Reinforcement learning for planning of a simulated production line

Gustaf Ehn, Hugo Werner

February 27, 2018

Abstract

Deep reinforcement learning has been shown to be able to solve tasks without prior knowledge of the dynamics of the problems. In this thesis the applicability of reinforcement learning on the problem of production planing is evaluated. Experiments are performed in order to reveal strengths and weaknesses of the theory currently available. Reinforcement learning shows great potential but currently only for a small class of problems. In order to use reinforcement learning to solve arbitrary or a larger class of problems further work needs be done. This thesis was written at Syntronic Software Innovations.
Keywords: Reinforcement learning, Machine learning, artificial neural networks, production planning.

Acknowledgements

We would foremost like to thank our supervisor Adam Andersson for his support and through feedback during the writing of this thesis. We would also like to thank our second supervisor Niels-Christian Overgaard as well as the people at Syntronic Gothenburg for their support and encouragement during this process.

Gustaf and Hugo
Lund Central Station, 15/2/2018

Contents

List of abbreviations and definitions

Table 1: List of abbriviations

RL	Reinforcement Learning
MDP	Markov Decision Process
SMDP	Semi Markow Decision Process
GSMDP	General Semi Decision Process
ANN	Artificial Neural Network
DQN	Deep Q-Network
DDQN	Double Deep Q-Network
SKU	Stock Keeping Unit
ReLU	Rectified Linear Unit

Table 2: List of notations

$\mathbf{1}_{\text{predicate}}$	Indicator function
$\mathcal{U}$	Uniform distribution
$p(\cdot\vert\cdot)$	Conditional probability
$\mathbf{P}$	Matrix with the state-transition probabilities between all states
$\mathcal{S}$	State space
$\mathcal{A}$	Action space
$\mathcal{R}$	Reward from an episode
γ	Discount factor
s	State
s'	State following s
a	Action
r	Reward received from transition $s, a \rightarrow s'$
π	Policy
$V(s)$	Value function
$Q(s, a)$	Q-value function
ϵ	Exploration probability
$\boldsymbol{\theta}$, $\boldsymbol{\theta}^{-}$	Weights for function approximation
$f(x; \boldsymbol{\theta})$	Function approximation
t_{terminal}	Total time of a simulation episode
n_{training}	Number of episodes run for training
f_{training}	Number of actions to be taken until the training network is trained next
f_{update}	Number of actions for which the target network is frozen before being updated
batch size	Number of transitions sampled every time a networked is trained
α	Learning rate
M	Number of materials
N	Number of shapes
P	Number of presses
O	Number of ovens
G	Max capacity of oven
K	Capacity of press
L	Number of operators
$\lambda_{\text{purchase rate}}$	Poisson rate parameter for the rate at which purchases occur
$\lambda_{\text{purchase amount}}$	Poisson rate parameter for the number of SKUs purchased at the event of a purchase

Chapter 1

Introduction

A child learns to crawl, walk and talk by interacting with its environment. The child learns how to control and manipulate its surroundings and learns the relationship between its actions and the impact of the actions. This process of trial and error with feedback is the core of how humans learn and the subfield of machine learning known as reinforcement learning. The basic concepts of reinforcement learning are covered in the first part of Chapter 3.

In recent years, reinforcement learning has been used to achieve some astonishing accomplishments, such as learning to play Go [4], chess and shogi[1] at a superhuman level [22] and playing Atari with only raw sensory data [2]. These results were achieved by combining reinforcement learning with artificial neural network function approximation.

The concept of artificial neural networks, ANN, is inspired by how the human brain works. The idea and theory of ANN dates back to the 1940s [26], but due to the computational cost it has been of limited use until recent years. With the development of more powerful hardware ANN has become a viable and useful tool. ANN are divided up into deep and shallow networks, see Section 3.6. When deep ANNs are used as function approximators in reinforcement learning techniques the learning system is called deep reinforcement learning. The theory of deep reinforcement learning relevant to the work in this thesis is covered in the end of Chapter 3.

There is currently an interest in automation of production and manufacturing. Whereas automation has become commonplace for manual tasks, assembly robots for example, more cognitive task such as planning and scheduling have remained mostly an area where humans have to do most of the work. The use of automation in these cognitive tasks have the potential to, with good margin, be superior to humans.

This process of automating more cognitive tasks falls under what is sometimes referred to as industry 4.0 [5]. Research and development in this area has attracted a lot of interest from both industry and governments [16, 23].

1.1 Problem

The problem that is investigated in this thesis is that of controlling stations in a production line. The production line has two types of stations, presses and ovens. The press takes a material and moulds it into a shape. The moulded units are then stored in a storage. The oven takes the shaped units from the storage and hardens them into the final products. The final products are referred to as stock keeping

[1]Japanese version of chess

units, SKUs. The SKUs are stored in stock until they are purchased. An example of a production line is shown in Figure 1.1.

The presses and ovens need to change setting if different SKUs are to be processed. Both processing and changing setting takes time. Thus the problem is choosing which SKU to process in the different stations in order to avoid the stock going empty. The production line and the simulation environment is described in detail in Chapter 5.

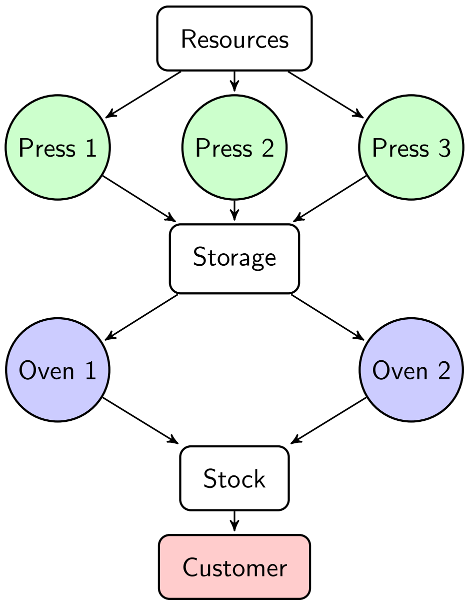

Figure 1.1: Example of a production line to be controlled.

1.2 Goals

The aim of this thesis is to use reinforcement learning to write a program that is able to learn an approximately optimal strategy for controlling the stations of a production line in order to meet the demand. In order to meet the demand the stock must contain enough units to satisfy all the customers trying make a purchase. The program should be able to learn the optimal strategy for different production lines. Formally our goal is to create a program that learns to minimise the number of times the SKUs sell out, leaving the stock empty, for any given production line. Our goals can be stated as:

- Formalise production planning in a reinforcement learning framework.
- Implement reinforcement learning algorithms that find an approximately optimal plan for the described production line with arbitrary number of press stations and ovens.
- Evaluate algorithms and their applicability to production planning.

1.3 Related work

As an introduction to reinforcement learning in general, we used the book by Sutton and Barto [24]. In the book Sutton and Barto methodically and thoroughly cover the basics of reinforcement learning and provide insight and intuition into the field of reinforcement learning. We also complemented our overview study of reinforcement learning with the lecture series by Silver [21], and the blog course by WildML [8]. Both the lecture series and the blog follows the book.

1.3.1 Deep reinforcement learning

Articles from the Google subsidiary Deepmind on deep reinforcement learning, such as [2, 4, 12], have been used as reading material to get knowledge and intuition of how deep reinforcement learning work.

1.3.2 Production planning using reinforcement learning

We have not found any projects that uses reinforcement learning to deal with the type of production planning problem examined in this thesis. There has however been some work done using reinforcement learning to solve other types of problems occurring in production planning.

Dranidis and Kehris treats a production system with multiple workstations in [9]. Their objective is to meet demand while keeping a good balance between the items manufactured. The production system is a simplification of a real production system. In the simplified system only two products are produced. When a decision needs to be made three actions are possible; produce product A, produce product B or wait. Their solution uses a neural network as a function approximation. For an explanation of neural networks see Section 3.6 and for an explanation of function approximation see Section 3.5. Dranidis and Kehris use one network for each action. Their result where good in the sense that their solution learned the objective, i.e. the solution found a nearly optimal solution according to their metric. However, they stated that more work was needed to be able to handle larger problems.

A basic job-shop scheduling problem (see [15]) was considered by Gabel and Riedmiller with reinforcement learning in [11]. They used a decentralised approach, similar to the work in this thesis, however their objective was to minimise the total production time for producing a given number of items. In production planing literature this is known as minimising the make span. They achieved good results, outperforming alternative benchmark solutions. There is no costumer demand present in their problem formulation, which make their problem differ from the problem dealt with in this thesis.

Das et al. [1] developed a reinforcement learning technique to optimise maintenance scheduling. They use a version of Markov decision processes called semi-Markov decision process (see Section 4.1) which does not fully adhere to the Markov property (see Section 3.2). Their objective was to plan when production should stop for maintenance. When production is stopped for maintenance, the risk of not meeting the costumers demand increases. If, however, maintenance is postponed too long the risk of a machine failure increases. A machine failure would create an even greater risk of not satisfying the demand then stopping production for maintenance. The production line is capable of producing several different items and the possible actions are to produce one of the items or to stop the production for maintenance. This approach takes under consideration the time a decision takes to carry out. This differs from the other production planning problems mentioned above. Considering the time to carry out a decision adds complexity to the problem. Since comparing the effectiveness of different choices also must take into account the time it took to complete that choice. For clarification of the difficulties arising when considering the time to carry out a decision see Section 4.1.

Chapter 2

Production Planning

In this chapter a short introduction to production planning is presented. Production planning is an ambiguous term. In this thesis it means how to most efficiently make use of a production line. This definition of production planning entails that 'soft' values, such as workers condition, is not taken into consideration.

2.1 Operations research

The area of science that is concerned with production planning is *operations research*. Operations research is, loosely speaking, the science of how to use quantitative methods to make things better. Because of the quantitative approach, operations research uses advanced mathematical tools, in particular optimisation, and can be considered to be a branch of applied mathematics. One could thus say that operations research is a mathematical way of looking at decision making. This approach is of course limited in many ways. The biggest limitation being that everything considered needs to be quantifiable, which is sometimes difficult. Personal satisfaction for example can be difficult to quantify.

Industrial processes, such as manufacturing and logistics, are often possible to model accurately. Thus there have been many cases of industrial processes being optimised successfully using operations research. Industrial processes can often be modelled to fit certain problem formulations.

Operations research is both concerned with modelling problems and solving or optimising them. As mentioned above many situations occurring in industrial process are similar and can thus be modelled by the same, or similar, models. For further reading on production planing and operations research see [10, 13].

Chapter 3

Reinforcement learning

Reinforcement learning (RL) is a subclass of machine learning which differs from classical supervised and unsupervised learning. It has a strong connection to behavioural psychology, more precisely to the law of effect which states the following, "Responses that produce a satisfying effect in a particular situation become more likely to occur again in that situation, and responses that produce a discomforting effect become less likely to occur again in that situation."[1]. The process of taking an action and receiving a response is what we know as experience. Through experience we learn to make better choices. This is the fundamental idea of RL; to train a computer to take good actions in a given situation by allowing it to experience the environment.

Supervised and unsupervised machine learning are mainly used for problems concerning classification, regression and pattern recognition in large data sets. RL algorithms on the other hand attempt to find an optimal behaviour in a given environment. However, parts of supervised learning is often integrated in order to improve and to scale the basic algorithms of RL.

When humans learn, the response is in the form of feelings, we eat something we like and get a satisfying feeling and when we eat something we do not like we get a dissatisfying feeling. The response system in RL algorithms is a numerical value, where a higher numerical value corresponds to a more satisfying feeling. Thus from experience the algorithms try to learn which actions to take in certain situations in order to receive the highest possible numerical value.

Reinforcement learning draws from several different areas to achieve the aforementioned goal. Due to the fact that RL has not received much attention before the last few years and the many different fields interacting in RL the terminology varies and changes depending on the background of the author. In this thesis we use the terminology used in [24].

3.1 Foundation

As mentioned above, RL focuses on algorithms which learn the optimal decision given a situation. In this thesis we refer to the learning and decision making entity as the *agent*. The agent observes the *environment* which is defined to be everything that the agent does not have direct control over but still influences the process. The agent interacts with the environment through *actions* and by observing the *state* of the environment. The consequences of an action is that the state of the environment might change. The feedback of an action is called *reward*. An agent interaction with an environment is visualised in Figure 3.1.

[1] Gray, Peter. Psychology, Worth, NY. 6th ed. pp 108–109

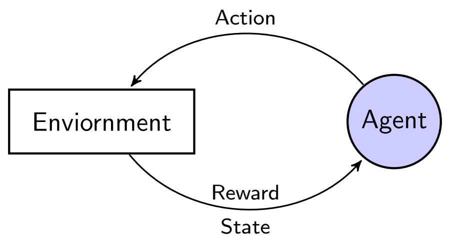

Figure 3.1: Visualization of an agents interaction with an environment.

By allowing the agent to try different actions it gains experience and can make a plan how to act in different situations. The agent's plan is referred to as the agent's *policy*. The goal for the agent is to find a policy which maximises the accumulated reward.

The process in which the agent interacts with the environment is formalised by the Markov decision process.

3.2 Markov decision process

In order to implement RL the process of the agent interacting with the environment needs to be formalised. For this purpose Markov decision process, henceforth abbreviated to MDP are central.

An MDP is an extension of a Markov process. An MDP is a Markov process with the addition of actions. The actions affect the transition probabilities such that the actions can, to some extent, control the process. This creates a framework where the process is partly random and partly controlled. The actions of the MDP are the actions described in the previous section and illustrated in Figure 3.1.

The entity deciding which actions to take is the agent. When an agent takes an action the state may change, entering a different state or returning to a state yields a reward which corresponds to how good the transition to that new state is.

An MDP is completely described by its state-space, action-space, transition probabilities, rewards and discount factor, this makes up the tuple $(\mathcal{S}, \mathcal{A}, \mathbf{P}, \mathcal{R}, \gamma)$. The state-space, $\mathcal{S}$, consists of all the possible states the process can be in. The number of possible states might be infinite. The action-space, $\mathcal{A}$ contains all the possible actions that are possible to take in the different states. Note that the possible actions, may vary between different states. The transition probabilities, $\mathbf{P}$, contains the probabilities that, given a specific action in a certain state, the process moves to a particular state. The rewards, $\mathcal{R}$ describes the value of transitioning from one state to another. The discount factor, γ, is a scalar between $[0, 1]$ which is used to weigh the importance of immediate reward against the accumulated reward. If $\gamma < 1$, then the reward is called discounted. Setting $\gamma = 0$ implies that only the next reward is taken into account and the closer γ is to one the more important are future rewards. Setting $\gamma = 1$ implies that the sum of all future rewards is the important.

Figure 3.2 illustrates one step in an MDP. The node on top is the current state denoted s. In state s there are two possible actions a_1 and a_2. For each action there is an uncertainty which will be the following state, denoted s'_i. For both actions there are two possible states, the probabilities for ending up in one or the other state is denoted by p_i. In the case that action a_1 is chosen the agent will end up in s'_1 or s'_2 with probability p_1 and $p_2 = 1 - p_1$, respectively.

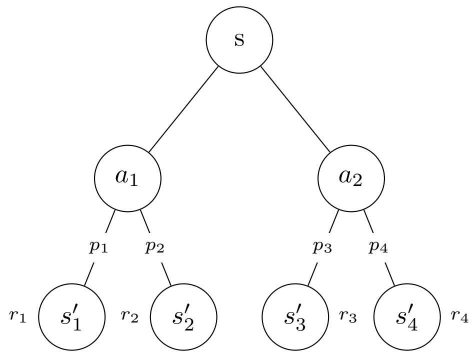

Figure 3.2: Visualisation of a step in an MDP

The sequence of actions and transition is repeated forever or until the process reaches a state which it can not leave. Such a state is said to be a *terminal* state. The evolution of an MDP from start to a terminal state is called an *episode*. MDPs which with certainty ends in a terminal state after a finite number of steps are said to be *episodic*.

An MDP assumes the Markov property, which states that the future is completely independent of the past, given the present. This is formally written

$$\mathbb{P}(s_{n+1}, r_{n+1} | s_n \dots s_1, a_n \dots a_1) = \mathbb{P}(s_{n+1}, r_{n+1} | s_n, a_n), \quad n \geq 0.$$

Each transition yields a reward, illustrated in Figure 3.1. Let r_t be the reward received from the transition at time t and let T be the time when the episode terminates. The discounted accumulated future reward from time t is then given by

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^{T-t} r_T = \sum_{k=0}^{T-t} \gamma^k r_{t+k}, \quad t \in \mathbf{N}.$$

This sum is governed by which actions are taken. The decisions of which actions to take is determined by the agent's policy, denoted as π. A policy can be written as a conditional probability of what action to take given the current state,

$$\pi(a|s) = \mathbb{P}[a_t = a | s_t = s], \quad a \in \mathcal{A}, \ s \in \mathcal{S}.$$

If the probability of all actions are either zero or one the policy is deterministic.

The *state-value* function, $v(s)$, is the expected discounted accumulated future reward from the current state, s. The state-value is dependent on the policy. Since the policy determines which actions to take, the accumulated future reward may differ depending on the policy. Due to the dependence of the policy, the policy is denoted as a subscript of the value function. The state-value function when following policy π is thus

$$v_\pi(s) = \mathbb{E}_\pi[R_t | s_t = s], \quad s \in \mathcal{S}. \tag{3.1}$$

Let $p(s'|s, a)$ be the probability of transitioning to s' given that the action a is chosen in state s. This allows the conditional probability of the next state s' given the current state s to be written as

$$p(s'|s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} p(s'|s, a). \tag{3.2}$$

Let $r_{s,s'}$ be the reward received for transitioning from s to s'. Then the expected reward for transitioning from state s, denoted $r_\pi(s)$, can be written as

$$r_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} p(s'|s,a) r_{s,s'}, \quad s \in \mathcal{S}.$$

Now (3.1) can be written

$$v_\pi(s) = r_\pi(s) + \gamma \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} p(s'|s,a) \mathbb{E}_\pi[R_{t+1}|s_{t+1} = s'], \quad s \in \mathcal{S}.$$

By identifying that

$$\mathbb{E}_\pi[R_{t+1}|s_{t+1} = s'] = v_\pi(s')$$

we can rewrite (3.1) as

$$v_\pi(s) = r_\pi(s) + \gamma \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} p(s'|s,a) v_\pi(s'), \quad s \in \mathcal{S}. \tag{3.3}$$

Since the value of a state can heavily depends on the action chosen, there is another measure of a state value that also includes the action chosen in that state. Given both the current state *and* action, the *action-value* function $q_\pi(s,a)$ is the value of taking action a in state s while following policy π.

The value of an action in a state is, as for $v(s)$, the accumulated expected future reward, also $q(s,a)$ is dependent on the policy for the same reasons as v. The action-value function, following policy π, can be expressed as

$$q_\pi(s,a) = \mathbb{E}_\pi[R_t|s_t = s, a_t = a].$$

To understand why the action-value function is of importance, consider one step of an MDP as visualised in Figure 3.2. The rewards r_1 and r_2 might be high, while r_3 and r_4 are low. By following a random policy, where the choice of action is 50/50, the state s might look average. But the state s is actually either good or bad depending on the action chosen. The action-value function captures this information since it is a function of both the state and the action. Analogous to the value function, the action-value function can be rewritten as

$$q_\pi(s,a) = r_\pi(s) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a'). \tag{3.4}$$

The algorithms presented in this thesis approximate the q-function.

Equations (3.3) and (3.4) constitutes Bellman's equations, which are solved with dynamic programming. Dynamic programming is the topic of the next section.

3.3 Dynamic programming

Dynamic programming, DP, is a computational method. In order to apply DP to a problem, the problem must fulfil the so called 'Principle of optimality'. The principle states that:

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision [7].

This means that the value of leaving a state can not be dependent on how the process ended up in that state, or to phrase it in another way, the past can not affect the future. Since MDPs fulfils the Markov condition, which states that the process is independent of the history, it is possible to apply DP to RL.

3.3.1 Dynamic programming for RL

Within reinforcement learning dynamic programming is used to find v and q. For finite processes, i.e., episodic MDPs where $\mathcal{A}$ and $\mathcal{S}$ are finite and a model of the process, i.e. the transition probabilities, is available, dynamic programming can be used. The bellman equations, (3.3) and (3.4) are linear systems which can be solved explicitly. However, solving the systems explicitly requires inverting a matrix which is computationally demanding. An alternative to computing the explicit solution is to use iterative dynamic programming algorithms. Two such methods that are use in RL and have been proven to converge to the optimal policy are policy iteration and value iteration (for details of these algorithms see [24]). In many interesting problems the processes are not finite and the transition probabilities are not available which renders DP not applicable. An alternative method when DP is not applicable is model-free algorithms which approximate v_π and q_π based on experience. The experience replaces the need of a model. One such algorithm is the Q-learning algorithm.

3.4 Q-Learning

Due to the fact that dynamic programming is not applicable in many cases alternative techniques have been developed. Q-learning is a technique used in RL to update the action value function $q(s, a)$ without the need of a model of the environment [25]. Q-learning is thus model-free and does not require knowledge of the system as dynamic programming does.

3.4.1 The Q-learning Algorithm

The Q-learning algorithm is an algorithm that systematically updates q_π. The basic idea of the algorithm is to first run an episode and store all states, actions and rewards. Once the episode has terminated the values of q are updated. A simplified algorithm is stated in the pseudocode below.

Simplified Q-Learning Algorithm

```
Set the learning rate $\alpha$
Initialise $q$ arbitrarily
**Run episode**:
  Take an action $a_t$ in the current state $s_t$ such that
  $a_t = \text{argmax}_{a \in \mathcal{A}} q_\pi(s_t, a)$
  Observe reward, $r_t$
  Save $s_t, a_t, r_t$
**For every observed pair** $(s_t, a_t)$**, update**:
```

$$q_\pi(s_t, a_t) \leftarrow q_\pi(s_t, a_t) + \alpha \left[\sum_{k=0}^{T-t} \gamma^k r_{t+k} - q_\pi(s_t, a_t) \right]$$

Every update only uses one sample action, therefore Q-learning avoids the curse of dimensionality. However, the simplified Q-learning algorithm shown above still has a few drawbacks. Firstly, every state, action and reward observed in an episode has to be stored in order to update q_π at the end of the episode. Secondly, updating q at the end of an episode makes learning quite slow. It would be beneficial for the learning speed to update q every time an action is taken and a reward is received. In order to handle the drawbacks mentioned above *bootstrapping* is introduced. To understand bootstrapping we need to write $q_\pi(s_t, a_t)$ as

$$q_\pi(s_t, a_t) = r_t + \sum_{k=1}^{T-t} \gamma^k r_{t+k}.$$

The idea of bootstrapping is to to approximate the sum, $\sum_{k=1}^{T-t} \gamma^k r_{t+k}$ with $\max_a q_\pi(s_{t+1}, a_{t+1})$ so that the updating rule in the Q-learning algorithm becomes

$$q_\pi(s_t, a_t) \leftarrow q_\pi(s_t, a_t) + \alpha \Bigg[r_t + \gamma \max_a q_\pi(s_{t+1}, a_{t+1}) - q_\pi(s_t, a_t) \Bigg].$$

With this new updating rule it is possible to update q_π after every action instead of at the end of an episode. The more frequent updating allows for faster learning as updates can occur during an episode. There is however a small downside to bootstrapping, it slightly reduces the accuracy of the updates. The accuracy is reduced due to the approximation of the sum.

Another problem with the simplified Q-learning algorithm is that it becomes biased towards positive decisions. If it takes an action in a state and gets a positive reward it tends to prefer those actions over unexplored actions. This is simply due to the fact that the policy π is greedy, it chooses the action believed to yield the highest reward. An action which is known to yield a positive reward appears better than an action with an unknown outcome. To overcome the problem of bias, the agent is forced to explore random actions with certain probability during the learning phase. Every time the agent is to choose an action during the learning phase, with probability ϵ it chooses an action at random. By introducing this randomness the agent is forced to explore the environment. This new exploratory policy is called ϵ-greedy. The Q-learning algorithm with bootstrapping and an exploratory policy can be seen below.

Q-Learning Algorithm - with ϵ-greedy policy

```
Set the exploration probability $\epsilon$
Set the learning rate $\alpha$
Initialise $q$ arbitrarily
Initialise new episode and state $s_0$
**Run forever**:
  Take an action $a_t$ in the current state $s_t$ according to $\epsilon$-greedy-$q_\pi$
  Observe reward $r_t$ and next state $s_{t+1}$
  Update $q_\pi$:
    $q_\pi(s_t, a_t) \leftarrow q_\pi(s_t, a_t) + \alpha \Big[ r_t + \gamma \max_a q_\pi(s_{t+1}, a) - q_\pi(s_t, a_t) \Big]$
  **If** $s_{t+1}$ is terminal
    *Break*
```

The q-function can be represented in a lookup table fashion. In the case of q-learning one would look up a state-action pair and find the corresponding q-value. A lookup table representation requires the state and the action representation to be discrete and finite in order to work. This is due to the fact that every state and action needs to be visited multiple times in order to be able to update and get a good approximation of the q-value. To visit all states in a very large but bounded state space is very time consuming and impossible with a unbounded or continues state-space. If the state space or action space is continuous the probability of even revisiting a unique state is zero. The look up table approach works great for small discrete state-action spaces but falls short otherwise. For most interesting problems the state space is very large or even continuous and thus requires the q-function to be represented with a function approximation.

3.5 Function approximation

A large state space creates a problem when representing the q-function using the lookup table method. In the continuous case, an injective mapping such as the lookup table is impossible and for a large state space it quickly becomes infeasible. In order to extend the practice of reinforcement learning to large state spaces, the mapping from state and action to q-value needs to be condensed in some manner. Function approximation is used for this purpose. A function approximation attempts to find a parameterized function $f(x;\boldsymbol{\theta})$ satisfying $f(x) \approx f(x;\boldsymbol{\theta})$. In this thesis function approximation is used to approximate the action-value function, $q_\pi(s,a)$. By using a function approximation, predictions of q-values for unseen states is possible. In Figure 3.3 the idea of a function approximation, as it is used in this thesis, is visualised. The left figure represents a lookup table where the pairs x_i and y_i are stored. The right figure represents function approximation, where only the parameters $\boldsymbol{\theta}$ are stored. An input is passed through the function $q(s,a;\boldsymbol{\theta})$ and the approximate value corresponding to the given input is obtained. A group of very versatile and powerful function approximators is artificial neural networks. They can be used to approximate nonlinear functions. Artificial neural networks is explained in the next section.

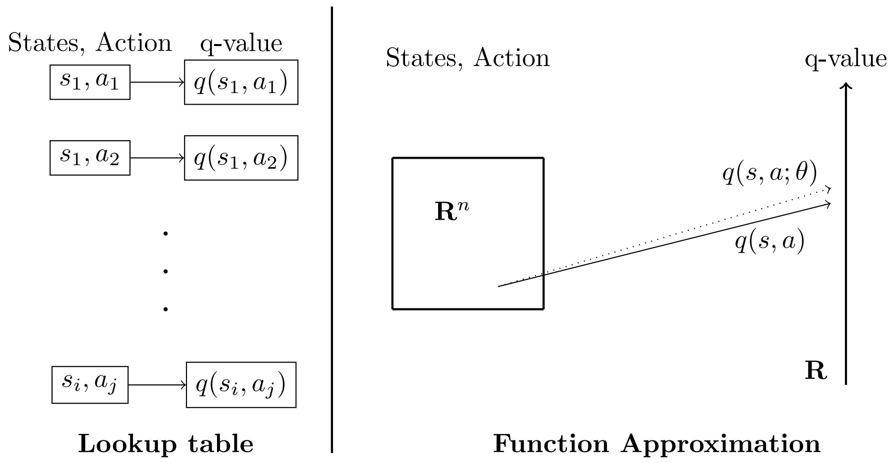

Figure 3.3: Illustration of a lookup table to the left and a function approximation to the right.

3.6 Artificial neural networks

Artificial neural networks, abbreviated ANN, are parameterized functions that process information in a similar way biological neural networks do. The building blocks of an ANN are nodes which would represent the neurons in a biological neural network. Every node, except the input nodes receives information from other nodes and processes the combined inputs into an output from the node. The network is composed of layers, where information is passed from layer to layer until the final layer, the output layer. Choosing the nodes and layers appropriately ANNs can be used to approximate very complex functions.

3.6.1 Multilayer Perceptron

One particular class of ANN is called *multilayer perceptron*, MLP. This type of network is used in this thesis. A MLP netwok is constructed of layers of nodes where every node in every layer is connected to every node in the following layer. An illustration of an MLP can be seen in Figure 3.4.

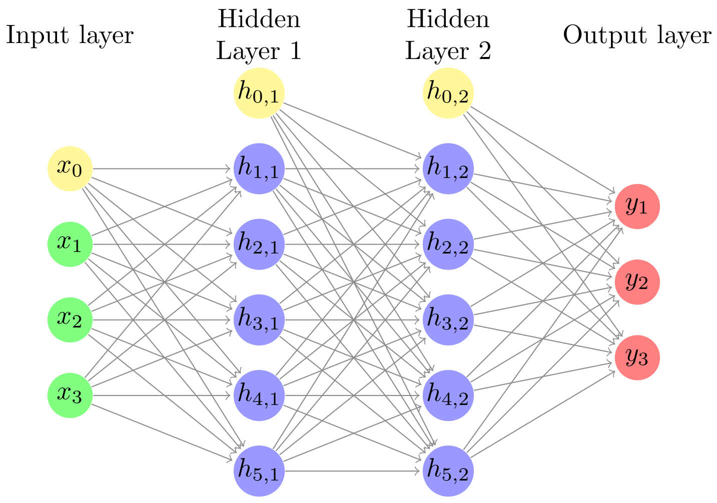

Figure 3.4: Illustration of a feed forward artificial neural network. The green nodes are input, blue are the hidden layer nodes, red nodes are the output and the yellow nodes are bias nodes.

A MPL with more than one hidden layer is called a deep artificial neural network. For every connection in the MLP there is a corresponding weight w_i and for every node there is a bias. The weights and the biases make up the parameters $\boldsymbol{\theta}$ that parameterize the q-function as described in Section 3.5.

The mathematical formulation of the computations performed by the MLP are described next. Let l_i be a layer with n nodes followed by layer l_{i+1} with m nodes. The connections of these two layers are represented by a weight matrix $W \in \mathbf{R}^{m\times n}$ and the biases for the layer is represented by the bias vector $\mathbf{b}^i \in \mathbf{R}^m$. Let H denote the activation function, it acts as a filter turning the node 'on' or 'off' depending on the input. Denoting the output of node k in layer l_i by a_k^i the node performs the operation

$$a_j^{i+1} = H\left(\sum_{k=1}^{m} w_{k,j} a_k^i + b_j^i\right).$$

In more convenient matrix vector notation this reads

$$\mathbf{a}^{i+1} = H\left(W\mathbf{a}^i + \mathbf{b}^i\right). \tag{3.5}$$

A common activation function H is the Rectified Linear Unit (ReLU) defined as

$$\mathrm{ReLU}(x) = \max(0, x).$$

ReLU allows for the 'on-off' effect but with a more gradual transition and no upper bound which enables the output to contain more information compared to a step function which would only output 0 or 1.

The information which enters an MLP in the input layer propagates forward as in (3.5) until the output layer is reached. Different activation functions can be used in the different layers. In the networks implemented in this thesis ReLU is used for the hidden layers and the output layers output the weighted sum without an activation function also known as a linear activation function.

3.6.2 Training

Training an ANN is the process of updating the weights and biases in order to increase the accuracy of the function approximation. During training the ANN is provided a data set, $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, with input and output pairs (x_i, y_i). With input x_i, the output of the neural network is denoted by $\hat{y}_i$. After training, $\hat{y}_i$ should approximate y_i. In particular, for good performance, the network should approximate y by $\hat{y}$ well for unseen data not in the training data set. A *loss function*, $L(y_i, \hat{y}_i)$, maps the error between the output $\hat{y}_i$ and y_i to a real number. A simple example of a loss function is

$$L(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2.$$

The loss function is used for the comparison between y_i and the approximation $\hat{y}_i$ but to determine the overall accuracy of the ANN we introduce the *cost function* $C(\boldsymbol{\theta})$. The cost function can be defined as

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{y}_i). \tag{3.6}$$

The goal of training is to update the weights in order to minimise the cost function.

Stochastic gradient descent

Once $C(\boldsymbol{\theta})$ is calculated the training and the updating of the weights and biases can take place. Due to the many parameters and the fact that minimising $C(\boldsymbol{\theta})$ is a non-convex problem, finding the global minimum among the many local minimum is impossible. A common approach is to apply some iterative optimisation method such as gradient decent in order to find a good local minima. Such a method computes the gradient with respect $\boldsymbol{\theta}$ and the weights are updated in the direction of the gradient. The cost function above takes the entire data set into account which leads to calculating the gradient for the entire data set. Calculating the gradient for large data sets can be slow. To increase the speed *stochastic gradient descent*, SGD, is customly used. It also reduces the risk to get stuck in a sub-optimal local minima. SGD computes the gradient for one sample or a small subset of the entire data set at a time. This subset is referred to as a *batch*. The gradient calculated on the batch is an approximation of the gradient for the entire data set. For every batch the parameter vector $\boldsymbol{\theta}$ is updated in the direction of the negative gradient corresponding to that batch. The size of the step in the direction of the gradient is determined by the learning rate, α. The updating for $\boldsymbol{\theta}$ is

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}).$$

Adaptive moment estimation

In this thesis the optimisation algorithm used is *Adaptive moment estimation*, commonly called Adam. It is also a stochastic gradient method but it makes use of previous gradients which can allow for quicker convergence [14]. In every iteration the algorithm keeps a decaying average m of the previous gradients and also a decaying average v of the past gradients squared. We let $g_t = \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta})$ be the gradient in time step t, the algorithm then calculates

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2.$$

Here β_1 and β_2 are hyperparameters controlling the decay of the average sums. The averages m_t and v_t are biased to the initialisation value of 0. The Adam algorithm corrects for the biases by replacing m and v with

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

These new corrected averages are then used to update the parameters $\boldsymbol{\theta}$ as follows

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\hat{v}_t} - \epsilon} \hat{m}_t.$$

Backward propagation

In an ANN there are potentially a very large number of weights and therefore potentially also a large number of partial derivatives to be computed every update. The common algorithm used to calculate the partial derivatives with respect to the weights and biases is the *backpropogation* algorithm, short for backward error propagation. The algorithm calculates the derivatives starting with the final layer of weights and propagates backwards, hence the name. Calculations from previous layers are used in the current layer instead of computing every derivative individually. This allows for quick and efficient computations. For details on the algorithm see [19]. For further reading about ANNs see [3].

3.7 Deep reinforcement learning

For problems with large state action spaces the lookup table representation of the action value function q is not applicable due to the reasons discussed at the end of Section 3.4. Instead of the lookup table a function approximation can be used to approximate $q(s,a)$ with the parametrised function $q(s,a;\boldsymbol{\theta})$. Artificial neural networks are powerful function approximators, as discussed in Section 3.6, and are therefore useful in RL. When deep artificial neural networks are used in RL it is called *deep reinforcement learning*. An RL algorithm which uses deep ANN is *Deep Q-network*. The details of the algorithm are presented below.

3.7.1 Deep Q-Network

Deep Q-Networks, abbreviated DQN, use deep neural networks as function approximation of the action-value function $q(s,a)$. They were first introduced in [2] for the application of playing Atari 2600 games. The input of the artificial neural network used is the state and the output is the estimated q-values of the state-action pairs.

When a DQN agent is trained the ANN is trained by minimising the cost function,

$$C(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, \hat{y}_i).$$

Here N is the batch size and $\boldsymbol{\theta}$ is the vector with the weights and the biases of the neural network. The specific choice of L is discussed below. The value which the ANN tries to approximate, y_i, is referred to as the target. The data collected from experience does not contain a target explicitly so in order to train an ANN the targets needs to be approximated. When the targets are approximated the data collected during experience is used. The target $y_i = y_t$, where t denotes the time step, is defined as

$$y_t \equiv r_{t+1} + \gamma \max_a q(s_{t+1}, a; \boldsymbol{\theta}). \tag{3.7}$$

While DQN allow for large and also continuous state-spaces there are however drawbacks to the basic algorithm which uses (3.7) as target. Firstly if the updates are performed on the events that occurred most recently, the data would be highly correlated and old training samples would lose their impact with every new sample. This would degrade the networks ability to generalise [19]. Secondly, the target uses the same network that is being updated to predict the value of the next state, s'. This leads to instability when s and s' are equal or even similar which is often the case [2].

The first problem mentioned above is solved by storing old transitions to a *replay memory.* The replay memory simply stores the transitions such that they can be used at later times. By sampling transitions from the replay memory the network can be trained on uncorrelated experiences and thereby increases its ability to generalise. This also allows the network to predict the correct values in states which might be visited less frequently when the agent's strategy gets better. In short, it increases the data utilisation, allowing the network to train more on fewer experiences.

The second problem is solved by introducing a second network, a *target network*, which is a copy of the first network, which we call the *training network.* The target network is only used to predict the value of taking the optimal action from s' when updating the training network. The target network is updated with a certain frequency by copying the weights from the training network. Hence freezing the target network while updating the training network. This method makes the the training network more stable [2] With the frozen parameters denoted as $\boldsymbol{\theta}^-$, the expression for y_t becomes

$$y_t \equiv r_t + \gamma \max_a q(S_{t+1}, a; \boldsymbol{\theta}^-).$$

A third problem still remains; DQN overestimates states[12]. The overestimation is a result of evaluating the optimal action from s' with the target network. When the action corresponding to the largest q-value in s' is chosen it is likely to be an overestimation of that action which causes the overestimation. A way to work around this is by choosing the best action from s' using the training network and then using the target network to evaluate the value of taking that action from s'. The action is thus picked based on the most recent updates, but the value of the action is estimated using the target network. This method is known as *Double DQN* [12]. Using Double DQN, DDQN, y_t becomes:

$$y_t \equiv r_t + \gamma q\big(s_{t+1}, \operatorname*{argmax}_a q(s_{t+1}, a; \boldsymbol{\theta}); \boldsymbol{\theta}^-\big).$$

Since the value of q The hyperparameters of the algorithm can be seen in Table 3.1 and the complete learning algorithm described above can be seen in the pseudocode below.

Table 3.1: DDQN Hyperparameters

ϵ	Exploration probability
t_{terminal}	Total time of a simulation episode
n_{training}	Number of episodes run for training
f_{training}	Number of actions to be taken until the training network is trained next
f_{update}	Number of actions for which the target network is frozen before being updated
$a_{\text{start training}}$	Number of transitions before training starts
batch size	Number of transitions sampled every time a networked is trained
α	Learning rate
γ	Discount factor

Double DQN Algorithm - with ϵ-greedy policy

```
Initialise new episode and state $s$
While $s \neq s_{terminal}$:
    Take an action, $a$ in the current state $s$ according to $\epsilon$-greedy-$q_\pi$
    Observe reward, $r$ and next state $s'$
    Save $< s, a, r, s' >$ to the replay memory
    numberOfActionsTaken $\leftarrow$ numberOfActionsTaken $+ 1$

    If numberOfActionsTaken $\geq a_{\text{start training}}$
        Update neural network weights $\boldsymbol{\theta}$ on a batch of
        transitions sampled from the replay memory:
            $y \leftarrow r + \gamma q\big(s', \text{argmax}_a\, q(s', a; \boldsymbol{\theta}); \boldsymbol{\theta}^-\big)$
            $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \text{Adam}\big(y, q(s, a; \boldsymbol{\theta})\big)$

    If numberOfActionsTaken (mod $f_{\text{update}}$) == 0
            $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$
    $s \leftarrow s'$
```

There is high risk that the error $y_t - \hat{y}$ may be large for a few samples. This is partly due to q being arbitrarily initialised but also due to the variance of r. With a squared error as the loss function the gradients of the cost function can become very big. This could yield large updates in the network, which would degrade the networks ability to generalise. One can simply reduce the learning rate in order to reduce the risk of having too large updates but this results in too small updates when the errors are small. To overcome this problem the loss function used is the Huber loss, it allows for a larger learning rate without risking the large updates. The Huber loss function maps large errors linearly and small errors quadratically which allows for large updates while minimising the risk of updates becoming too large. The Huber loss function is defined as

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x)| \leq \delta, \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

Here δ determines the transition between quadratic and linear. A comparison between the Huber loss function and the squared error, $(y - \hat{y})^2$ is visualised in Figure 3.5. The gradient of the squared error increases as the error grows while the gradient for the Huber loss remains constant beyond δ.

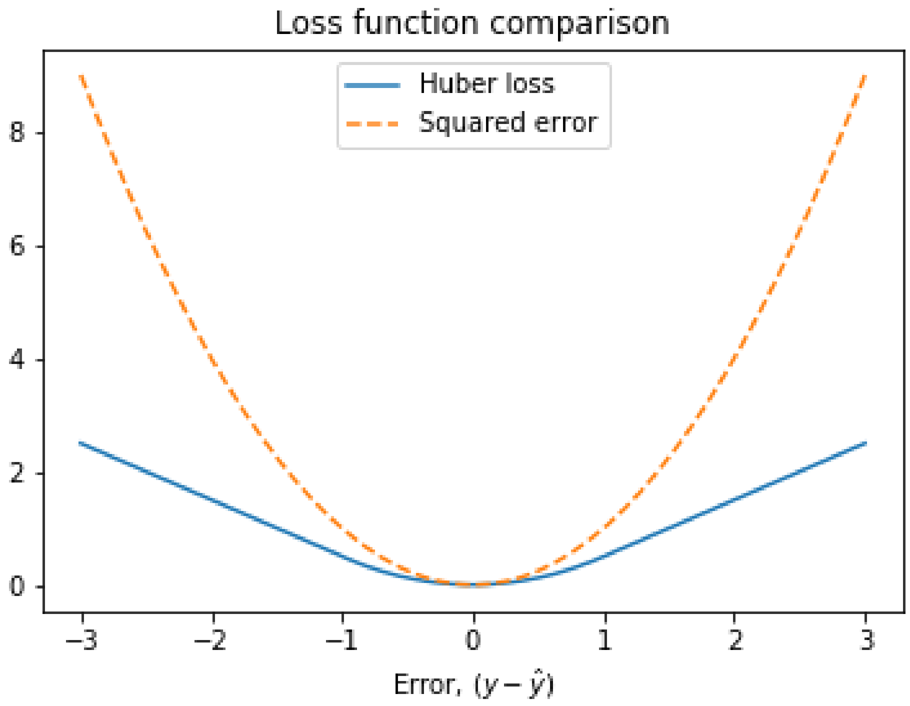

Figure 3.5: Comparison of the squared error and Huber loss with $\delta = 1$

Chapter 4

Markovian processes

In Chapter 3 the foundations of RL was laid out. The underlying process in RL was said to be MDPs. However, all problems can not be accurately modelled as MDPs. Therefore, the MDP framework needs to be extended in order to formalise and apply RL to a bigger class of problems.

In this thesis the name Markov decision process is reserved to mean the process described in Section 3.2. The term Markovian decision process is used to mean all processes the can be formulated as decision problem that fulfils the Markov property to some extent. The Markovian processes relevant to the work in this thesis are presented in this chapter.

4.1 Semi-Markov decision process

In a semi-Markov decision process, as the name suggests, the Markov property is not fully satisfied. In a SMDP the Markov property only holds at decision points, the points in time at which a decision is taken. The transitions in in MDP are discrete, thus there is no real concept of time of a transitions in an MDP. In an SMDP the time of a transition is taken into to account. The time between actions in an SMDP is variable, potentially stochastic and dependent of the state and action. Figure 4.1 shows the difference between an MDP and SMDP.

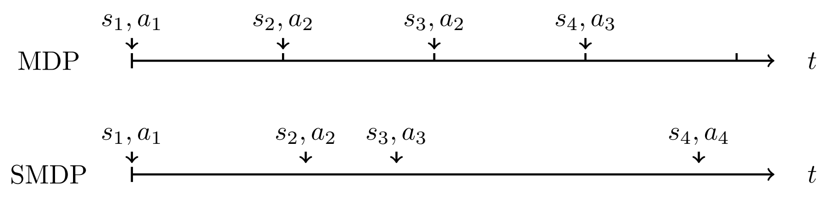

Figure 4.1: Time line of two environments, the one on top showing the uniform time between actions of an an MDP and the lower showing the variable time between actions of an SMDP

The transition time, also called sojourn time, can be express as

$$\tau_{n+1} = t_{s_{n+1}} - t_{s_n}.$$

The extension from MDP to SMDP is formally expressed as the transition probabilities P for the SMDP being a joint distribution over both the transition probability and the transition time,

$$P(s_{n+1}, \tau_{n+1} | s_n, a_n).$$

Because the transition to the next state is dependent on when the action was taken the Markov property is violated, when the process is interpreted as a continuous time stochastic process, hence the name semi-Markov decision process. The main difficulty when dealing with SMDPs is how to account for the variable transition time when designing the reward function. As the time of an action increases, the variance of the next state increases. The shorter time an action takes the smaller the potential difference of the states before and after an action. The combination of long and short actions makes it difficult to learn an optimal policy. The actions which take a short time and are taken in a good state are likely to end up in a very similar state, thus a good state. On the other hand if an action which takes more time is chosen there is a higher risk of ending up in a bad state. For the first action little time passes and little changes in the environment, thus as soon as positive reward is received, by taking the shortest action from that state it is likely that the agent ends up in a similar good state and receives another positive reward. Therefore comparing actions of significantly different lengths is difficult. For a thorough explanation of the complexities of SMDPs we refer to [1, 6, 18].

4.2 Generalised semi-Markov decision process

A generalised semi-Markov decision process, as introduced in [27], can be viewed as several SMDPs all effecting the same environment. The asynchronous SMDPs affect the state of the environment but do not directly affect each other. This can be illustrated with an example.

Consider a help desk in a store. Customers arrive at random and if there is no line they get service immediately, otherwise they wait for their turn. Depending on the customer's problem, the time it takes to help customers varies. If a customer walks into the shop when another customer is getting help the state of the environment changes but it does not affect how long it takes to help the current customer. In this environment, events of the different process happen intermittently but do not affect the other process. If there is one help desk, there are two processes that govern the size of the queue, the arrival of new customers and the servicing of customers. These processes are asynchronous. If there were two help desks they would both affect the queue, but they would not affect each other, thus in that case there would be three asynchronous processes governing the size of the queue.

In the case with one help desk, both processes affect the state of the queue, but the next state of the queue is dependent on when the latest customer was starting to get served, thus the Markov property is lost.

Since the Markov property is lost, it is difficult to solve GSMDPs. The process has to be transformed to processes which are solvable, for example MDPs or SMDPs. One possible way to reattain the Markov process is to include the necessary history in the state representation [20].

The use of a GSMDP framework allows for an elegant model of problems which can be formulated as asynchronous SMDPs. The GSMDP formalisation provides a common description for a class of problems, for which general solutions can be made. For a more thorough description of GSMDP we refer to [18, 27].

Chapter 5

Production line

In this thesis we apply RL algorithms to a production planning problem in order to analyse the applicability and the scalability of the algorithms. In the previous chapters the algorithms and theory is presented. In this chapter the environment which the algorithms try to optimise is presented.

The implementation of the production line was provided by Syntronic. The source code for the simulation environment was available to the authors.

5.1 Environment description

The environment is a simulation of a production line which can produce a number of different products. The finished products are referred to as stock keeping units, SKU. During the production line the unfinished products are referred to as units. The production line considered in this thesis consists of five stages. Before becoming an SKU the units pass through every stage. The stages and their order in the production line can be seen in Figure 5.1.

Stage 1	Resources
	$\downarrow$
Stage 2	Presses
	$\downarrow$
Stage 3	Storage
	$\downarrow$
Stage 4	Oven
	$\downarrow$
Stage 5	Stock.

Figure 5.1: The five stages of the simulated production line in the order of occurrence.

An SKU begins as a piece of raw material stored in the first stage, resources. In the second stage raw material is processed by a press. When the press has processed the raw material the units are stored in the third stage of the production line, the storage. The units in the storage can then be processed by an oven, the fourth stage of the production line. Once the oven has processed the units they are finished and referred to as SKUs. The SKUs are stored in the fifth and final stage of the production line, the stock, where they remain until they are purchased. For an easy conceptual overview of a production line with multiple presses and ovens see Figure 1.1.

5.1.1 Stock keeping units

The production line produces products which when completed are refereed to as stock keeping units. The SKUs are defined by shape and material.

5.1.2 Resources

The resources refer to the raw material used to produce the SKUs. The resources are unlimited.

5.1.3 Press stations

A press moulds raw material into the shape of a desired SKU. A production line can have multiple presses, in that case the presses are identical and are controlled individually. A press can process all of the possible combinations of the shapes and materials and has a unique setting for each combination. For the press to be able to process a specific shape and material the press needs to change to the corresponding setting first. The time it takes to change setting and process a unit is determined by uniform distributions. There is one distribution determining the time to change between shape settings, another for the material settings and a third for the actual processing time.

5.1.4 Storage

Once a press has processed raw material the units are stored in storage until they are processed by the oven. There is no limit on the number of units which can be stored in the storage.

5.1.5 Ovens

The oven harden the units moulded by the press. A production line can have multiple ovens, in that case the ovens are identical and are controlled individually. The ovens can process all possible combinations of shape and material. An oven can only produce one combination of shape and material at a time. However, the setting of the oven depends only on the material of the unit which is to be processed. If the material of two units is the same but the shape is different they can be processed subsequently without the need to change setting of the oven. Different settings of the oven corresponds to different temperatures. The time it takes to change setting depends on the temperature difference of the settings the oven is changing between. The time it takes to increase and decrease the temperature 1 degree is uniformly distributed. The time it take to process units is also determined by a uniform distribution.

5.1.6 Stock and purchases

When the units have been processed by an oven they are stored as SKUs in stock. The stock has no limit on the number of SKUs which can be stored in it. The SKUs which are in stock can be purchased by a customer. The frequency at which purchases of each SKU occur is determined by the amount of time between the purchases. The time between the purchases of each SKU is given by an exponential distribution with the probability density function

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0, \\ 0 & x < 0. \end{cases}$$

In the event that a specific SKU is purchased the time until the next purchase of that type of SKU is determined by the exponential distribution above with with λ corresponding to that SKU. Only one type of SKU is bought per purchase. The amount purchased at the event of a purchase is given by a

Poisson distribution with intensity $\lambda > 0$. The parameter λ may vary between the SKUs. The Poisson distribution is defined as

$$P(x = k) = e^{-\lambda}\frac{\lambda^k}{k!}, \quad k \geq 0.$$

5.2 Summary of parameters and environment features

The complete set of parameters of the production line environment are listed in Table 5.1. The features that define the state of the production line are listed in Table 6.2.

Table 5.1: List of environment parameters

M	Number of materials, $m \in \{1, 2..., M\}$
N	Number of shapes, $n \in \{1, 2..., N\}$
P	Number of presses
O	Number of ovens
G	Capacity of oven
K	Capacity of press
$\mathcal{U}(t^1_{\text{low}}, t^1_{\text{high}})$	Uniform distribution of the time it takes change shape setting in press
$\mathcal{U}(t^2_{\text{low}}, t^2_{\text{high}})$	Uniform distribution of the time it takes to change material setting in press
$\mathcal{U}(t^3_{\text{low}}, t^3_{\text{high}})$	Uniform distribution of the time it takes to process a unit in a press
$\mathcal{U}(t^4_{\text{low}}, t^4_{\text{high}})$	Uniform distribution of the time it takes change temperature of oven 1 degree
$\mathcal{U}(t^5_{\text{low}}, t^5_{\text{high}})$	Uniform distribution of the time it takes to process a unit in an oven
$\lambda^{m,n}_{\text{purchase rate}}$	Poisson rate parameter for the rate at which purchases occur
$\lambda^{m,n}_{\text{purchase amount}}$	Poisson rate parameter for the number of SKUs purchased at the event of a purchase

Table 5.2: Features defining a state

Press station (For station i):	
Material Setting	$p^i_{\mathrm{m}} \in \{1, 2, ..., M\}$
Shape setting	$p^i_{\mathrm{s}} \in \{1, 2, ..., N\ \}$
SKUs in press	$p^i_n \in \{0, 1\}$
Expected time remaining	$p^i_{\mathrm{t}} \in \mathbb{R}$
Occupied	$p^i_{\mathrm{o}} \in \{0, 1\}$
Oven (For each oven i)	
Temperature setting	$o^i_{\mathrm{T}} \in \{0, 1, 2, ..., M\}$
SKUs in oven	$o^i_{\mathrm{n}} \in \{0, 1\}$
Expected time remaining	$o^i_{\mathrm{t}} \in \mathbb{R}$
Occupied	$o^i_{\mathrm{o}} \in \{0, 1\}$
Stock Keeping Units	
Number of SKU m, n in storage	$S^{m,n}_{\mathrm{storage}} \in \mathbb{N}$
Number of SKU m, n in stock	$S^{m,n}_{\mathrm{stock}} \in \mathbb{N}$
Number of SKU m, n in all presses	$S^{m,n}_{\mathrm{press}} \in \{0, K, 2K..., PK\}$
Number of SKU m, n in all ovens	$S^{m,n}_{\mathrm{oven}} \in \{0, G, 2G, ..., OG\}$

5.3 Markovian Formulation

Applying reinforcement learning algorithms to a problem requires the problem to be of a certain type. It must be possible to retrieve the tuple defining the MDP. As an example the algorithms mentioned in Chapter 3 requires the problem to be able to provide tuples: (s, a, r, s'), where s' is the state following the action a taken in state s and r is the reward received during that transition.

Figure 5.2 shows the process of a production line with two presses and two ovens. The vertical lines correspond to a state when a station is done with it's previous action, a reward is received and a new action is chosen.

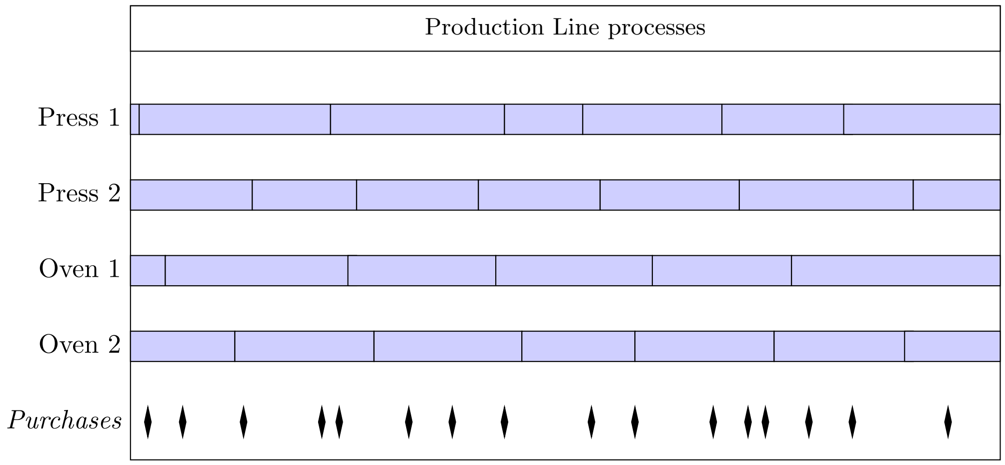

Figure 5.2: Visualisation of the parallel processes in the production line. The vertical lines correspond to the station being done and a new action being chosen.

Since the times between action and state transition for different stations overlap two subsequent states may not concern the same station. The overlapping causes problems when interpreting the reward received in the actual subsequent state due to the fact that the reward that is received may not reflect how good the latest action was. The different stations can be seen as individual processes. Thus the stations working in parallel can be modelled as asynchronous SMDPs.

By defining s and s' so that every such pair only concerns one station, s is when an action concerning a station is taken and s' is when that action is complete. The process is thus modelled as concurrent and asynchronous SMDPs. This is visualised in Figure 5.3. Using this formalism means that every tuple (s, a, s', r), s and s' are states with events concerning a unique station. This formulation allows for a stronger correlation between the actions and the rewards, which increases the agents ability to learn.

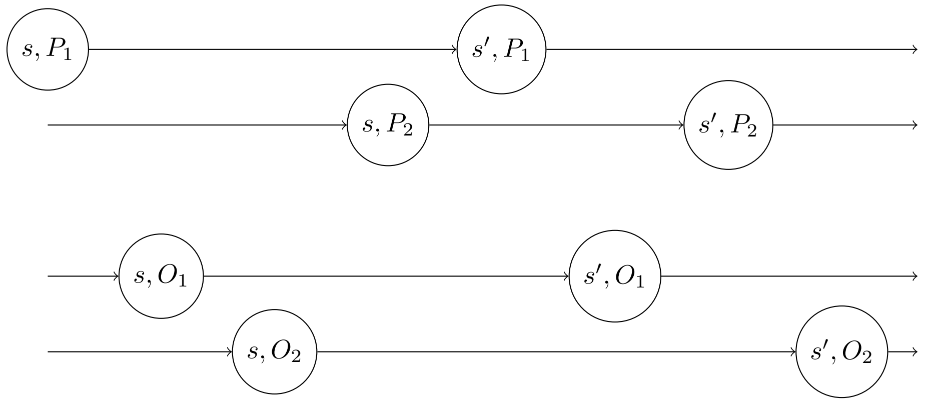

Figure 5.3: Separating the stations processes, each is modelled as an SMPD such that for every tuple $(s, a, s', r))$, s and s' concern the same station.

With each station modelled as a separate SMDP, the production line consists of asynchronous SMDPs and thus the production line is modelled as a GSMDP as describe in Section 4.2. By including an estimate of the time remaining for the other processes in the state representation, the agent have full information of the state and a optimal action is possible to deduce.

5.4 Agent interaction

An agent can interact with the production line and thus control the production of SKUs. The production line can be broken down into two parts that need to be controlled, the presses and the ovens. When either a press or an oven is done with the previous action the next action needs to be chosen. In a state when a press is done, the possible actions are always the same due to the unlimited amount of resources. In a state where an oven is done, the possible actions depend on which units are available in storage. Since the ovens only produce with maximum capacity, G, the storage needs to contain at least G units for an oven to be able to processes that type of unit. Thus if there are fewer than G units the action of producing that unit is not possible. In the case that there are too few of every type of unit, the oven only has one possible action; wait until a press is done and has produced more units. For both the presses and the ovens, the settings are changed if the chosen action requires a different setting than the current setting of the station. Thus the agents do not have the possibility to only change setting, but need to produce immediately after the change is done.

Chapter 6

Experiments

In order to reveal strengths and weakness of the algorithms presented in this thesis, they are implemented and used to solve the problems at hand. Their performance is then analysed and evaluated. The algorithms are used to find an optimal policy in three different simulation environment configurations.

The agents have been written in Python by the authors. The configuration and computations concerning the neural networks are implemented with the deep learning library TensorFlow[1]. The experiments have been run on machines with Intel Core i7-6500U processors with 16 GB RAM.

6.1 Experiment configurations

The details of the experiments carried out are described below.

6.1.1 Production line configurations

The algorithms developed in this thesis are tested on suite of different production lines. This is to investigate their strengths and weaknesses. In total the different configurations are used to test the different algorithms.

The first configuration has 3 ovens, 2 presses and can produce 4 different SKUs, the SKUs consist of the combinations of two shapes and two materials.

The second configuration has 12 presses, 8 ovens, 4 materials and 4 shapes, which yields 16 different SKUs. In configuration two the ovens have a larger temperature difference between the highest and the lowest temperature setting. Since the time needed to change an oven's setting depends on the temperature difference of the settings, the larger temperature span increases the time needed to change between the highest and lowest setting.

In both the first and the second configurations the demand of the different SKUs are equal. In the third configuration, however, the demand is not equal for all SKUs. In configuration three one of the SKUs have an demand ten times higher then the other SKUs. Apart from the demand, configuration three is identical to configuration one.

The distributions determining the time it takes to produce and change setting are the same for all configurations. The details of the different simulation environment configurations can be seen in table Table 6.1.

[1]`www.tensorflow.org`

6.1.2 Stockouts

A stockout is defined as an SKU not being available in stock. Every SKU not available in stock counts to the total number of stockouts. The stock is checked and the stockouts counted every time an oven has completed an action. If there are two out of four possible SKUs available in stock when an oven is in between actions it counts as two stockouts. These two stockouts are counted to the total number of stockouts during the episode.

The way the stockouts are counted there is a risk of missing stockouts. This is because the stock is checked for stockouts when an oven is available to produce again. What the oven produced just before is now already in the stock. This means that it is possible for an SKU to sell out without it being registered. Consider the following scenario: An oven starts to produce a batch of units, which when finished are SKUs of type A. During the time it takes for the oven to finish the process, SKU A sells out. When the oven is finished, the newly produced batch of SKU A is added to the stock. The stock is now checked for stockouts, since the newly produced batch of SKU A is there, no stockout is registered.

6.1.3 Episode initialisation

The initial states of every episode during the training and testing phase are defined as follows. The settings of the stations is randomly chosen. Every station is empty and ready to pick an action. The storage is initialised such that there is 20 of each SKU. The stock is initialised such that there are 20 of each SKU.

6.1.4 Training and testing

The simulation environment does not have a natural terminal state but in order to train and compare the algorithms an episode is set to be five hours in the simulation environment time. The agents are first trained and then tested. The difference between the training and testing phase is that the agent is not forced to explore and the policy is not updated during testing. During training the agents are forced to explore the environment, done by setting $\epsilon > 0$. During training epsilon is linearly reduced every episode of the first half of the training episodes. Starting at $\epsilon = 1$ it is reduced linearly until $\epsilon = 0.005$ which occurs after half of the training episodes. During testing the exploration is removed by setting $\epsilon = 0$ and thus only exploiting the learned policy.

The testing phase consists of running an agent and simulation environment 100 episodes. The metric used to compare the agents performance is the mean number of stockouts that occur during the 100 episodes.

Table 6.1: Production line settings

	Configuration 1	Configuration 2	Configuration 3
Shapes	2	4	2
Materials	2	4	2
Ovens	2	8	2
Presses	3	12	3
Oven			
Capacity (units)	15	15	14
Production time (min)	$\mathcal{U}(17,19)$	$\mathcal{U}(17,19)$	$\mathcal{U}(17,19)$
Change temp in oven (s/°C)	$\mathcal{U}(3,4)$	$\mathcal{U}(3,4)$	$\mathcal{U}(3,4)$
Temp settings (°C)	800, 900	800, 900, 1000, 1100	800, 900
Press			
Capacity (units)	10	10	10
Production time	$\mathcal{U}(4,5)$	$\mathcal{U}(4,5)$	$\mathcal{U}(4,5)$
Change material setting (min)	$\mathcal{U}(2,4)$	$\mathcal{U}(2,4)$	$\mathcal{U}(2,4)$
Change shape setting (min)	$\mathcal{U}(5,6)$	$\mathcal{U}(5,6)$	$\mathcal{U}(5,6)$
Purchases			
$\lambda_{\text{purchase rate}}$	$\begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}$	$\begin{bmatrix} 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 \end{bmatrix}$	$\begin{bmatrix} 15 & 15 \\ 15 & 15 \end{bmatrix}$
$\lambda_{\text{purchase amount}}$	$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 \\ 1 & 10 \end{bmatrix}$

6.2 Agents

In Chapter 3 the concept of the agent was introduced as the controlling entity. The agents are powered by their algorithms. In this section the agents used for the experiments are presented. Due to the nature of the problem it can be divided into to parts, controlling the presses and controlling the ovens. For this reason two agents are used to control the production line, one for the presses and one for the ovens. The first two algorithms, 'Random Agent' and 'Hard Coded Agent' do not use reinforcement learning but are implemented to establish benchmarks.

6.2.1 Random Agent

The Random Agent acts randomly. It does not take the state into account when making choices. It is implemented to set a benchmark which can be used when evaluating the performance of the more advanced algorithms.

Random Agent

```
Initialise new episode and state $s_0$
while time of model < length of one episode:
  if a press is available:
    Choose randomly an SKU to produce
  else if an oven is available:
    if storage is not empty
      Choose randomly one of the possible SKUs to produce
    else
      Wait until a press has produced something
```

6.2.2 Hard Coded Agent

The Random Agent is likely to set a lower bound in terms of performance. The Hard Coded agent is therefore a slightly more advanced agent which chooses action with consideration to the state of the environment. The decision process of the agent is explicitly coded, therefore the agent lacks the ability to learn. Every time the agent makes a decision it evaluates the state and scores the possible actions. The action with the highest score is then chosen. The pseudocode below shows the decision process of the Hard Coded Agent. The following notation is used

Table 6.2: Notation of features

Number of SKU m, n in storage	$S^{m,n}_{\text{storage}} \in \mathbb{N}$
Number of SKU m, n in stock	$S^{m,n}_{\text{stock}} \in \mathbb{N}$
Number of SKU m, n in all presses	$S^{m,n}_{\text{press}} \in \{0, K, 2K..., PK\}$
Number of SKU m, n in all ovens	$S^{m,n}_{\text{oven}} \in \{0, G, 2G, ..., OG\}$

Hard Coded Agent

```
Initialise new episode and state $s_0$
while time of model < length of one episode:
  if a press is available:
    Evaluate how good every possible action is:
    For every action $a_{m,n}$:
      score($a_{m,n}$)= $100 \cdot \mathbf{1}_{S^{m,n}_{\text{Stock}}<5}$
             $-0.2S^{m,n}_{\text{stock}}$
             $-10S^{m,n}_{\text{presses}}$
             $-10S^{m,n}_{\text{storage}}$
             $-10S^{m,n}_{\text{oven}}$
             $-30$(number of presses with settings to produce SKU $m,n$))
    Take the action with the highest score.
  else if an oven is available:
    Evaluate how good every possible action is:
    For every action $a_{m,n}$:
      score($a_{m,n}$)= $100 \cdot \mathbf{1}_{\text{number of SKU } m,n \text{ in stock and oven } <5}$
             $-S^{m,n}_{\text{Stock}}$
             $-S^{m,n}_{\text{Ovens}}$
             $-50\cdot$(number of presses with setting to produce SKU $m,n$)
      Take the action with the highest score.
```

6.2.3 Multi-agent tabular design

The tabular design uses a reduced state representation. This is because of the limitations of tabular design mentioned above. The reduced state representation uses only a few of the features defining the state of the production line. When an agent is acting on limited information, that is the information is insufficient to deduce what the optimal action is, the system is said to be partially observed.

State representation

The decision processes for the presses are only presented the reduced state of presses settings and storage. The decision processes for the ovens are presented the oven the oven's setting, storage and stock. In order to reduce the number of states further, the values of the number of SKUs in stock and storage are segmented. The state representation of the tabular agent for the press can be seen in Table 6.3 and the state representation for the oven can be seen i Table 6.4.

Table 6.3: Segmentation for press-agent

Number of SKU m, n in stock:	Segment
$[0, 15)$	1
$[15, 30)$	2
$[30, 45)$	3
$[45, \infty)$	4

Table 6.4: Segmentation for oven-agent

Number of SKU m, n in stock:	Segment
0	1
$[1, 15)$	2
$[15, 30)$	3
$[30, \infty)$	4

To capture the work performed by the other stations, the amount the other stations are currently producing is added to storage or stock depending on station. This means that if the storage is completely empty and and two out of three presses are currently producing a total of 20 SKU (n, m), the third press will perceive this as there being 20 units of SKU (n, m) in storage. The same method is used for the ovens. What the other ovens are producing is perceived as units in stock.

Reward functions

The reward function for the agent controlling the presses punishes the agent with -1 for every SKU which has fewer than 15 units in storage. Using the notation in Table 6.2 the reward function is defined as:

$$r_{\text{press}}(s) = -\sum_{k=1}^{M}\sum_{k=1}^{N} \mathbf{1}_{S^{m,n}_{\text{storage}}<15} \quad s \in \mathcal{S}.$$

For the agent controlling the ovens the reward function is similar to the function above but concerns the stock. The threshold for being punished is set to zero corresponding to the empty stock. The reward function for the agent controlling the oven is defined as:

$$r_{\text{oven}}(s) = -\sum_{k=1}^{M}\sum_{k=1}^{N} \mathbf{1}_{S^{m,n}_{\text{stock}}=0}, \quad s \in \mathcal{S}.$$

Training

The learning was performed with tabular Q-learning. The details of Q-learning algorithm can be seen in Subsection 3.4.1. The exploration probability, ϵ, is set to zero. To ensure exploring, all new state-action pairs are assigned the maximum value when first encountered, zero in this case. This ensure exploring, since unexplored states have a high q-value, they are chosen. Their value then decreases toward the true value when the q-function is updated. This is often an effective method to ensure exploration since the exploration occurs systematically instead of randomly which is the case when a ϵ-greedy policy is used for exploration. The training was done during 6000 episodes where every episode lasted for 5 hours of simulation time.

6.2.4 Deep Q-Learning Agents

The tabular agent presented above has a major drawback, it requires a reduction of the state space in order to be computationally tractable. The reduction implies loss of information, which could potentially be used to make better decisions. To make use of all relevant data of the state space an agent using DDQN, presented in Subsection 3.7.1 is implemented.

Reward functions

The different reward functions which are used and tested with the DDQN agent controlling the oven are:

1. $r_1(s) = -\sum_{i=1}^{M}\sum_{j=1}^{N}\mathbf{1}_{S^{i,j}_{\text{Stock}}=0}$
2. $r_2(s) = -\sum_{i=1}^{M}\sum_{j=1}^{N}\max\big(-0.1S^{i,j}_{\text{Stock}}+1,\ 0\big)$
3. $r_3(s) = -\sum_{i=1}^{M}\sum_{j=1}^{N}\Big(e^{-0.8S^{i,j}_{\text{Stock}}}+\mathbf{1}_{S^{i,j}_{\text{Stock}}=0}\Big)$
4. $r_4(s) = -\sum_{i=1}^{M}\sum_{j=1}^{N}\max\big(e^{-0.8S^{i,j}_{\text{Stock}}-\lambda_{\text{purchase amount}}},1\big)+\mathbf{1}_{S^{i,j}_{\text{Stock}}=0}$

The first reward function, $r_1(s)$, gives a negative reward when stockouts occur which reflects that stockouts are bad. Reward function two and three give a large negative reward for stockouts but also attempts to capture the fact that the more SKUs in stock the less likely a stockout is to occur. Thus the reward increases as the number of SKUs in stock increases. Reward function four is similar to reward function two and three but also takes into account the demand of the SKUs, such that the reward is adjusted accordingly. The reward used for the agent controlling the press punishes the agent for having fewer than 15 of an SKU in storage and also if there is a big difference in the number of SKUs in storage. The reward function for the press agent is

$$r(s) = -\sum_{i=1}^{M}\sum_{j=1}^{N}\mathbf{1}_{S^{i,j}_{\text{storage}}<15}+0.25\cdot\mathbf{1}_{S^{i,j}_{\text{stock}}=0}-\mathbf{1}_{\max(S_{\text{stock}})-\min(S_{\text{stock}})>20}\cdot\left(\frac{\max(S_{\text{stock}})-\min(S_{\text{stock}})}{30}\right)^2.$$

Neural network architecture

Neural networks are very flexible in terms of design. The design needs to be carefully chosen to achieve the desired performance. In total the function approximation consists of four neural networks. One agent is trained for the presses and one for the ovens. Both have a Q-network and a training network.

All the neural networks consist of an input layer, two hidden layers and an output layer. The networks for the two agents differ in the number of nodes in each layer. These are different for for our three production line configurations. The hidden layers use the rectified linear function as activation function and the output layers uses a linear function as activation function. The number of nodes in each layer of the networks can be seen in Table 6.5 below. The number of nodes and number of layers were chosen by trial and error.

Table 6.5: Nodes in the ANNs used

Parameter	Configuration 1 & 3		Configuration 2	
	Press	Oven	Press	Oven
Input	27	15	252	155
Hidden layer 1	108	60	1008	620
Hidden layer 2	108	60	1008	620
Output	4	5	16	17

The weights in the networks are initialised with a normal distribution with zero mean and 0.1 standard deviation. The biases are initialised as 0.01.

The output of the neural network is the Q values for each of the possible actions. The neural networks for the oven have one more output node compared to the neural networks for the press. The extra node corresponds to the action of waiting until a press is done. This action is set to only be available when the storage is empty.

State representation oven

The features of the environment used as input to train the network for the agent controlling the oven are S_{stock}, $S_{storage}$, current actions of all the ovens and the expected time remaining until they are finished.

The features are passed to the network as a vector. Before the vector is passed to the network the features are preprocessed. The number of each SKU, S^i_{stock} is divided by a factor 10. Each value $S^i_{storage}$ is converted to 1 if storage contains the oven's capacity of SKU i, otherwise the value is set to -1.

The current actions of each oven is represented as a vector with a length equal to the number of SKUs in the simulation environment where each position, i, of the vector represents the action of producing SKU i. If an oven's current action is to produce SKU i the ith element in the vector is set to one and all other values are set to -1.

The time remaining, t_r, until each oven is done with its current action is calculated by dividing the expected finishing time, t_f, of the action minus the elapsed time, t_e, with the average time it takes to produce an SKU, t_a. The remaining time, t_r, is thus calculated in the following way

$$t_r = \frac{t_f - t_e}{t_a}.$$

The setting of the oven which needs a new action is represented with vector the same length as the number of possible settings. The position corresponding to the current setting is set to one. The other elements in the vector are set to -1. An example of a vector which is passed to the neural network for the oven when configuration one is used:

$$[\underbrace{1.2, 0.1, 2.0, 1.8}_{\text{Stock}}, \underbrace{1, 1, -1, 1}_{\text{Storage}}, \underbrace{-1, -1, 1, -1}_{\text{Action of other oven}}, \underbrace{0.82}_{\text{Time remaining}}, \underbrace{-1, 1}_{\text{Setting}}]$$

By including the action and time remaining for the other oven the Markov property is regained, and the agent have full information when deciding the next action.

State representation press

The state representation for the press is much the same as the oven's state representation. For the DDQN agent handling the press the state representation consists of what is in storage and stock, the setting of the other presses and their time remaining until they are finished with their current job. Also the press's own setting is represented. The preprocessing for the press differs somewhat from the ovens preprocess. The number of SKU in stock, S^i_{stock}, is divided by 10 as for the oven. However for the press what are currently being produced by the ovens are added to the stock inventory, S^i_{stock}. The press thus sees what being processed in the ovens as already done and in stock. The storage content is divided by 10. The setting for the other presses and the time remaining is represented in the same way as for the ovens. The press's own setting is also described in the same fashion used for the ovens.

An example of a vector which is passed to the neural network for the press when configuration one is used:

$$[\underbrace{1.2, 0.1, 2.0, 1.8,}_{\text{Stock}} \underbrace{1.5, 1, 2, -1,}_{\text{Storage}} \underbrace{-1, -1, 1, -1}_{\text{Action of other press}}, \underbrace{0.64}_{\text{Time remaing}},$$
$$\underbrace{-1, 1, -1, -1}_{\text{Action of other press}}, \underbrace{0.82}_{\text{Time remaing}} \underbrace{-1, 1, -1, -1}_{\text{Setting}}]$$

Hyperparameters

The replay memory is set to have a capacity of 500000. If the replay memory gets filled the oldest entry is removed. The hyperparameter determining the exploration rate, ϵ, is variable during training and decreases linearly for the first half of the training episodes. For the second half ϵ remains constant.

Table 6.6: Hyperparameters

DDQN	
ϵ	$\max(1 - \frac{2\cdot\text{episode}_{\text{current}}}{\text{episode}_{\text{total}}}, 0.005)$
t_{terminal}	5
n_{training}	1000
f_{training}	1
f_{update}	10000
batch size	32
α	1e-7
γ	0.99
ADAM optimiser	
β_1	0.9
β_2	0.999
ϵ_{ADAM}	1e-08

Chapter 7

Results

In this chapter the results from running the agents in the environment configurations described in Chapter 6 are presented. The testing phase consists of running 100 episodes of each configuration with the different agents. Every episode in the testing phase is five hours of the environments time. The objective is to minimise the number of stockouts. The results show the mean number of stockouts during the testing phase.

7.1 Configuration One

Recall that configuration one from Subsection 6.1.1 consists of a production line with three presses, two ovens, two materials and two shapes. The number of stockouts for the different agents is shown in Figure 7.1. The results are numerically presented in Table 7.1.

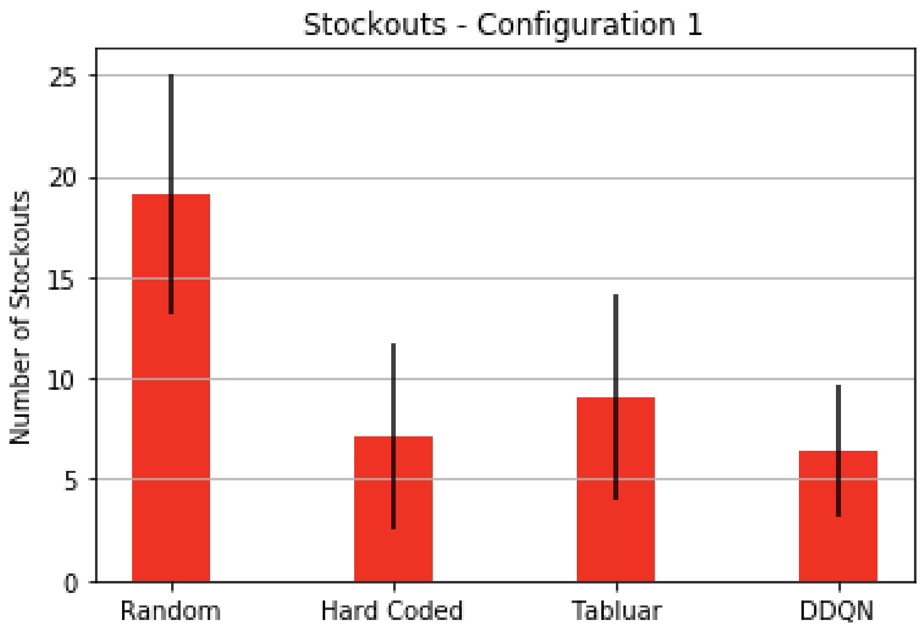

Figure 7.1: The bar graph shows the mean number of stockouts that occur during an episode when the agents control the production line with configuration one. The results are averaged from 100 episodes for each agent. The DDQN agent uses reward function 3.

Table 7.1: Results from production line one

	Stockout	
	Mean	Std
Random	19.12	5.95
Hard Coded	7.17	4.60
Tabular	9.98	7.28
DDQN	5.72	2.48

Table 7.2: The table shows the same results as in Figure 7.1 but also includes the energy consumption.

The results show that both agents utilising RL are able to learn strategies which are superior to acting randomly. The tabular agent trains for 6000 episodes and the DDQN agent is trained during 500 episodes, which illustrates one of the benefits gained with function approximation.

7.2 Configuration Two

The second production line configuration described in Subsection 6.1.1 is larger than configuration one, it consists of twelve presses, eight ovens, four shapes and four material. For this larger configuration, the tabular agent had to be omitted due to the large state space. Training the tabular agent in this environment is infeasible due to the immense amount of time needed to visit all the states and try all actions multiple times. Further more, storing and accessing the look up table efficiently becomes a problem of its own, due to the large amount of data. Therefore the Figure 7.2 and Table 7.3 only show the results from the random, hard coded and DDQN agents.

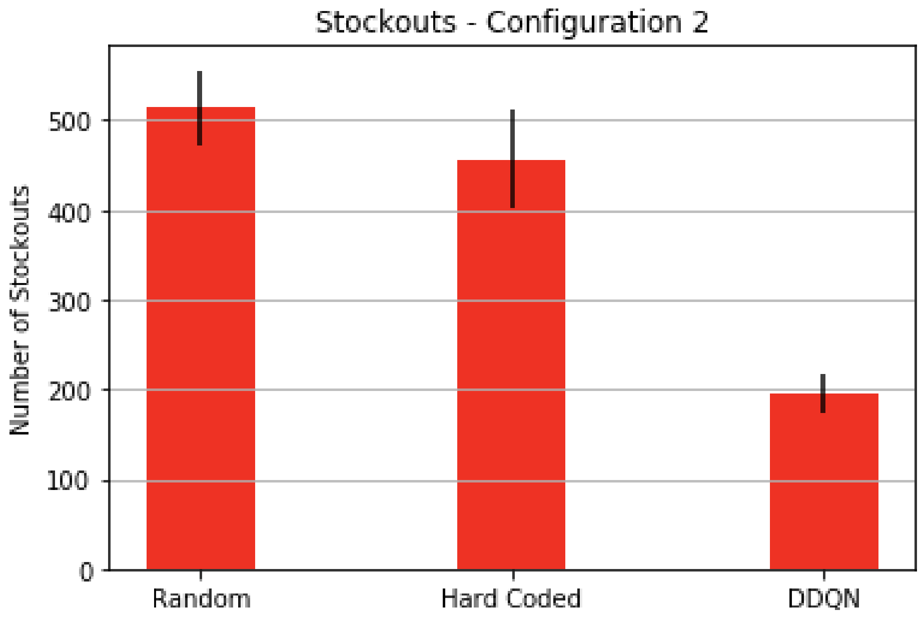

Figure 7.2: The bar graph shows the average number of stockouts that occur during an episode when the agents control the production line with configuration two. The results are averaged from 100 episodes for each agent. The DDQN agent uses reward function 3.

Table 7.3: Results from production line two

	Stockout	
	Mean	Std
Random	510.71	41.14
Hard Coded	468.09	54.95
DDQN	195.49	22.09

The hard coded agent does not adapt to the changed production line, and preforms only slightly better then the random agent. The DDQN agent on the other hand is able to learn a policy which is superior to the benchmark set by the random and harcoded agents. Moreover, the benefits of the function approximation is further noticeable as it is infeasible to use the tabular agent in this configuration while DDQN performs well.

7.3 Configuration Three

The third configuration described in Subsection 6.1.1 is similar to configuration one but the demand of the SKUs is not equal. Configuration three also has four SKUs but the demand of one SKU is higher than the demand of the other three. The DDQN is tested with reward function 3 and reward function 4. Reward function 4 uses information about the demands of the SKUs, see Subsection 6.2.4.

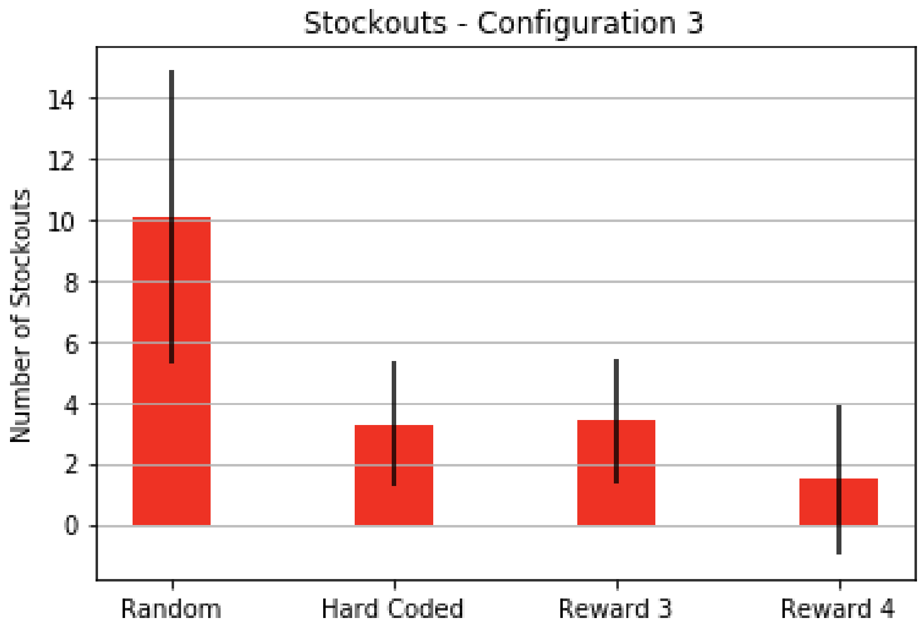

Figure 7.3: The bar graph shows the average number of stockouts from 100 episodes when there is an unequal demand; the expected number of SKUs purchased is one for three of the SKUs and 10 for one of the four SKUs.

The results show that the a reward function which incorporates information about the demands improves the DDQN agent's ability to avoid stockouts. Including information about the demand means that the agents does not have to learn both the demand and a suitable policy at the same time.

7.4 Comparison of reward functions

Figure 7.4 shows the results when the DDQN agent uses the different reward functions presented in Subsection 6.2.4 in configuration one. As a comparison the results of the random agent is included.

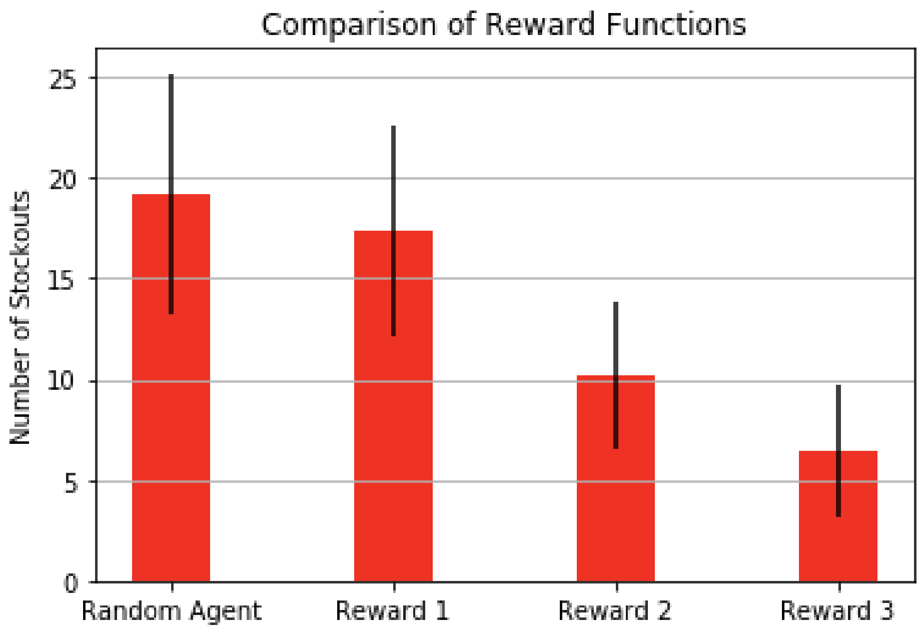

Figure 7.4: The bar graph show the number of stockouts obtained per episode averaged over 100 episodes. It illustrates the importance of reward shaping. The configuration 1 is used, the same as is used in Figure 7.1.

The reward function four is omitted in the comparison as it was specifically designed for the unequal demand found in configuration three. The results show that with a coarse reward function the agent performs on par with an agent acting randomly. With a reward function which gradually differs between good and bad states the agents ability to learn and its performance is improved. It is clear form Figure 7.4 that the choice of reward function is of importance.

Chapter 8

Discussion

The goal of this thesis is to evaluate the application of reinforcement learning as a tool for production planing. In the previous chapter the results of the experiments show that the algorithms have a potential for this application. In this chapter the results are further discussed and insights gained during the process of working on this thesis presented.

8.1 Results

8.1.1 Configuration One

Section 7.1 shows that the agents which use RL learn how to act in the environment in order to avoid stockouts without any prior knowledge. Stockouts do occur but significantly fewer times compared to the agent acting randomly. Even thought the tabular agent uses a reduced state representation, it is able to perform significantly better than the agent acting randomly. The tabular agent is however not able to meet the result of the hard coded agent. The extra information available to the agent using DDQN creates a significant improvement over the tabular agent. Not only does the DDQN agent perform much better than the tabular agent, it is also able to learn its policy much faster. The DDQN agent is for 500 episodes compared to the tabular agent which is trained for 6000 episodes. The updates of function approximation affects all states but especially similar states which makes training much faster.

Designing a policy for the hard coded agent is an easy task for the small production line configuration. The small number of SKUs and an equal demand make it quite simple. This explains why the hard coded agent performs almost as well as DDQN and with a little more work it could possibly perform as good or even better.

8.1.2 Configuration Two

The results clearly show that the hard coded agent from configuration one simply can not be used to control configuration 2. It only performs slightly better than the agent which acts at random. The DDQN agent on the other hand performs fairly well. In comparison with the hard coded agent it gets less than half the number of stockouts. The difference is significantly larger for configuration two than for configuration one. Due to the many more choices which could also have a larger effect. In configuration one the maximum temperature difference was 300 degrees whereas in configuration two it is 1100 degrees. The larger the difference between an oven's current temperature and the desired temperature, the longer it takes to make the change. This is something the agent using DDQN is able to learn while the hard coded does not take it into account. It is possible to include in the hard coded agent but one would need to weigh the importance of the temperature difference compared to the other features used to make the decisions.

There are no results for the tabular agent for configuration two. The tabular agent simply can not handle the large state space even with a binary representation. Visiting all the unique states numerous times is a time consuming task making training very slow. For the tabular agent every visited state-action pair needs to be stored thus a large amount of data is quickly generated, data which needs to be readily accessible. This is possible to do but put more focus on data handling which is a separate topic not covered in this thesis would be needed.

8.1.3 Configuration Three - Unequal demand

The results of the unequal demand further shows that the agent using DDQN can learn different configurations. Even though it learns and performs fairly well there is still room for improvement. By giving the agent better feedback by slightly altering the reward function, the agent is able to learn an even better policy. As seen in Section 7.3, the agent using reward function 4 performs better than the one using reward function 3. This gives support to the idea of either providing more information through the reward function or having a reward function which changes as the agent experiences the environment. A variant of this idea is suggested in [17], where states which end up in a bad state within a number of actions are identified. That identified states can then be used to punish the agent for being close to a bad state.

8.1.4 Different reward functions

The results of testing the different reward functions show the importance of the reward function's design. By only giving the agent a negative reward when stockouts occur the agent receives little information about the actions that do not result in a stockout. The better the feedback of an action is the easier the agent will learn the task. Better feedback means more information about how good the action was. This on the other hand has to be done with care. If not, the reward may include information which may interfere with finding an optimal policy.

8.2 Production planning

Since the problems arising in production planning often can be formulated as Markovian decision processes, the use of RL may appear natural. With the recent development of deep reinforcement learning there appears to be great potential for RL as a tool for solving problems with large state spaces. Therefore, deep reinforcement learning appears useful for solving problems appearing in industry, such as production planing which often has a large state space. However, as seen in this thesis the production planing problem does not fit perfectly into the MDP framework, which many of the latest developments have been focused on. The operations research community is however, used to dealing with problems lying outside the MDP framework. Therefore the development of RL could benefit greatly, in the authors opinion, from the production planning- and operations research communities. This is because of the experience these communities have working with Markovian decision processes. Their results and knowledge have not yet been incorporated into the latest developments in RL.

8.3 Reinforcement learning

The problem investigated in this thesis comes from a practical problem, it is thus not an toy problem to illustrate or exemplify a reinforcement learning method. This introduces challenges, among which is the problem that the process investigated might not naturally fit in to the developed and tested RL theory. This problem is accentuated by the fact that state of the art in RL currently is mostly backed up by empirical results. Assessing the results is thus troublesome, since the potential sources of error are numerous.

Another difficulty introduced by the practical origin of the problem is that several areas on the frontier of current research has to be handled all at once. Again this introduces challenges, partly because the theory is underdeveloped, but also because the interplay between different special cases is unknown.

The DDQN algorithm is limited in its action space. The problem DDQN tries to solve is essentially a regression problem over the possible actions. As the number of actions grows the number of nodes in the output layer grows as well. To retain accuracy the network needs to be increased in size. At some point the training time will become infeasible when the action space becomes too large but this point is unknown to the authors. Existing RL methods capable of large, even continuous, action spaces all relies on spatial correlations in the state space. This is not the case for the problem investigated in this thesis.

Deep reinforcement learning is, at the present time, moved forward by empirical results achieved by simulations [1, 2, 4]. If a new algorithm preforms better on a benchmark, then it is considered the new state of the art. Little effort goes into formally proving stability, convergence or speed of convergence. The lack of a rigorous theoretical foundation ensuring the behaviour and outcome of an algorithm, makes the application of RL to real physical systems questionable, at least for the time being.

The step from solving toy problems to handling practical problems, also accentuates the need for correct modelling. Since RL algorithm learns from experience it is preferable to train the algorithm within a simulation. This means that a model of the problem must be constructed. As always when using models, it is the model that is solved. Any discrepancies between the model and the real problem may lead to an insufficient solution.

As mentioned in Chapter 4 the MDP framework is limited and there is a large class of problems which can not accurately be modelled as MDPs. The research into how to model and solve problems, which do not fit the MDP framework, is limited [18]. Expanding the theory of MDP extension have the potential to vastly expand the applicability of RL.

8.4 Simulation Environment

The simulation environment was provided by Syntronic. The aim of the simulation program was to be used to investigate and identify difficulties occurring when applying RL to production planning. It was designed using intuition and common sense, the compliance of the simulation environment to production planning theory and practice is thus unknown. In this sense the results produced may be of limited use for production planing since there might be assumptions that are not compliant to standard practices within the production planing community. For the simulation environment to be a viable tool for development of production planing software, it should be investigated from an operations research point of view and, if necessary, modified accordingly.

The simulation environment used for the experiments in this thesis has a limited action space. It lacks the possibility to wait instead of producing. The action of waiting could be the optimal action in a state when an oven would otherwise need to change setting due to a lack of units requiring the current setting of the oven. The press may almost be finished producing such units so instead of changing the setting of the oven the optimal action could be to wait until the press is done. However it requires further development of the algorithms discussed in this thesis. The action to wait introduces a potential of having very large time differences between actions, see Section 4.1 for details of this problem.

8.5 Agents

The use of multiple agents makes the program flexible to some changes in the production line. This is, as argued in [11], an advantage as this makes the program scalable in the number of layers in the production line. Since an extra layer in the production line could with very little extra work can be added and then controlled by adding an extra agent. However the interplay between DDQN agents have not, to the authors best knowledge, been investigated and is therefore a suitable subject for further research.

8.6 Future work

There exist many potential directions to progress from the results of this thesis. Some of the more interesting difficulties to further investigate are the following:

- Hyperparameter tuning
- Design of reward function
- Large differences in time duration of actions
- Theory of GSMDPs.
- Scalability

All of theses difficulties are important and interesting enough to be studied on their own.

Chapter 9

Conclusion

The goal of this thesis was to evaluate the potential of reinforcement learning as a tool for solving production planning problems. From the experiments and results, we conclude that reinforcement learning has great potential. However getting reinforcement learning to work in any environment can be difficult. The production line setup had to be conformed into a MDP in order to achieve positive results. To be able to handle a larger class and more general problems, e.g. SMDP and GSMDP, the theory of Markovian decision processes and the corresponding reinforcement learning theory needs to be developed further. Especially the application of deep reinforcement learning to problems which does not fit naturally into the MDP framework, are in need of further investigation. To the best of the authors knowledge deep reinforcement learning has not been applied to generalised semi-Markov decision processes prior to this thesis.

In order to make RL a viable tool in production planing on the scale common in industry, the difficulties mentioned in Section 8.6 need to be researched. Increased knowledge into these areas would allow RL to more easily be applied to production planning.

In conclusion the authors find the application of RL to production planning promising, but further research is required before real world, large scale problems can be handle efficiently.

Bibliography

[1] Das et al. "Solving Semi-Markov Decision Problems Using Average Reward Reinforcement Learning." In: *Management Science* 4 (1999), p. 560. ISSN: 00251909.

[2] Mnih et al. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (2015), pp. 529–533. ISSN: 00280836.

[3] Ng et al. *UFLDL Tutorial.* 2017-12-18. URL: `http://ufldl.stanford.edu/tutorial/`.

[4] Silver et al. "Mastering the game of Go without human knowledge." In: *Nature* 550.7676 (2017), pp. 354–359. ISSN: 1476-4687.

[5] Cornelius Baur and Dominik Wee. "Manufacturing's next act". In: *McKinsey Quarterly, Jun* (2015).

[6] M Baykal-Gürsoy. "Semi-Markov Decision Processes". In: *Wiley Encyclopedia of Operations Research and Management Science* (2010).

[7] Richard Ernest Bellman. *Dynamic programming.* A Rand Corporation research study. Princeton, N.J. : Princeton Univ. Press, 1957, 1957.

[8] Denny Britz. *Learning Reinforcement Learning (with Code, Exercises and Solutions).* 2017-12-18. URL: `http://www.wildml.com/2016/10/learning-reinforcement-learning/`.

[9] D Dranidis and E Kehris. "A production scheduling strategy for an assembly plant based on reinforcement learning". In: *Proceedings of the 5th WSES International Conference on Circuits, Systems, Communications and Computers. WSEAS.* 2001, p. 29.

[10] H. A. Eiselt and Carl-Louis Sandblom. *Operations Research. [Elektronisk resurs] : A Model-Based Approach.* Springer Texts in Business and Economics. Berlin, Heidelberg : Springer Berlin Heidelberg : Imprint: Springer, 2012., 2012. ISBN: 9783642310546.

[11] Thomas Gabel and Martin Riedmiller. "Adaptive reactive job-shop scheduling with reinforcement learning agents". In: *International Journal of Information Technology and Intelligent Computing* 24.4 (2008).

[12] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-learning." In: (2015).

[13] Alexander Hübl. *Stochastic Modelling in Production Planning: Methods for Improvement and Investigations on Production System Behaviour.* Wiesbaden : Springer Fachmedien Wiesbaden : Imprint: Springer Gabler, 2018., 2018.

[14] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[15] Jens Kuhpfahl. *Job Shop Scheduling with Consideration of Due Dates: Potentials of Local Search Based Solution Techniques.* Springer, 2015. Chap. Job Shop Scheduling – Formulation and Modeling.

[16] Örjan Larsson. “Future Smart Industry-perspektiv på industriomvandling”. In: *VINNOVA Rapport VR 2015: 04* (2015).

[17] Zachary C Lipton et al. “Combating Deep Reinforcement Learning’s Sisyphean Curse with Reinforcement Learning”. In: *arXiv preprint arXiv:1611.01211* (2016).

[18] Jeremy James McMahon. “Time-dependence in Markovian decision processes.” PhD thesis. 2008.

[19] Michael A Nielsen. *Neural networks and deep learning.* 2015.

[20] Emmanuel Rachelson et al. “A Simulation-based Approach for Solving Generalized Semi-Markov Decision Processes.” In: *ECAI.* 2008, pp. 583–587.

[21] David Silver. *UCL Course on RL.* 2015.

[22] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: *arXiv preprint arXiv:1712.01815* (2017).

[23] Brenna Sniderman, Monika Mahto, and Mark J Cotteleer. “Industry 4.0 and manufacturing ecosystems: Exploring the world of connected enterprises”. In: *Deloitte Consulting* (2016).

[24] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning : an introduction.* Adaptive computation and machine learning. Cambridge, Mass. : MIT, cop. 1998, 1998. ISBN: 0262193981.

[25] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards.” In: (1989).

[26] Neha Yadav, Anupam Yadav, and Manoj Kumar. *An introduction to neural network methods for differential equations.* Springer, 2015. Chap. History of Neural Networks.

[27] Håkan LS Younes and Reid G Simmons. “A formalism for stochastic decision processes with asynchronous events”. In: *Proc. of AAAI Work. on Learn. and Plan. in Markov Processes* (2004), pp. 107–110.