm which is defined, with greedy target policy and ϵ-greedy behavior policy, in algorithm 1. Off-policy learning means that the algorithm does not necessarily follow the policy, but tries other actions and then compares them to the policy in order to see which ones are better. The policy that it compares to is called the target policy and the policy that it actually follows is called the behavior policy. TD learning, unlike Monte Carlo where the terminal state has to be reached before evaluation, is the idea that one can evaluate the Q-value after each action. This is called *bootstrapping* and one important result is that regardless of which policy is applied, the action-value function Q will converge to the optimal action-value function q_*, which consists of action-values $Q(s, a)$ that are optimal for each state and action [4].

Algorithm 1 Tabular Q-learning

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, $\forall s \in \mathcal{S}$, $a \in \mathcal{A}$, arbitrarily except that $Q(terminal, \cdot) = 0$
repeat (for each episode)
    Initialize $s$
    repeat (for each step in episode)
        Choose $a$ from $s$ using policy derived from Q (e.g. $\epsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
        $s \leftarrow s'$
    until $s$ is terminal
until convergence of Q-values
```

The idea behind algorithm 1 is that it starts with predefined values for step size/learning rate α, and ϵ. Then the Q-values are initialized arbitrarily for all possible states and actions, except for the terminal Q-value which must be zero, resulting in a table of Q-values. What happens next is that the initial state is chosen and the algorithm performs an action as defined by its behavior policy. It then updates the Q-value for the state it came from and the action it performed based on the old Q-value, the reward r, the discount factor γ, and a Q-value for the next state s' from an action chosen by its target policy. It will perform these behavior policy actions until it reaches the terminal state, after which it will start over from the initial state but with an updated Q-value table. This will repeat until the Q-value table no longer changes over time and converges.

2.2 Function approximation and deep Q-learning

The tabular Q-learning algorithm described in Section 2.1 is most suitable for a small number of states and actions because in order to learn a Q-value for a certain (s,a) pair, that element must be visited during training. For an increasing number of states and actions, the probability of visiting a certain element in the Q-matrix decreases and for continuous states this probability is zero. Thus many elements in the Q-matrix will never be updated. However, it is often the case that for models with large amount of states, the desired behavior for a state is similar to the behavior of neighboring states. This means that it would be possible to approximate a Q-value based on neighboring states even though it has never been visited. For example, if a state of an inverted pendulum includes the angle of the rod and the goal is to balance it, the best choice of direction in which to move the rod would be the same even if the angle were a few degrees different.

More formally, instead of representing the Q-values as a lookup table they will be represented as a parameterized functional $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$ where π is a known policy and $\mathbf{w}$ is a weight vector. The weights can be chosen to represent a suitable model for function approximation. They can be the weights in a linear combination of features important for approximating the action-value function, or they can be connection weights in a multi-layer *artificial neural network* (ANN) that takes the state as input and gives the approximated Q-values as output in a non-linear mapping. *Deep Q-learning* refers to the applications where deep neural networks are used to translate states into Q-values. Linear models are efficient when it comes to both data and computation whereas nonlinear function approximation using multi-layer ANNs has been a vital part in many of the applications that have gained much attention in recent years such as DeepMind's AlphaGo [4].

According to Sutton, Barto and Bach [4], the tools discussed so far cannot be implemented without the risk of instability for the updates. They denote the three elements of function approximation, bootstrapping update targets and off-policy training as the *Deadly Triad* and explain that instability can only be avoided if a maximum of two elements of the deadly triad are used at once. This statement will be considered and investigated in this project.

2.2.1 Linear action-value function approximation

Theory and conclusions in this section can be compared to [4, pp. 195-241]. In linear action-value function approximation the state and action is represented as a feature vector $\mathbf{x}(s, a)$ and the approximation is in turn defined as

$$\hat{q}(s, a, \mathbf{w}) = \mathbf{x}(s, a)^T \mathbf{w}. \tag{2.1}$$

During training of the reinforcement algorithm the weights can be updated to minimize the mean squared error $J(\mathbf{w})$ between the approximation $\hat{q}(s, a, \mathbf{w})$ and the true action-value function q_π resulting in

$$J(\mathbf{w}) = \mathbb{E}[(q_\pi(s,a) - \hat{q}(s,a,\mathbf{w}))^2]. \tag{2.2}$$

Using *Stochastic Gradient Descent* and taking a step $\Delta\mathbf{w}$ in the opposite direction of the gradient of $J(\mathbf{w})$ w.r.t $\mathbf{w}$, results in the update rule

$$\Delta\mathbf{w} = \alpha(q_\pi(s,a) - \hat{q}(s,a,\mathbf{w}))\nabla_w\hat{q}(s,a,\mathbf{w}), \tag{2.3}$$

where $\nabla_w\hat{q}(s,a,\mathbf{w})$ is the gradient of $\hat{q}(s,a,\mathbf{w})$ w.r.t $\mathbf{w}$. Using equation (2.1) this simplifies to

$$\Delta\mathbf{w} = \alpha(q_\pi(s,a) - \mathbf{x}(s,a)^T\mathbf{w})\mathbf{x}(s,a). \tag{2.4}$$

Lastly, since the true action-value function $q_\pi(s,a)$ is unknown it must be replaced with a *target*. The target in TD-learning is $r + \gamma\hat{q}(s',a',\mathbf{w})$, where a' is chosen to be the action that results in the largest approximated Q-value for state s'. This results in the final update rule

$$\Delta\mathbf{w} = \alpha(r + \gamma\mathbf{x}(s',a')^T\mathbf{w} - \mathbf{x}(s,a)^T\mathbf{w})\mathbf{x}(s,a). \tag{2.5}$$

Note that when using bootstrapping in RL the target $r + \gamma\hat{q}(s',a',\mathbf{w})$ depends on $\mathbf{w}$. However, when calculating the gradients in the back-propagation method the targets $q_\pi(s,a)$ are considered as constants w.r.t. the weights. This means that the effect on changing the weights is only taken into account on the estimate and not on the targets. Therefore, this method is not by definition the true SGD and is thus sometimes referred to as a *Semi-gradient method* [4].

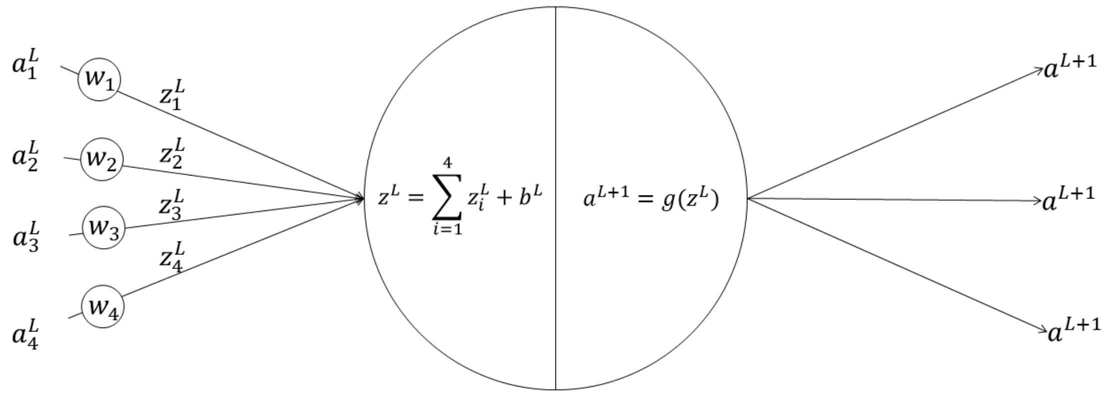
The method is implemented according to algorithm 2.

Algorithm 2 Linear function approximation

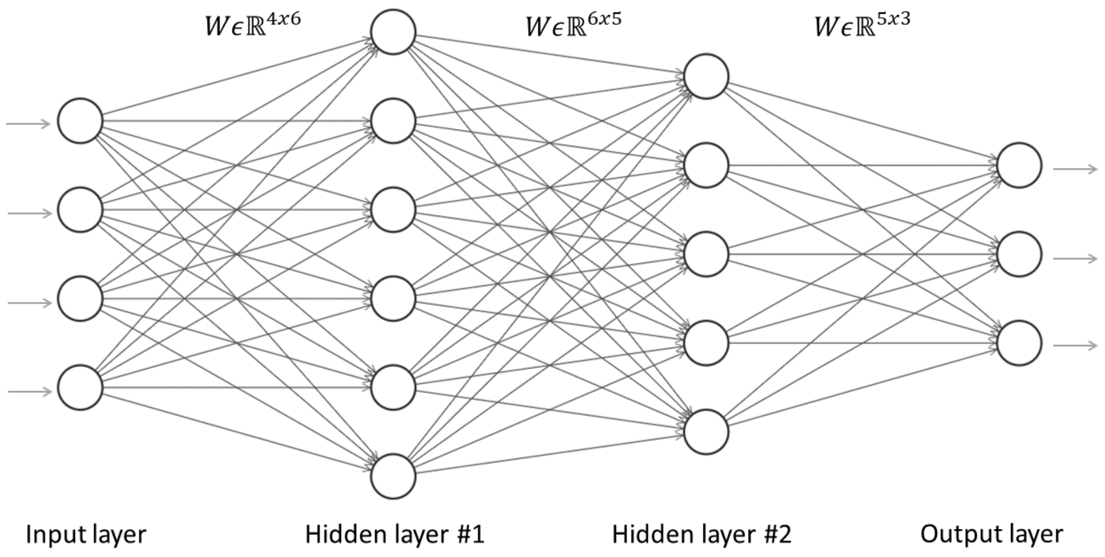
```
Algorithm parameters: step size $\alpha$, small $\epsilon$, discount factor $\gamma$, loss threshold $\delta$
Initialize $\mathbf{w}$ arbitrarily
repeat (for each episode)
    Initialize $s$
    repeat (for each step $i$ in episode)
        Choose $a$ from $s$ using policy derived from $\hat{q}$ (e.g. $\epsilon$-greedy)
        Take action $a$, observe $r$, $s'$, $a'$
        Form the feature vectors $\mathbf{x}(s,a)$ and $\mathbf{x}(s',a')$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha(r + \gamma\mathbf{x}(s',a')^T\mathbf{w} - \mathbf{x}(s,a)^T\mathbf{w})\mathbf{x}(s,a)$
        Store loss in vector $\mathbf{J}(i) = (r + \gamma\mathbf{x}(s',a')^T\mathbf{w} - \mathbf{x}(s,a)^T\mathbf{w})^2$
        $s \leftarrow s'$
    until $s$ is terminal
    Compute mean of loss vector $J(\mathbf{w}) = mean(\mathbf{J})$
until $J(\mathbf{w}) < \delta$
```

2.2.2 Nonlinear action-value function approximation

(a)



(b)



$W \epsilon \mathbb{R}^{4x6}$
$W \epsilon \mathbb{R}^{6x5}$
$W \epsilon \mathbb{R}^{5x3}$
Input layer
Hidden layer #1
Hidden layer #2
Output layer

Figure 2.1: Figure 2.1a shows one neuron in a hidden layer. In this case the preceding layer has four neurons and the succeeding layer has three neurons. The weights w_i in circles are multiplied with the respective activation a_i^L producing z_i^L. Figure 2.1b depicts an entire neural network consisting of neurons.

Instead of manually constructing the features as described in the previous section, training the layers of an ANN provides an automatic creation of features appropriate for a given application [4]. The features are represented in the hidden layers as connection weights which usually are updated using back-propagation in order to minimize (or maximize) an objective function such as TD-errors, defined in equation 2.6, or expected rewards. This method is sometimes referred to as *deep reinforcement learning* due to the use of multiple layers in a neural network for function approximation. Feed forward deep neural networks are used in this project. They consist of simple computational elements called neurons, see Figure 2.1a. Each neuron exists in one of the network's layers and a neuron in one layer is typically connected to neurons in the neighbouring layers. If each neuron is connected to every neuron in the neighbouring layers the network is called *Fully connected.* Figure 2.1b depicts a fully connected feed forward neural network.

In a fully connected neural network each neuron's input is a linear combination of the outputs a_i^L from each neuron in the previous layer using weights w_i associated with the particular neuron. The result z is passed through an activation function $g(z)$ usually a sigmoid function, hyperbolic tangent (tanh) or a rectified linear unit (ReLU). This is necessary in order to break the linearity in the network; without nonlinear activation functions the network would only be able to perform linear mappings [7]. The output, often called activation are passed as input to each neuron in the next layer. Further details about neural networks used in this project will be covered in the upcoming sections. A more extensive introduction of feed forward neural networks can be read in the article written by Svozil, Kvasnicka and Pospichal [8].

Letting the network represent a nonlinear function approximating the Q-values taking the feature vector defined similarly as for linear function approximation as input and scalar $\hat{q}$ as output, algorithm 3 can be implemented.

Algorithm 3 Nonlinear function approximation

```
Algorithm parameters: Optimizer parameters (see Section 2.2.2.2), activation function, #hidden layers, #hidden units/neurons in each layer, small $\epsilon$, discount factor $\gamma$, loss threshold $\delta$

Initialize network weights arbitrarily
repeat (for each episode)
    Initialize $s$
    repeat (for each step $i$ in episode)
        Choose $a$ from $s$ using policy derived from $\hat{q}$ (e.g. $\epsilon$-greedy)
        Take action $a$, observe $r$, $s'$, $a'$
        Form the feature vectors $\mathbf{x}(s, a)$ and $\mathbf{x}(s', a')$
        Forward propagate $\mathbf{x}(s, a)$ and $\mathbf{x}(s', a')$ through network
        Retrieve $\hat{q}(s', a', \mathbf{w})$ and $\hat{q}(s, a, \mathbf{w})$
        Update network weights using backpropagation (see section 2.2.2.1)
        Store loss in vector $\mathbf{J}(i) = (r + \gamma\hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))^2$
        $s \leftarrow s'$
    until $s$ is terminal
    Compute mean of loss vector $J(\mathbf{w}) = mean(\mathbf{J})$
until $J(\mathbf{w}) < \delta$
```

2.2.2.1 Backpropagation

The network must be trained to be able to map the input to the desired output. This is performed by updating each neuron's weights in the direction that reduces the loss function $J(w)$ that is to be minimized. The loss function in the case of value function approximation in RL can be the same as in the linear case i.e. the mean squared TD-error

$$J(\mathbf{w}) = (r + \gamma \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))^2. \tag{2.6}$$

To achieve this, *Stochastic Gradient Descent* (SGD) is a common method (often referred to as optimization algorithm or *optimizer*) which for a sample or a batch of the training data set updates the weights by taking a step α in the opposite direction of the loss gradient, i.e.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} J. \tag{2.7}$$

For a certain layer L corresponding to the left side of Figure 2.1a it holds by the chain rule that for each weight w_k^L in that layer

$$\frac{\partial J}{\partial w_k^L} = \frac{\partial J}{\partial z_k^L}\frac{\partial z_k^L}{\partial w_k^L} = \frac{\partial J}{\partial a^{L+1}}\frac{\partial a^{L+1}}{\partial z_k^L} a_k^L = \frac{\partial J}{\partial a^{L+1}}\frac{\partial g}{\partial z_k^L} a_k^L, \tag{2.8}$$

where most interestingly

$$\frac{\partial J}{\partial a^{L+1}} = \sum_i \frac{\partial J}{\partial z_i^{L+1}}\frac{\partial z_i^{L+1}}{\partial a^{L+1}} = \sum_i \frac{\partial J}{\partial z_i^{L+1}} w_i^{L+1} \ . \tag{2.9}$$

This shows that the gradients in layer L depends on the gradients and the weights of the next layer $L+1$. Hence, it is natural to start in the last layer, compute

$$\frac{\partial J(\mathbf{w}_N)}{\partial \mathbf{w}_N} \tag{2.10}$$

where $\mathbf{w}_N$ are the weights in the output layer and then proceed backwards through the network until all gradients are calculated and the full weight update can be computed. As a result, the output error propagates from the output layer through the hidden layers to the input layer which is why the algorithm is called backpropagation [8].

2.2.2.2 Optimizers

As described above, SGD is a common choice of optimizer which is due to its simplicity. For many applications, other optimizers performs much better. Ruder describes many of them, such as *Adagrad*, *RMSprop* and *Adam* [9]. For many applications, specially with large networks and sparse gradients, he recommends the adaptive learning rate methods in general and Adam in particular. Adam has proven to be a proper choice for this application and is thus the one explained in this section. The name is derived from adaptive moment estimation and was designed by D. P.

Kingma and J. Lei Ba to combine the advantages of RMSprop and AdaGrad [10]. For the algorithm, they recommend the following tuning parameters: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$. The authors observed that since $\mathbf{m}_t$ and $\mathbf{v}_t$ are initialized to zeros they are biased towards zero, specially in the early time steps, and when β_1, β_2 are close to 1. They propose a correction for this in the way that $\hat{\mathbf{m}}_t$ and $\hat{\mathbf{v}}_t$ are calculated below. The algorithm is presented below.

Algorithm 4 Adam

```
Require: $\alpha$: learning rate
Require: $\beta_1$, $\beta_2$: $\in [0, 1)$ Exponential decay rates for the moment estimates
Require: $J(\mathbf{w})$: Loss function with parameters $\mathbf{w}$
Require: $\mathbf{w}_0$: Initial parameter vector
  $m_0 \leftarrow 0$ Initialize 1$^{st}$ moment vector
  $v_0 \leftarrow 0$ Initialize 2$^{nd}$ moment vector
  $t \leftarrow 0$ Initialize time step
  repeat
      $t \leftarrow t + 1$
      $\mathbf{g}_t \leftarrow \frac{\partial J(\mathbf{w}_{t-1})}{\partial \mathbf{w}_{t-1}}$
      $\mathbf{m}_t \leftarrow \beta_1 \cdot \mathbf{m}_{t-1} + (1 - \beta_1) \cdot \mathbf{g}_t$
      $\mathbf{v}_t \leftarrow \beta_2 \cdot \mathbf{v}_{t-1} + (1 - \beta_2) \cdot \mathbf{g}_t \odot \mathbf{g}_t$
      $\hat{\mathbf{m}}_t \leftarrow \frac{\mathbf{m}_t}{1-\beta_1^t}$
      $\hat{\mathbf{v}}_t \leftarrow \frac{\mathbf{v}_t}{1-\beta_2^t}$
      $\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \alpha \cdot \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t}+\varepsilon}$
  until $\mathbf{w}$ has converged
  return $\mathbf{w}$
```

2.2.2.3 Overfitting

When training the model, one wants to find weights that generalize well i.e. when the model is presented to new data not seen during training, the input-output mapping is still satisfying. When *overfitting*, the model performs well on the training data set but bad on the test data set. This can either be a result of having built an unnecessary complex model for the application or a result of training on a data set that is not diverse enough, or a combination of the two.

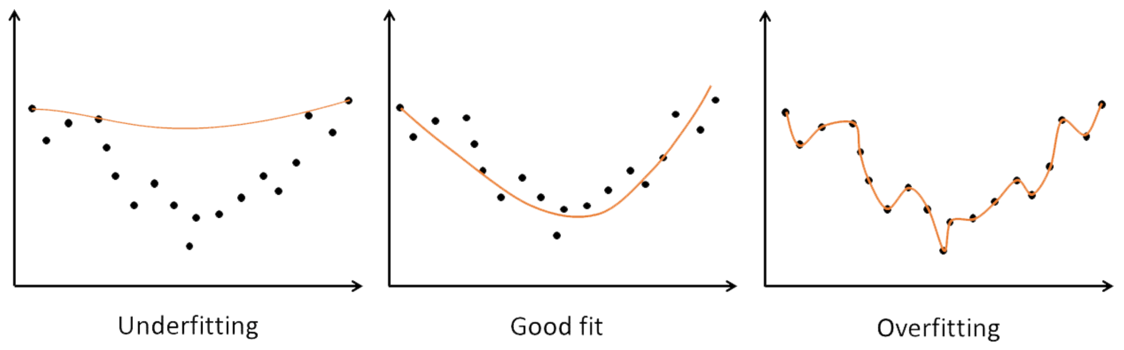


Figure 2.2: Visualization of underfitting (to the left): when the model is too simple. Overfitting (to the right): when the model is too complex. A good fit (in the middle): a model that is a proper representation of the data.

The phenomenon can be compared to a non-linear interpolation of data points in a two dimensional problem as in Figure 2.2 [8]. On the other hand, Advani and Saxe [11] have performed an analysis of generalization error of high-dimensional neural networks and found some interesting results that contradicts this intuition. They found that there is what they call a *frozen subspace* which is large for complex networks. When the number of weights is much greater than the number of data samples there are many directions in which there are no training data and as a consequence those weights have no gradients and are never learnt. Thus, if the initialization of the weights are set to small numbers, the impact of undesirable high dimensions is kept negligible. The frozen subspace actually protects against overfitting and when the network complexity increase, overfitting is often reduced. Lastly, and less importantly in these applications, *underfitting* is the opposite to overfitting. It may occur when the model is too simple for the application making it impossible to detect the patterns present in the data set.

2.2.2.4 Choosing network architecture

The best architecture for a neural network highly depends on the application. While there are no general methods for deciding the best architecture some rule of thumb methods have been documented. Heaton [12] states that there are no theoretical motives to use more than two hidden layers in a feed forward neural network because with two layers one can represent an arbitrary decision boundary and approximate any smooth mapping to any accuracy. He proposes further different rules for setting the number of neurons in each hidden layer. One rule states that the number of neurons should be between the input and output layer dimension while another states that it should be less than twice the size of the input layer. Many more parameters have to be chosen, such as learning rate, activation function and optimizer. Stathakis [13] lists four commonly used approaches to reach a proper network setup:

Trial and error Commonly used but with risk of finding a suboptimal network structure.

Heuristic search Use knowledge from previous experience, often in the form of formulas estimating the number of required neurons in the hidden layers as a function of the number of inputs and outputs. This method can also be used to get a range of topologies to be searched and evaluated.

Exhaustive search This method is simply searching through all topologies which is not very reasonable to do in a real application because each network takes a long time to evaluate. Moreover, each network should be evaluated multiple times due to the fact that neural networks performs differently even when everything is kept constant due to the randomness when initializing the weights.

Pruning and constructive algorithms Either this algorithm starts with no links between neurons (weights) or redundantly many and incrementally adds links or removes links until a network that produces satisfactory results is achieved. Optimal Brain Damage [14] and Optimal Brain Surgeon [15] are two examples of commonly used pruning algorithms.

2.3 Simulated closed loop manufacturing system

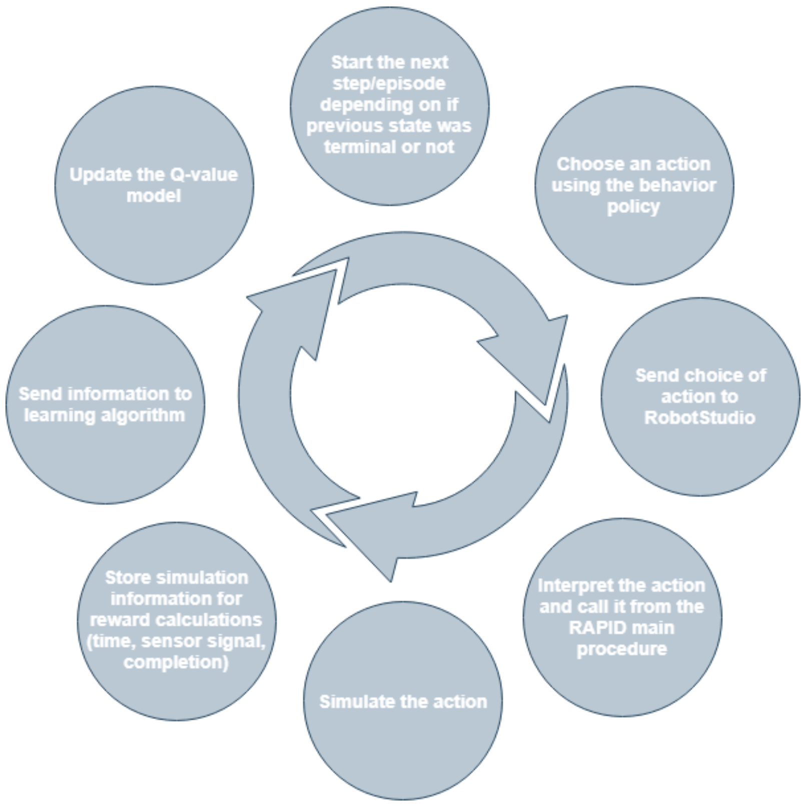

Start the next step/episode depending on if previous state was terminal or not
Choose an action using the behavior policy
Send choice of action to RobotStudio
Interpret the action and call it from the RAPID main procedure
Simulate the action
Store simulation information for reward calculations (time, sensor signal, completion)
Send information to learning algorithm
Update the Q-value model

Figure 2.3: This loop shows more detailed steps of the training loop based on data from RobotStudio.

In the loop shown in Figure 2.3, data from the simulation (RobotStudio) is used to calculate the Q-values which in turn affects the behavior (policy) of the simulated process. This is what is referred to as the *simulated closed loop* in the first research question in Section 1.1 and is a milestone to reach the real closed loop manufacturing system. The loop is applied to all implementations in this thesis; from tabular Q-learning to all function approximations. One can compare the loop to algorithms 1, 2 and 3 to see that this loop uses the same structure as those algorithms but implemented with RobotStudio and of course generalized.

2.4 Reward shaping

When the goal is to build an AI application that generalizes well to varying environments, design of the reward function is crucial. Dewey [16] points out that the more general and autonomous a RL agent becomes, the design of rewards that elicit desired behaviors becomes both more important and more difficult. However, in this project, generality when it comes to the reward function will not be a priority. Instead, reward shaping will be exploited which is a method of integrating knowledge through a strategic design of the reward function so that the algorithm is guided faster towards the goal [17]. First, some considerations must be taken into account because of the risk that the engineer guides the algorithm into suboptimal solutions and loses the optimal one that may yet be unknown to the designer of the reward function. Since it is desired that the agent finds the optimal solution even though (and specially if) it was unknown to the engineer, the reward function must not force the algorithm into suboptimal solutions. Fortunately, the concept of potential-based reward shaping can be used which has proven not to alter the optimal policy [18]. This holds if a reward $F(s, s')$ is provided in addition to the regular reward r where F is defined as the difference of some real-valued potential function ϕ between a source state s and a destination state s' according to

$$F = \gamma\phi(s) - \phi(s'), \tag{2.11}$$

where γ is the same discount factor as in the original algorithm. As a result, the update rule in the tabular Q-learning algorithm takes the form

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + F(s,s') + \gamma \min_{a'} Q(s',a') - Q(s,a)]. \tag{2.12}$$

Similarly, for the function approximation methods, the TD-target is modified to

$$r + F(s,s') + \gamma\hat{q}(s',a',\mathbf{w}). \tag{2.13}$$

One example of a potential function is $\phi(s) = -d(s)$ where $d(s)$ is the distance from state s to the goal state. According to the heuristic, the potential function should be higher the closer the state is to the goal state which is why the distance is negated [19].

2.5 Experience replay

In general, machine learning algorithms need large amounts of data to converge and RL, specially deep RL, is not an exception. When applying RL on simulations that are slow, one therefore has to deal with a potential scarcity of data. One solution is to simulate many parallel instances and orchestrate learning with some update strategy. This is explained more in Section 2.6 and investigated thoroughly in Section 4. However, this project also concerns possibilities and challenges when it comes to training on a physical station. Parallel training on physical stations are of course limited to the number of physical stations available which usually is a small number. This means that it is hard to collect the large amount of data required. A method that can increase data efficiency proposed by Lin [20] is called *Experience Replay* in which the data collected are stored in a memory called *Replay Memory*. The data in the replay memory is presented to the RL algorithm repeatedly so that data already experienced will be re-experienced. Lin explains that the result will be that the rewards received will propagate faster which speeds up the learning process. A warning though, as Lin writes, is that this method can be harmful when the environment changes rapidly. Experience replay has been successfully implemented in several physical real-time control problems by Sander, Buşoniu and Babuška [21] which demonstrate that experience replay RL methods can be well suited for control of physical systems.

2.6 Parallel instance orchestration

In this application, every action is simulated in RobotStudio in order to retrieve the reward. Hence, the simulation will be the main bottleneck in the algorithm. A way to overcome the speed limitation in the simulation software is to run many parallel instances where each instance simulates an action and sends the reward to the algorithm. One important factor that makes this strategy possible is that the update of the Q-values is independent of the update sequence. This means that the Q-value of an action occurring late in an assembly sequence can be updated before a Q-value of an early action. This will often be the case when multiple instances are running in parallel. Nevertheless, the update strategy of the Q-matrix must be considered carefully. It is often the case that a global Q-matrix is updated from multiple training instances. As shown in the update step of algorithm 1, the new Q-value depends on the previous one. Suppose then that instance A starts simulating a certain action at time t_A, based on a policy evaluated with the global Q-matrix, and instance B starts simulating the same action based on the same global Q-matrix at time t_B where $t_A < t_B$. Then, instance B will overwrite the Q-value that has already been updated by instance A and as a result the experience gathered from instance A will be lost because the update from instance B is based on an already aged Q-matrix. The impact of this depends on the application and hence the suitability of different update strategies varies and must be evaluated for the specific application. Strategies evaluated in this project are described in more detail in Section 4.

2.6.1 Central learner and distributed learners

In this context, the terms *actors* and *learners* are often used for separating the mechanism that actually performs the different actions in its environment and the learning algorithm that uses the results gathered by the actors. In this project the actors are the simulated robots. When the actors, are the bottleneck, as in this case with tedious RobotStudio simulations, a centralized learner is preferred because this will avoid any loss of data collected by the actors as opposed to the case described above with one fast instance A and one slow instance B. Each actor will update a global Q-matrix based on the latest parameters after each action. Also the the exploration will be based on the latest updates on the global Q-matrix from all instances. However, in a scenario where the learner is the bottleneck which might be the case when sufficiently many parallel simulations instances run, the update strategy is not that straight forward. It might then be more computationally efficient to collect data in batches and distribute learning on multiple GPUs. Challenges then arise when it comes to how to orchestrate updates from different learners.

Espeholt et al [22] propose an architecture called IMPALA (Importance Weighted Actor-Learner Architecture) that together with the V-trace off-policy algorithm deals with these issues. In this tool actors, i.e. instances of RobotStudio simulations, gather experience and send it in the form of states, actions and rewards to a learner which in this project corresponds to the algorithm implemented in Visual Studio. Either the actors send minibatches of trajectories of data via a queue to a centralized learner and the actors receive the latest policy parameters from the learner, or the policy parameters and the data trajectories are distributed across multiple synchronized learners. In both strategies, the actors and learners are completely decoupled; the actors collect data until a desired mini batch size is stored at the same time as the learners update the model parameters. As a result, the policy on the actors lag behind the policy on the learners and to correct for this deviation a correction method is needed. Espeholt et al deal with this by introducing the V-trace algorithm that compensates for the data collected by the actors being off-policy. In their article, Espeholt et al show that their novel correction method V-trace outperforms traditional methods.

2.7 Similar projects and their relevance

In this part, research projects that relate closely to this project are presented. They contribute with partly answering the research questions and are reflected upon in Section 6.

2.7.1 Tabular Q-learning and function approximation

In the assembly station model in Section 5, the learning algorithm was extended from tabular Q-learning to function approximation for increased flexibility when it came to the state definition. Unintuitively, using deep neural networks for function approximation might converge faster than tabular Q-learning. Ehn and Werner [23] found that function approximation converged faster than tabular Q-learning due to the fact that each update of the model updates all states (specially similar states) instead of only one state at a time which speeds up training. Another observation from the work of Ehn and Adam is that in order to define a problem solvable for tabular Q-learning they had to reduce the amount information defining a state which lead to less accurate results while all information easily could be stored when applying function approximation. This indicates that for more complex applications the advantages of non-linear function approximation might become more evident.

2.7.2 Offline optimization and reinforcement learning

As RL is an algorithm that updates parameters in order to minimize or maximize a loss function it is closely related to traditional optimization. However, there are some important differences that will be covered briefly in this section based on a comparison of the two methods made by Özçelikkale, Koseoglu and Srivastava [24]. In their study, they optimize the energy allocation of an energy beacon to different sensors and also the data transmission powers of the sensors. The loss to be minimized is defined as the field reconstruction error at the sink. Traditional optimization methods require full knowledge of the problem or that modeling assumptions are made. If this can be achieved then there are methods that guarantee convergence to an optimal solution. RL methods do not require full knowledge of the system, instead the nature of the system is automatically learnt within the algorithm. In general, this comes with a cost of an increased number of iterations before convergence. On the other hand, as Özçelikkale, Koseoglu and Srivastava point out, it can be argued that since the optimization approach requires prior knowledge of the model, which is often gained through interactions with the system, overhead training in some sense must be conducted. As a conclusion for their application, the study shows that if the number of iterations is disregarded, off-line optimization and RL reach similar performance. Another interesting yet expected result is that if the assumed model in the optimization problem differs from the actual model, the performance decreases. In RL such discrepancies do not exist because no model assumptions are made. A final note that they consider as future work touch upon the fact that the RL approach can treat modeling and optimization as separate tasks. Then a simulation model of the real world is used for initial training to achieve a plausible model which is later transferred to the real world for final optimization.

2.7.3 Simulation gap

Simulation models are only representations of the real world and include many simplifications. When training an RL algorithm on simulation models it is therefore important to consider how to deal with simulation biases. J. Kober et al write that RL algorithms exploit model inaccuracies if they are beneficial in order to maximize the reward function which can lead to results that are overfitted and perform well in the simulation but that cannot be implemented on the real system. However, one way to reduce this risk is to introduce stochastic distributions in the models [25]. In this project the gap between simulation and reality will have to be sufficiently small so that the result can be applicable to the real station even though it has been synthesized mainly through training on the simulation environment.

2.7.4 AI technology from video games

The core ideas, goals and methods behind this thesis are all supported by previous research and studies. This is primarily because the setup for this thesis where an assembly station has a limited amount of actions and a goal of assembling a product can be viewed as a video game. One can therefore refer to research in previous RL projects related to different gaming scenarios. For instance Firoiu et al [26] applied RL to a video game where the algorithm had a set number of 54 actions. It then had to beat its opponent through these actions, and would get a reward for doing so. This is similar to the robot and the operator having a set number of actions to assemble the product, but instead of beating an opponent, the algorithm has to assemble the product quickly and with high quality. The famous paper by Mnih et al [27] where they developed a RL algorithm to play many Atari games, also shows the possibility and flexibility of these algorithms for similar scenarios. The main difference between these papers and this project is the input model where both Fiouiu et al and Mnih et al had input in the form of pixelated images, which meant a great deal of data to analyze and motivated them to use deep RL. However, the simulated model used in this thesis will have relatively few states, which simplifies the needed methods and computing power.

2.7.4.1 GOAP

A research project by Yu et al [28] presents a method for mechanical assembly planning using AI technology from games. They used Goal-Oriented Action Planning (GOAP) which is based on STRIPS (STanford Research Institute Problem Solver) in order to find the optimal assembly sequence of pump components. GOAP is an automated planner and one of the most important features is its dynamic replanning capability which was used in the project in order to enable the system to change the assembly sequence based on the behavior of the operator. It is argued that this flexibility feature is what makes GOAP superior to finite state machine (FSM) techniques. Owens [29] also compares GOAP with FSM where he points out that in FSM all the connections between states and actions must be predefined as opposed to GOAP where the states and actions are decoupled and the relations are learnt by the system itself. A complex FSM can be extremely difficult to modify in order to add or remove possible actions whereas in GOAP the absence of a fully defined connected model allows the user to add and remove actions to the action space without the need to modify the model; GOAP will learn to adapt to the new conditions. This is a substantial advantage when working with mechanical assembly sequence planning of complex products when it comes to the ease of building the search model and it also allows the algorithm to find sequences that might not have been found if it were not an obvious solution for the modeling engineer. Owens explains that

GOAP is an artificial intelligence system that finds a sequence of actions to satisfy a particular goal where the sequence depends not only on the goal state but also on the current state of the agent. In an assembly application, this allows the agent to modify the sequence adapting it to the current operator with a certain behavior.

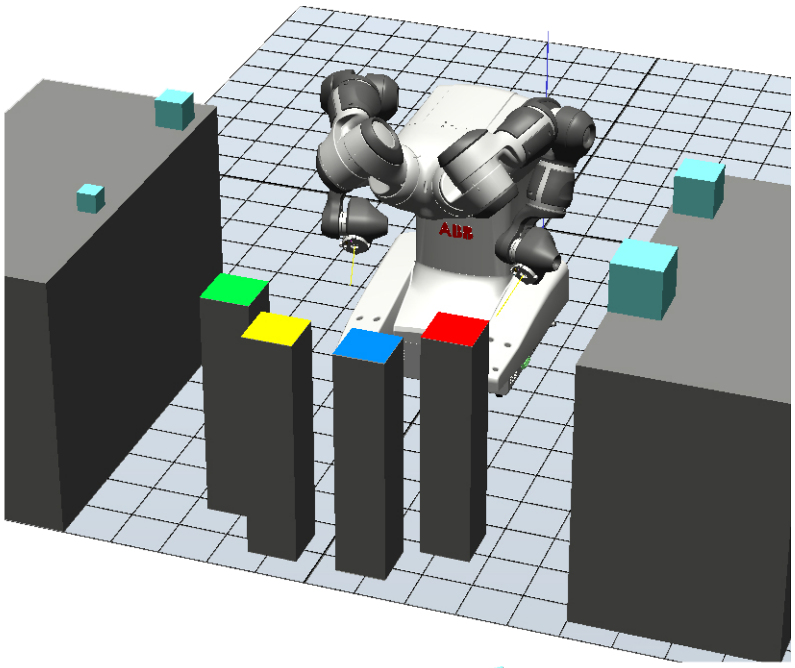
2.7.5 Planning Domain Definition Language

Planning Domain Definition Language (PDDL) is a standardized artificial intelligence planning language first deveoped by Drew McDermott et al 1998 for the International Conference on AI Planning and Scheduling (AIPS) planning competition [30]. The language is developed with a high level of neutrality and as a result many extensions of the first version in different directions have been developed and released in connection with the competition. Basically, PDDL includes a problem definition with objects, initial state and goal state, and a domain definition with predicates (boolean properties of the objects) and actions. Finally there is a solution checker that returns a sequence of actions.

3

Learning optimal product assembly sequence

(a)



(b)

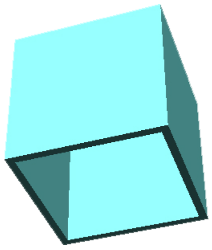


Figure 3.1: Figure 3.1a shows the RobotStudio model used to find an optimal assembly sequence. Figure 3.1b shows how each box is hollow underneath, which makes it impossible to stack a larger box on top of a smaller one.

Methods and algorithms covered in the previous chapter has been applied on two simulation models in order to find answers to the research questions. The model in Figure 3.1a was developed with the purpose of demonstrating the feasibility of learning an optimal assembly sequence. The idea behind the model is that the robot can choose from one of four hollow boxes, visualized in Figure 3.1b, and has to place them on the red pillar so as to build the highest possible tower. Since the boxes vary in size, there is only one optimal sequence of actions. By activating the built-in physics engine in RobotStudio, the robot can learn how the objects physically interact with each other. For example, a large box will fall down and enclose a small one if the robot tries to stack them in the wrong order. To learn this, tabular Q-learning according to algorithm 1 is used.

In this specific case, the application itself is without inherit value but it is conceptually useful. This because it is easily scalable in difficulty as one can for example choose how many pillars the robot can place the boxes on, if the robot can repeat actions, or if it can place the boxes on different heights. Each increase in complexity will result in the simulations taking more time, which limits the amount of data that can be gathered in a given time. For that reason, this thesis will focus on the moderately difficult case for this model and will not allow the robot to perform the same action twice, or place the boxes on different heights. Another particularly useful aspect of this model is that it simplifies assembly processes that can be complicated and long, into something that is easy to understand and explain but still has the same underlying idea. All product assemblies can essentially be reduced to a deterministic chain of events in the sense of ”Do A, then do B”. One can for example imagine that building a tower on the red pillar is equivalent to constructing a bolted joint, where you first have to drill a hole in the sheet metal, place the bolt and tighten the nut. Any other sequence of actions will not work, as in the case with the described model.

The model is also reused when evaluating the effects of running multiple parallel simulation instances, described in Section 4. This model has a small number of states and actions which makes tabular Q-learning a suitable choice to use in the learning algorithm. Hence, function approximation is not applied here but on the assembly station in Section 5.

3.1 Setup

State	#Boxes on red pillar	#Boxes on non-red pillars	Reward	Optimal Q-value for corresponding action
Initial	0	0	0	-
Error	1,2,3	>0	-50	-50
Height 1	1	0	20	116.56
Height 2	2	0	35	128.75
Height 3	3	0	50	125
Height 4	4	0	100	100

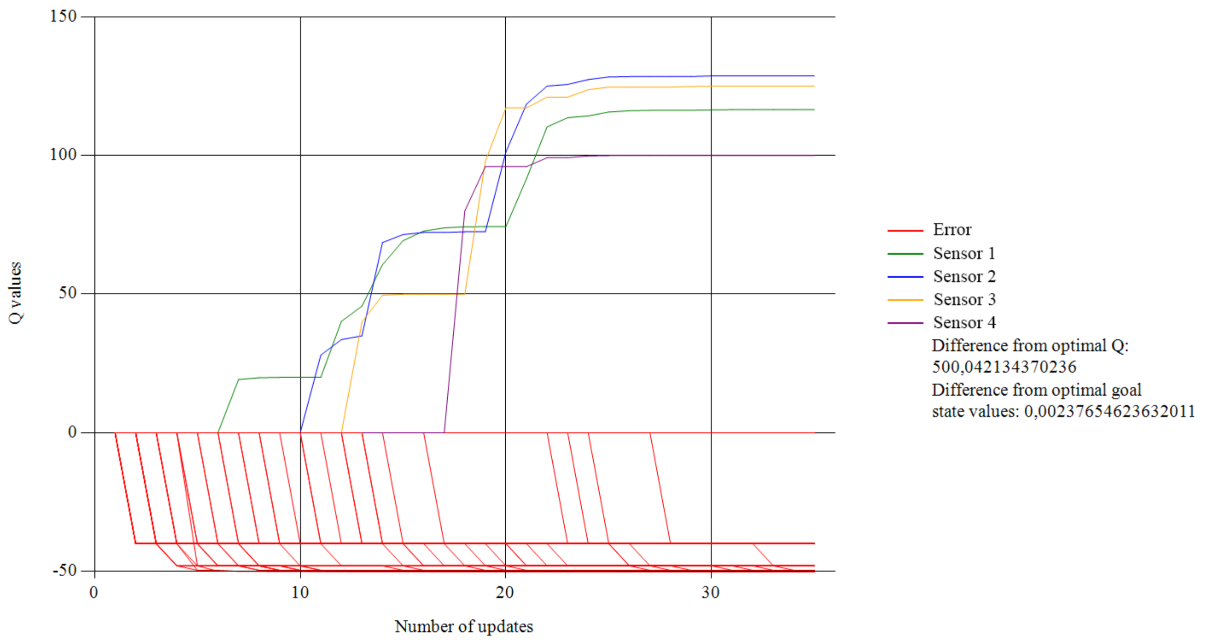
Table 3.1: State definitions and reward system for the system shown in Figure 3.1. The state relates to the number of boxes placed on the red pillar. The more boxes placed on the red pillar, the higher the reward. A negative reward is achieved whenever a box is placed on a non-red pillar. In the last column, approximations of the optimal Q-value for the action leading to the respective state is given.

The actions were defined in the form ”Pick X and place on pillar Y” where X is defined as the set of items Box 1, Box 2, Box 3, Box 4 whereas Y is defined as the set of pillars Green, Yellow, Blue, Red.

The states were defined based on the number of items stacked on the red pillar. Four object sensors were placed at different heights on the red pillar to determine how many boxes have been stacked there. If an action did not lead to an additional sensor being activated, the agent ended up in an error state. The goal state, Height 4 in Table 3.1, was reached when all four sensors were activated which only occurred when the boxes were stacked in the correct sequence on the red pillar. If an action put the agent in the error state, it would be punished with a negative reward of -50. State 2 and 3 would yield positive rewards of 30 and 50 respectively because they corresponded to states from where it was possible to reach one of the goal states. A reward of 100 was given if the goal state was reached. These definitions are clearly summarized in Table 3.1. The magnitude of the values were selected arbitrarily, however their relations to each other was chosen through trial and error as they affect convergence time.

3.2 Convergence analysis

(a)



(b)

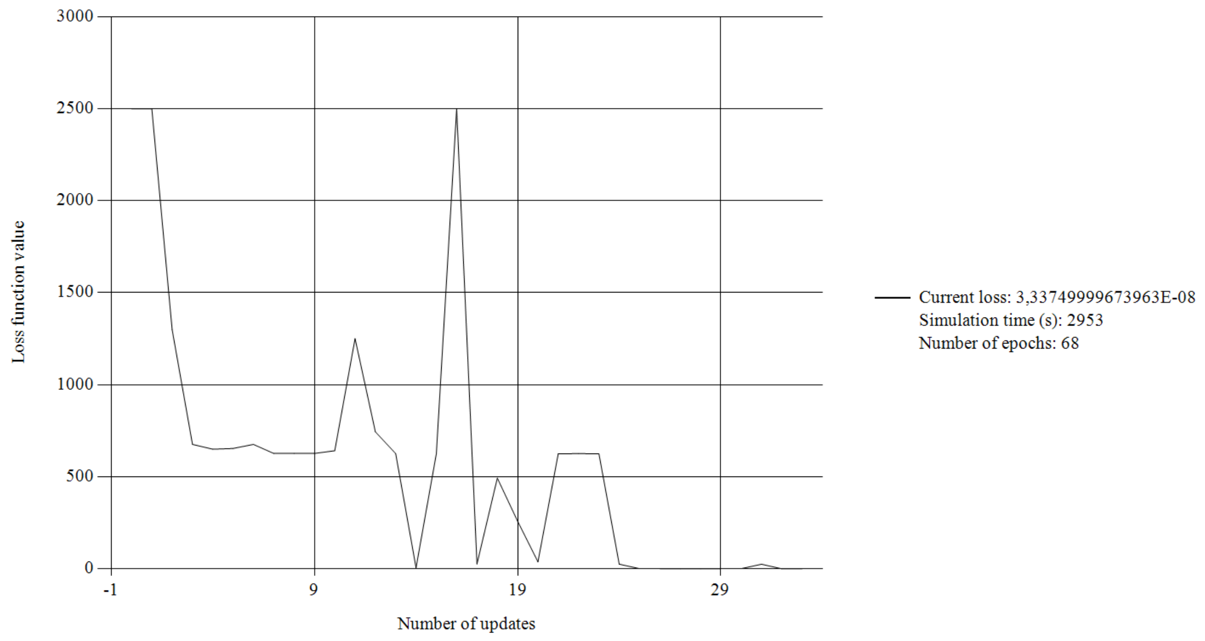


Figure 3.2: Figure 3.2a and Figure 3.2b show the Q-values and MSE respectively during training of the model. For reference, the optimal Q-values are given in Table 3.1.

Using this setup, the convergence behavior in Table 3.2a and 3.2b was observed. The model is considered converged when the largest change of any Q-value during the last two epochs is less than 0.1. These values were chosen based on the desire to decrease the probability of the algorithm getting stuck in a local optimum and being satisfied with a sub-optimal solution. In combination with an ε-greedy behavior policy, the algorithm managed to find a solution close to the optimal values in Table 3.1 in a short amount of time. The Q-values and the loss in the plots are updated after each epoch i.e. each time all boxes have been picked and placed. Hence, in this case the number of updates (on the x-axis) is equal to the number of epochs.

Figure 3.2a shows the result of each update during training. The red lines relate to actions that did not manage increase the number of sensors activated and thus leading to the error state, while the non-red lines correspond to actions activating the sensor given in the legend and thus leading to a state from which the goal state can be reached. The same definitions hold for all subsequent plots related to this simulation model. Every action that leads to the error state will eventually converge to -50. Since only a stack on the red pillar is rewarded, Height 1 corresponds to placing the biggest box on the red pillar, Height 2 when the second biggest box is placed on top of the biggest one on the red pillar etc. The difference from optimal Q is calculated as

$$\sum_{a=1}^{n_a} Q_a - Q_a^* \ , \tag{3.1}$$

where n_a is the total number of actions, Q_a and Q_a^* are the calculated Q-value and optimal Q-value for action a respectively. The simulation time is not the real time but the the simulated time that depends on how fast the simulation runs. It is observable from the figure that the different successful states are achieved in correct succession, with the algorithm first learning that the largest box should be placed on the red pillar, then that the next largest box should be placed on top of it, and so on. Furthermore, one can also see how not all Q-values were explored as there is at least one state with an initial Q-value. This is directly connected to the relatively greedy behavior policy that was chosen. It is assumed that since minimization of production lead time is the driving factor, the algorithm should finish when the optimal solution has been found. Nevertheless, there are cases where optimal Q-values for all actions wants to be known, see Section 5, and in those cases the policy should be completely exploratory.

As can be seen in the plot legend of Figure 3.2b convergence was reached after 2953 simulated seconds. Furthermore, comparing both Figure 3.2a and 3.2b shows a clear connection between the sudden and large change in loss and the discovery of a new successful state. Looking at Figure 3.2a around the 17th update when the algorithm locates Height 4, one can see a massive loss spike in Figure 3.2b. The loss chart also motivates the convergence check that was chosen since it is clear that at some point before the goal state was found the algorithm was almost at 0 loss. Not checking convergence at every epoch increases the probability of the algorithm finding the optimum before being satisfied with the local optimum.

The results from this model show that it is possible to use tabular Q-learning in a simulated environment using a built-in physics engine to learn an optimal assembly sequence. It was noticed that for faster and guaranteed convergence, only one goal state is preferred. Additionally, time can be saved if states that are not error states are only reachable through a deterministic sequence of actions.

4

Parallelization in reinforcement learning applications

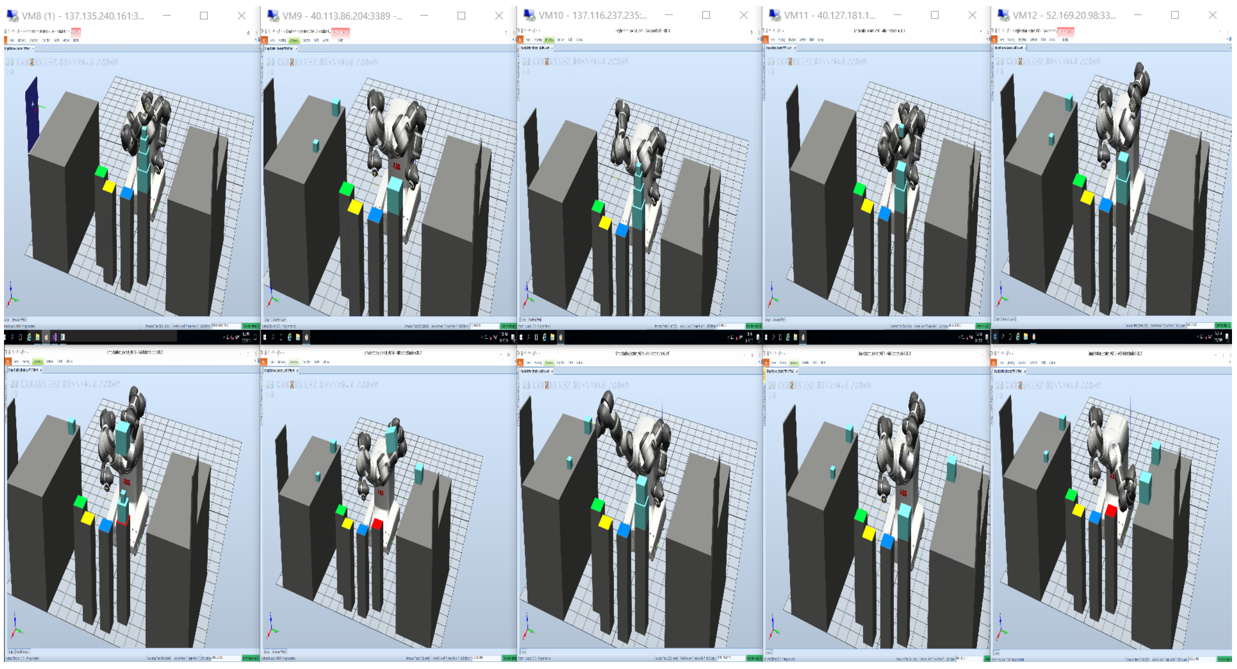

VM8 (1) - 137.135.240.161:3...
VM9 - 40.113.86.204:3389 -...
VM10 - 137.116.237.235:...
VM11 - 40.127.181.1...
VM12 - 52.169.20.98:33...

Figure 4.1: 10 instances of the model in Section 3 simulated in parallel using VM on Azure.

As the experiments conducted in Section 2.1, and indeed RL experiments in general, take a great deal of time, parallelization of multiple RobotStudio instances is explored as a possible strategy to decrease convergence time. Different methods of how to orchestrate parallel actors (simulation instances) and how to update the Q-values based on experience from them are investigated. The model described in Section 3 is used to analyze the effects of different strategies when scaling up to learning on multiple parallel simulation instances. Hence this analysis has been implemented with the same tabular Q-learning method as in Section 3. Figure 4.1 shows 10 instances of this model working in parallel in order to build the Q-matrix efficiently. Furthermore, it is important to note that there is no intentional control of the different instances, so which instance is faster or slower is unknown and therefore the synchronous actors are, perhaps unconventionally, randomized as well. This is however motivated by the fact that this increases the generalization of the system as the instances can change their speeds without having to change anything in the algorithm.

Algorithm 5 Centralized learner with asynchronous/synchronous actors

```
Initialize global Q-matrix
repeat
    Choose actions for $n$ instances based on global Q-matrix
    for each instance $i \in n$ do                       ⎫
        Simulate action in instance $i$                  ⎪
        Retrieve simulation data                         ⎪
        Send to RL agent (global)                        ⎬ in parallel
        RL agent updates global Q-matrix                 ⎪
        if Synchronized strategy then                    ⎪
            Wait for all other instances                 ⎭
until convergence of Q-values, checked every other epoch
```

Algorithm 6 Distributed learner with asynchronous/synchronous actors

```
Initialize global Q-matrix
Initialize $n$ local Q-matrices based on global Q-matrix
repeat
    Choose actions for $n$ instances based on their local Q-matrices
    for each instance $i \in n$ do                       ⎫
        Simulate action in instance $i$                  ⎪
        Retrieve simulation data                         ⎪
        Update local Q-matrix                            ⎪
        if Synchronized strategy then                    ⎬ in parallel
            Wait for all other instances                 ⎪
        if 2 epochs have passed for instance $i$ then    ⎪
            Merge global and local Q-matrix of instance $i$ ⎭
until convergence of Q-values, checked every other epoch
```

Four different update strategies have been evaluated, primarily based on the theoretical background presented in Section 2.6.1 and algorithms 5 and 6:

Centralized learner with asynchronous actors Each actor will send collected data to a central learner after each action. The learner updates the Q-values as soon as it receives data. The implementation follows the pseudo-code shown in algorithm 5.

Centralized learner with synchronized actors The updates are calculated as soon data is received on the learner just as in the previous strategy. Here, however, each actor will perform all actions in an epoch and wait for all other instances to finish before it starts the next epoch, meaning that all actors always start each epoch at the same time. The implementation follows the pseudo-code shown in algorithm 5.

Distributed learners with asynchronous actors Local Q-values are calculated in each instance, and after two epochs they are inserted in a global Q-matrix in the order in which each instance finishes every two epochs. The implementation follows the pseudo-code shown in algorithm 6.

Distributed learner with synchronized actors Local Q-values are calculated in each instance as in the previous strategy but inserted in the global Q-matrix first when all instances have finished two epochs. Then all instances proceed with the next two epochs synchronously. The implementation follows the pseudo-code shown in algorithm 6.

A tool for orchestrating updates from multiple actors called IMPALA was described in Section 2.6 which is well-suited for multitask RL with large amounts of data. In this project the applications are simpler and data is scarce which makes it inappropriate to apply IMPALA, but when the concepts in this project are scaled sufficiently, it might be an efficient strategy. Instead of applying IMPALA and the V-trace algorithm, the aforementioned methods were implemented.

In Section 2.6 the risk of losing data when running parallel instances was explained. This risk is present in both the synchronizing and the asynchronous strategy. Since the order in which instances access the global Q-matrix is random it is hard to reach general conclusions of how the effect of this differs between the two strategies. However, since the absolute value of the Q-values in the box model has proven to increase monotonically this can easily be avoided by updating Q-values from an instance only if the update would increase the absolute value of the global Q-value. In that way, updates will only be saved if they are closer to the optimum. This is generally not an option but in this special case it was a good way to highlight strategy attributes that would be hard to observe otherwise.

Another problem that was noticed for one instance but caused increased disturbance with multiple instances is that when a specific state can be reached through different sequences of actions, instances sometimes work against each other. Then, it becomes harder to learn what sequence is desired since the profitable sequence will increase a certain action value while a unprofitable sequence will decrease it. For example, Height 1 can be reached by putting any of the smaller boxes underneath the largest box, but that would be undesirable. This emphasizes the insight that time can be saved if states are defined so that if they belong to a desired sequence, they should not be reachable from undesired sequences. In this project, this was realized by sending all the actions that did not lead to an increase of the height of the stack to an error state. Note that this is not a requirement for RL to work, however it is a way to increase that probability while decreasing convergence time substantially.

4.1 Parallel instance analysis

Parameter	**Value**
Convergence tolerance, δ	0.1
Learning rate, α	0.8
Discount factor, γ	0.75
Exploration rate, ϵ	0.6

Table 4.1: Test parameters used for each learner in the parallel instance tests.

1 instance				
Test	Instances	Simulation time (s)	Error	Avg. simulation time (std. dev.) Avg. error (std. dev.)
1	1	2790	460.62	2902 (317.30) 491.55 (54.22)
2	1	2953	500.04	
3	1	3392	448.10	
4	1	2854	466.75	
5	1	2521	582.26	

Table 4.2: The table presents the simulation time and error (see equation 3.1) for five runs with one instance. This serves as a reference, emphasizing the effects of parallelizing.

Table 4.1 lists the values used in all the parallel instance tests for the parameters corresponding to the Q-learning algorithm presented in Section 2.1. In Table 4.2 data for one instance is collected for comparison with multiple instances. Simulation time is the simulated time in seconds passed before the Q-values have converged. The error is defined as in equation 3.1. One notices quite large errors in this case which is due to the fact that error states have most likely not converged. This is seen when looking at Figure 3.2a which shows the Q-values for 1 instance as well. The optimal values have converged but the algorithm takes too long to find the other Q-values, which results in it converging with larger errors. The average and standard deviation of the test runs are presented in the last column. For all tests with multiple instances convergence is checked every two epochs and defined as reached when the maximum difference between the present Q-values and the Q-values two epochs before is less than δ as shown in Table 4.1. The reason why convergence is specifically checked every two epochs, is simply because it was found that checking convergence every epoch leads to a greater probability of getting stuck at a local optimum.

For evaluating the strategies described above, tests with 5 and 10 instances were conducted to observe the effects of scaling the number of instances. Moreover, tests were also conducted where the robot was only allowed to place boxes on the red pillar. In that way the impact of the strategies based on varying problem complexity could be analyzed.

5 instances				
Test	Strategy	Simulation time (s)	Error	Avg. simulation time (std. dev.) Avg. error (std. dev.)
1	Cent. async.	2262	19.44	2133 (367.25) 34.27 (24.38)
2	Cent. async.	1904	74.74	
3	Cent. async.	1729	39.56	
4	Cent. async.	2083	21.36	
5	Cent. async.	2685	16.23	
6	Cent. sync.	2169	26.45	2035 (97.04) 68.43 (38.81)
7	Cent. sync.	2004	75.63	
8	Cent. sync.	1912	124.77	
9	Cent. sync.	2003	77.71	
10	Cent. sync.	2086	37.58	
11	Dist. async.	2693	18.53	2416 (230) 33.76 (23.37)
12	Dist. async.	2538	73.41	
13	Dist. async.	2080	36.64	
14	Dist. async.	2334	19.78	
15	Dist. async.	2435	20.47	
16	Dist. sync.	1741	72.36	1858.6 (95.17) 105.37 (32.99)
17	Dist. sync.	1904	107.94	
18	Dist. sync.	1994	154.59	
19	Dist. sync.	1829	114.17	
20	Dist. sync.	1825	77.81	

Table 4.3: The table shows the results of 5 measurements for each strategy described in Section 4 using 5 parallel instances.

As anticipated, Table 4.3 shows that using parallel instances, regardless of the strategy chosen, reduces both the convergence time itself as well as its variation between measurements. Not equally as expected is that the error reduces even more drastically; converged values are much closer to the optimum when running multiple instances. Since each of the 5 parallel instances converges approximately at the same time, their combined simulation time is much larger than the time it took for one instance. This means that the 5 parallel instances have explored a much larger number of actions, even though the individual convergence time is less. By looking at the total number of epochs explored before convergence, one can argue that it is harder to converge with many instances. This is because the random exploration that accompanies the ε-greedy policy increases when the number of instances increases. For each state in one instance, there is an ε high probability that the action is chosen randomly. At the same time, for each state in one instance, there are five instances in that state in the corresponding multiple instance simulation, each of which will explore randomly with probability ε. This increased exploration rate is the explanation behind the dramatically reduced error and the more modest reduction in simulation time. The same tendencies is observed moving from 5 instances to 10 instances. Lastly, it is clear that the asynchronous/synchronous strategies differ, both for centralized and distributed learning. Further comparison between the strategies will follow.

10 instances				
Test	Strategy	Simulation time (s)	Error	Avg. simulation time (std. dev.) Avg. error (std. dev.)
1	Cent. async.	1488	3.45	1464.2 (42.16) 8.89 (11.31)
2	Cent. async.	1389	29.05	
3	Cent. async.	1479	3.41	
4	Cent. async.	1483	5.51	
5	Cent. async.	1482	3.05	
6	Cent. sync.	1571	3.17	1603.8 (100.83) 5.99 (5.39)
7	Cent. sync.	1643	10.64	
8	Cent. sync.	1572	2.66	
9	Cent. sync.	1481	0.63	
10	Cent. sync.	1752	12.85	
11	Dist. async.	1841	1.3	1950 (131.0) 16.06 (26.62)
12	Dist. async.	1821	0.4	
13	Dist. async.	2080	62.94	
14	Dist. async.	1926	1.38	
15	Dist. async.	1910	12.14	
16	Dist. sync.	1400	73.48	1443 (110.6) 43.62 (35.97)
17	Dist. sync.	1560	7.56	
18	Dist. sync.	1309	17.14	
19	Dist. sync.	1555	89.52	
20	Dist. sync.	1389	30.42	

Table 4.4: The table shows the results of 5 measurements for each strategy described in Section 4 using 10 parallel instances.

The result from Table 4.3 and 4.4 proves not only that the simulation time necessary for convergence and the error decreases with increased number of instances but also that the result is more stable. Comparing to Table 4.2, it can be deduced that using 10 parallel instances decreases the convergence time by 50% and reduces the error by at least a factor of 50. Stability in this context means that the variation in simulation time between the tests reduces and the possibility of exploring the entire set of actions increases. It is also evident that the difference between the strategies is larger for distributed learning compared to centralized learning. Another result is that faster convergence correlates with larger errors and that when the number of instances scales to 10, the errors are larger for distributed learning than for centralized learning. Interestingly, the distributed learning with synchronized actors is very fast and can often beat the centralized learners when it comes to simulation time. However, the speed comes with the cost of large errors, whose importance can be questioned since those errors are driven by not having explored all the error states.

10 instances, reduced problem complexity				
Test	Strategy	Simulation time (s)	Error	Avg. simulation time (std. dev.) Avg. error (std. dev.)
1	Cent. async.	430	0.16	447.2 (71.95) 0.5 (0.96)
2	Cent. async.	430	0.02	
3	Cent. async.	516	0.02	
4	Cent. async.	344	2.22	
5	Cent. async.	516	0.08	
6	Cent. sync.	344	0.1	412.8 (38.46) 0.024 (0.04)
7	Cent. sync.	430	0.02	
8	Cent. sync.	430	0	
9	Cent. sync.	430	0	
10	Cent. sync.	430	0	
11	Dist. async.	602	0.5	756.8 (112.13) 0.272 (0.218)
12	Dist. async.	860	0.52	
13	Dist. async.	688	0.1	
14	Dist. async.	860	0.1	
15	Dist. async.	774	0.14	
16	Dist. sync.	688	0.08	670.8 (38.46) 0.268 (0.21)
17	Dist. sync.	688	0.16	
18	Dist. sync.	688	0.44	
19	Dist. sync.	602	0.54	
20	Dist. sync.	688	0.12	

Table 4.5: The table shows the results of 5 measurements for each strategy described in Section 4 using 10 parallel instances with reduced problem complexity by only enabling the robot to place the boxes on the red pillar i.e. only the correct stacking order is learned.

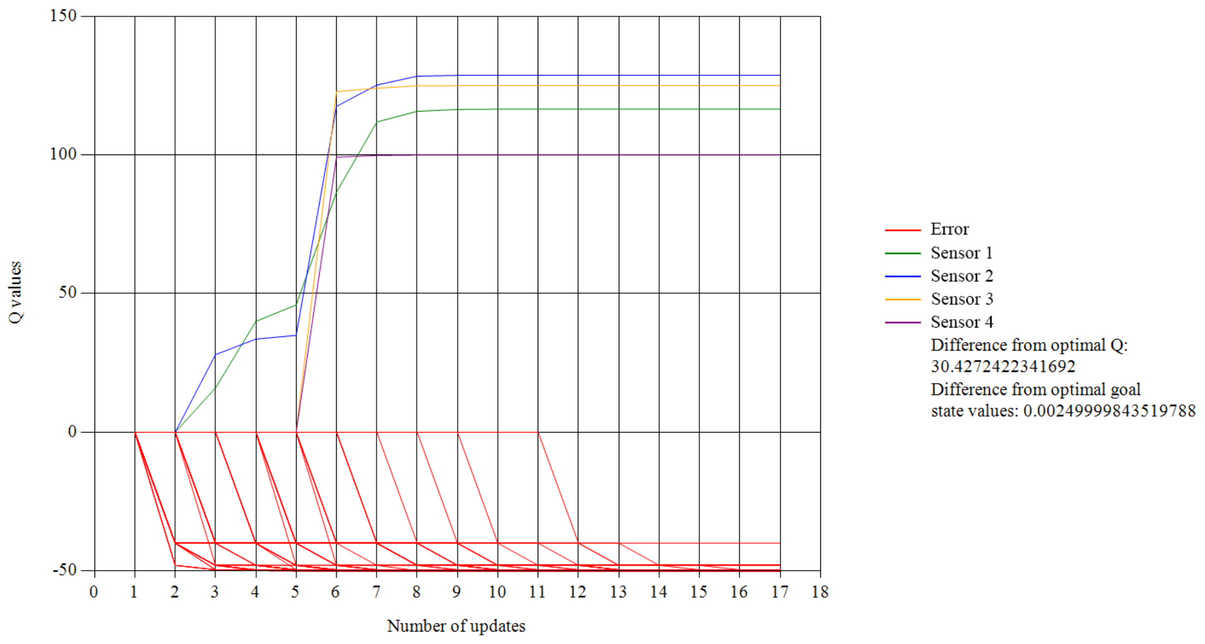
Table 4.5 shows that when the complexity of the problem decreases, the centralized learners gain advantage over the distributed and more specifically, the distributed learners with synchronized actors seems to lose earlier advantages.

#Inst.	Model	Comparison	Avg. Increased Performance
5	4 Pillars	Cent. & Dist.	Cent. async. 1.13 times faster than dist. async. Dist. async. 1.01 times more accurate than cent. async. Dist. sync. 1.09 times faster than cent. sync. Cent. sync. 1.54 times more accurate than dist. sync.
		Async. & Sync.	Cent. sync. 1.05 times faster than cent. async. Cent. async. 1.99 times more accurate than cent. sync. Dist. sync. 1.3 times faster than dist. async. Dist. async. 3.1 times more accurate than dist. sync.
10	4 Pillars	Cent. & Dist.	Cent. async. 1.33 times faster than dist. async. Cent. async. 1.8 times more accurate than dist. async. Dist. sync. 1.11 times faster than cent. sync. Cent. sync. 7.28 times more accurate than dist. sync.
		Async. & Sync.	Cent. async. 1.09 times faster than cent. sync. Cent. sync. 1.48 times more accurate than cent. async. Dist. sync. 1.35 times faster than dist. async. Dist. async. 2.7 times more accurate than dist. sync.
10	1 Pillar	Cent. & Dist.	Cent. async. 1.69 times faster than dist. async. Dist. async. 1.83 times more accurate than cent. async. Cent. sync. 1.62 times faster than dist. sync. Cent. sync. 11.16 times more accurate than dist. sync.
		Async. & Dist.	Cent. sync. 1.08 times faster than cent. async. Cent. sync. 20.8 times more accurate than cent. async. Dist. sync. 1.13 times faster than dist. async. Dist. sync. 1.01 times more accurate than dist. async.

Table 4.6: A comparison between the strategies, based on the results in Table 4.3, 4.4 and 4.5. For each model, the centralized strategies are compared with the distributed strategies and the asynchronous strategies are compared with the synchronized strategies.

Considering the distributed learners, using the asynchronous strategies, the actors proceed with their next epochs before the slower instances have finished their epochs. As a result, the fast instances explore based on less experience compared to the synchronized actors that wait for the slower instances and learn from their experience as well. This makes the synchronized actors explore more intelligently and find the highly rewarded sequence faster. As previously mentioned, the importance of the fact that this leads to larger errors can be discussed since it is often irrelevant to converge the bad action values. In some cases however, for example when learning a human behavior as described in Section 5, it is desired to explore broadly to be sure to cover the true characteristics of the operator, which might make the asynchronous actors an appropriate choice. In addition, this explanation can also motivate why the centralized learners gain advantages over the distributed in Table 4.5. Since a simpler problem does not require as much intelligence, the advantage of exploring with more information about the environment does not matter as much. In Table 4.6 the effect that the distributed learner with synchronized actors becomes faster than the asynchronous actors has been observed to increase with increased problem complexity and increased number of instances.

(a)



(b)

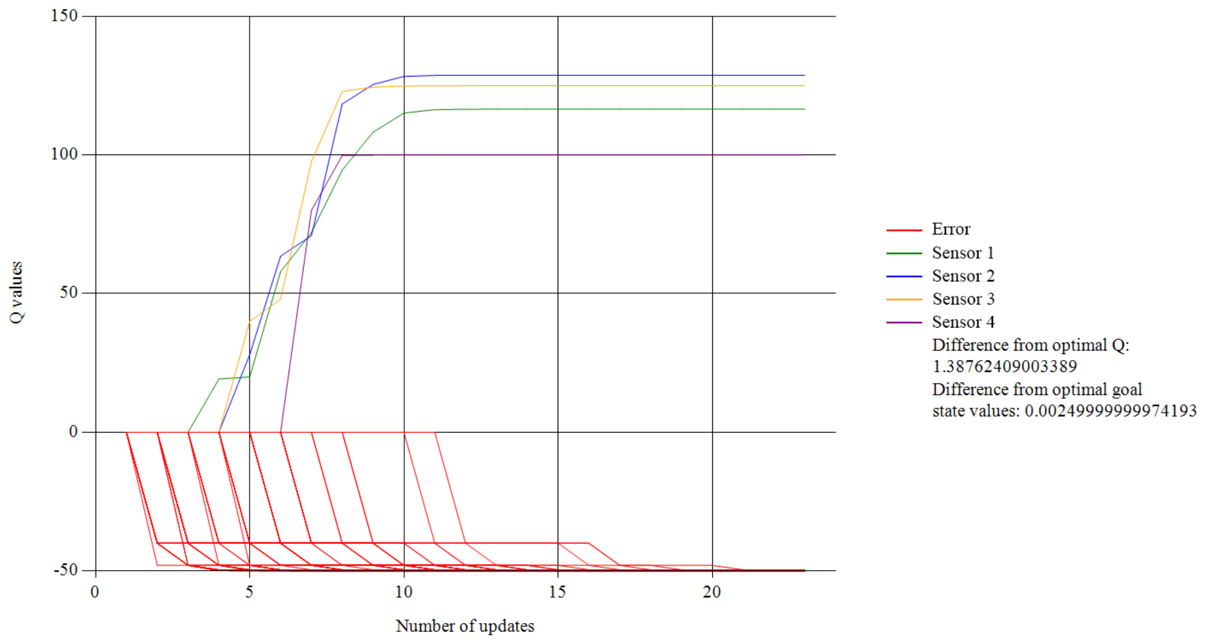


Figure 4.2: Figure 4.2a shows the Q-values during training of distributed synchronized learners and Figure 4.2b shows the Q-values during training for the distributed asynchronous learners.

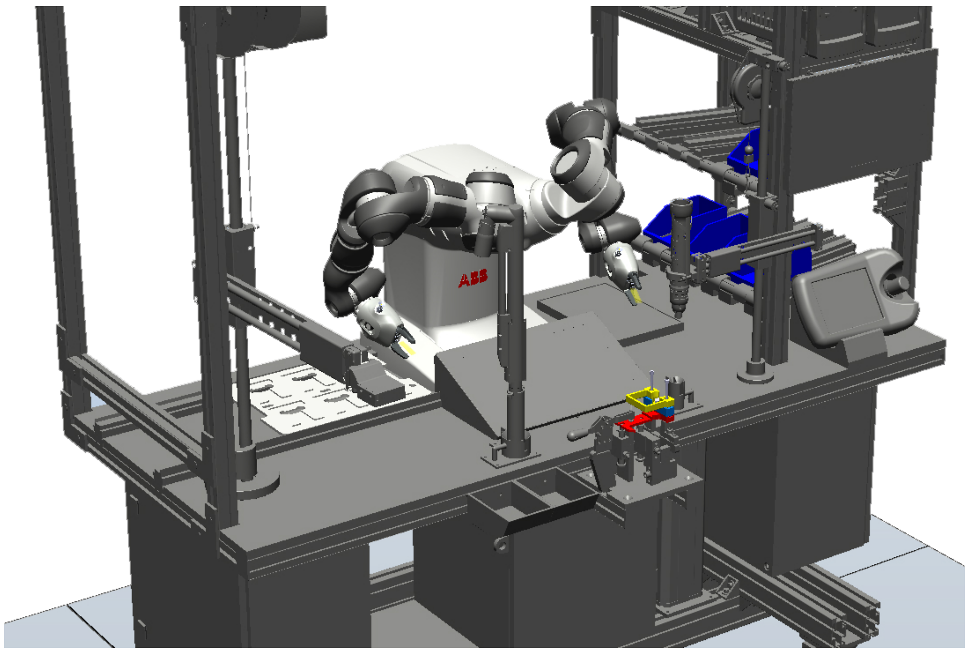
In figures 4.2a and 4.2b some differences between the distributed synchronized and the distributed asynchronous strategy can be highlighted. The synchronized actors have found the first rewarded actions by the third update while the asynchronous actors found the first rewarded action after the fourth update. This is a general pattern observed and can be explained by the fact that before the synchronized actors start an epoch, they will wait for experience gathered by every instance to be collected in the previous epoch. This waiting increases the probability that a higher number of error states have been discovered when starting the next epoch compared to the case where fast instances start the next epoch without waiting for the other instances to potentially discover new error states. As a result, if the synchronized actors act greedily, there are less potentially good actions to choose from which increases the probability of finding the correct action earlier. The same holds for the succeeding rewarded actions. Even though the asynchronous actors were lucky and found Height 3 before the synchronized actors did, they ran past the asynchronous actors by finding Height 4 earlier due to them waiting for the slower actors to rule out some of the bad actions not yet discovered. When it comes to speed, the synchronized actors are almost always preferred.

Finally, from Table 4.6 it is also clear that for low complexity problems, the centralized learner with synchronized actors outperforms the alternatives. Note that the importance of the accuracy measure for the model with only one pillar is considered to be very limited since the errors are extremely small for all strategies.

5

Training adaptable robot for collaborative assembly

(a)



(b)

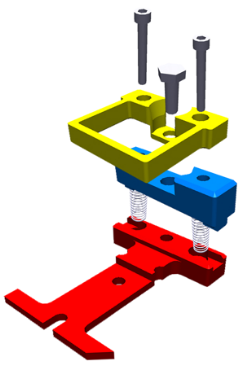


Figure 5.1: Figure 5.1a shows the assembly station in RobotStudio that incorporates human and robot collaboration in order to assemble the product that is shown in Figure 5.1b.

While the model described in chapter 3 was used for proving the possibility of learning an assembly sequence and providing results on parallel learning strategies, the assembly station shown in Figure 5.1a was better suited for more complex scenarios requiring more advanced algorithms. The model uses both robot and human actions in order to assemble the product shown in Figure 5.1b. The product consists of the bottom red part (part A), followed by two springs, then the blue part (part B), the yellow square part (part C), two side screws and a middle screw. One of the greatest advantages with RL is that it can automatically learn models otherwise cumbersome or infeasible to formulate mathematically as optimization problems. An application with such complexity would be able to learn patterns in human behaviour and choose actions based on those patterns such that the collaboration between the robot and the human becomes as efficient as possible. Since the assembly station is based on a real collaborative robot assembly station, it becomes reasonable to integrate human attributes in its state definition and make it learn how to act in relation to them. The previous model in Section 3 is a typical application for tabular Q-learning; other RL methods are hard to motivate for such simple applications. However, when human characteristics are added to the state definition, the complexity can be scaled up to levels where tabular Q-learning must be replaced by other methods. Scalability is important for the generality of the results in this project, hence the methods with this ability merit investigation.

In the assembly station the robot learned the behavior of a human operator and chose actions that suited the operator in an optimal way in order to assemble the product in Figure 5.1b while minimizing production lead time. The first human operation serves as a classification stage, in which the robot observes the skill of the operator and remembers that for the rest of the assembly. The classification maps to a certain behavioral pattern in the rest of the assembly and the goal is to learn this pattern and adapt to it. In this project the classification stage encompassed the placement of the springs, which could be placed at three different speeds. The idea was that the placement of the springs at a certain speed represented a certain operator profile which was not obvious. For instance, if the operator placed the springs slowly, then they would place part B fast. The robot had to choose actions that suited the operator's profile. The actions the robot could choose from was whether or not to help the operator by presenting a box with the relevant parts, or at which speed the robot should perform tasks that had to be synchronized with the operator. The task to be synchronized are in this case the robot picking the screwdriver and screwing the middle screw while the operator places the screw in the fixture. The robot must learn not to be too fast for the operator to prevent screwing too early and not too slow to keep production lead time low. Tedious operations and badly synchronized tasks were punished according to equation 5.1 and 5.2. The synchronization step was used as a visual confirmation of the robot having adapted to the operator, as different operator profiles required different actions from the robot in order to be optimized. If the operator places part C slowly, then the robot cannot screw fast because then it would attempt to screw a product without any screw in place.

To realize the behavior described above, the reward function was defined as

$$r = -\frac{t_a}{10} - |t_{op} - t_{rob}| - 15000\xi \tag{5.1}$$

for the actions involving placing and screwing the middle screw (the actions in the bottom three states in Table 5.2). $\xi = 1$ if the synchronization fails (i.e. if $t_{rob} < t_{op}$) and $\xi = 0$ otherwise. t_a is the total duration of these actions i.e. the largest of the operator action duration t_{op} and the robot action duration t_{rob}. For the rest of the actions, which are performed alone the reward is simply defined as

$$r = -\frac{t_a}{10}. \tag{5.2}$$

The specific multipliers in equation 5.1 and 5.2 were chosen accordingly to achieve a large enough ratio between the punishment of bad synchronization and the time penalty. Of course, they can be scaled differently. The large punishment of the synchronization can be motivated as to create a safe margin for the synchronized operations if there are any uncertainties or randomness in the human behaviors. Based on measurements of action times and knowledge of how the human operators are simulated, optimal Q-values were calculated for evaluation of the learning process. They are presented in Table 5.2. Note that this would not be possible to calculate in a more realistic application when the operator behavior is unknown. Moreover, the values shown in the table are generated based on the largest possible robot speed and three distinct operator speed profiles. The operator profiles were chosen so as to make it beneficial for the robot to alternate actions depending on the operator and are defined in Table 5.1.

Time-efficiency was chosen as the optimization parameter to be minimized in this project due to simplicity and intuitiveness, however this could be altered to many other factors e.g. product assembly quality. Alike the optimization parameter, the classification stage can be defined in many different ways, e.g with more operations or with parameters such as age, weight or years of experience. The classification can be dynamic so that it changes on a daily or weekly basis to cover temporary conditions such as exhaustion or sickness.

Operator profiles			
	Operator 1	**Operator 2**	**Operator 3**
Place springs (Classification)	Slow	Medium	Fast
Pick and place B	Fast	Medium	Slow
Pick and hold C	Medium	Fast	Slow
Pick and place Midscrew	Slow	Fast	Medium
Pick and place Sidescrews	Slow	Medium	Fast
Place and press C on AB	Slow	Fast	Medium

Table 5.1: Operator profiles. Based on observations of the operator placing the springs, the skill level of the other human operations are learned.

Optimal Q-values for each action			
State	**Present box**		**Do nothing**
Springs placed operator 1	-1927		-2500
Springs placed operator 2	-2234		-1815
Springs placed operator 3	-1412		-1500
B placed operator 1	-2440		-2696
B placed operator 2	-2144		-1527
B placed operator 3	-1107		-1603
C held operator 1	-3049		-2808
C held operator 2	-1250		-1530
C held operator 3	-1351		-2122
Middle screw placed in C operator 1	-3296		-2583
Middle screw placed in C operator 2	-880		-1410
Middle screw placed in C operator 3	-1668		-1938
	Screw slow	**Screw medium**	**Screw fast**
Side screws placed operator 1	-2572	-16519	-17715
Side screws placed operator 2	-5912	-2471	-1009
Side screws placed operator 3	-6616	-3175	-1713

Table 5.2: Optimal Q-values are shown for the possible robot actions in each state. The actions are related to the three distinct operator profiles chosen for the model. The action "Do nothing" corresponds to a passive robot, allowing the operator to perform the operation by itself.

Many Q-values are in a close interval and thus hard to extinguish in the subsequent plots of the Q-values. However, the actions in the last three states that risk the large punishment if they do not synchronize with the operator are easy to verify in the plots. Convergence was defined as reached when the value of the loss function (MSE of the TD-error) was less than 1000 which proved to be enough for learning the correct sequence of actions in all methods described below.

To evaluate the model and finding which methods are possible and suited to this and similar applications, the three strategies below have been evaluated. Implementation and results from the respective strategy are presented in Section 5.1 and 5.2.

Tabular Q-learning The Q-values are stored in a table in which each row represents a state and each column represents the eligible actions in the respective state. The method was implemented according to algorithm 1 in Section 2.1.

Linear function approximation The Q-values are represented as a linear function. The function takes a feature vector representing a certain action in a certain state as input, and returns the Q-value for that state and action as a linear combination of the elements in the feature vector. The method was implemented according to algorithm 2 in Section 2.2.1.

Nonlinear function approximation The Q-values are represented as a nonlinear function generated through a neural network. The input is a feature vector as in the linear case which is fed to the network that then returns its corresponding Q-value. The method was implemented according to algorithm 3 in Section 2.2.2.

Finally, experience replay as described in Section 2.5 was evaluated for both linear and nonlinear model training. Two separate methods were developed for this purpose, referred to as *Standard* and *Threshold* experience replay strategies. In the Standard experience replay strategy, the data is saved to a replay memory until the memory reaches a specified size at which point the update algorithm proceeds to loop through the data a number of times. Afterwards, it will empty the replay memory and wait until it has reached the specified size again. The same general principle is applied to the Threshold strategy as well, however in this strategy the replay memory will only be emptied if the loss is under a certain threshold. This adds a dynamic sense to the algorithm since the emptying of the replay memory will not be tied to a constant.

The linear model has proven to converge both with and without experience replay. In Section 5.2.1.1, the convergence of the linear model is compared between no experience replay and the Threshold strategy. For the nonlinear model, experience replay is required in order for the model to converge. Hence, only the Standard and Threshold strategies are compared for the nonlinear model in Section 5.2.2.1. The specific number of loops in the Standard strategy was chosen to be 20, and the threshold in the Threshold strategy was set to 35000. These values were chosen through trial and error. The number of operator profiles was limited to three because of implementation difficulty, however the number of profiles can be potentially infinite, something that is discussed and addressed with function approximation in Section 5.2.

5.1 Tabular Q-learning

(a)

Action	Description
1	Do nothing
2	Present box with part B
3	Present box with part C
4	Present box with the middle screw
5	Present box with the side screws
6	Pick screwdriver and screw middle screw, speed level slow
7	Pick screwdriver and screw middle screw, speed level medium
8	Pick screwdriver and screw middle screw, speed level fast

(b)

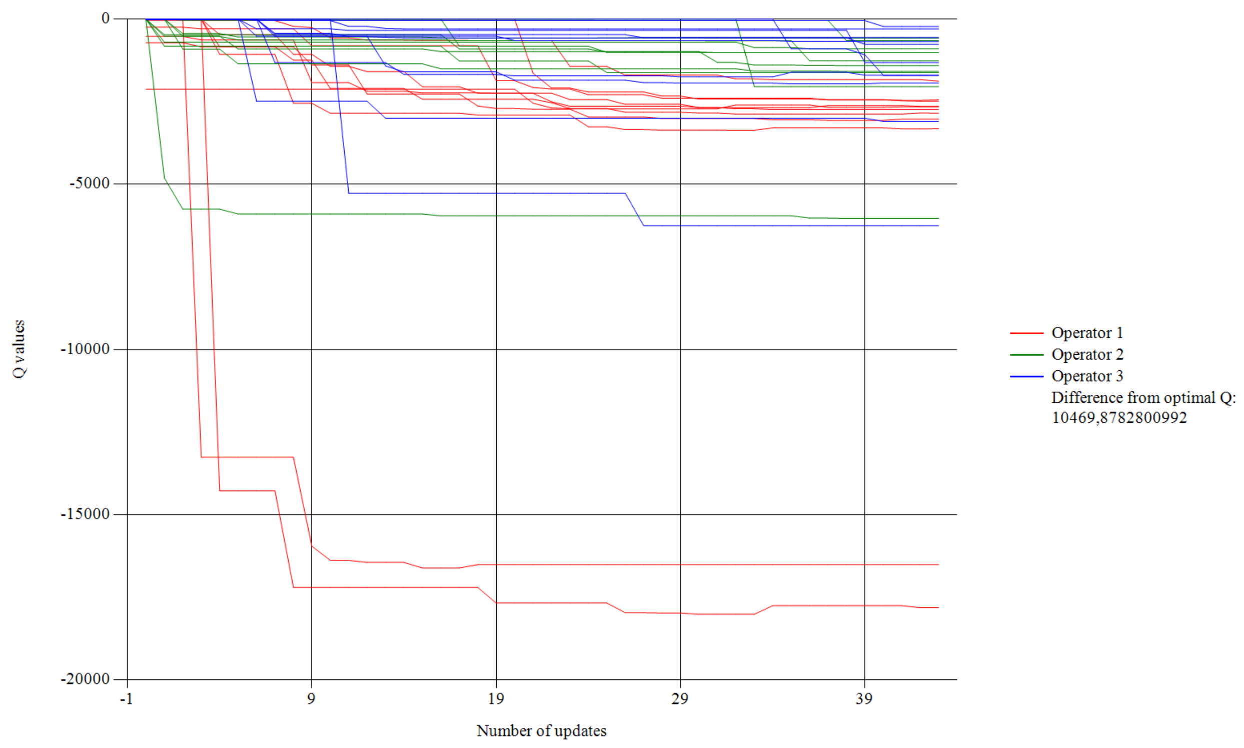
State	Description	Possible Actions
1	Springs placed operator 1	1, 2
2	Springs placed operator 2	
3	Springs placed operator 3	
4	B placed operator 1	1, 3
5	B placed operator 2	
6	B placed operator 3	
7	C picked operator 1	1, 4
8	C picked operator 2	
9	C picked operator 3	
10	Middle screw placed in C operator 1	1, 5
11	Middle screw placed in C operator 2	
12	Middle screw placed in C operator 3	
13	Side screws placed operator 1	6, 7, 8
14	Side screws placed operator 2	
15	Side screws placed operator 3	

Table 5.3: Table 5.3a shows the action numbering and describes the actions available in the assembly station. Table 5.3b shows the state numbering, describes them and denotes the possible actions in each state.

All actions available for the robot are listed in Table 5.3a but only some of these actions are possible to choose depending on the current state. The states include the situations where the robot has to make a decision how to act and they are described in Table 5.3b together with the actions possible to choose in each state. In this project, learning only happens on the robot, while the operator is considered as part of the environment, which is why only robot actions are included in the Q-learning models.

5.1.1 Convergence analysis

(a)



(b)

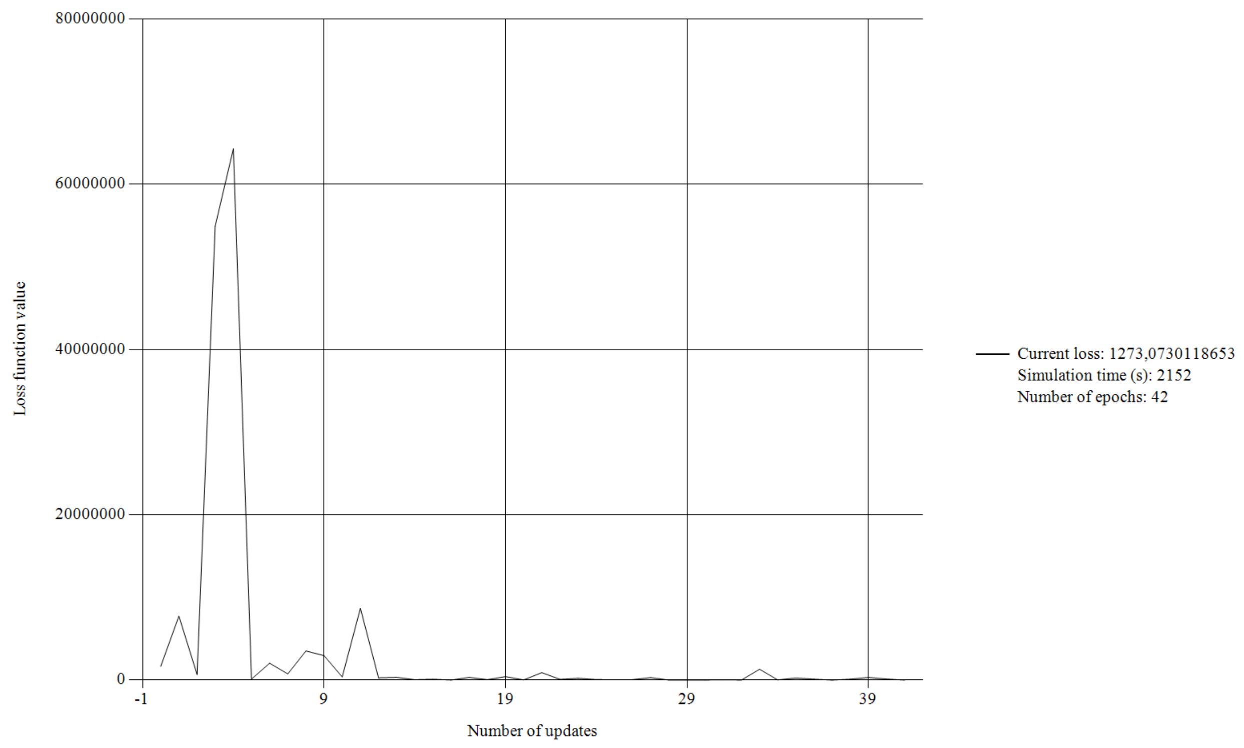


Figure 5.2: Figure 5.2a and Figure 5.2b show the Q-values and MSE respectively during training of the assembly station using tabular Q-learning.

Using tabular Q-learning for three operators, convergence was reached after 2152 simulated seconds (Figure 5.2). This is faster than the model described in Section 3 which is mainly due to fewer action values (34 compared to 56) but is affected by other factors as well such as exploration rate and the reward system. Clearly, the four values below -5000 indicate that the robot has been punished for bad synchronization and will not repeat those mistakes when following the learned target policy.

5.2 Q-learning with function approximation

As described in Section 2.2, for problems dealing with very large state-spaces, tabular Q-learning proves cumbersome and inefficient. Instead of having to define a Q-matrix with a set number of states and actions, function approximation has been used. This approach is based on defining a feature vector that contains information of the state and action of which the Q-value is sought for. Additionally, a model that takes the feature vector as input and produces the Q-value as output is defined. Two model classes will be explored in this project: linear and nonlinear function approximation, the latter implemented as a neural network. The training of the model is performed through optimization of the model weights in order to minimize some loss function, in this case the mean square TD error.

For this application, the feature vector was defined as in Table 5.4. Note that the operator profile is only defined by the time it took for the operator to place the springs and does not have to be categorized into operator 1, 2 and 3 as have been done in the tabular approach. Hence, with the feature vector it is easy to cover an unlimited number of operator profiles.

Element	Value
1	The time in seconds it took for the operator to place the springs
2	1 if the springs are placed and the choice of action is 2, else 0.
3	1 if the springs are placed and the choice of action is 1, else 0.
4	1 if part B is placed and the choice of action is 3, else 0.
5	1 if part B is placed and the choice of action is 1, else 0.
6	1 if part C is placed and the choice of action is 4, else 0.
7	1 if part C is placed and the choice of action is 1, else 0.
8	1 if the middle screw is placed and the choice of action is 5, else 0.
9	1 if the middle screw is placed and the choice of action is 1, else 0.
10	1 if the side screws are placed and the choice of action is 6, else 0.
11	1 if the side screws are placed and the choice of action is 7, else 0.
12	1 if the side screws are placed and the choice of action is 8, else 0.

Table 5.4: Feature vector element description used for both linear and nonlinear function approximation. It is motivated by and connected to Table 5.3

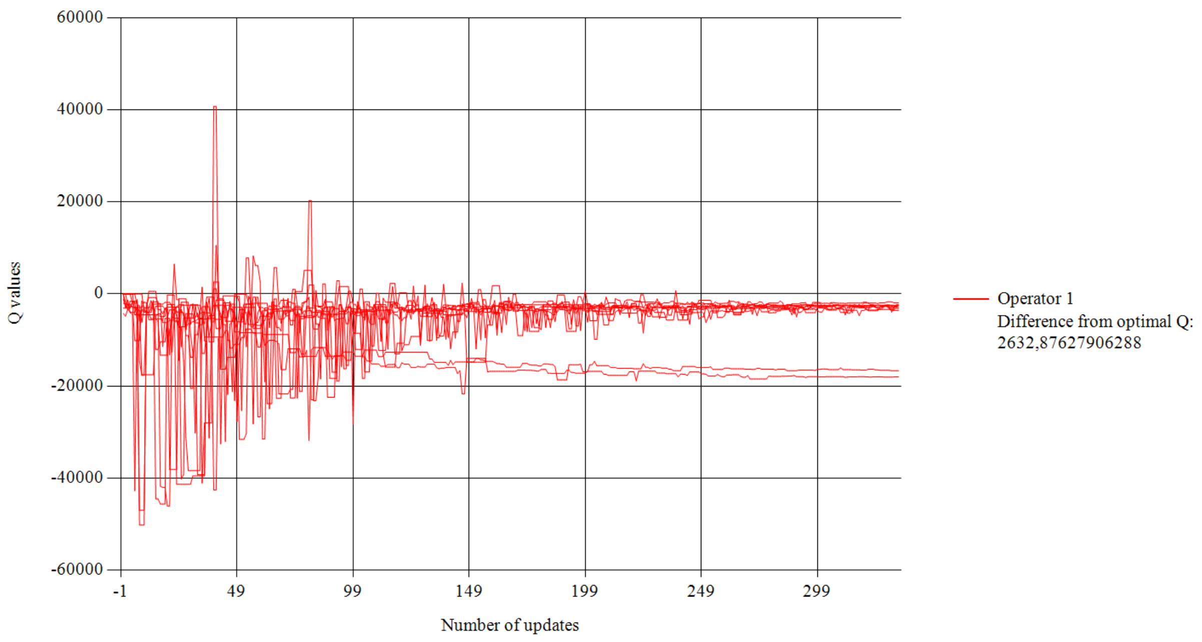
For example, the feature vector

$$\mathbf{x}(s,a) = [8.2\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0]^T$$

corresponds to part B being placed, it taking 8.2 seconds for the operator to place the springs correctly in the product and the action chosen is that the robot presents the box with part C to the operator. Details of how this vector is combined with the weights and how they are optimized are described in Section 2.2.1.

5.2.1 Linear function approximation

(a)



(b)

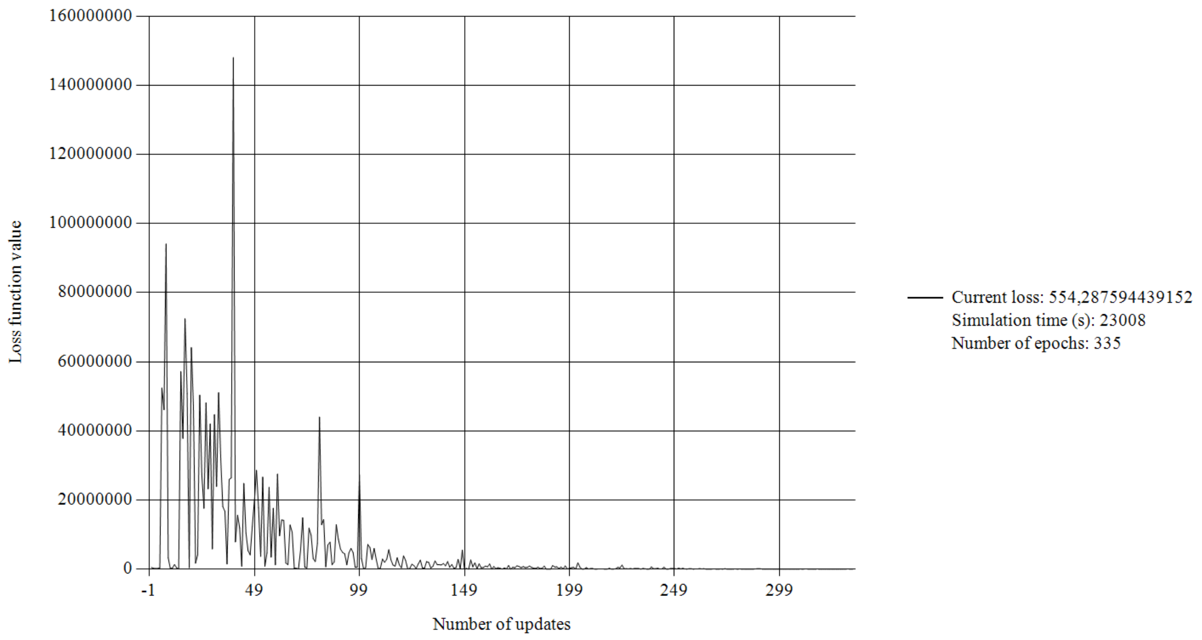
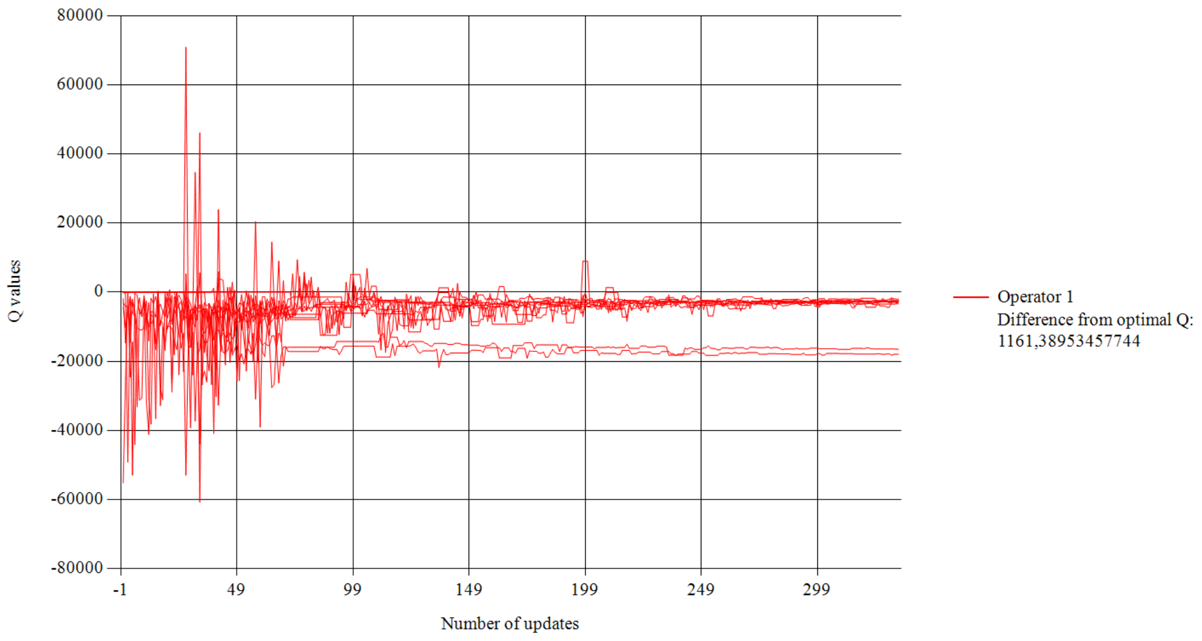


Figure 5.3: Figure 5.3a and Figure 5.3b show the Q-values and MSE respectively during training of the assembly station using linear function approximation for operator 1 without experience replay.

Using this model the weights converged after 23008 simulated seconds if only one operator was simulated, as seen in Figure 5.3. Note specifically the two action values that are considerably less than the others, which characterizes operator profile number one. However, simulating multiple operators, the weights never converged while it was observed earlier that tabular Q-learning managed to converge for all three operators simultaneously in only 2152 seconds as seen in Figure 5.2. The fact that only one operator converged with this feature definition is logical since the same weight must alone map different Q-values which is impossible. This is a result of all elements except the first one of the feature vector remaining unchanged for similar actions between different operators. Results from training only operator 1 are presented above. To obtain Q-values for operator 2 and 3 they have to be trained separately. Results for operator 2 and 3 are presented in appendix A.1. It would be possible to define a feature vector that can be used to train all three operators in the same linear model but that would require separate feature elements for each operator. In that case it would not be possible to gain the advantages over the tabular Q-learning model that function approximation normally does.

5.2.1.1 Convergence using experience replay

(a)



(b)

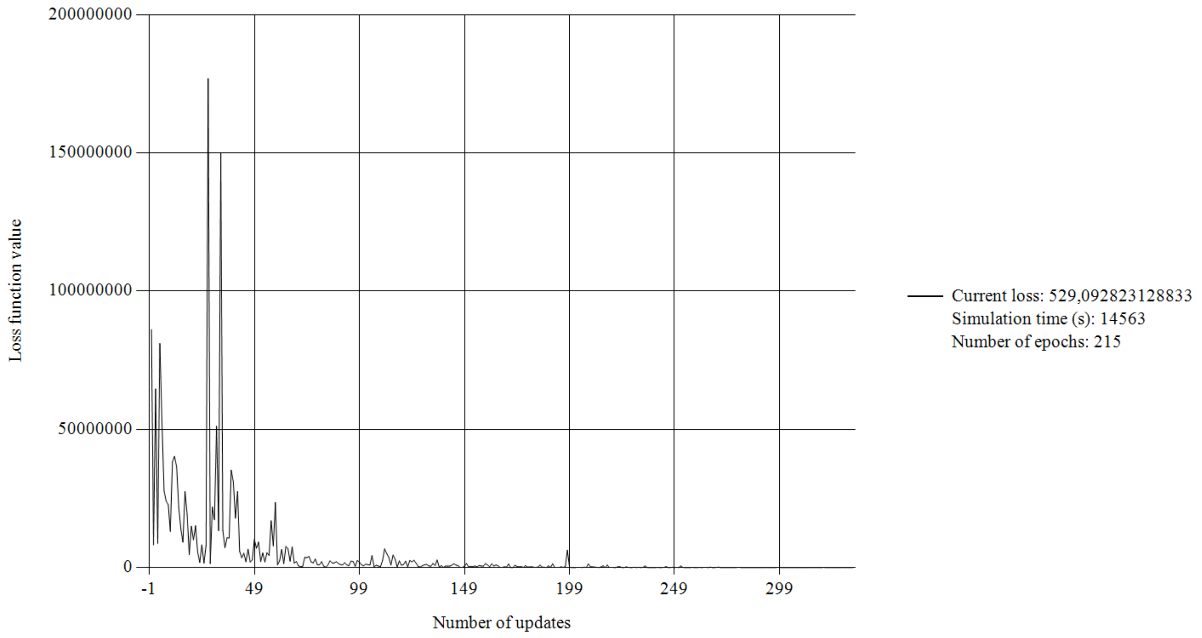


Figure 5.4: Figure 5.4a and Figure 5.4b show the Q-values and MSE respectively during training of the assembly station using linear function approximation for operator 1 with the use of experience replay.

A comparison has been made between training without experience replay (Figure 5.3) and applying the Threshold experience replay strategy (Figure 5.4). While these graphs show the result for operator 1, corresponding graphs for operator 2 and 3 are found in appendix A.1. It is clear that experience replay reduced the time to convergence considerably (14563 compared to 23008 seconds) and that during these times approximately the same number of updates were performed. Furthermore, it can be observed that without experience replay there are negative peaks below −20000 until around update number 100 while the corresponding limit when applying experience replay is found at around update number 70. It can also be noted that with experience replay, both the Q-values and and the loss have a slightly more oscillating behavior. This is an indication that experience replay tends to pass a number of local optima before it finds the global optimum.

Replay memory strategy comparison for linear model				
Test	Strategy	Simulation time (s)	Error	Avg. simulation time (std. dev.) Avg. error (std. dev.)
1	No exp. replay	16367	3046	14632 (3084) 1683 (939)
2	No exp. replay	18731	1735	
3	No exp. replay	11602	1095	
4	No exp. replay	14830	570	
5	No exp. replay	11628	1969	
6	Threshold	4795	1018	4677 (2288) 1377 (535)
7	Threshold	7876	800	
8	Threshold	1586	2134	
9	Threshold	3816	1682	
10	Threshold	5313	1251	

Table 5.5: A comparison between training without experience replay and applying the Threshold experience replay strategy for the linear model of operator 2.

The largest difference between the strategies was observed for operator 2. Therefore, a more exhaustive analysis with five runs for each strategy was conducted for operator 2 presented in Table 5.5. It appears that experience replay not only reduces the convergence time but also reduces the error i.e. the sum of the distances from the optimal Q-values found in Table 5.2. Since the convergence limit on the loss is the same for both strategies, this result implies that experience replay counteracts the risk of getting stuck in local optima. This is interesting because it was noticed in the plots for operator 1 that during training, experience replay tends to pass more closely to local optima than in the case without experience replay. A final note is that the results among the runs becomes less variant.

5.2.2 Nonlinear function approximation using neural networks

Unlike the previous case with linear function approximation, the nonlinear approach allows multiplication of weights with each other which means that fewer feature elements will be necessary to converge to the same optimal Q-value. With this model it would therefore be possible to have an adaptable solution that would not require any specific code in order to adapt to potentially infinitely many different operators.

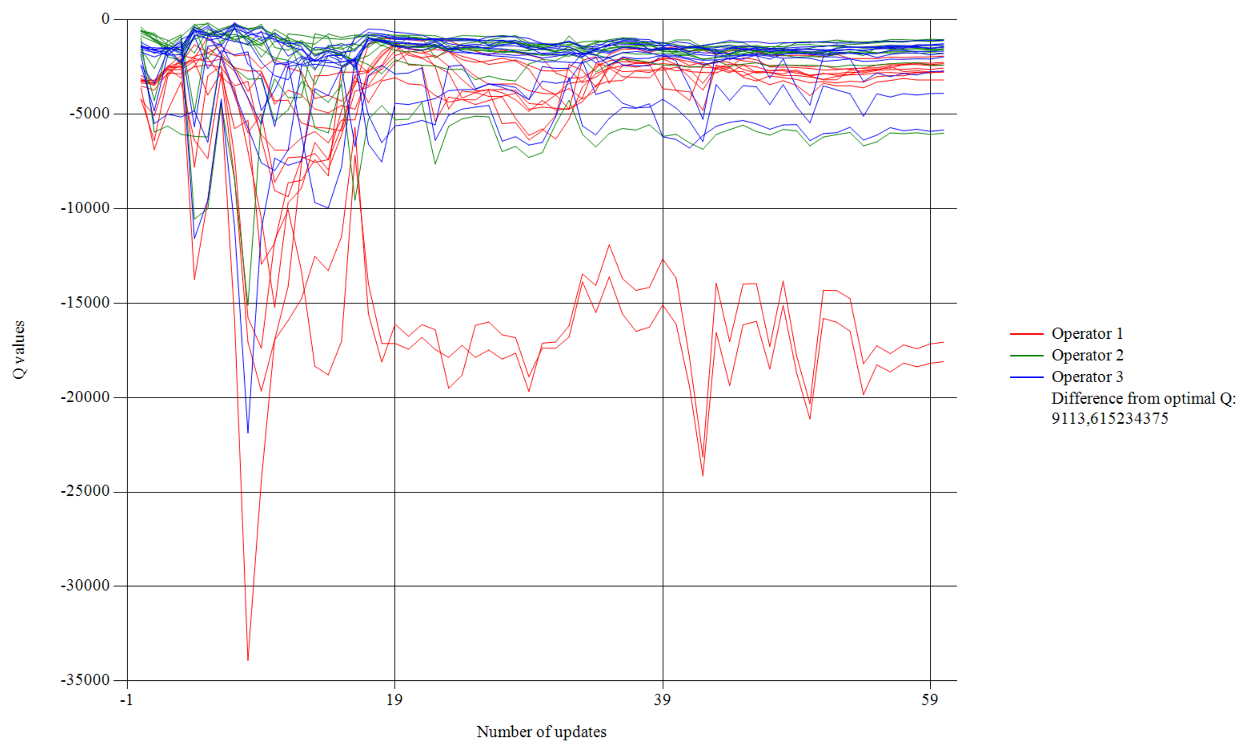
As explained in Section 2.2.2.4 there are no general methods for finding an optimal network architecture. In this project, heuristic search was used based on the recommendations from Heaton [12]. As the evaluation of each network is extremely tedious in this application, a model for fast initial evaluation is developed where the RobotStudio simulations are replaced by feature vectors and corresponding rewards stored in a data file. The data is based on earlier simulations with some noise added making the data more similar to actual simulations. This model is discussed in more depth and presented in Section 5.2.2.2.

#Layers	Hidden dim.	Activation	α	Optimizer	#Updates	#Failures
2	8	ReLU	0,015	Adam	-	5
2	60	ReLU	0,015	Adam	9240	2
2	100	ReLU	0,015	Adam	3400	3
2	140	ReLU	0,015	Adam	4300	1
3	100	ReLU	0,015	Adam	1620	0
4	**100**	**ReLU**	**0,015**	**Adam**	**1080**	**0**
5	100	ReLU	0,015	Adam	1360	0
4	100	ReLU	1.5e-10	SGD	-	5
4	100	ReLU	0,99	RMSProp	-	5
4	100	ReLU	0,001	Adam	5340	0
4	100	ReLU	0,02	Adam	1260	0
4	100	Tanh	0,015	Adam	-	5
4	100	Sigmoid	0,015	Adam	-	5

Table 5.6: This table list the performance of the neural networks evaluated. A training is considered failed when the number of updates without convergence exceeds 20 000.

As opposed to the linear case, the simple feature vector as defined in Section 5.2.1 can be used as input to one single model that maps the correct output (Q-values) for all three operators. This indicates that non-linear models have great potential of scalability to a vast amount of operator profiles. Based on the heuristics some reasonable hyperparameters of neural networks were evaluated. The result is presented in Table 5.6. An interesting result is that the first test using the general recommendations could not be used in this application. Instead a much bigger network is necessary for convergence. Fastest convergence was achieved using four layers, each with 100 hidden units. When implementing large neural networks, the risk of overfitting should be considered. However, as explained in Section 2.2.2.3 overfitting can actually be reduced if the number of weights exceed a certain point that depends on the number of training data samples. In this simple application, the data is scarce and it is highly probable that the network found contains many more parameters than data samples explored during training. Importantly, the weights have been initiated to small numbers which is necessary for the undesirable high dimensions to be kept negligible.

(a)



(b)

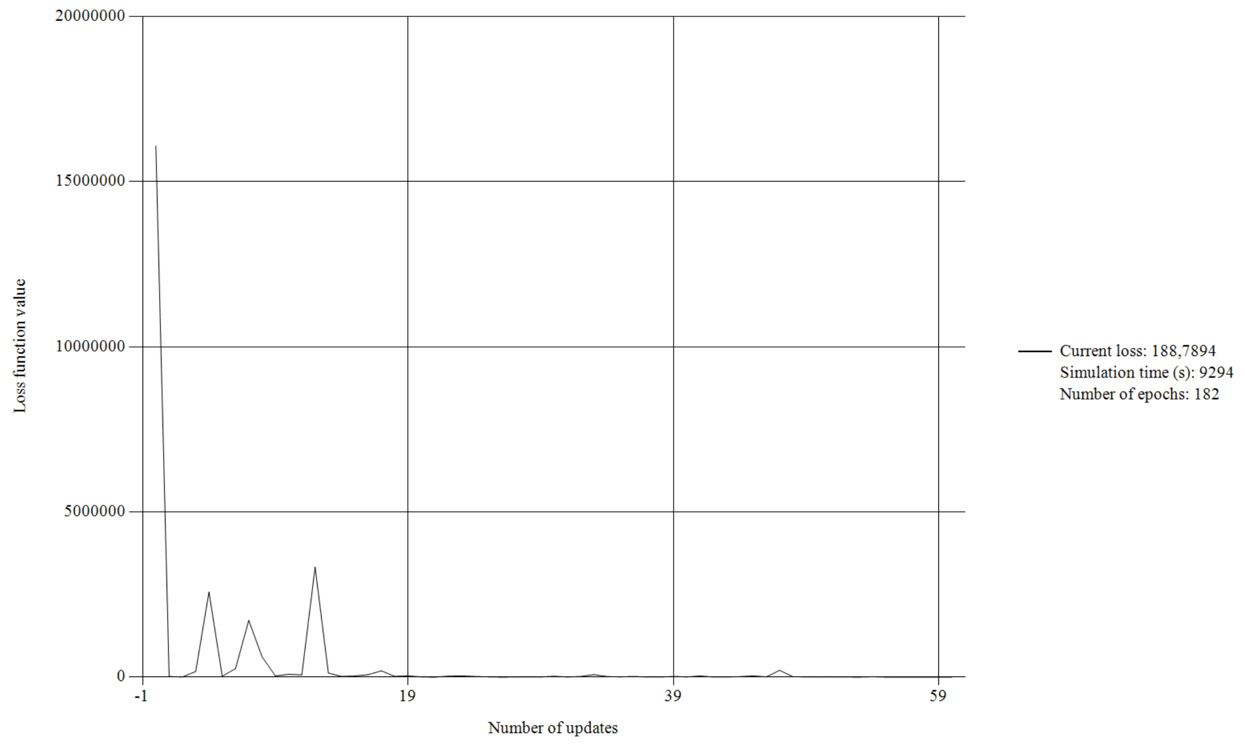
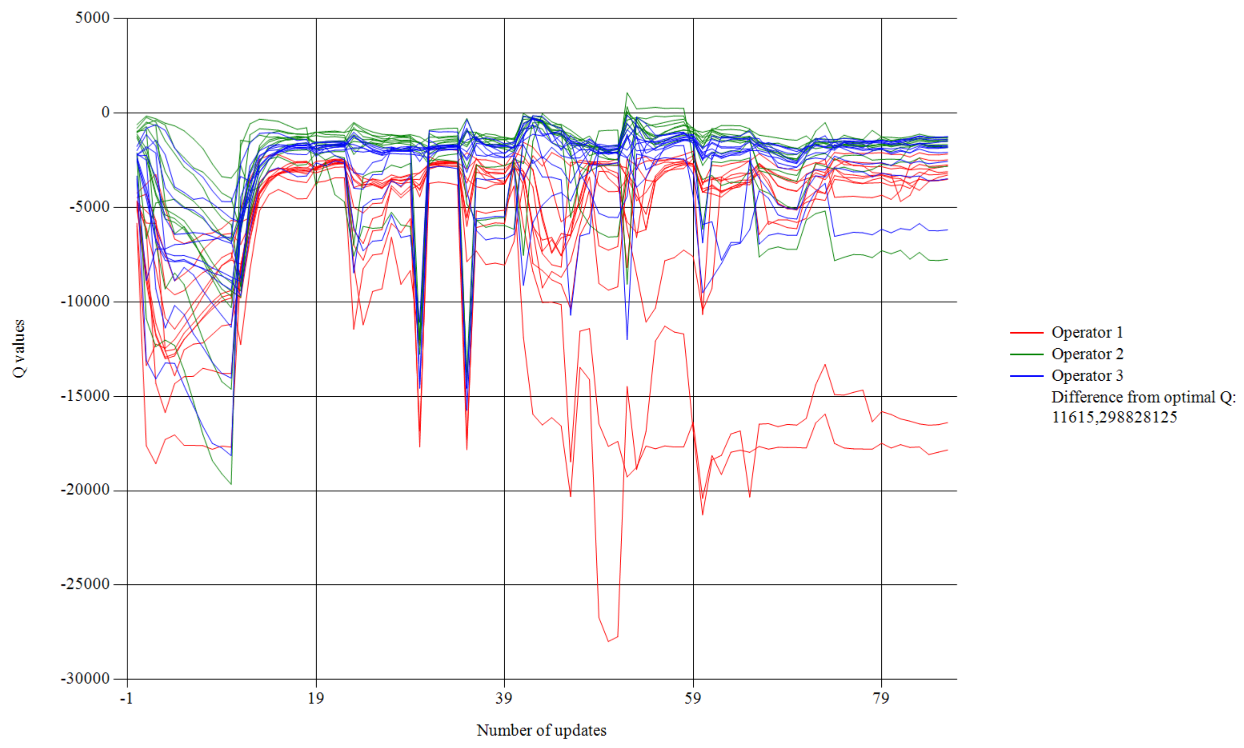


Figure 5.5: Figure 5.5a and Figure 5.5b show the Q-values and MSE respectively during training of the assembly station using nonlinear function approximation with the high performance neural network that is marked with bold figures in Table 5.6. Here, the Standard experience replay strategy as described in the introductory part of this section is applied.

Convergence dynamics for the best neural network found is presented in Figure 5.5. Compared to the tabular convergence in Figure 5.2a it takes around 4 times longer in simulation time for the nonlinear model to converge with similar level of accuracy. It is also clear that the nonlinear Q-values are more scattered as a result of each update affecting all Q-values. The result in Figure 5.5 comes from using the Standard experience replay strategy.

5.2.2.1 Comparing two experience replay strategies

(a)



(b)

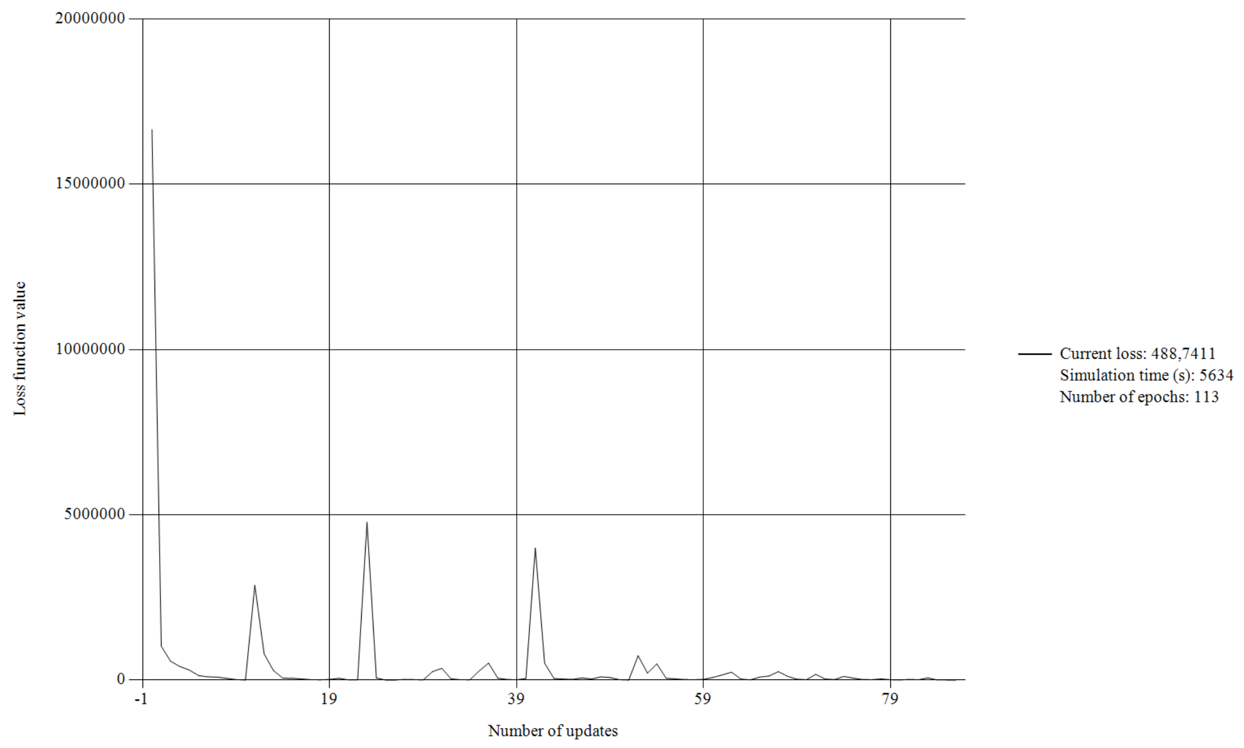


Figure 5.6: Figure 5.6a and Figure 5.6b show the Q-values and MSE respectively with the same network as in Figure 5.5 but with the so called Threshold experience replay strategy as explained in the introductory part of this section.

In Figure 5.6 the effect of the Threshold experience replay strategy is clear. A local optimum is found very fast; after around 10 updates. By then the model reaches the loss threshold at which the data trained on is erased which of course inflates the loss. The replay memory needs to be erased a number of times, i.e. a number of local optima have to be passed, before approaching the global optimum. This makes the number of updates larger compared to the standard experience replay method (see Figure 5.5). By observing the Q-values for operator 1 that are large in magnitude it can be noticed that the global optimum is approached after approximately 40 updates in Figure 5.6a whereas it is approached already after approximately 20 updates in Figure 5.5a. However, reuse of old mini batches of data enables the algorithm to perform a larger number of updates in less time as opposed to having to wait for a new mini batch after each update. As a result, the simulation time before convergence decreases considerably in this case: 5634 seconds compared to 9294 seconds.

Replay memory strategy comparison for nonlinear model				
Test	Strategy	Simulation time (s)	Error	Avg. simulation time (std. dev.) Avg. error (std. dev.)
1	Standard	5795	24181	9326 (5886) 12352 (5260)
2	Standard	18328	10147	
3	Standard	2968	18336	
4	Standard	6830	7921	
5	Standard	19115	9921	
6	Standard	14690	7264	
7	Standard	4546	11949	
8	Standard	5764	10568	
9	Standard	5934	14123	
10	Standard	9294	9114	
11	Threshold	1414	67840	4105 (3090) 44101 (43631)
12	Threshold	165	149016	
13	Threshold	2096	29986	
14	Threshold	5634	11615	
15	Threshold	7073	14133	
16	Threshold	844	65109	
17	Threshold	8122	9835	
18	Threshold	5547	25665	
19	Threshold	8017	8943	
20	Threshold	2142	58869	

Table 5.7: The table shows a comparison between the Standard and Threshold experience replay strategies through 20 tests.

Since the nonlinear model has proven to have great potential of scalability, a more rigorous analysis of the experience replay strategies was motivated. Indeed, the observations when comparing Figure 5.5 and 5.6 are verified in Table 5.7. The simulation time in the Threshold strategy is considerably less than the one in the

Standard strategy. On the other hand, the threshold approach produces much larger errors (equation 3.1). Since the convergence requirement is at the same MSE (equation 2.6) for the two strategies this implies that the threshold approach easier gets stuck in local optima. This can be seen by looking at the measurements in Table 5.7 as one notices a few conspicuous values of the Threshold strategy. There is a measurement that converges after 165 seconds with an error of 149016, and another one that converges after 844 seconds with an error of 65109. It is possible that one could refine the Threshold strategy to avoid such early convergence, for instance checking convergence with larger intervals, and in that way arrive at a strategy better than both the Standard and Threshold ones.

5.2.2.2 Model without RobotStudio

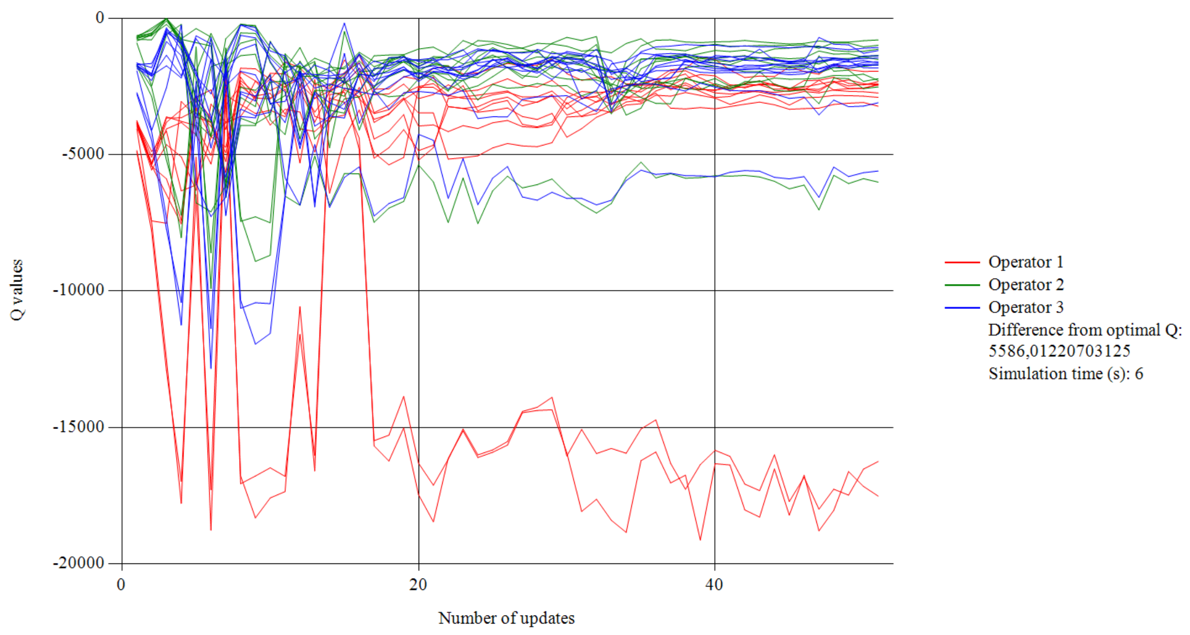


Figure 5.7: This figure shows the Q-values during training of the model that reads data based on earlier measurements from a file instead of receiving data continuously in real-time from the simulation.

The plot in Figure 5.7 is provided to validate that the model without the simulations is sufficiently close to the model with the simulations so that it can be used when searching for a suitable neural network. It also serves to show that it is possible to solve the adaptability problem offline using predefined values. Comparing the Q-value progress plots for both models (Figure 5.5a and 5.7) shows that this model is valid substitute for the RobotStudio one, i.e. a noise level was found that fit well to the one in the simulation model.

5.2.3 Transferability to the physical station

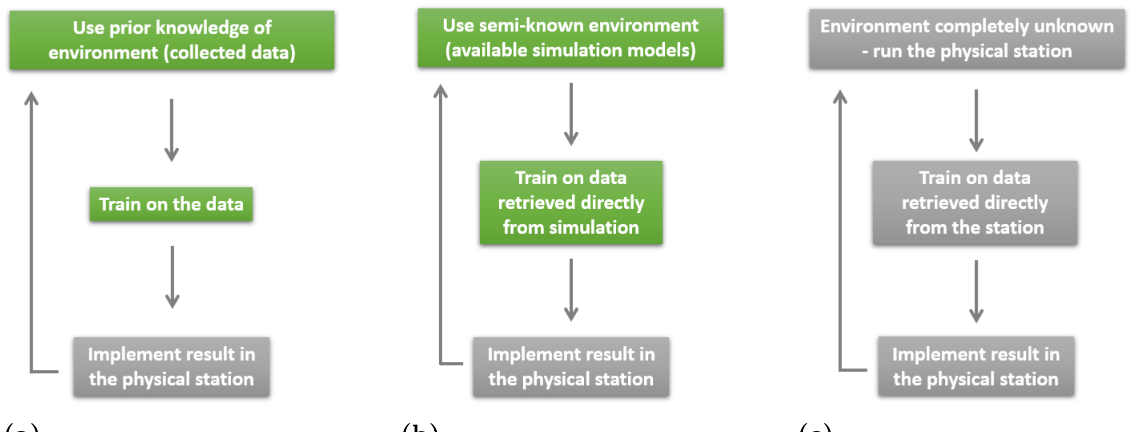
Even though more work needs to be done for the models used in this project to be transferable to the physical station, it has been considered during development of the tools used. The human actions have been simulated by making objects fly along paths with as realistic speeds as possible (using *Smart Components* which is a tool in RobotStudio) as to mirror how a human would move the objects. Ideas of how to make this even more realistic is discussed in Section 6.4.

The main issue regarding transferability to the physical station is how the RL algorithm would communicate which actions the physical robot should explore. Conveniently, the exact same communication tools used in this project for communication between the algorithm and the robot's virtual controller in the simulation environment, can be used for communication between the algorithm and the physical controller. Both the virtual controller and the physical controller execute the same RAPID code and it is possible to establish a server in and communicate via a WebSocket directly in RAPID. Syntax used for these tools are listed in appendix A.2.

6
Discussion

Many applications can be solved using already existing AI tools such as PDDL as described in Section 2.7.5. However, since this project was based on RobotStudio, which is developed in C#, it was thought that using the same development tools could make this solution and future development smoother and more modular. Furthermore, C# has an extensive catalogue of features and documentation which makes it easier and faster to work with. Specially when it comes to perhaps more specialized tools such as WebSockets that might not be developed in AI tools like PDDL. Far from saying that these tools cannot be used in this application, their omittance is more due to time constraints and their niche nature.

In Section 1.1, three research questions are stated. The first research question concerns how closed loop manufacturing can be realized using RL and challenges in developing a simulation model that can be used to combine training on simulation and on a corresponding physical station. In Section 5.2.3 and 6.4 issues and solutions regarding the transferability to a physical station have been discussed. Furthermore, in Section 5.2.2 it was explained that for quick evaluation of different neural network architectures, the RobotStudio simulations were completely replaced by data received from the simulations with some noise added. The data could of course have been collected from the physical station instead of the simulation model for more realistic results. One can therefore question why the tedious simulations are needed for training in the first place. Actually, in several situations it is valuable to be able to train on simulations instead of (or in addition to) the physical station. If the physical station does not yet exist training can be performed on the simulation model. Training can continue on the physical model when built if desired. Further, the physical station might not be available for training or measurements due to risk of injuries/damage or inaccessibility. Lastly, when the complexity of the problem is such that it is hard to create data suitable for training, it is easier to simply interact with the environment directly, either on the physical station, in a simulation model or a combination. One example is that more complex human operating behavior might be hard to measure or recreate and it would be easier to learn continuously from simulations using VR. Another example is when the robot has to interact with irregular objects, objects that are posed randomly or objects with unknown features. Then it is easier to interact directly with the environment or a physics engine in simulations to learn an optimal way of working.


Use prior knowledge of environment (collected data)
Train on the data
Implement result in the physical station
Use semi-known environment (available simulation models)
Train on data retrieved directly from simulation
Implement result in the physical station
Environment completely unknown - run the physical station
Train on data retrieved directly from the station
Implement result in the physical station

(a) **(b)** **(c)**

Figure 6.1: Three approaches to create a closed loop manufacturing system using RL. Figure 6.1c shows training on data based on earlier measurements. Figure 6.1b shows training based on data retrieved from simulations. Figure 6.1a shows training based on data received directly from a physical assembly station.

In this project, three approaches that relate differently to closed loop manufacturing have been treated as visualized in Figure 6.1. The green areas represents successfully implemented areas in this project. The aim has been to enable future development towards the physical implementation in Figure 6.1c or a combination of training on simulations (Figure 6.1b) and reality. Further discussion regarding this can be found in Section 6.4.

One of the most important frameworks successfully implemented in this project concerns the simulated closed loop as mentioned in the first research question in Section 1.1. It can be thought of an inner loop that enables the training process in the central block in Figure 6.1b.

The second research question concerns what methods are suitable for learning an optimal assembly sequence and recognizing patterns in the behavior of a human operator. It has been shown that tabular Q-learning is fastest given a limited number of states and actions. For problems with few variations yet non-trivial solutions, tabular Q-learning is advised. This contradicts what Ehn and Werner [23] found in their research (see Section 2.7.1). One reason is that in this project where the size of the weights varies greatly, few weights that are far from their optimal values affect the updates of all other weights negatively which makes learning slower. As soon as the state definition becomes more complex, the advantages of function approximation should be exploited. Comparing Section 5.1.1 and 5.2.1, it becomes apparent that the performance of linear function approximation does not beat tabular Q-learning. It takes more iterations before convergence and it is hard to find a scalable feature representation that works in a linear mapping from the states and actions to the respective Q-values. However, this has successfully been solved with nonlinear function approximation using neural networks. A compact and scalable feature representation maps to the respective Q-values within a reasonable amount of iterations. The research question also asks how these strategies compare to traditional optimization methods, and this has been treated in Section 2.7.2 and 2.7.4.1. As similar research projects have already found, for complex and unintuitive applications, when traditional optimization stops being a realistic approach, RL is a perfect transition. As Özçelikkale, Koseoglu and Srivastava [24] pointed out, the risk of errors due to erroneous environment assumptions is eliminated in RL since no assumptions are required. For learning more realistic and complex human behavior than implemented in this project, traditional optimization is inconvenient to implement since it would require knowledge of the environment. Another advantage underlined by Owens [29] is the flexibility and scalability as possible actions and state features can be added independently without having to redefine all connections as would be the case of a finite state machine.

When the problem complexity is increased learning becomes slower, why techniques for parallelizing learning have been investigated. This relates to the third research question, which concerns how the training process can be improved by using parallel simulation instances and how they should be orchestrated. It can be concluded that if possible, parallelization should be applied, since any method evaluated improves both speed and accuracy compared to using only one instance. The decision of which orchestration strategy to implement is then a question of how to balance speed and accuracy. For very simple problems, the centralized strategies seem to be the obvious choice. However, for simple problems, parallelization is less necessary. Problems will often be more complex than the model in which the robot stacks boxes on one pillar only. Then, the distributed learning strategies should be evaluated for the specific application. If speed is more important than accuracy, the distributed synchronized actors might be a good choice. Contrariwise, if accuracy is a priority, one of the centralized strategies will be a suitable choice. For the cases in this project, it is hard to motivate the distributed asynchronous learners which is always the slowest option and rarely the most accurate. While some of the patterns observed and commented may seem unambiguous, parameters such as exploration rate, interval between model updates, convergence definition and the discrepancy in speed between the instances might direct the result differently.

6.1 Experience replay

Experience replay proved highly successful for the linear model. It not only reduced the simulation time before convergence but also the error (or the risk of getting stuck in a local optimum). For simpler functions such as the linear one, gradients on a few weights have a large impact on the deviation between the function and the data. That causes old data to be rapidly forgotten. By continuously re-experiencing old data, the risk of forgetting them decreases which reduces the risk of converging at suboptimal solutions. This however is contradicted by comparing figures 5.3 and 5.4, which indicates that the algorithm passed local optima more often with experience replay than without. It can be concluded that experience replay increases the risk of getting stuck in local optima but when the model complexity is low, taking that risk is necessary in order to find the global optimum. Furthermore, the linear approximator was only trained on one operator at a time, that is the data did not vary much which makes it more probable that local optima coincides with the global optimum.

In a more complex model such as the nonlinear model with neural networks, gradients have a more local impact on the deviation between the model and the data. Thus, old data are more easily remembered and the risk of getting stuck in local optima decreases. Furthermore, the nonlinear model was trained on much more varying data and on larger batches than the linear model which resulted in a wide representation of the data set. Hence, the problem of forgetting old data becomes rare. Even so, the Threshold experience replay strategy seemed to undergo a significant risk of getting stuck in local optima. As opposed to the linear model, complex models have a larger number of local optima and overexploiting experience replay increases the risk of updating the model towards one of them, satisfying the convergence requirement too early. On the other hand, as briefly discussed in Section 5.2.2.1, one can imagine enhancing the Threshold strategy by altering the convergence check or perhaps by having a dynamic threshold in order to investigate if it is possible to find an improved strategy.

6.2 Reward strategies

In the box model potential-based reward shaping as described in Section 2.4 was applied. The rewards in Table 3.1 are examples of a potential-based shaping function being used and can be interpreted as some sort of inverted distance to the goal where higher rewards are delivered the closer to the goal the learning agent gets. Instead of only rewarding the agent once arriving at the goal state this strategy helped to speed up the learning.

Similarly in the assembly station model potential-based reward shaping was applied but as a function of production lead time instead of a distance. Higher rewards were achieved the faster the goal state was reached along with a bonus reward for successful synchronization on parallel operations.

Insofar as heuristic knowledge exists for a problem, a potential-based reward shaping function is recommended to be used as long as it is defined according to equation 2.11 and 2.12. As proved and demonstrated by Y. Ng, Harada and Russel [18] this can speed up the learning greatly without altering the optimal policy.

6.3 The deadly triad

As described in Section 2.2, instability in the sense of the Deadly Triad is achieved when function approximation, bootstrapping and off-policy training are applied simultaneously in an algorithm. In this project however, all three elements have been applied simultaneously on the assembly station model which still converged. In fact, it is not uncommon that the three elements of the deadly triad have been combined successfully according to van Hasselt et al [31]. Compared to their study, the applications in this project are very small when it comes to the state space. At the same time the use of reward shaping guides learning rather strongly. That might explain why these applications have been proved to be stable even though no consideration of the results from their study have had to be taken. If the application is scaled to more complex situations it is advised to increase the number of steps before bootstrapping which van Hasselt et al found to be the clearest prevention from divergence. Otherwise the network size in the case of nonlinear function approximation or the bootstrapping method (e.g. Double Q-learning) can be experimented with.

6.4 Further applications and future work

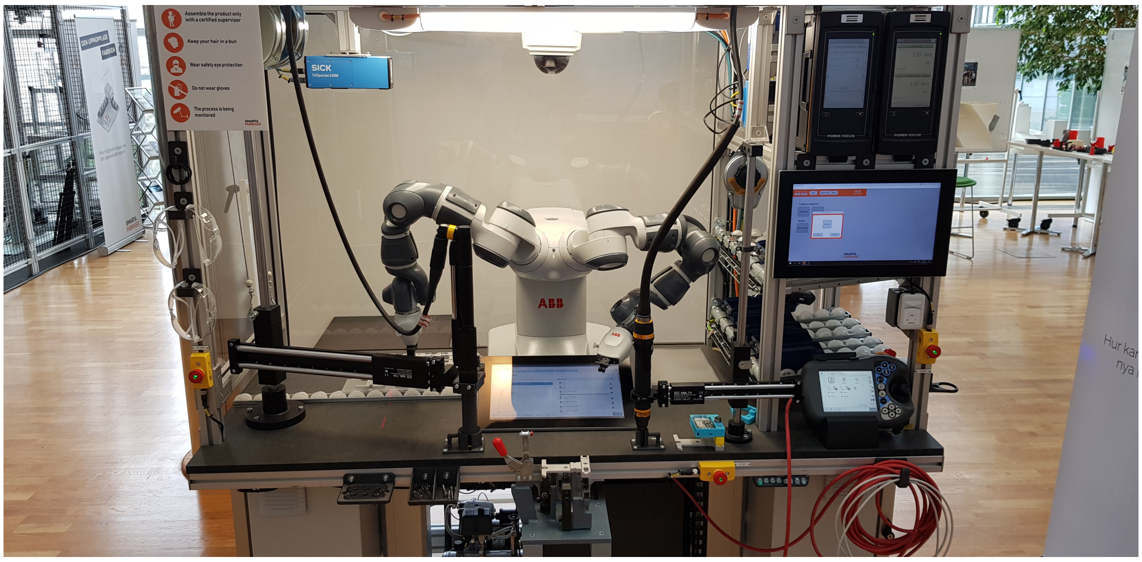


Figure 6.2: The physical assembly station that has been simulated in this project and can be used in future development.

In this project, the learning algorithms have been applied on a simulated virtual model. However, the assembly station is a close representation of a physical station seen in Figure 6.2. Furthermore, the tools created are in no matter bound to a virtual model or RobotStudio. With minor modifications the learning could be performed on the physical station as well which would be a very interesting case for future work.

One great feature of RobotStudio is that the exact same RAPID code in the virtual controller can be used in a real controller which makes it easy to alter between simulation and reality. The same holds for communication between the script that implements the algorithms and the robot. This opens up many opportunities for combining training in simulation and in the real world. As noted by Özçelikkale, Koseoglu and Srivastava [24] RL can effectively treat modeling and optimization as separate tasks. In this case, basic modelling can be performed on a simulated environment and final model tweaks and optimization can be performed on the real world. A challenge will be to minimize the simulation gap (see Section 2.7.3) so that the real world learning can profit from prior learning on simulations, specially when it comes to simulating human behavior. An intermediate task before moving completely to the real world could be to use virtual reality tools for simulating human behavior where an actual human interacts with the simulation environment. There are advantages with this approach e.g. it can be used before the physical station has been built and there is no risk for injuries in the early stages of evaluation.

The two cases built in this project serve only as examples and proof of concepts. The application could even scale to multitasking where the agent learns assembly sequences and human behavior patterns for many products instead of only one product. Only ones imagination limits what actions to define, what information a feature vector should store and what reward system to implement.

7 Conclusion

RL algorithms do not require a known model of the environment and thus gain advantage over traditional optimization techniques for complex problems such as human behavior. Although many simplifications in this project make the problems feasible to solve by traditional optimization, RL has the potential to manage more realistic applications. It has been discovered that tabular Q-learning is a fast method if the problem can be reduced to a small number of states and actions. If the problem becomes complex, with large number of states and actions, tabular Q-learning is impractical and in some cases infeasible. Instead, function approximation using a deep neural network, trained using experience replay, is a perfect choice of method. For the applications in this project, 4 layers with a large number of hidden units together with ReLU activation and Adam optimizer had great impact on decreasing the convergence time.

To increase speed further, training can be applied on parallel simulation instances using an orchestration strategy best suited for the problem. The research in this project has shown that parallel learning and environment exploration reduce convergence time and the deviation from the global optimal solution. Using 10 parallel instances on a moderately difficult problem decreased the convergence time by 50% and reduced the error by at least a factor of 50 compared to exploring with only one instance. It has been shown that for simple problems, centralized outperforms distributed learning. Additionally, it has been found that synchronized is always faster than asynchronized distributed learning. Synchronizing the actors leads to more intelligent exploration and thus faster convergence. However, faster convergence comes with the cost of larger errors in cases where error is nontrivial. Finally, transferability to the physical station was considered by keeping the simulation gap reasonably small and establishing a modular communication method.

The findings presented in this report are promising for future research. They show that it would be possible to learn not only an assembly sequence of a complex product but also how to adapt the process to a complex representation of a human operator. Instead of a collaborative robot being guided by humans, intelligent robots will start to guide and understand their human coworkers, enabling great possibilities for future industries.

Bibliography

[1] J. Walker, *Machine learning in manufacturing – present and future use-cases*, Last accessed 15 November 2018, 2018. [Online]. Available: `https://www.techemergence.com/machine-learning-in-manufacturing/`.

[2] B. Rusch. (2018). Abb launches start-up accelerator for industrial ai, [Online]. Available: `https://www.hannovermesse.de/en/news/abb-launches-start-up-accelerator-for-industrial-ai-106435.xhtml` (visited on 05/10/2019).

[3] L. Hibbert. (2018). How collaborative robots will usher in the era of industry 5.0, [Online]. Available: `https://www.theengineer.co.uk/collaborative-robots-industry-5-0/` (visited on 06/19/2019).

[4] R. Sutton, A. Barto, and F. Bach, *Reinforcement Learning: An Introduction.* MIT Press, 2018, ISBN: 9780262039246. [Online]. Available: `https://books.google.se/books?id=6DKPtQEACAAJ`.

[5] J. O'Hare, *Sensing your way through closed-loop manufacturing*, Last accessed 15 November 2018, 2017. [Online]. Available: `https://www.qualitydigest.com/inside/management-article/get-scoop-closed-loop-manufacturing-042517.html`.

[6] D. Silver, *Lecture 4: Model-free prediction*, University Lecture, 2015. [Online]. Available: `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MC-TD.pdf` (visited on 06/19/2019).

[7] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning", *arXiv preprint arXiv:1811.03378*, 2018.

[8] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks", *Chemometrics and intelligent laboratory systems*, vol. 39, no. 1, pp. 43–62, 1997.

[9] S. Ruder, *An overview of gradient descent optimization algorithms*, `http://ruder.io/optimizing-gradient-descent/index.html`, Accessed: 2019-04-24.

[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[11] M. S. Advani and A. M. Saxe, "High-dimensional dynamics of generalization error in neural networks", *arXiv preprint arXiv:1710.03667*, 2017.

[12] J. Heaton, *Introduction to neural networks with Java.* Heaton Research, Inc., 2008.

[13] D. Stathakis, “How many hidden layers and nodes?”, *International Journal of Remote Sensing*, vol. 30, no. 8, pp. 2133–2147, 2009.

[14] Y. LeCun, J. S. Denker, and S. A. Solla, “Optimal brain damage”, in *Advances in neural information processing systems*, 1990, pp. 598–605.

[15] B. Hassibi and D. G. Stork, “Second order derivatives for network pruning: Optimal brain surgeon”, in *Advances in neural information processing systems*, 1993, pp. 164–171.

[16] D. Dewey, “Reinforcement learning and the reward engineering principle”, in *2014 AAAI Spring Symposium Series*, 2014.

[17] M. Grześ, “Reward shaping in episodic reinforcement learning”, in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, International Foundation for Autonomous Agents and Multiagent Systems, 2017, pp. 565–573.

[18] A. Y. Ng, D. Harada, and S. Russell, “Policy invariance under reward transformations: Theory and application to reward shaping”, in *ICML*, vol. 99, 1999, pp. 278–287.

[19] M. Grzes and D. Kudenko, “Theoretical and empirical analysis of reward shaping in reinforcement learning”, in *2009 International Conference on Machine Learning and Applications*, IEEE, 2009, pp. 337–344.

[20] L.-J. Lin, “Self-improving reactive agents based on reinforcement learning, planning and teaching”, *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[21] S. Adam, L. Busoniu, and R. Babuska, “Experience replay for real-time reinforcement learning control”, *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 2, pp. 201–212, 2012.

[22] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, “IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures”, *CoRR*, vol. abs/1802.01561, 2018. arXiv: `1802.01561`. [Online]. Available: `http://arxiv.org/abs/1802.01561`.

[23] H. Werner and G. Ehn, “Reinforcement learning for planning of a simulated-production line”, *Master’s Theses in Mathematical Sciences*, 2018.

[24] A. Ozçelikkale, M. Koseoglu, and M. Srivastava, “Optimization vs. reinforcement learning for wirelessly powered sensor networks”, in *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, IEEE, 2018, pp. 1–5.

[25] J. Kober, J. A. Bagnell, and J. Peters, “Reinforcement learning in robotics: A survey”, *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013. DOI: `10.1177/0278364913495721`. eprint: `https://doi.org/10.1177/0278364913495721`. [Online]. Available: `https://doi.org/10.1177/0278364913495721`.

[26] V. Firoiu, W. F. Whitney, and J. B. Tenenbaum, “Beating the world’s best at super smash bros. with deep reinforcement learning”, *CoRR*, vol. abs/1702.06230, 2017. arXiv: `1702.06230`. [Online]. Available: `http://arxiv.org/abs/1702.06230`.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning”, *CoRR*,

vol. abs/1312.5602, 2013. arXiv: `1312.5602`. [Online]. Available: `http://arxiv.org/abs/1312.5602`.

[28] H. Yu, T. Lim, J. Ritchie, R. Sung, S. Louchart, I. Stănescu, I. Roceanu, and S. de Freitas, "Exploring the application of computer game theory to automated assembly", *Procedia Computer Science*, vol. 15, pp. 266–273, 2012.

[29] B. Owens. (2014). Goal oriented action planning for a smarter ai, [Online]. Available: `https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793` (visited on 03/17/2019).

[30] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "Pddl-the planning domain definition language", 1998.

[31] H. van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil, "Deep reinforcement learning and the deadly triad", *CoRR*, vol. abs/1812.02648, 2018. arXiv: `1812.02648`. [Online]. Available: `http://arxiv.org/abs/1812.02648`.

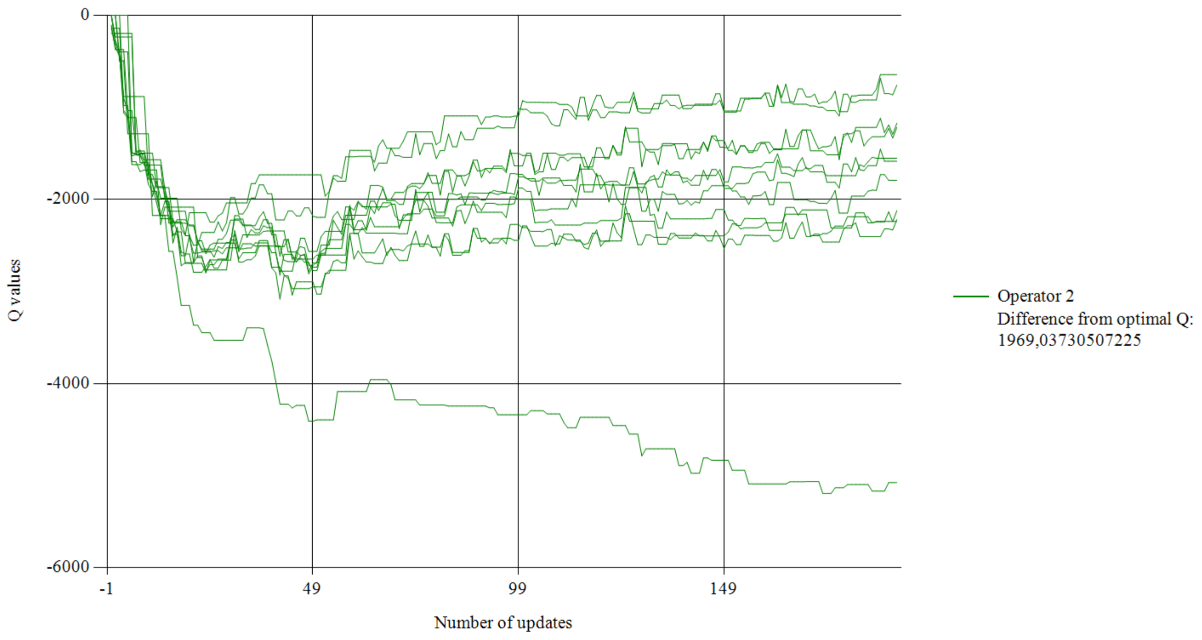
A

Appendix

A.1 Linear function approximation for operator 2 and 3

The plots of the Q-values and loss, with and without experience replay, for operator 2 and 3 as referenced in 5.2.1.1 are shown in this section.

(a)



(b)

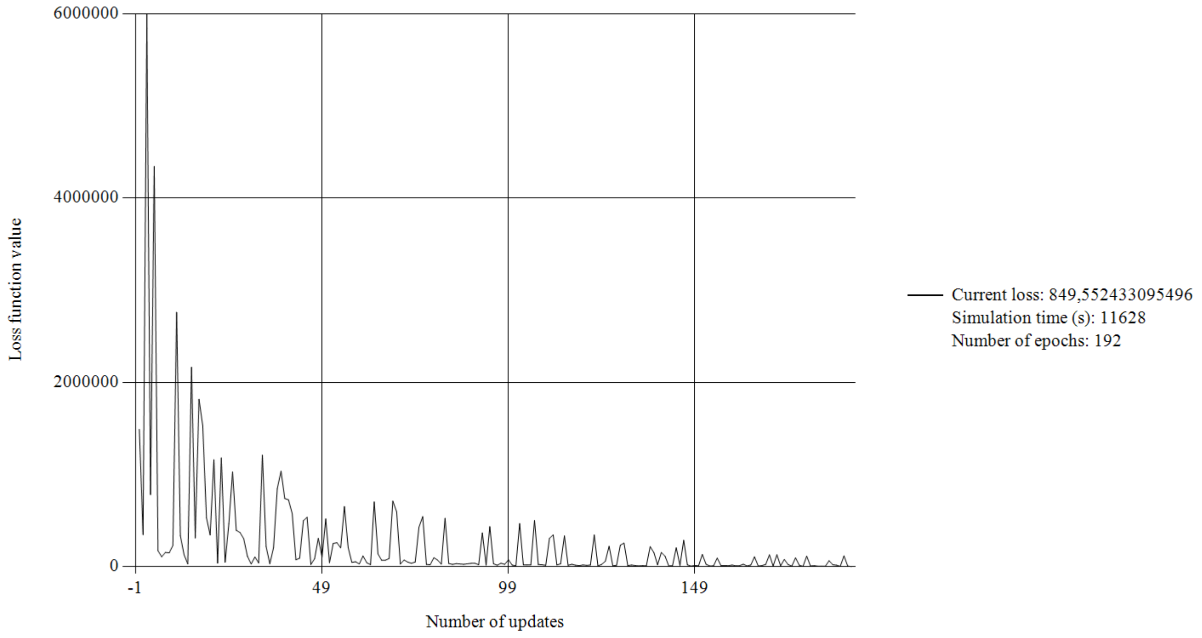
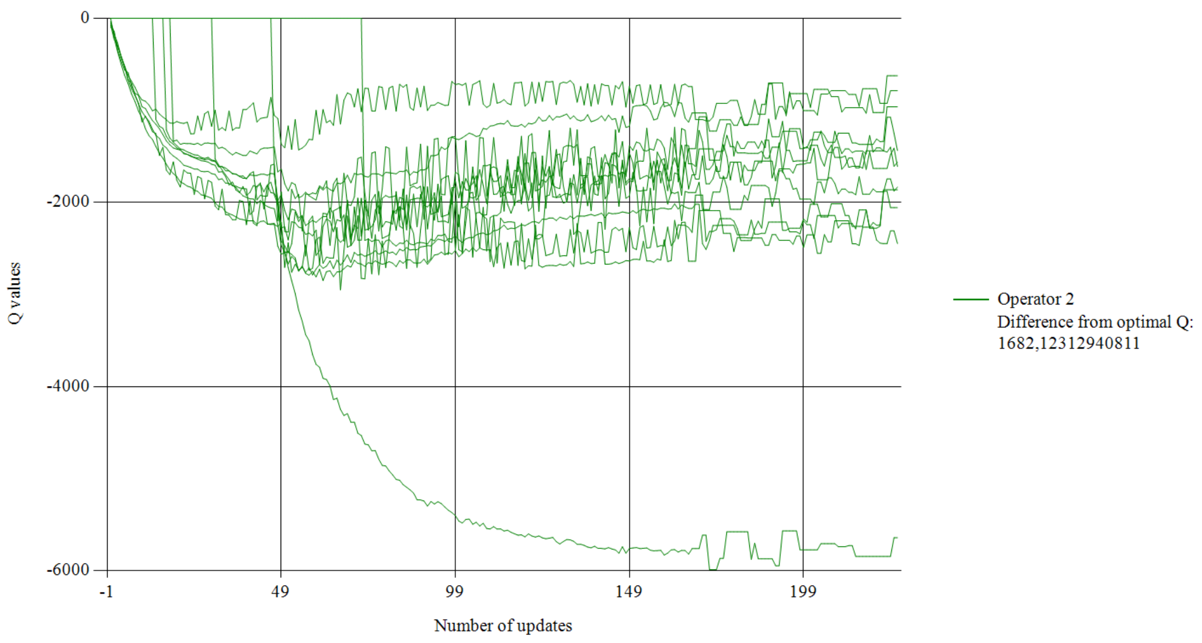


Figure A.1: Figure A.1a and figure A.1b show the Q-values and Mean Square Error respectively for operator 2 during training of the assembly station using linear function approximation without experience replay.

(a)



(b)

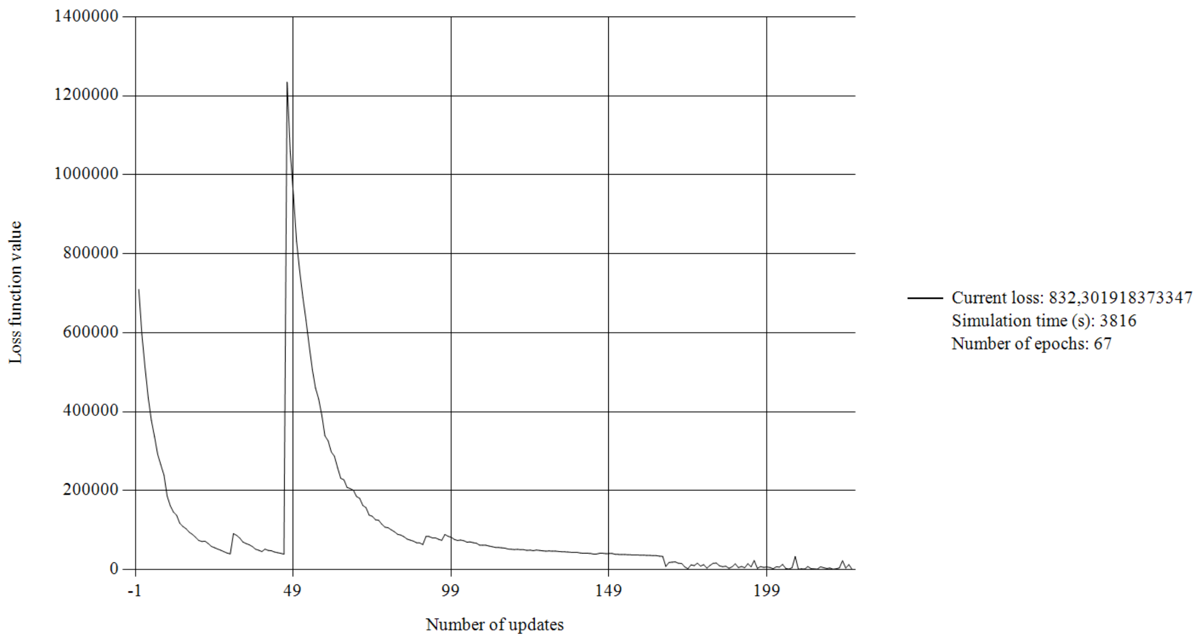
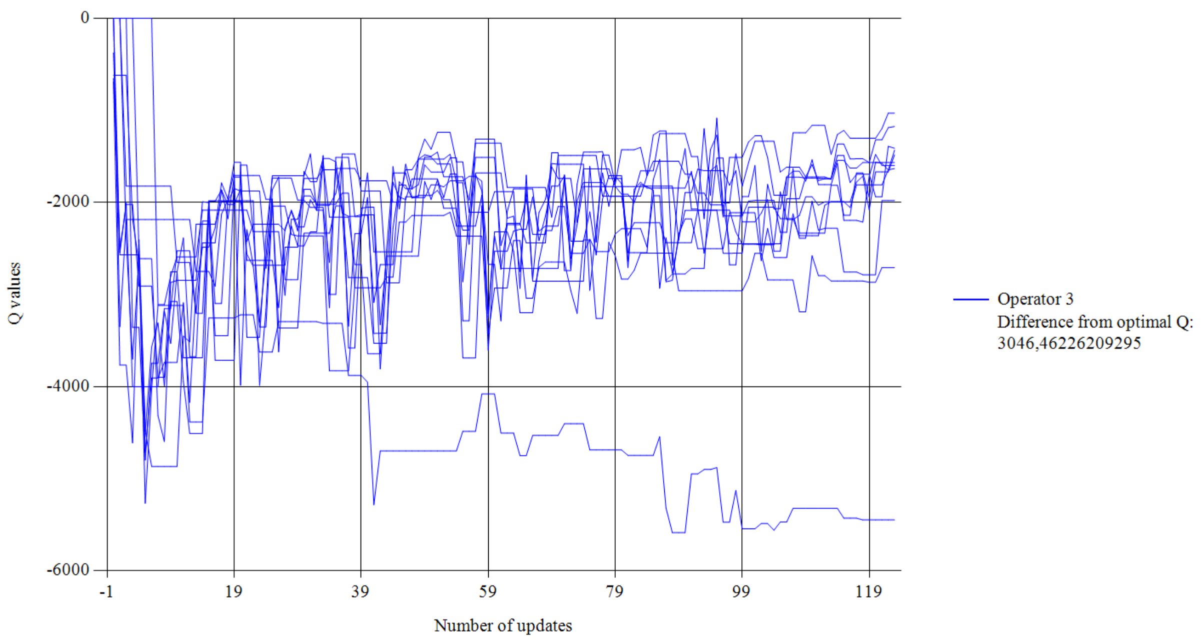


Figure A.2: Figure A.2a and figure A.2b show the Q-values and Mean Square Error respectively for operator 2 during training of the assembly station using linear function approximation with experience replay.

(a)



(b)

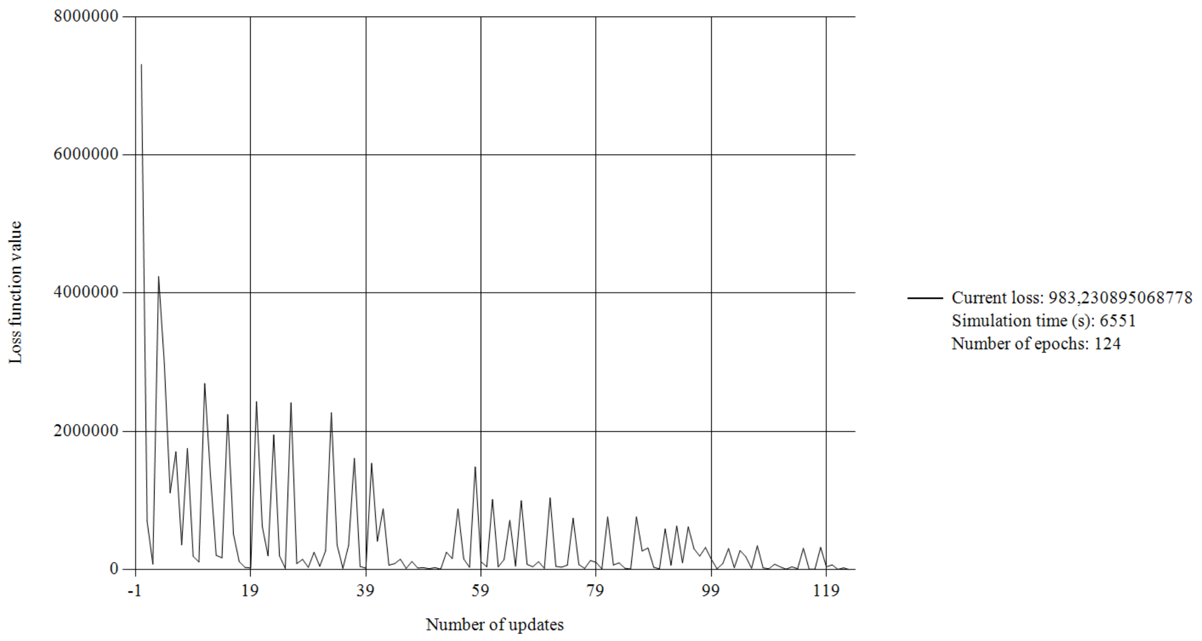
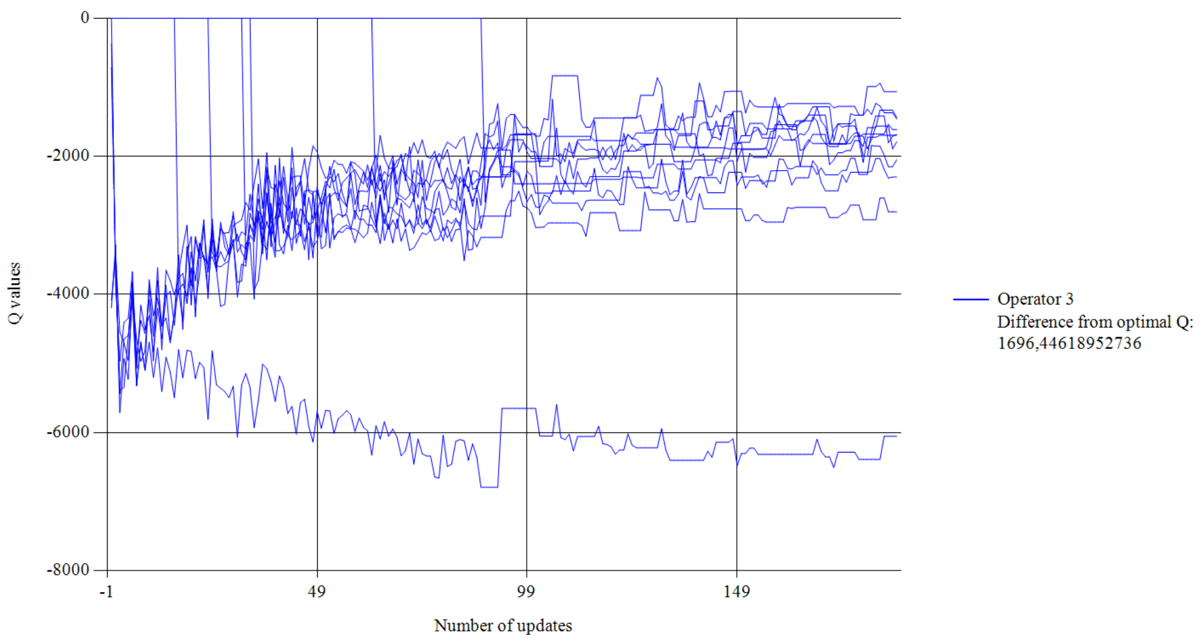


Figure A.3: Figure A.3a and figure A.3b show the Q-values and Mean Square Error respectively for operator 3 during training of the assembly station using linear function approximation without experience replay.

(a)



(b)

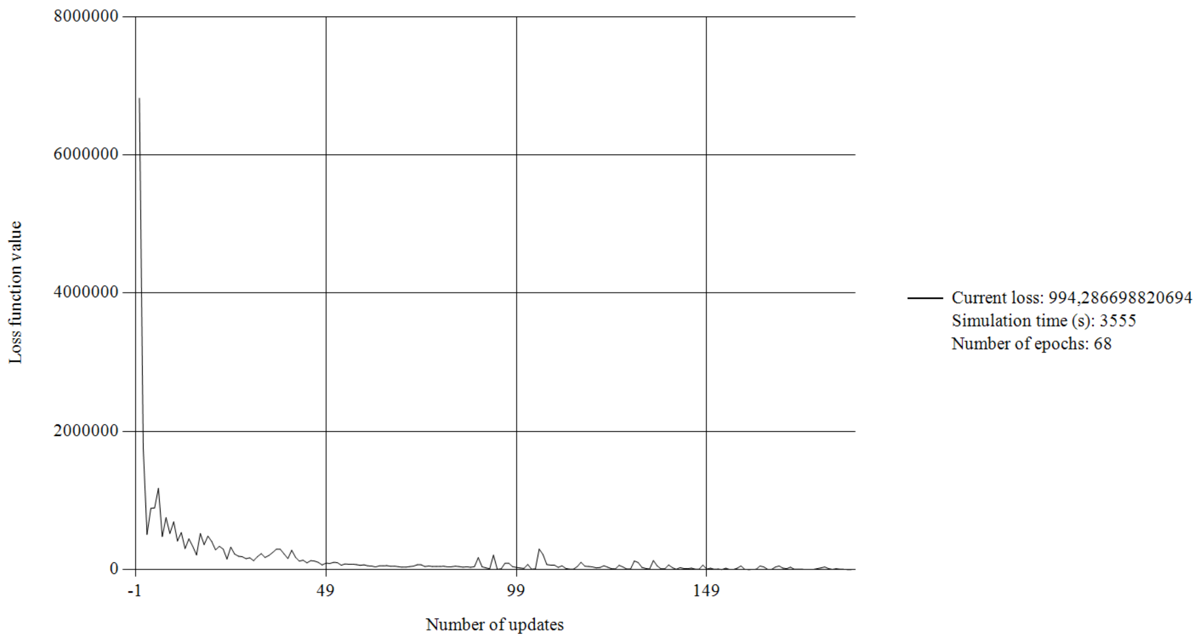


Figure A.4: Figure A.4a and figure A.4b show the Q-values and Mean Square Error respectively for operator 3 during training of the assembly station using linear function approximation with experience replay.

A.2 Communication between RL algorithm and robot controller

As referenced in section 5.2.3 the following code was used for communication between the RL algorithm and the robot controller.

Establish a client used in the RL algorithm:

```
string serverIP;
int serverPort;

TcpClient myClient = new TcpClient(serverIP, serverPort);
NetworkStream nws = myClient.GetStream();
```

Send message to the controller from the RL algorithm:

```
string messageToSend;
int byteCount = Encoding.ASCII.GetByteCount(messageToSend);
byte[] sendMessage = new byte[byteCount];
sendMessage = Encoding.ASCII.GetBytes(messageToSend);
nws.Write(sendMessage, 0, sendMessage.Length);
```

Receive message from the controller to the RL algorithm:

```
byte[] buffer = new byte[myClient.ReceiveBufferSize];
int bytesRead = nws.Read(buffer, 0, myClient.ReceiveBufferSize);
string message = Encoding.ASCII.GetString(buffer, 0, bytesRead);
```

Establish a server in the controller:

```
VAR socketdev server;
VAR socketdev client;
string serverIP;
num serverPort;

SocketCreate server;
SocketBind server,serverIP,serverPort;
SocketListen server;
SocketAccept server,client;
```

Send message to the RL algorithm from the controller:

```
VAR string message;

SocketSend client,\Str:=message;
```

Receive message to the controller from the RL algorithm:

```
VAR string message;
VAR rawbytes data;

Socketreceive client,\RawData:=data,\Time:=WAIT_MAX;
UnpackRawBytes data,1,message,\ASCII:=RawBytesLen(data);
```