Approximating optimal solutions for Job Shop Scheduling Problems with unrelated machines in parallel using generalizable deep Multi-Agent Reinforcement Learning

Submitted in partial fulfillment of the requirements
for the degree of master of science

Merel Wemelsfelder
12566365

MASTER INFORMATION STUDIES
Data Science

Faculty of Science
University of Amsterdam

Date of defense: 2020/06/19

1^{st} *Examiner*
dr. Joost Berkhout
j2.berkhout@vu.nl

Faculty of Science
Vrije Universiteit

2^{nd} *Examiner*
Cristian Rodriguez Rivero, Ph.D.
c.m.rodriguezrivero@uva.nl

Faculty of Science
University Of Amsterdam

ABSTRACT

In production plants, scheduling plays an important role in reducing production-loss, while ensuring that customer due dates are met. The investigated part of this particular problem is represented by a Job Shop Scheduling Problem with unrelated machines in parallel. The aim of this study is to find approximately optimal solutions for this problem, by using a Multi-Agent Reinforcement Learning algorithm. A Neural Network is used as its policy value function, in order to make the algorithm applicable to many different problem instances, and thus create generalizability. It is found that the algorithm indeed performs much better on large instances, which it has never seen before, when initialized with weights that were pre-trained on other, small, instances, than when using random weights. The performance of the designed algorithm is also compared to that of a Mixed-Integer Linear Programming (MILP) algorithm. It is concluded that it performs worse than the MILP for small problem instances, but better for large ones, especially when little computing time is allowed.

Code: github.com/MerelWemelsfelder/RL_scheduling.git

1 INTRODUCTION

In production plants, the efficient scheduling of manufacturing jobs plays an important role in meeting demand and reducing production-loss, while ensuring that customer due dates are met [6]. This is challenging since products often share machine capacity, as a result of a limited number of production resources [3, 42].

Conway et al. [11] have introduced a notational framework, to enable describing a variety of scheduling problems by the characteristics of jobs and machines, and the criteria by which a schedule will be evaluated. In this framework, a Job Shop Scheduling Problem (JSSP) is the task of assigning each operation to a specific position on the time scale of a specific machine. In cases that contain more machines, and where at least some of the jobs have a sequence of operations to be performed, the collection of machines is said to constitute a flow shop. The assignment of operations is then called a Flow Shop Scheduling Problem (FSSP). Over time, this framework has been extended. The Flexible Job Shop Scheduling Problem (FJSSP) and the Flexible Flow Shop Scheduling Problem (FFSSP) are extensions of the JSSP and FSSP respectively. In both cases, instead of single machines in series, the operations have to pass work stations with at each work station a number of identical machines in parallel [3, 33].

This thesis is oriented towards production scheduling in the animal-feed industry, using production process knowledge and information on plant structure made available by ENGIE Services [6]. The production process at ENGIE can be represented by a FFSSP. Usually one of the work stations is the bottleneck, meaning that optimizing only a single work station would lead to efficiency improvements. This work station can be represented by a JSSP with unrelated machines in parallel. Each machine, or resource, consists in itself of a flow shop of units. Optimizing the solution to this partial problem will be the main goal of this thesis.

Solving JSSPs is a complex issue. The total number of possible solutions for a problem with n jobs and m machines is $\frac{(m+n-1)!}{(m-1)!}$, and since no deterministic polynomial-time algorithm has been found to solve this kind of problem [7], they are classified as NP-hard [6, 16, 22]. Techniques such as Mixed-Integer (Linear) Programming (MILP) have demonstrated the ability to obtain optimal solutions for well-defined problems [6, 26, 31, 39], but since this can not be done in polynomial time, this approach is in most cases limited to a theoretical level. However, these techniques are very appropriate for finding optimal solutions for small problem instances. We will therefore use this property of the MILP algorithm, combined with a commercial solver, to guide our own algorithm during the learning process.

Several methods have been proposed to approximate solutions, such as Tabu Search [8, 10, 12, 25, 40], Evolutionary Algorithms [2, 20, 21, 27, 32] and Reinforcement Learning (RL). Zhang and Dietterich [48] have shown that RL has the potential to quickly find high-quality solutions to scheduling problems. Further investigating the use of RL for this purpose is therefore the focus of this thesis. Reinforcement Learning can be implemented with either one or multiple agents. Although the use of multiple agents can come with many challenges considering policy learning [24], Gabel [16, 17], Beke [4] and Jiménez [26] advocate the use of multi-agent systems for solving scheduling problems. By implementing multi-agent learning with full observability, we can profit from its advantages, but reduce the negative consequences. Besides, this will make the algorithm easily extendable to a more decentralized multi-agent system, which is in some cases more appropriate to represent real-world plants.

In order to find schedules in a computationally efficient manner, the RL algorithm has to learn how to quickly solve problems that it has not seen before, by first learning how to solve a limited number of other instances.These unseen problem instances may result from the increase or decrease of the number of functioning machines, the arrival of new

jobs that have to be processed, or the need to solve a scheduling problem in a similar but different plant. Investigating the generalizability of the algorithm is therefore expected to contribute largely to the algorithm's applicability. The approach to achieve this will be by using deep RL, meaning that an artificial Neural Network (NN) is used for policy value estimation. This technique disconnects policies from individual agents and actions, and rather makes them dependent on the general aspects of the environment.

Our approach is innovative with reference to existing approaches in literature, differing in various aspects. Zhang and Dietterich [48], and Zeng and Sycara [47] both use single-agent RL for doing repair-based scheduling, instead of building schedules from scratch using multiple agents. Both Jiménez [26] and Beke [4] use multi-agent RL to address scheduling problems similar to ours, but Q-learning is used as an approach, in contrast to our Neural Network. Besides, their main focus lies in handling uncertainty and generating robust schedules, instead of focusing on the generalizability of the algorithm. The approach of Gabel and Riedmiller [16, 17] is most similar to ours, by also implementing deep multi-agent RL for different types of scheduling problems. However, they strive to generate reactive schedules in order to schedule in real-time, using a partially observable environment, in order to create complete decentralization of the system. Furthermore, none of these studies use a MILP algorithm (or similar) to assist their algorithm during training.

To conclude, it can be stated that our approach is renewing, and will shed light on new ways to address scheduling problems. This will be done by answering the following research question: How can generalizable Multi-Agent Reinforcement Learning be applied to approximate optimal solutions for Job Shop Scheduling Problems with unrelated machines in parallel? To achieve this, first the existing literature on this topic will be investigated in Section 2. Section 3 elaborates on implementing the algorithm and the design choices that had to be made. In Section 4 the results from testing the algorithm's performance are presented. The thesis is concluded by a discussion of the study as a whole in Section 5, followed by drawing our final conclusions in Section 6.

2 RELATED WORK

2.1 Scheduling

Aspects that make production planning at (animal-feed) plants complicated include a varying production speed, limited storage capacity for intermediate or finished products and different customer priorities [6]. Operations might have a release date, and machines might have a set-up time between operations [4, 26].

A distinction can be made between predictive and reactive scheduling. Predictive schedules are created in a closed world, assuming that events are predictable and thus there are no uncertainties to be dealt with. Reactive scheduling addresses the problem of maintaining a schedule in a dynamic world, reacting upon events when they occur [9]. In most real-world production processes, scheduling is an ongoing process, where uncertainties are inherent and unexpected events may occur [13, 45]. However, investigating the robustness of schedules is beyond the scope of this thesis. For investigating optimality and generalizability, generating only predictive schedules suffices.

2.1.1 *Scheduling problems.*

As already mentioned in Section 1, Conway et al. [11] have introduced a notational framework to enable describing a variety of scheduling problems. Definitions of the two most relevant problems types are given below, being obtained from the book 'Scheduling' by Pinedo [33]. A visualization of these definitions, including two other closely related problem types, can be found in Appendix A.

Definition 2.1. *Job Shop Scheduling Problem (JSSP)* In a job shop with l machines, jobs have to visit the machines in a pre-defined order. A distinction is made between job shops in which each job visits each machine at most once, and job shops in which a job may visit each machine more than once.

Definition 2.2. *Flexible Flow Shop Scheduling Problem (FFSSP)* Instead of l machines in series there are m work stations with at each work station a number of identical machines in parallel. Product operations have to visit all work stations, and have to be performed in the same order of work stations. For each work station that job j visits, it needs processing on only one machine and any machine can do.

Of these problem types, the Flexible Flow Shop Scheduling Problem (FFSSP) is most comparable to the particular scheduling problem at ENGIE. However, usually one of the work stations is a bottleneck, meaning that if the scheduling could be optimized for this one work station, the quality of the schedule as a whole would also be improved. This study will therefore focus on optimizing scheduling performance for a single work station, corresponding to a JSSP with unrelated machines in parallel (JSSP-UMP) [33]. For a JSSP to consists of unrelated machines in parallel means that the time $\tau(o_{ji})$ that job j spends on machine i is dependent on both the processing needed to complete job j and the particular speed that machine i operates in when processing job j.

2.1.2 *Objectives.*

Many researchers on the topic of production process optimization agree on the general goals of a scheduling system [6, 40, 48]. The primary goal should be to find a feasible schedule which minimizes the sum of tardiness [5, 23]. More general, an objective function should be composed which

can be minimized or maximized. This may be a sum of variables or the maximum or minimum of some variable function. Typical objective functions include some of the following performance measures [26]:

- Minimization of the makespan C_{max}, which is equivalent to the time when the last job leaves the system.
- Difference between a job's due date D_j and completion time C_j.
  - Earliness measures the difference if the job is finished early: $E_j = max(D_j - C_j, 0)$
  - Tardiness measures the difference if the job is late: $T_j = max(C_j - D_j, 0) = max(L_j, 0)$
- Earliness-Tardiness: This measure has been the focus of some studies where it is desirable that jobs finish close to their due dates ($\sum_{j=1}^{n} E_j + \sum_{j=1}^{n} T_j$).

2.2 Markov Decision Processes

Scheduling problems can be formally expressed as Markov Decision Processes (MDP), where the goal is to find an optimal policy that minimizes the long-term average cost [30, 34]. Many distinguishable types of MDPs exist. Cooperative multi-agent systems are frequently modeled by Decentralized MDPs (DEC-MDP) [19]. In this thesis, we will be using a factored DEC-MDP with changing action sets to represent the scheduling environment. Some fundamental definitions will now be given as context for this framework. Definition 2.3 was adopted from Puterman [35], and definitions 2.4 and 2.5 from Gabel and Riedmiller [16, 19].

Definition 2.3. A *Markov Decision Process* (MDP) is a 4-tuple $[\mathcal{S}, \mathcal{A}, P, R]$ where $\mathcal{S}$ denotes the set of environmental states and $\mathcal{A}$ the set of actions the agent can perform. Function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ denotes immediate rewards $R(s, a, s')$ that arise when taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ and transitioning to $s' \in \mathcal{S}$. The probability $P(s'|s, a)$ of ending up in state s' when performing action a in state s is specified by the probability distribution $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$.

Definition 2.4. A factored *Decentralized Markov Decision Process* (DEC-MDP) is defined as a 7-tuple $[A_g, \mathcal{S}, \mathcal{A}, P, R, \Omega, O]$, where $A_g = \{1, \ldots, l\}$ is a set of l agents. $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_l$ is the set of potentially executable actions for all agents, where $a_i \in \mathcal{A}_i$ denotes the current action taken by agent i. The components P and R are as in Definition 2.3 except that they are now defined over joint actions $(a_1, \ldots, a_l) \in \mathcal{A}$. $\mathcal{S}$ denotes the set of environmental states, which can be factored into l components $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_l$, where $\mathcal{S}_i$ belongs to agent i. $\Omega = \Omega_1 \times \cdots \times \Omega_l$ represents the set of possible observations of all agents, and $(\omega_1, \ldots, \omega_l) \in \Omega$ denotes a joint observation, with ω_i as the observation for agent i. O is the observation function that determines the probability $O(\omega_1, \ldots, \omega_l|s, a, s')$ that agents 1 through l perceive observations ω_1 through ω_l upon the execution of a in s and entering s'.

We refer to the agent-specific components $s_i \in \mathcal{S}_i$, $a_i \in \mathcal{A}_i$, and $\omega_i \in \Omega_i$ as the local state, action, and observation of agent i, respectively. Since our problem is defined to be fully observable, each agent's local observation always truly identifies the global state.

Definition 2.5. A DEC-MDP with factored state space $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_l$ is said to feature *changing action sets*, if the local state of agent i is fully described by the set of actions currently executable by that agent ($s_i = A_i \backslash \{a_0\}$) and A_i is a subset of the set of all available local actions $\mathcal{A}_i = \{a_0, a_{i1}, \ldots, a_{ik}\}$. Thus $\mathcal{S}_i = \mathcal{P}(\mathcal{A}_i \backslash \{a_0\})$, where $\mathcal{P}$ denotes the powerset. Here, a_0 represents a null action that does not change the state and is always in A_i. Subsequently, we abbreviate $\mathcal{A}_i^r = \mathcal{A}_i \backslash \{a_0\}$.

2.3 Reinforcement Learning

According to Russel [38], an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. In Reinforcement Learning (RL), the agents that were already introduced in Section 2.2 are allowed to learn (optimal) behavior through trial-and-error interaction with the environment [26]. To do this, the following steps must be performed by each agent iteratively [4, 16]:

(1) The agent perceives an input state $s \in S$
(2) The agent determines which action $a \in A$ is best to execute, by using a decision-making policy π.
(3) The chosen action is performed.
(4) The agent obtains a scalar reward r from its environment (reinforcement).
(5) The agent learns by processing the received reward

2.3.1 *Multi Agent Learning.*

Multi-Agent Learning (MAL) comes with many advantages. These include the ability to distribute the required computations over a number of entities, an increased amount of system robustness, and an increased number of system parameters that may be altered to improve system performance [4, 16]. Most MDPs featuring multiple agents are partially observable, meaning that agents are only able to observe their own states and actions, and have no or limited knowledge about the states and actions of other agents [41]. A core challenge in such systems is how to manage proper communication between agents, in order to coordinate their decisions [44].

In the particular case of ENGIE, it is only required to consider predictive scheduling, in contrast to reactive scheduling. As a result, the environment can be made fully observable,

meaning that the agents are allowed to observe the entire environment state as well as the actions of other agents. This makes the system noticeably similar to a single-agent system. However, letting the system consist of multiple agents creates the possibility to express the global reward as a sum of local rewards, allowing for complete decentralization of decision-making [16]. This has proven to be particularly useful in applications where no global control can be instantiated and where communication between distributed working centers is impossible [17, 28]. Despite of this issue being beyond the scope of this thesis, making the algorithm suitable to include this property in the future might contribute to its general applicability.

2.4 Policy Learning

We can distinguish between two types of learning systems: policy search-based methods and value function-based methods [16]. In policy search-based Reinforcement Learning, an agent employs an explicit representation of its behavior policy π, and aims at improving it by searching the space of possible policies. Value function-based methods follow the idea of first learning the optimal value function f_{π^*} for a task, and then derive the agent's policy values from this function.

2.4.1 *Policy search-based.*

The key point of cooperative multi-agent reinforcement learning is to have independent agents that try to improve their local policies with respect to a common goal [16]. A policy-search based algorithm that is designed to do this is Joint Equilibrium Policy Search (JEPS), which was proposed by Gabel and Riedmiller [16–19]. In this algorithm, each agent i possesses a matrix of policy values π_i for state-action pairs (s, a), and the learning process for approximating the optimal local policy π_i^* is based on updating these values.

Since the effect of taken actions only becomes clear after finishing a complete schedule, all actions that are taken during the scheduling process are stored, and the policy values are updated afterwards. Every time a schedule has been generated, a score is calculated by the objective function. In JEPS, this score does not determine the degree of increase or decrease of policy values, but only whether they will be updated or not. Heuristic $H(r)$ is used to do this, returning 1 if the current global reward equals or exceeds the maximal reward obtained so far, and 0 otherwise [15]. Only if $H(r) = 1$, the policies of the agents will be updated.

Despite the fact that JEPS has proven to be an effective approach for scheduling problems, two important problems arise when attempting to apply it to our problem. First, with an increasing number of actions available to the agents, the size of the state space grows exponentially [16], making JEPS only a proper candidate for application to small problem instances. Second, since each agent possesses its own policy matrix, learned information is difficult to transfer to new problem instances with different agents and actions. This conflicts with our ambition to create an algorithm that is generalizable to many problem instances. Since value function-based methods better meet these requirements, as will be explained in the following section, these are chosen to be used for policy learning.

2.4.2 *Value function-based.*

In value function-based methods, it is attempted to learn the value function of the optimal policy π^*, denoted $f^* = f_{\pi^*}$, rather than directly learning π^*. Once this function is learned, it can be applied as follows. If an agent has to choose its next action, it computes the value $f^*(a)$ for all executable actions, and consequently chooses the action that maximizes this value. The function uses information about the current status of the environment as input, disconnecting the resulting policy from the particular agent and action, and rather considering the more general aspects of the environment.

There are many possible functions that would be able to output policy values, using information about some agent's states and actions as input, considering that the function only has to be able to transform numerical input into numerical output. Zhang and Dietterich [48], who were pioneering in applying RL for scheduling, used an artificial Neural Network (NN) to represent the value function. The first advantage of this technique is its ability to approximate non-linear functions [37]. Furthermore, training the hidden layers of a NN is a way to automatically create features appropriate for a given problem, so that hierarchical representations can be produced without relying exclusively on hand-crafted features [41]. The last reason for considering NNs is their notorious capability of generalization [16].

We will now briefly introduce Neural Networks, in order to explain the way they can be used as a policy value function for the RL model. This introduction is adapted from Abu-Mostafa et al. [1]. To start, Figure 1 shows a graph-representation of a Neural Network (NN), where the input vector $\mathbf{x}$ is transformed into output $h(\mathbf{x})$.

A NN consists of L layers, denoted as $\ell = 1, \ldots, L$. Input layer $\ell = 0$ is usually not considered a layer as it is only meant for feeding input to the NN. When used in our RL algorithm, this input will be a vector of variables describing an agent, action and their relation. Each layer has dimension $d^{(\ell)}$, meaning it has $d^{(\ell)} + 1$ nodes. The extra node represents the bias, which always has an output of 1. In total, the architecture of the network is given by $[d^{(0)}, d^{(1)}, \ldots, d^{(L)}]$. All nodes of two consecutive layers $\ell - 1$ and ℓ are connected by a weight matrix $W^{(\ell)}$, except for the bias node, which does not get any input. These connections are indicated by the colored lines in Figure 1. The output $\mathbf{x}^{(\ell-1)}$ of layer $\ell - 1$ is multiplied by these weights to generate a vector of input

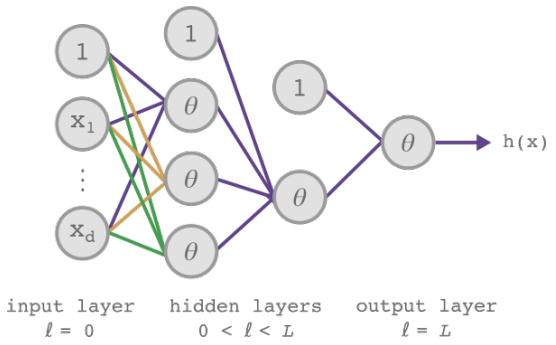


Figure 1: Visualization of a Neural Network, consisting of an input layer, two hidden layers, and an output layer.

signals $\mathbf{s}^{(\ell)}$ for layer ℓ. Each node with an input signal contains an activation function θ, which again transforms the input $\mathbf{s}^{(\ell)}$ into a new output $\mathbf{x}^{(\ell)}$. Common transformation functions are the logistic function [14, 43], which belongs to the family of sigmoid functions, and the rectified linear unit (ReLU) [36]:

$$\text{logistic function:} \qquad \theta(\mathbf{s}) = \frac{1}{1+e^{-\mathbf{s}}}$$
$$\text{ReLU:} \qquad \theta(\mathbf{s}) = max(0, \mathbf{s})$$

We will be using the sigmoid function for our algorithm. Further information on how this decision was made can be found in Appendix D.5.

The output of the Neural Network is computed by forward propagation. Following from what is described above, and by initializing the input layer with input vector $\mathbf{x}$, forward propagation can be performed by executing the following chain:

$$\mathbf{x} = \mathbf{x}^{(0)} \xrightarrow{W^{(1)}} \mathbf{s}^{(1)} \xrightarrow{\theta} \mathbf{x}^{(1)} \dots \xrightarrow{W^{(L)}} \mathbf{s}^{(L)} \xrightarrow{\theta} \mathbf{x}^{(L)} = h(\mathbf{x})$$

The algorithm learns by updating its weight matrices, which is done using gradient descent. To compute the gradient of the in-sample error E_{in}, we need its partial derivative with respect to each weight matrix. Backpropagation allows to efficiently compute these, by writing partial derivatives of layer ℓ using the partial derivatives of layer $\ell + 1$:

$$\frac{\partial \mathsf{e}}{\partial W^{(\ell)}} = \mathbf{x}^{(\ell-1)}(\sigma^{(\ell)})^T,$$

where $\sigma^{(\ell)}$ is the sensitivity for layer ℓ. We only need $\sigma^{(L)}$ to seed the backward process, which can be calculated as follows:

$$\sigma^{(L)} = 2(\mathbf{x}^{(L)} - y)\theta'(\mathbf{s}^{(L)}),$$

where $\mathbf{x}^{(L)}$ is the output from forward propagation, and y is the true output value, which the algorithm attempts to approximate. In our case, $\mathbf{x}^{(L)}$ is the estimated policy value, but the true optimal policy value y is unknown. If NNs are to be used to estimate policy values, another way should be found to compute or properly estimate $\mathbf{x}^{(L)} - y$.

2.5 Exploration and exploitation

A distinction can be made between two opposing aspects of a learning process: exploring and exploiting. Greedy action selection *exploits* current knowledge to maximize immediate reward, by selecting the actions whose estimated reward is greatest. If instead one selects an action without considering their expected reward, this is called *exploring*, because it enables the estimation of a non-greedy action's value. In any specific case, whether it is better to explore or exploit depends in a complex way on the values of the estimates, uncertainties, and the number of remaining steps [41].

A way of balancing exploration and exploitation by applying the ϵ-greedy strategy [4, 41, 48]. This holds that each time an action has to be chosen to execute, a random action will be selected with probability ϵ. In the other cases, the greedy action will be taken. Zhang and Dietterich [46] have chosen to initially set the value of ϵ to 1. After each action, ϵ is decreased by an amount $\Delta\epsilon$ until it reaches a final value of 0.05.

2.6 From scheduling to Reinforcement Learning

In order to practically implement the algorithm, a translation had to be made from the environment that was described by Conway et al. [11], and the RL model that is desired to build. Since this mapping does not necessarily have to be understood before reading the rest of this thesis, but might contribute to a better understanding of the approach, it can be found in Appendix B. This appendix also contains a summary of all relevant notations that have been stated in the current section, and can thus be used as a quick look-up guide for the next sections.

3 METHODOLOGY

In this section we will elaborate on the implementation of the deep RL algorithm. In order to build the algorithm, many design choices had to be made, which are substantiated in this section. After clarifying the assumptions and constraints of the problem as a whole in Section 3.1, the objective function is composed in Section 3.2, which is directly used to compare the algorithm's performance for several settings regarding policy learning, discussed in sections 3.3 and 3.4. The methodology is concluded with Section 3.5, explaining how the performance of the resulting algorithm, all settings being fixed, will be tested. A description of the resources that were used for implementation can be found in Appendix F, followed by the pseudo-code of the final algorithm in Appendix G.

3.1 Assumptions and constraints

As a starting point, a few assumptions and constraints have to be taken into account. These were based on Beke [4], Berkhout [6] and Kyparisis [29].

Assumptions:

- Job preemption is not allowed. Once a job starts to be processed on a unit, this processing must continue without interruption until the operation is completed.
- The processing times of all jobs on all units are deterministic and known.
- The capacities of finished product silos do not severely obstruct the optimal production schedule.
- The agent network is static. All resources are available at time $z = 0$, and remain available (e.g. there are no breakdowns).

Constraints:

- A machine set-up and change-over time between operations exists.
- *Precedence constraints*: Of two consecutive processing operations of the same job j, the processing of the first operation must be finished before the second can start.

$$t_{jiq} + \delta(o_{jiq}) \leq t_{jiq+1}$$

- *No-overlap constraints*: Only one job can be processed at a unit at a time. To all jobs j and k that have to be processed on the same unit u_{iq}, the following applies:

$$t_{jiq} + \delta(o_{jiq}) \leq t_{kiq}$$
$$or$$
$$t_{kiq} + \delta(o_{kiq}) \leq t_{jiq}$$

3.2 Objective function

In order to train the model, an objective function should be composed which can be minimized or maximized. In the particular case of ENGIE, completing jobs before their deadline is reached is of much more importance than minimizing the total processing time of all jobs. In other words, the algorithm should focus on learning how to minimize the summed tardiness T_{sum}, and minimize the makespan C_{max} as a secondary priority.

Our benchmarking procedure, the MILP algorithm, uses an objective function of $100 * T_{sum} + 1 * C_{max}$. However, regardless of whether the weight for T_{sum}, when used by our RL algorithm, is 1, 10 or 100, the optimal T_{sum} is nearly always found within 1000 training epochs. After that, only the makespan is further decreased, which appears to happen most effectively when using a weight of 10 for T_{sum}. As a result, the following objective function was chosen:

$$\text{objective value} = 10 * T_{sum} + 1 * C_{max}$$

The objective value, being the output of the objective function, can be interpreted as a score to indicate the quality of the schedule. In the case of our objective function, a lower score indicates a better schedule, and the score should therefore be minimized.

3.3 Policy learning

In Reinforcement Learning, a policy is learned to approximate an optimal output, corresponding with either a minimal or maximal objective value. As explained in Section 2.4, a value function-based approach is used for policy learning, for which a Neural Network is used as a policy value function. Accordingly, the matrix W is applied as the matrix of connection weights between all layers.

Several aspects of the algorithm can influence its performance. These design choices include deciding on the input variables for the policy value function, setting the architecture and activation function for the Neural Network, and tuning the values for the hyperparameters ϵ and γ. The most important design choices that were made will be discussed in this section.

3.3.1 Hyperparameters ϵ and γ.

The value for ϵ determines the probability that an agent will choose a random action to execute, instead of the action with the highest policy value. As introduced in Section 2.5, Zhang and Dietterich [48] have proposed not to set ϵ to a fixed value, but to decrease it over time. This might be better than starting with a lower ϵ, since the weights of the policy value function are also randomly initialized, and executing completely random values each time step will lead to a larger variety of tried actions than following a randomly generated policy. This strategy was implemented by initializing ϵ with 1 and executing $\epsilon = \epsilon * (1 - \text{decrease})$ each 1000 epochs.

Tests were executed to decide on both the value for ϵ-decrease and γ. The hyperparameter γ represents the learning rate of the algorithm. Its value determines the extent to which the weights of the policy function are updated after each training epoch. The algorithm was trained for 30 minutes, for all combinations of ϵ-decrease $\in \{0.025, 0.05, 0.1\}$ and $\gamma \in \{0.1, 0.2, 0.3, 0.5, 0.7\}$, using a testing instance of $n = 10$, $m = 1$, $l_1 = 6$, $g_1 = 6$. The performance of the algorithm appeared to be best for both 0.025 and 0.1 as values for decrease of ϵ, and worse for 0.05. An ϵ decrease of 0.1 brings the value for ϵ close to 0 very quickly, meaning that it starts acting based on the learned policy much sooner than for a decrease of 0.025. Considering that the values of 0.025 and 0.1 produce similar results, it is decided to set the decrease of ϵ to 0.025 for training, since the training phase should be more about exploring. Exploitation should happen more extensively when loading the pre-trained weights, so the

decrease of 0.1 will be used in the validation phase, where the performance of the final algorithm is tested.

When using the ϵ decrease of 0.025, setting the value of γ to 0.3 leads to the best performance. However, to improve generalizability of the algorithm, it would be best to train the algorithm on multiple instances. This should not be done by simply changing the training instance every several epochs. The reason for this is that the weights of the policy value function are usually initialized with random numbers, which converge towards optimal values by training the algorithm. After training the algorithm for some time, it would therefore not make a difference anymore whether the weights would have been initialized with random or pre-trained weights. Therefore, if one desires to train the weights over a series of multiple instances, the learning rate γ should be decreasing each time the training instance changes, so the effects of updates done in the first part of training will still be visible at the end.

The best value for this γ-decrease depends on how well a certain set of trained weights performs when used as initialization for scheduling some other instance. The algorithm was trained on instances:

$n = 5, l_1 = 3$	$n = 11, l_1 = 6$	all including
$n = 7, l_1 = 3$	$n = 13, l_1 = 6$	$g_1 = 6, m = 1$

The decrease of γ, which is applied by executing $\gamma = \gamma * (1 - \text{decrease})$ each time the instance changes, is varied over γ-decrease $\in \{0.1, 0.3, 0.5, 0.7\}$, starting from $\gamma \in \{0.3, 0.5, 0.7, 0.9\}$. After training these 16 sets of weights, one for each combination of γ and its decrease, these weights were used as initialization weights for testing.

Next, the algorithm was executed using the pre-trained weights for all combinations of γ and γ-decrease, with $\epsilon = 0$. Each combination was tested for 30 seconds for each of the following instances:

$n = 6, l_1 = 4$	$n = 25, l_1 = 11$	all including
$n = 12, l_1 = 5$	$n = 53, l_1 = 19$	$g_1 = 6, m = 1$

The average objective value that was found for all testing instances was calculated, and for each combination of γ and γ-decrease, the found objective values were compared with the mean for each instance. From this comparison, it results that the combination of $\gamma = 0.7$ and γ-decrease $= 0.3$ leads overall to the best results. This is further illustrated in Appendix D.1.

3.3.2 *Input of the Neural Network.*

In order to estimate policy values, the policy value function needs to process input variables that contain information about current states and actions. By executing tests with different configurations of input variables, a choice could be made which configuration is most effective to use. A total of 28 variables were tested, which can be found in Appendix C.

The following eight configurations of input variables were assembled to test as input for the Neural Network. The italicized names will be used to refer to these variable sets, and the numbers refer to the variables in Appendix C.

- *all variables* = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 22, 24, 25, 26, 27, 28}
- *similarity selection* = {1, 2, 5, 6, 9, 11, 12, 13, 16, 17, 18, 19, 20, 22, 24, 25, 26, 27, 28}
- *absolute* = {1, 6, 7, 18, 19, 20, 22, 24, 25, 26, 27}
- *relative* = {2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 28}
- *significant weights* = {18, 19, 20, 21, 23, 22, 24, 26}
- *significant weights extended* = {2, 6, 9, 12, 13, 18, 19, 20, 22, 24, 26, 27}
- *generalizability* = {1, 9, 11, 16, 17, 28}

The first set contains all 28 variables. The sets *similarity selection*, *absolute* and *relative* were respectively composed by removing variables that were highly similar, taking only the variables with absolute values, and taking only the variables with relative values. The remaining sets were composed by first investigating various matrices of trained weights. After training, weights that are related to variables that are considered unimportant by the Neural Network will be assigned a value close to 0. Important values will be assigned either very low (negative) or very high (positive) weight values. The *significant weights (extended)* sets were composed of variables that were assigned such low or high weights by the Neural Network. The *generalizability* set contains the variables that received very similar weights after training on different instances. This suggests that the algorithm would be more generalizable when using these input variables, than it would be when using variables that get completely different weights when trained on different instances.

For all sets of variables, the algorithm was trained on all instances consisting of $n \in \{8, 11, 14\}$, $m = 1$, $l_1 \in \{4, 7\}$, $g_1 = 6$. The value for ϵ was initialized to 1.0, with a decrease of 0.025 each 1000 epochs, and γ was set to 0.3. Although the Neural Network's architecture has to be different for each input set, as a result of differing input sizes, they were kept as similar as possible. All architectures contain three hidden layers, and an approximately equal ratio between layer dimensions. For each instance, the average objective value found at each training epoch, for all input sets, was calculated and compared to the individual values found for each epoch. For all input sets, these differences were then averaged over all instances, of which the result is shown in Figure 2. From this, the conclusion was drawn that the input sets *relative*, *generalizability* and *similarity selection* all perform well after many epochs, but that the *relative* set reaches low objective values the fastest.

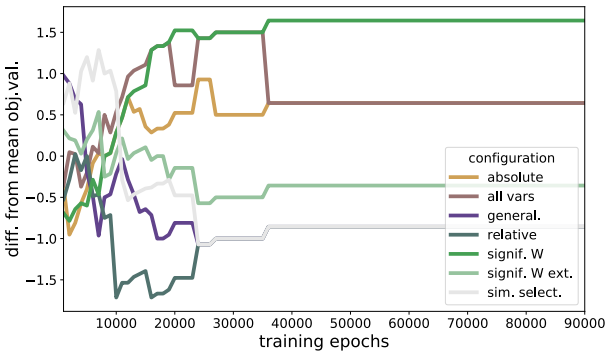


Figure 2: Difference between the objective values and their mean, averaged over six instances, for the algorithm using different sets of input variables for the NN.

However, it is not only important how well a certain configuration of the algorithm performs on the particular instance that it is trained on, but also how well it generalizes to other instances. Therefore, the following experiment was performed for the sets *all variables*, *generalizability* and *relative*. The algorithm's NN weights were initialized with each of the weight matrices that were generated by training the algorithm on instance $n = 14$, $m = 1$, $l_1 = 4$, $g_1 = 6$. Consequently, the algorithm was executed for 1 minute to generate schedules for all instances $n = \{8, 11, 14\}$, $m = 1$, $l_1 = \{4, 7\}$, $g_1 = 6$. There appears to be very little difference in generalization capabilities for the algorithm using *all variables*, the *generalizability* set, or the *relative* set. An illustration of this can be found in Appendix D.2. Since the performance of *relative* has been better overall, this set of variables is chosen to continue using as input for the Neural Network.

3.3.3 *Structure of the Neural Network.*

As explained in Section 2.4.2, the updating of Neural Network weights is done by backpropagation, which is initialized with the sensitivity $\sigma^{(L)}$ to seed the backward process. This value is calculated using $\mathbf{x}^{(L)} - y$, where $\mathbf{x}^{(L)}$ is the output from forward propagation. In our case y would be the optimal policy value for some action, which is unknown.

In Section 2.4.1, heuristic $H(r)$ was introduced, which is a boolean indicating whether the current global reward r equals or exceeds the best reward r_{best} obtained so far. The approach to get a value for y was inspired on this heuristic. At the end of each epoch, a schedule is generated and the reward r for this schedule is determined using the objective function. Subsequently a score is calculated using r and r_{best}, indicating the extent to which r is better or worse than r_{best}. This score is used to derive an estimate of y from $\mathbf{x}^{(L)}$, by assuming that the true policy value should be higher or lower than the predicted value, to the same extent as r was respectively better or worse than r_{best}.

$$\text{score} = \frac{r_{best} - r}{min(r_{best}, r)}$$

$$y = \mathbf{x}^{(L)} + (score * \mathbf{x}^{(L)})$$

For obtaining an even better estimate of y, it would be best to replace r_{best} with the actual optimal objective value, which can be found by the MILP algorithm, but only for small problem instances. Since our algorithm is designed to be generalizable, it is possible to train the weights of the Neural Network on small instances, and extend their use to larger instances. It is therefore decided to replace r_{best} by the optimal objective value found by the MILP algorithm during the training of weights, and to keep using r_{best} when the RL algorithm applied to larger instances, after loading the pre-trained weights. For further illustration, a performance comparison between the two can be found in Appendix D.4.

Finally, we will have to decide on the architecture of the Neural Network, meaning the number of layers L that the network consists of and the dimension $d^{(\ell)}$ of each layer ℓ. The dimension $d^{(0)}$ of the input layer is always equal to the number of variables that is used as input for the NN, so in our case the number of variables in the *relative* set, which is 15. The dimension $d^{(L)}$ of the output layer depends on the number of values that is desired to get as a result from forward propagating through the NN. In our case, the desired output is an estimated policy value, which is a scalar value, so $d^{(L)} = 1$. Between the input and output layer, hidden layers can be added to make the algorithm able to represent more complex, non-linear functions.

Tests were performed on nine instances, comparing four network architectures. For each instance, the average best found objective value from all architectures was calculated for each training epoch, which was compared to each individual objective value found using a particular architecture. The difference between each of these best found objective values and the average for all architectures is shown in Figure 3. The layer dimensions are of the form $[d^{(0)}, d^{(1)}, \ldots, d^{(L)}]$. The architecture $[15, 11, 6, 1]$ appears to perform best, and thus will be used for the final algorithm.

3.4 The action of doing nothing

Since 'executing' the action of doing nothing, instead of starting to process a job, might be beneficial in some circumstances, this option was originally implemented in the algorithm. In policy search-based approaches this action receives a value just like all job-processing actions, which can also be updated as such. In value function-based approaches, however, information about the current situation is used to calculate policy values. As can be observed in Section 3.3.2, where these inputs were discussed, these values mainly contain information about processing times, blocking time, and

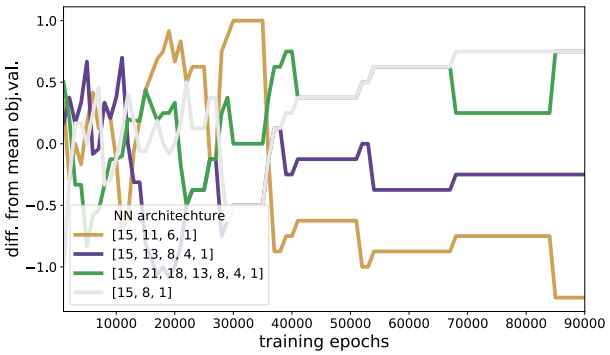


Figure 3: Comparison of the objective values found using different NN architectures.

expected tardiness. All of these values are always 0 for the action of doing nothing, which gives the NN little information about the actual results of executing this action. Therefore, the possibility to do nothing, while being in a state where there are available jobs to be processed, is removed as an executable option.

3.5 Validation

Now all parameters are set, the algorithm's performance has to be evaluated. This is done using the MILP algorithm combined with a commercial solver, which was used by Berkhout [6], as a benchmark. Both the RL algorithm and the MILP are executed for a timespan of 30 seconds, 1 minute, 3 minutes and 30 minutes, in order to compare their performance measured by the objective function.

The RL algorithm was tested using both pre-trained and random weights as initialization for the Neural Network. The weights were pre-trained on multiple instances, making use of the decreasing value for γ. This value was initially set to 0.7, and decreased with $0.3 * \gamma$ at each instance change. The value for ϵ was initialized with 1.0 and decreased with $0.025 * \epsilon$ each 1000 training epochs. The following instances were used for training on the JSSP-UMP:

$n = 5, l_1 = 3$	$n = 11, l_1 = 6$	all including
$n = 7, l_1 = 3$	$n = 13, l_1 = 6$	$g_1 = 6, m = 1$

The training of weights for the FFSSP was done using the same values for n and l_v, only setting g_v to 4 for each v, and varying $m \in \{2, 5\}$.

When testing the algorithm's performance after loading these weights, the value for ϵ is varied between 0, 0.2, and 0.5, combined with an ϵ-decrease of $0.1 * \epsilon$ each 1000 epochs. This is done in order to test the (in)dispensability of instance-specific weight tuning after training them on other instances. The performance of the RL algorithm and the MILP are compared for the following seven problem instances, scaling from a very tiny up to a massive size. For solving the JSSP-UMP, all instances include $m = 1$ and $g_1 = 6$. For the FFSSP this changes to $g_v = 4$ and $m \in \{2, 3, 5\}$.

$n = 4, l_1 = 2,$	$n = 32, l_1 = 13,$	$n = 64, l_1 = 21,$
$n = 8, l_1 = 4,$	$n = 47, l_1 = 17,$	$n = 81, l_1 = 26.$
$n = 18, l_1 = 8,$		

4 RESULTS

In this section, the objective values resulting from executing both the deep RL algorithm and the MILP will be compared, for all instances and settings that were mentioned in Section 3.5.

Since our aim has been to create an algorithm that generalizes well over different instances, it is expected that the algorithm performs better when initializing its Neural Network weights with pre-trained weights, instead of with random values. This has, at least to some extent, proven to be true, which is illustrated in Appendix D.3. All performance comparisons to be presented will therefore only consider the algorithm using pre-trained weights.

Figure 4 compares schedules found by the RL and MILP algorithms for all testing instances and computing times, using $\epsilon = 0.0$. Figures for $\epsilon = 0.2$ and $\epsilon = 0.5$ can be found in Appendix E.1. The values on the y-axis represent the relative differences between objective values. For each unique combination of instances and computing times, these are calculated as follows:

$$OR = median(\text{obj.val.s}_{RL})$$
$$OM = median(\text{obj.val.s}_{MILP})$$
$$\text{diff} = \frac{OR - OM}{max(OR, OM)}$$

This difference will approximate -1.0 when the median of objective values when using RL is much smaller than those when using the MILP, and be close to 1.0 if it is much larger.

For small instances, the MILP algorithm performs better irrespective of computation time. As the instance size increases, the RL algorithm starts performing better than the MILP. However, as the computing time increases a lot, this effect decreases and the MILP also starts producing lower objective values, but this only goes for computing times that might not be realistic nor economical to be spending in real-life situations.

When comparing the performance of the *load* RL algorithm using different values for ϵ, it appears to perform notably better for higher ϵ values. This suggests that, although the algorithm has proven to be generalizable to some extent, it still produces much better results if a little instance-specific exploring and weight tuning is also allowed.

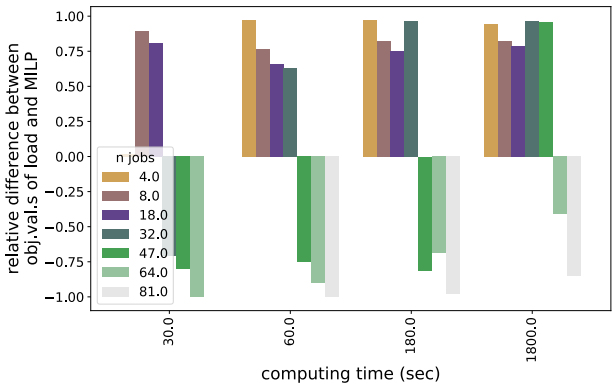


Figure 4: Comparison of the objective values found using the deep RL algorithm and the MILP for different limitations of computing time, executed with $\epsilon = 0.0$.

A performance comparison between the RL and MILP algorithms when applied to the FFSSP can be found in Appendix E.2. For the FFSSP it can be concluded that for almost all computing times, our RL algorithm performs much worse than the MILP algorithm. This difference is less for either a small number of work stations, or a very short computing time. In general, the algorithm does not perform very well for multiple work stations.

5 DISCUSSION

The application of Reinforcement Learning for solving scheduling problems is still in its infancy, especially when combined with using a Neural Network as a policy value function. Several papers have been published on this topic, but due to the large variety of scheduling problem types, ways of representing the environment, and techniques to approximate optimal policies, there is still much to discover in this field.

In this thesis, we have been able to verify techniques that have been proposed in published literature, such as the use of a Neural Network as a policy value function, a decreasing ϵ-value, and the representation of the JSSP-UMP by a factored decentralized MDP with changing action sets. New aspects have also been introduced, such as the use of optimal solutions found by a MILP algorithm to guide the training of weights for our policy value function, and training on multiple instances by decreasing the value of γ.

Section 2.3.1, containing related work on Multi-Agent Learning, states that we have chosen to let the system consist of multiple agents, in order to allow for decentralization of decision-making. It has not been the aim to implement this in the current study, but this would be a proper start of a follow-up study.

Besides, in this thesis only predictive scheduling has been investigated. The current implementation of the algorithm does however already consider, at each time step separately, which resources are currently idle and which jobs are waiting to be processed on them. It is therefore expected that the algorithm is very suitable to generate real-time reactive schedules as well. A future study may implement and evaluate its performance in such an environment.

Moreover, Section 2.3.1 contains the statement that using multiple agents comes with the ability of distributing the computational load over a number of entities, using parallel computing. However, for the current study this is rather cumbersome, since all agents have the the exact same set of possibly executable actions at their disposal, and they all use the same policy value function. If they would also choose actions in parallel, each decision point would come with a high risk of multiple agents deciding to execute the same action. This might be more convenient to implement for the proposed real-time version of the algorithm, assuming that it would be applied to an environment with a larger variety of agents and jobs.

6 CONCLUSION

This thesis has been written in order to answer the question how generalizable Multi-Agent Reinforcement Learning can be applied to approximate optimal solutions for Job Shop Scheduling Problems with unrelated machines in parallel. Generalizability was achieved by using a Neural Network as a policy value function for the RL algorithm. This has made it possible to train the algorithm on small instances and use the trained weights to generate schedules for larger instances. Mixed-Integer Linear Programming, which is known for being able to find optimal solutions for small problem instances, has appeared to largely outperform the designed algorithm for such small instances. However, starting from a certain threshold instance size, our RL algorithm becomes increasingly better with increasing instance size. It can therefore be concluded that optimal solutions are approximated, at least to some extent, for Job Shop Scheduling Problems with unrelated machines in parallel.

REFERENCES

[1] Y.S. Abu-Moustafa, M. Magdon-Ismail, and H.T. Lin. 2012. *Learning from data*. Vol. 4. AMLBook, New York, USA.

[2] M. Basseur, F. Seynhaeve, and E.G. Talbi. 2002. Design of multi-objective evolutionary algorithms: Application to the flow-shop scheduling problem. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC'02)*, Vol. 2. IEEE, 1151–1156.

[3] D. Behnke and M.J. Geiger. 2012. *Test instances for the flexible job shop scheduling problem with work centers*. Research Paper. Helmut-Schmidt-Universität, Hamburg, Germany.

[4] T. Beke. 2013. *Multi-Agent Reinforcement Learning in a flexible Job Shop Environment: The VCST Case*. diploma thesis. Gent Universiteit, Gent, Belgium.

[5] J. Berkhout. 2019. Lecture 9: Advanced modeling. Slideshow corresponding to the 9th lecture of the course "Statistics, Simulation & Optimization", of the Master Information Studies at the University of Amsterdam.

[6] J. Berkhout, E. Pauwels, R. van der Mei, J. Stolze, and S. Broersen. 2020. Short-term production scheduling with non-triangular sequence-dependent setup times and shifting production bottlenecks. *International Journal of Production Research* (2020), 1–25.

[7] D.P. Bovet and P. Crescenzi. 1994. Introduction to the Theory of Complexity.

[8] P. Brandimarte. 1993. Routing and Scheduling in a Flexible Job Shop by Tabu Search. *Annals of Operations research* 41, 3 (September 1993), 157–183.

[9] P. Burke and P. Prosser. 1991. A distributed asynchronous system for predictive and reactive scheduling. *Artificial Intelligence in Engineering* 6, 3 (1991), 106–124.

[10] J.B. Chambers and J.W. Barnes. 1996. *Tabu Search for the Flexible-Routing Job Shop Problem*. Technical Report Series ORP9610. The University of Texas, Austin, TX. Graduate Program in Operations Research and Industrial Engineering.

[11] R.W. Conway, W.L. Maxwell, and L.W. Miller. 1967. *Theory of Scheduling*. Addison-Wesley Publishing Company, USA.

[12] S. Dauzère-Pérès and J. Paulli. 1997. An Integrated Approach for Modeling and Solving the General Multiprocessor Job-Shop Scheduling Problem using Tabu Search. *Annals of Operations Research* 70 (April 1997), 281–306.

[13] A. Duenas and D. Petrovic. 2008. An approach to predictive-reactive scheduling of parallel machines subject to disruptions. *Annals of Operations Research* 159, 1 (2008), 65–82.

[14] S. Elfwing, E. Uchibe, and K. Doya. 2018. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks* 107 (2018), 3–11.

[15] N. Fulda and D. Ventura. 2004. Incremental Policy Learning: An Equilibrium Selection Algorithm for Reinforcement Learning Agents with Common Interests. In *Proceedings of the 2004 IEEE International Joint Conference on Neural Networks (IJCNN)*. IEEE Press, Budapest, Hungary.

[16] T. Gabel. 2009. *Multi-agent reinforcement learning approaches for distributed job-shop scheduling problems*. Ph.D. Dissertation. Universität Osnabrück.

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

[18] T. Gabel and M. Riedmiller. 2008. Joint Equilibrium Policy Search for Multi-Agent Scheduling Problems. In *Proceedings of the 6th Conference on Multiagent System Technologies (MATES 2008)*. Springer, Kaiserslautern, Germany, 61–72.

[19] T. Gabel and M. Riedmiller. 2008. Reinforcement Learning for DEC-MDPs with Changing Action Sets and Partially Ordered Dependencies. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008)*. IFAAMAS, Estoril, Portugal, 1333–1336.

[20] J. Gao, M. Gen, L. Sun, and X. Zhao. 2007. A Hybrid of Genetic Algorithm and Bottleneck Shifting for Multiobjective Flexible Job Shop Scheduling Problems. *Computers & Industrial Engineering* 51, 1 (2007), 149–162.

[21] M. Gen, W. Zhang, L. Lin, and Y. Yun. 2017. Recent advances in hybrid evolutionary algorithms for multiobjective manufacturing scheduling. *Computers & Industrial Engineering* 112 (2017), 616–633.

[22] C.P. Gomes. 2000. Artificial intelligence and operations research: challenges and opportunities in planning and scheduling. *The Knowledge Engineering Review* 15, 1 (2000), 1–10.

[23] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.R. Kan. 1979. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics* 5 (1979), 287–326.

[24] L. Hunsberger and B.J. Grosz. 2000. A combinatorial auction for collaborative planning. In *Proceedings fourth international conference on multiagent systems*. IEEE, 151–158.

[25] J. Hurink, B. Jurisch, and M. Thole. 1994. Tabu Search for the Job-Shop Scheduling Problem with Multi-Purpose Machines. *OR Spektrum* 15, 4 (December 1994), 205–215.

[26] Y.M. Jiménez. 2012. *A generic multi-agent reinforcement learning approach for scheduling problems*. Ph.D. Dissertation. Vrije Universiteit Brussel.

[27] I. Kacem, S. Hammadi, and P. Borne. 2002. Approach by Localization and Multiobjective Evolutionary Optimization for Flexible Job-Shop Scheduling Problems. *IEEE Transactions on Systems, Man and Cybernetics* 32, 1 (2002), 1–13.

[28] T. Kemmerich and H. Kleine Büning. 2011. On the Power of Global Reward Signals in Reinforcement Learning. In *Multiagent System Technologies*, F. Klügl and S. Ossowski (Eds.). Springer, Berlin, Heidelberg, 53–64.

[29] G.J. Kyparisis and C. Koulamas. 2006. Flexible flow shop scheduling with uniform parallel machines. *European Journal of Operational Research* 168, 3 (2006), 985–997.

[30] S. Mahadevan, N. Marchalleck, T.K. Das, and A. Gosavi. 1997. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *Machine Learning: International workshop then conference*. Morgan Kaufmann Publishers, Inc., 202–210.

[31] B. Naderi, S. Gohari, and M. Yazdani. 2014. Hybrid flexible flowshop problems: Models and solution methods. *Applied Mathematical Modelling* 38, 24 (2014), 5767–5780.

[32] F. Pezzella, G. Morganti, and G. Ciaschetti. 2008. A Genetic Algorithm for the Flexible Job-Shop Scheduling Problem. *Computers & Operations Research* 35, 10 (2008), 3202–3212.

[33] M. Pinedo. 2012. *Scheduling: Theory, Algorithms, and Systems*. Vol. 5. Springer, New York.

[34] M.L. Puterman. 1990. Markov decision processes. *Handbooks in operations research and management science* 2 (1990), 331–434.

[35] M. Puterman. 2005. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Interscience, USA.

[36] P. Ramachandran, B. Zoph, and Q.V. Le. 2018. Searching for activation functions. *ArXiv* (2018).

[37] M. Riedmiller. 2005. Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method. In *Machine Learning: ECML 2005, 16th European Conference on Machine Learning*. Springer, Porto, Portugal, 317–328.

[38] S.J. Russell and P. Norvig. 2010. *Artificial Intelligence: a Modern Approach* (third ed.). Pearson Education, Inc., New Jersey.

[39] T. Sawik. 2012. Batch versus cyclic scheduling of flexible flow shops by mixed-integer programming. *International Journal of Production Research* 50, 18 (2012), 5017–5034.

[40] C.R. Scrich, V.A. Armentano, and M. Laguna. 2004. Tardiness minimization in a flexible job shop: A tabu search approach. *Journal of Intelligent Manufacturing* 15, 1 (2004), 103–115.

[41] R.S. Sutton and A.G. Barto. 2017. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press, Cambridge, Massachusetts.

[42] E. A. Toso, R. Morabito, and A.R. Clark. 2009. Lot sizing and sequencing optimisation at an animal-feed plant. *Computers & Industrial Engineering* 57, 3 (2009), 813–821.

[43] R. Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning* 8, 4 (1992), 229–256.

[44] P. Xuan, V. Lesser, and S. Zilberstein. 2001. Communication Decisions in Multi-Agent Cooperation: Model and Experiments. In *Proceedings of the 5th International Conference on Autonomous Agents (Agents 2001)*. ACM Press, Montreal, Canada, 616–623.

[45] B. Yang and J. Geunes. 2008. Predictive–reactive scheduling on a single resource with uncertain future jobs. *European Journal of Operational Research* 189, 3 (2008), 1267–1283.

[46] Z. Zang, W. Wang, Y. Song, L. Lu, W. Li, Y. Wang, and Y. Zhao. 2019. Hybrid Deep Neural Network Scheduler for Job-Shop Problem Based on Convolution Two-Dimensional Transformation. *Computational Intelligence and Neuroscience* (2019), 1–19.

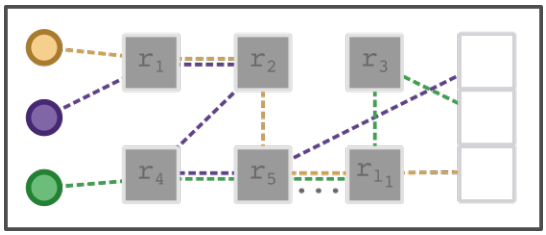
[47] D. Zeng and K. Sycara. 1995. Using Case-Based Reasoning as a Reinforcement Learning Framework for Optimization with Changing Criteria. In *Proceedings of the 7th International Conference on Tools with Artificial Intelligence (ICTAI 1995)*. IEEE Press, Takamatsu, Japan, 56–62.

[48] W. Zhang and T.G. Dietterich. 1995. A reinforcement learning approach to job-shop scheduling. *IJCAI* 95 (August 1995), 1114–1120.

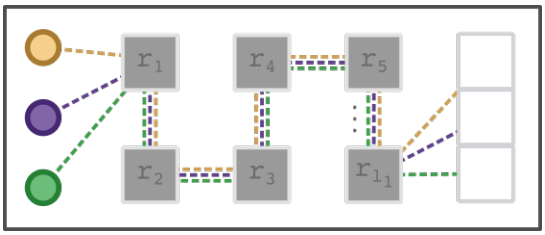
APPENDIX

A SCHEDULING PROBLEM TYPES

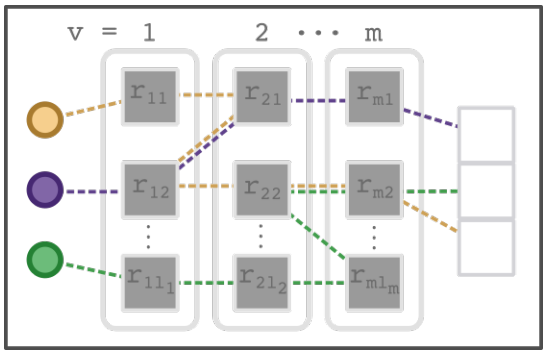
Below the four scheduling problem types that are discussed in this paper are visualized. The colored dots represent jobs. The gray squares (A-I) are resources, which are grouped in work stations (1-3) for two problem types. The white squares represent the end-stage of production for the jobs. For each job, the dotted line in the corresponding color shows in which order of resources the job is processed.



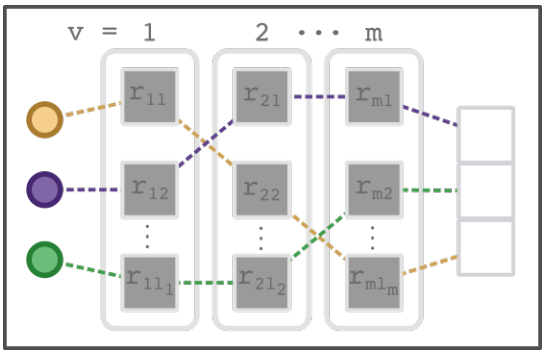
(a) Job Shop Scheduling Problem (JSSP): The jobs can visit the resources in any order, as long as this order is pre-defined.



(b) Flow Shop Scheduling Problem (FSSP): All jobs have to be processed on each of the resources, and the order of resources has to be the same for all jobs.



(c) Flexible Job Shop Scheduling Problem (FJSSP): The resources are ordered in work stations. The jobs can visit the work stations in any order, as long as this order is pre-defined. Within a work station, any resource can be chosen.



(d) Flexible Flow Shop Scheduling Problem (FFSSP): The resources are ordered in work stations. All jobs have to be processed on each of the work stations, and the order of work stations has to be the same for all jobs. Within a work station, any resource can be chosen.

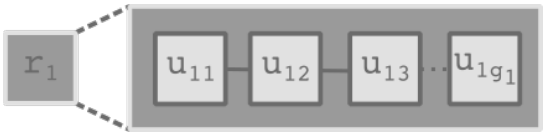


Figure 5: For the ENGIE problem in particular, all resources contain a flow shop of units. If a job enters the resource, it has to be processed on all subsequent units of that resource, meaning it can only enter at the first unit and leave at the last. New jobs can already enter the resource if only the first unit is available.

B FRAMEWORK AND NOTATION

In this section, the notational framework for this thesis be laid out. First, the notation for the Job Shop Scheduling Problem with unrelated machines in parallel (JSSP-UMP) and Flexible Flow Shop Scheduling Problem (FFSSP) is introduced. Then this will be combined with a notation for the Markov Decision Process (MDP) for Reinforcement Learning (RL).

B.1 JSSP-UMP and FFSSP

The notation for the FFSSP was adapted from Kyparisis [29]. In a JSSP-UMP, there is only one "work station", so variables related to the number of work stations are therefore only relevant for the FFSSP, and can be left out when referring to the JSSP.

m = number of work stations (=1 for JSSP-UMP)
$v = 1 \dots m$, index of work station (always 1 for JSSP-UMP)
l_v = number of resources at work station w_v
$i = 1 \dots l_v$, index of resource at work station w_v
g_v = number of units inside each resource of work station w_v
$q = 1 \dots g_v$, index of unit at resource r_{vi}
$w_v = \{r_{v1}, \dots, r_{vl_v}\}$, work station
$r_{vi} = \{u_{vi1}, \dots, r_{vig_v}\}$, ith resource at work station w_v
u_{viq} = qth unit at resource r_{vi}

JSSP-UMP		**FFSSP**		**description**
$\mathcal{J}$	$= \{1, \dots, n\}$	$\mathcal{J}$	$= \{1, \dots, n\}$	set of all jobs
$\mathcal{W}$	-	$\mathcal{W}$	$= \{w_1, \dots, w_m\}$	set of work stations
$\mathcal{R}$	$= \{r_1, \dots, r_{l_1}\}$	$\mathcal{R}$	$= \{r_{11}, \dots, r_{ml_m}\}$	set of resources for all work stations
$\mathcal{U}$	$= \{u_{11}, \dots, u_{l_1 g_1}\}$	$\mathcal{U}$	$= \{u_{111}, \dots, u_{ml_m g_m}\}$	set of units for all resources
O_j	$= (o_j)$	O_j	$= (o_{j1}, \dots, o_{jm})$	chain of operations for job j per work station
O_j	$= (o_{ji1}, \dots, o_{jig_1})$	O_j	$= (o_{j1i1}, \dots, o_{jmig_m})$	chain of operations for job j per unit

$\delta(o_{jiq})$	= processing time of job j on unit u_{iq}	C_{max}	$= max(c_1, \cdots, c_n)$, makespan
$\tau(o_{ji})$	= processing time of job j on resource r_i ($\Sigma(o_{jiq}, \dots, o_{jig_1})$)	B_j	= release date of job j
t_{jiq}	= starting time of the processing of job j on unit u_{iq}	D_j	= due date of job j
t_{ji}	= starting time of the processing of job j on resource r_i (= t_{ji1})	T_j	= tardiness of job j
c_{jq}	= completion time of the processing of job j on unit q	T_{sum}	= summed tardiness of all jobs
c_j	= completion time of the processing of job j		

B.2 JSSP-UMP as DEC-MDP

It will now be shown how the components of a JSSP can be employed to construct a corresponding DEC-MDP. This translation from scheduling problems to RL is based on the notations and algorithms created by Gabel and Riedmiller [16, 19], which were also used by Jiménez [26].

- To each of the resources r_i, one agent i is associated that observes the local state at its resource and controls its behavior. Consequently, we have as many agents as resources in the JSSP ($|A_g| = |\mathcal{R}|$).
- Actions correspond to starting the processing of job operations. A local action of agent i reflects the decision to further process one particular job out of the set $A_i \subseteq \mathcal{J}$ of jobs currently waiting at r_i.
- As a result of actions denoting the further processing of waiting jobs, the set of actions available to an agent varies over time. While $A_i \subseteq \mathcal{A}_i^r$ denotes the currently available actions for agent i, $\mathcal{A}_i^r$ is the set of all potentially executable actions for this agent. Furthermore, the local state s_i of agent i is fully described by the changing set of jobs currently waiting at resource r_i for further processing. Thus, $s_i = A_i$ and $S_i = \mathcal{P}(\mathcal{A}_i^r)$, where $\mathcal{P}$ denotes the powerset.

- After having finished an operation of a job, this job is transferred to another resource, which corresponds to influencing another agent's local state by extending that agent's action set. Additionally, if a job has to be processed on only one of multiple parallel resources, starting the processing on one of those resources means that the job has to be removed from the other resources' states.

DEC-MDP with changing action sets		**JSSP-UMP**
A_g	$= \{1, \ldots, l_m\}$, set of all agents	resources $\mathcal{R}$
$\mathcal{A}_i$	set of potentially executable actions for agent i	the set of jobs that can be processed on resource r_i + doing nothing
$\mathcal{A}_i^r$	set of potentially executable actions for agent i without the action of doing nothing a_0	the set of jobs that is possible to be processed on resource r_i
A_i	set of currently selectable actions for agent i	the set of jobs waiting to be processed on resource r_i + the action of doing nothing
$\mathcal{A}$	$= \mathcal{A}_1 \times \cdots \times \mathcal{A}_{l_m}$, all possibly executable actions	all unprocessed jobs $j \in \mathcal{J}$
a_i	action executed by agent i at a particular timestep	a job $j \in \mathcal{J}$
$\mathcal{S}_i$	all possible states for agent i: $\mathcal{P}(\mathcal{A}_i \backslash \{a_0\})$, where a_0 is the action of doing nothing, and $\mathcal{P}$ denotes the powerset	all possible combinations of jobs that can be processed on resource r_i
$\mathcal{S}$	$= \mathcal{S}_1 \times \cdots \times \mathcal{S}_{l_m}$, all possible agent states	$\mathcal{P}(\mathcal{J})$, all possible combinations of jobs waiting to be processed
s_i	$= A_i \backslash \{a_0\}$, current state of agent i	the set of jobs currently waiting to be processed on a resource r_i

B.3 Reinforcement Learning

The notations for RL were adapted from Sutton [41] and Gabel and Riedmiller [19].

Reward function $R(s, a, s')$ gives the reward for executing a in s and going to s'

$\omega = (\omega_1, \ldots, \omega_m) \in \Omega$ denotes a joint observation with ω_i as the observation for agent i

$\pi = \langle \pi_1, \ldots, \pi_m \rangle$ is the joint policy of local policies, where π^* is the optimal policy

f_π is the policy value function, f_{π^*} being the optimal function, which estimates values of π

$z = (1, \ldots, Z)$ is the serie of timesteps the RL algorithm needs to complete the schedule

Learning rate $\{\gamma \in \mathbb{R} | 0 < \gamma \leq 1\}$ is a trade-off between learning speed and obtaining a good policy: high values decrease learning time, small values decrease the risk of the algorithm converging prematurely

$\{\epsilon \in \mathbb{R} | 0 \leq \epsilon \leq 1\}$ probability of taking a random action in an ϵ-greedy policy, where 1 = exploring, 0 = exploiting

B.4 Neural Networks

L = number of layers, excluding the input layer

$\ell = 1, \ldots, L$, index of layer ℓ

$\mathbf{s}^{(\ell)}$ = input vector that is fed into the nodes of layer ℓ

$\mathbf{x}^{(\ell)}$ = output vector of layer ℓ

$d^{(\ell)}$ = dimension of layer ℓ

$W^{(\ell)}$ = weight matrix, transforming output $\mathbf{x}^{(\ell-1)}$ into input $\mathbf{s}^{(\ell)}$

θ = activation function, transforming input $\mathbf{s}^{(\ell)}$ into output $\mathbf{x}^{(\ell)}$

C NEURAL NETWORK INPUT VARIABLES

This appendix section contains the complete set of variables that were used as input for the policy value function. Different subsets of this collection were tested on their resulting algorithm performance.

(1) processing time of job j on resource i, on a scale from minimal to maximal processing time of j on all resources
(2) number of standard deviations that the processing time of job j on resource i is away from the mean processing time of job j on all resources (positive or negative)
(3) number of standard deviations that the processing time of job j on resource i is away from the mean processing time of job j on all resources, taking the time into account that other resources will still be unavailable
(4) number of standard deviations that the processing time of job j on resource i is away from the mean processing time of job j on all resources, taking the blocking on other resources into account
(5) number of standard deviations that the processing time of job j on resource i is away from the mean processing time of job j on all resources, taking the time into account that other resources will still be unavailable, and taking the blocking on other resources into account
(6) processing time of job j on resource i, on a scale from minimal to maximal processing time of all jobs on resource i
(7) processing time of job j on resource i, on a scale from minimal to maximal processing time of all jobs on resource i, only considering the jobs that still have to be processed
(8) number of standard deviations that the processing time of job j on resource i is away from the mean processing time of all jobs on resource i (positive or negative)
(9) number of standard deviations that the processing time of job j on resource i is away from the mean processing time of all jobs on resource i (positive or negative), only considering the jobs that still have to be processed
(10) number of standard deviations that the processing time of job j on resource i is away from the mean processing time of all jobs on resource i (positive or negative), taking the blocking of all these jobs on resource i into account
(11) number of standard deviations that the processing time of job j on resource i is away from the mean processing time of all jobs on resource i (positive or negative), only considering the jobs that still have to be processed, and taking the blocking of all these jobs on resource i into account
(12) number of standard deviations that the blocking that will result from scheduling job j on resource i is away from the mean blocking that will result of scheduling job j on all resources
(13) number of standard deviations that the summed idle time of units on resource i, that will result from scheduling job j on resource i, is away from the summed idle time of units on all other resources, that would result from scheduling job j on that resource
(14) number of standard deviations that the blocking that will result from scheduling job j on resource i is away from the mean blocking that will result of scheduling all jobs on resource i
(15) number of standard deviations that the summed idle time of units on resource i, that will result from scheduling job j on resource i, is away from the summed idle time of units on resource i, that would result from scheduling all jobs on resource i
(16) number of standard deviations that the blocking that will result from scheduling job j on resource i is away from the mean blocking that will result of scheduling all jobs on resource i, only considering the jobs that still have to be processed
(17) number of standard deviations that the summed idle time of units on resource i, that will result from scheduling job j on resource i, is away from the summed idle time of units on resource i, that would result from scheduling all jobs on resource i, only considering the jobs that still have to be processed
(18) the blocking that will result from scheduling job j on resource i
(19) the summed idle time of units on resource i that will result from scheduling job j on resource i
(20) the time that the first unit of resource i will be occupied as a result from scheduling job j on resource i
(21) the average blocking time that will be caused in the future by the processing of job j on resource i
(22) the median blocking time that will be caused in the future by the processing of job j on resource i
(23) the average summed idle time of units on resource i that will be caused in the future by the processing of job j on resource i
(24) the median summed idle time of units on resource i that will be caused in the future by the processing of job j on resource i
(25) how many jobs are already being processed on resource i
(26) indicating the expected tardiness of job j if processing on resource i starts now
(27) time until due date of job j, relative to the complete timespan from $z = 0$ to due date of job j
(28) number of standard deviations that the due date of job j is away from the due dates of all jobs that still have to be processed

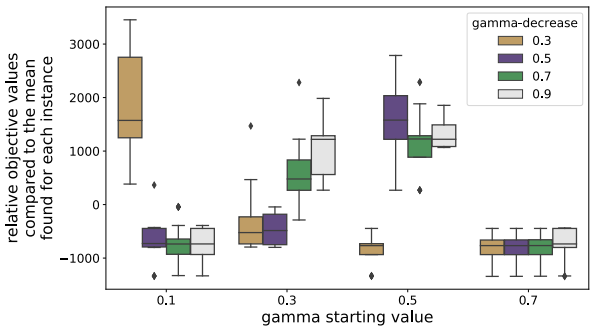


Figure 6: Comparison of values for γ and its decrease value, relative to the mean of each instance. The weights were trained on four instances, and tested on four different instances.

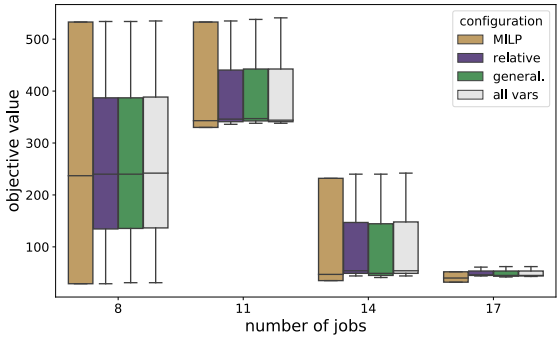


Figure 7: Result of training the algorithm for 1 minute on six different instances, using three sets of input variables for the policy value function. The algorithm was initialized with pre-trained weights.

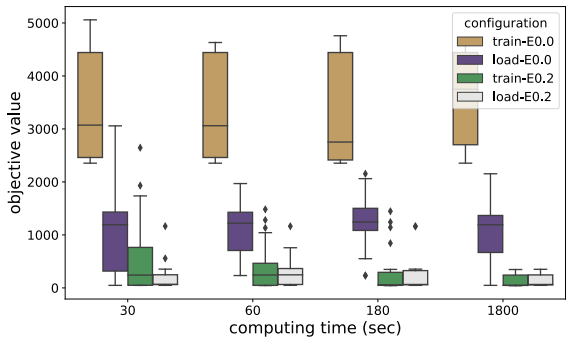


Figure 8: Comparison between the objective values found by executing the algorithm with pre-trained NN weights (*load*) and with randomly initialized NN weights (*train*), for $\epsilon = 0.0$ and $\epsilon = 0.2$.

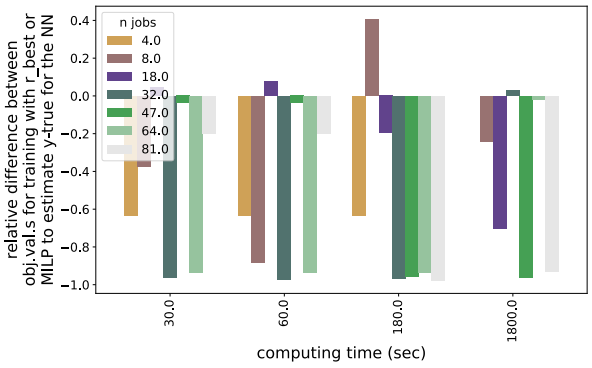


Figure 9: Comparison between the objective values found when using r_{best} to estimate the "true output" for the Neural Network, and when using the optimal value found by the MILP algorithm. Both use $\epsilon = 0$. Positive scores indicate training with r_{best} performed better, negative scores indicate training with the optimal objective value performed better.

D FIGURES SUPPORTING DESIGN CHOICES

D.1 Decrease of γ

Figure 6 illustrates a comparison between tested values for γ and its decrease value, for training the algorithm on multiple instances. For all testing instances, the mean objective value that was found for all tests was calculated. Then, for each combination of γ and γ-decrease, the mean was subtracted from the found objective value for all instances. The figure's y-axis represents the difference between the two.

D.2 Input of the Neural Network

Figure 7 and Table 1 contain a comparison of the generalizability of different sets of NN input variables. They that there is little difference between the generalization capabilities of the three sets.

	8	11	14	17
MILP	237.0	343.0	47.0	40.0
relative	240.0	346.0	54.0	45.0
general.	240.0	347.0	49.0	45.0
all vars	242.0	344.0	54.0	45.0

Table 1: Median of objective values over several instances, comparing between input variable sets for the Neural Network. This table represents the same data as Figure 7.

comp. time	30		60		180		1800	
ϵ	**0.0**	**0.2**	**0.0**	**0.2**	**0.0**	**0.2**	**0.0**	**0.2**
train	3072.0	242.0	3060.9	244.0	2752.7	59.0	3749.7	49.5
load	1191.5	67.0	1221.9	246.5	1244.0	65.0	1191.5	61.5

Table 2: Median of objective values over all testing instances. This table represents the same data as is visualized in Figure 8.

D.3 Comparison 'load' and 'train'

Figure 8 and Table 2 contain a comparison of tests where the RL algorithm was executed using the pre-trained or random weights as initialization for the Neural Network. When using $\epsilon = 0.0$, the objective value is consistently much lower when using pre-trained weights instead of randomly initialized weights. This indicates that the algorithm is at least generalizable to some extent, since the weights were trained on instances that are different from the ones it was tested on. However, when $\epsilon = 0.2$, the results produced with pre-trained weights are only better if very little computing time is allowed, and slightly worse for longer computing times.

D.4 Use optimal objective value for training

Figure 9 shows a comparison between the performance of the RL algorithm being trained in two different ways. The "true" output value of the Neural Network was either estimated using r_{best}, the best objective found by the algorithm so far, and using the optimal objective value found by the MILP. Since the former was subtracted from the latter, the figure illustrates that using the optimal objective value leads to a better performance.

D.5 Choosing the Neural Network activation function

A Neural Network needs an activation function, in order to transform the input of each node into an output. The algorithm was trained for four instances, using three different network architectures. For all epochs of each test, the difference between the objective values obtained by using the ReLU and sigmoid activation function was calculated, and for each epoch these differences were averaged over all four instances. The result is shown in Figure 10. Since the values obtained by using ReLU were subtracted from the values obtained by using the sigmoid function, the figure illustrates that objective values are generally higher when using ReLU. It is therefore decided to use the sigmoid as an activation function for the nodes of the NN.

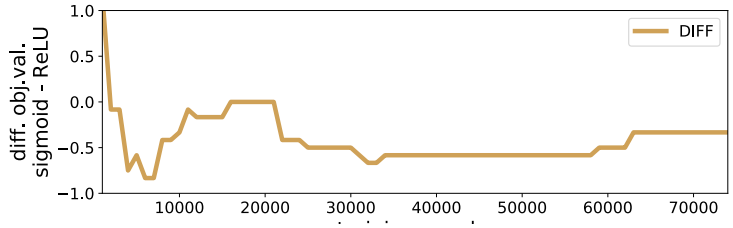
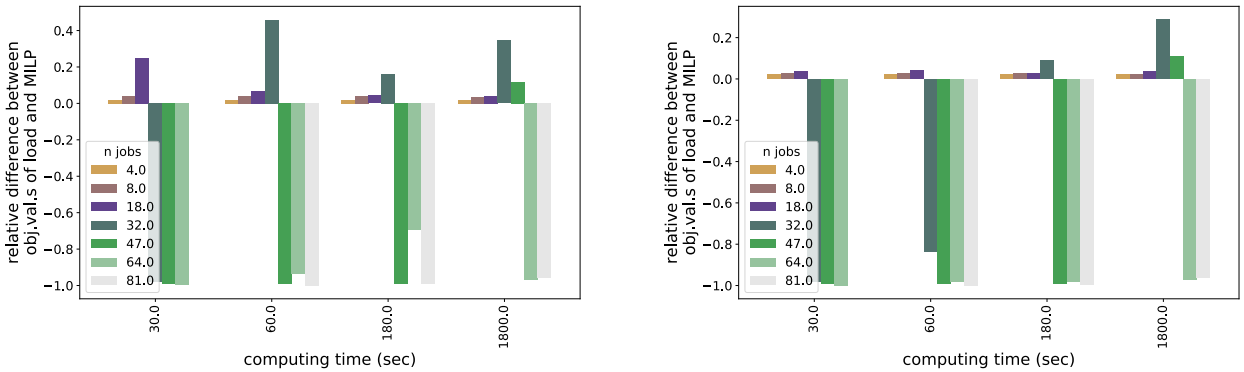


Figure 10: Comparison between the use of a sigmoid or ReLU activation function.

E FIGURES ILLUSTRATING THE RESULTS

E.1 Results for the JSSP-UMP

Figure 11 contains a comparison of the objective values found by the RL and MILP for all testing instances and computing times, using $\epsilon = 0.2$ and $\epsilon = 0.5$. The values on the y-axis represent the relative differences between objective values, calculated as for Figure 4 in Section 4.

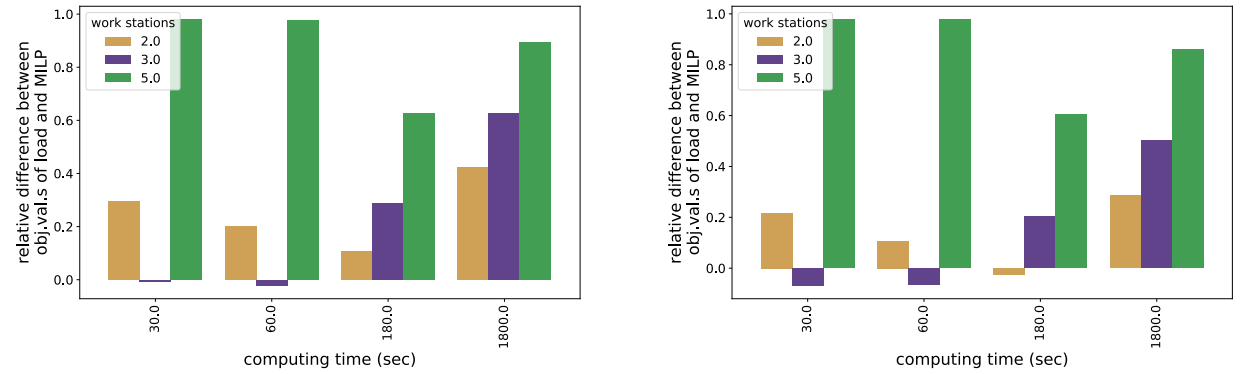


(a) Executed with $\epsilon = 0.2$ **(b) Executed with $\epsilon = 0.5$**

Figure 11: Comparison of the objective values found using the deep RL algorithm and the MILP for different limitations of computing time.

E.2 Results for the FFSSP

Figure 12 shows the results of testing performance of the algorithm when applied to FFSSPs. The figures show that for almost all computing times, our RL algorithm performs much worse than the MILP algorithm. This difference is less for either a small number of work stations, or a very short computing time.



(a) Executed with $\epsilon = 0.0$ **(b) Executed with $\epsilon = 0.2$**

Figure 12: Comparison of the objective values found for solving the FFSSP, using the deep RL algorithm and the MILP for different limitations of computing time.

F RESOURCES

F.1 Specifications

For the implementation and execution of the algorithm, a laptop was used with the following specifications:
Model: BTO, N85_N870HL
Processor: Intel© Core™i5-7300HQ CPU @ 2.50GHz × 4
Memory: 7.7 GiB
Hard Drives: 152.8 GB
Operating System: Linux Mint 19.1 Cinnamon
Cinnamon Version: 4.0.10

F.2 Methods

The complete implementation of the algorithm was written in `Python`. Only general packages were imported to support certain computations, such as `numpy`, `itertools` and `math`. The Reinforcement Learning algorithm and the Neural Network were implemented manually. The code can be found on Github, via the link below. Appendix G contains the corresponding pseudo-code.

github.com/MerelWemelsfelder/RL_scheduling.git

G IMPLEMENTATION PSEUDO-CODE

Code 1 Creating and training the RL algorithm

Settings: m, l_v and g_v for $v = 1 \dots m$, n, δ for each job-unit combination, γ, ϵ, γ-decrease, ϵ-decrease, allowed computing time $\mathcal{T}$, phase $\in \{load, train\}$

```
RL ← MDP(n, m, l_v, g_v)
while computing time < T do
    DONE ← False
    z ← 0
    RL.RESET()
    while not DONE do
        RL, DONE ← RL.STEP(z, γ, ε)
    r ← reward based on objective function
    update value function weights
    if r < r_best then
        r_best ← r
```

Code 2 MDP elements

```
class WORKSTATION()
    function INIT(v, l_v, g_v)
        v ← i                                   ▷ work station index
        resources ← [RESOURCE(i, g_v) for i ∈ {1, ..., l_v}]
    function RESET(waiting)
        resources ← [resource.RESET(waiting) for resource ∈ resources]

class RESOURCE()
    function INIT(i, g_v)
        v ← v                                   ▷ work station index
        i ← i                                   ▷ resource index
        units ← [UNIT(i, q) for q ∈ {1, ..., g_v}]
        policy ← arbitrary initial policy
    function RESET(waiting)
        units ← [unit.RESET() for unit ∈ units]
        state ← waiting

class UNIT()
    function INIT(i, q)
        v ← v                                   ▷ work station index
        i ← i                                   ▷ resource index
        q ← q                                   ▷ unit index
    function RESET(waiting)
        processing ← None                       ▷ job currently proc.
        if v = 0 and q = 0 then
            state ← waiting
        else
            state ← []
class JOB()
    function INIT(j, duedate)
        j ← j                                   ▷ job index
        D ← duedate                             ▷ due date
    function RESET( )
        done ← False                            ▷ processing finished
```

Code 3 Schedule

```
class SCHEDULE()
    function INIT(n, m, l_v, g_v)
        T ← empty vector of length n            ▷ tardiness
        t ← empty vector of length n            ▷ starting times
        c ← empty vector of length n            ▷ compl. times
        schedule ← empty schedule
    function OBJECTIVES( )
        C_max ← max(c) - min(t)                 ▷ makespan
        T_sum ← sum(T)                          ▷ summed tardiness
```

Code 4 MDP

```
class MDP()
    function INIT(n, m, l_v, g_v)
        jobs ← [JOB(j) for j ∈ {1, ..., n}]
        workstations ← [WORKSTATION(v, l_v, g_v) for v ∈ {1, ..., m}]
        NN = NEURALNETWORK(layer architecture)

    function RESET( )
        jobs ← [job.RESET() for job ∈ jobs]
        for ws ∈ workstations do
            ws ← ws.RESET(jobs)
        schedule ← SCHEDULE(n, m, l_v, g_v)
        DONE ← False
        NN_inputs ← []
        NN_outputs ← []

    function STEP(z, γ, ϵ)
        for ws ∈ workstations do
            for unit ∈ resources ∈ ws do
                if processing on unit is completed then
                    set unit to idle
                    move job to next unit
                    if unit = last unit of last ws then
                        schedule.c[j] ← z
                        schedule.T[j] ← max(z - job.D, 0)

        if all jobs are done then
            return self, True

        for all units with unit.q > 1 do
            if unit is idle and unit.state ≠ [] then
                resume first operation in line
        for all units with unit.q = 1 do
            if unit is idle then
                with probability ϵ do
                    job ← random action
                else
                    policy ← NN.forward for all actions
                    job ← action with max policy value
                NN_inputs ← NN inputs for job
                NN_outputs ← estimated policy value for job
                job.t ← z
                remove job from all first-unit states
                add processing of j on i to schedule

        return self, False
```