Vehicle management in a modular production context using Deep Q-Learning

Lucain Pouget*[1†], Timo Hasenbichler[2†], Jakob Auer[1], Klaus Lichtenegger[2] and Andreas Windisch[2,3,4,5,6]

[1*]Skalar Systems GmbH, Dr.-Robert-Graf-Straße 9, Graz, 8010, Austria.
[2]FH JOANNEUM – University of Applied Sciences, Data Science and Artificial Intelligence, Eckertstraße 30i, Graz, 8020, Austria.
[3]Know-Center GmbH, Inffeldgasse 13, Graz, 8010, Austria.
[4]Institute of Interactive Systems and Data Science, Graz University of Technology, Inffeldgasse 13, Graz, 8010, Austria.
[5]Physics Department, Washington University in St. Louis, One Brookings Drive, MO, St. Louis, 63130, USA.
[6]RL Community, AI Austria, Wollzeile 24/12, Vienna, 1010, Austria.

Contributing authors: lucain.pouget@skalarsystems.com; timo.hasenbichler@protonmail.com; jakob.auer@skalarsystems.com; klaus.lichtenegger@fh-joanneum.at; awindisch@know-center.at;
[†]These authors contributed equally to this work.

Abstract

We investigate the feasibility of deploying Deep-Q based deep reinforcement learning agents to job-shop scheduling problems in the context of modular production facilities, using discrete event simulations for the environment. These environments are comprised of a source and sink for the parts to be processed, as well as (several) workstations. The agents are trained to schedule automated guided vehicles to transport the parts back and forth between those stations in an optimal fashion. Starting from a very simplistic setup, we increase the complexity of the environment and compare the agents'

performances with well established heuristic approaches, such as first-in-first-out based agents, cost tables and a nearest-neighbor approach. We furthermore seek particular configurations of the environments in which the heuristic approaches struggle, to investigate to what degree the Deep-Q agents are affected by these challenges. We find that Deep-Q based agents show comparable performance as the heuristic baselines. Furthermore, our findings suggest that the DRL agents exhibit an increased robustness to noise, as compared to the conventional approaches. Overall, we find that DRL agents constitute a valuable approach for this type of scheduling problems.

1 Introduction

1.1 Modular Production

Modular production systems based on Automated Guided Vehicles (AGVs) allow a drastically increased flexibility of the production plant or setup. Instead of a long line of machines connected by conveyor belts, production islands (consisting of one or several machines) are spread over the production plant and connected via AGVs. The inputs and outputs of these production islands usually also contain buffers. The increased flexibility allows advancements like multiple product variants in the same plant, alternative stations (two stations of the same type as performance improvement or for failure safety) or the reuse of expensive stations at different stages of the production.

1.2 The Job Shop Scheduling Problem

In modular production environments, the optimized scheduling of the AGVs (the decision which AGV is assigned to which transport task) is crucial for the system's overall performance. This task leads to the Job Shop Scheduling Problem (JSSP), which is – like the related Travelling Salesman Problem – a computationally "hard" (NP-complete) problem of enormous practical importance, [**?**]. Due to the typically factorial growth of possible arrangement with the size of the problem, exact solutions can only be found for very small problems. Since the AGV to Job scheduling has to be done in real-time, heuristic strategies are used. This is usually done with algorithms like nearest neighbor [**?**] or cost tables [**?**]. All of these methods have various advantages and drawbacks, but they are all reaching a limit with increasing complexity and system size. Varying driving times (because of human machine interaction or other vehicles in the systems) or processing times highly affect the systems performance and stability. Wrong or suboptimal scheduling decisions can lead to deadlocks in the system. We show that Deep-Q based deep reinforcement learning agents are capable to overcome the aforementioned problems and also allow the calculation of optimal solutions in real-time.

1.3 Deep Q-Learning

Reinforcement learning aims to train an agent to interact with the environment in a way that maximizes the expected cumulative reward. The policy of the agent $\pi(\theta)$ is defined as a function that maps from a given state s to a suitable action a. Q-learning is a value-based approach, which means that the agent will try to approximate the expected cumulative reward (the "value") for each possible pair of (state/action). This value is formalized as

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a\right] \tag{1}$$

The expected cumulative reward $Q(s, a)$ in state s for an action a is defined as the sum of the immediate reward $r(s, a)$ and the highest Q-value for the next states among all possible actions. A discount factor $\gamma \in [0, 1]$ is applied to the future reward in order to control how to rate future rewards compared to the immediate ones.

Due to the usually large number of state-action pairs, filling the whole table of Q values is possible only for very small problems. An approach to overcome this obstacle is *Deep Q-learning*. This is a type of Q-learning where the value function is approximated by a deep neural network, that is, by a network with several layers. This is justified by the universal function approximation property of sufficiently deep neural networks, [**?**].

Each time the Agent makes a decision, an experience (initial state, action, reward, next state) is saved into a buffer (the Replay Memory). The weights of the network are trained using backpropagation on mini-batches, sampled from the Replay Memory, [**?**].

1.4 Structure of the article

This paper is structured as follows: In section 2, the environment used for training the DRL agents is discussed in detail, including the particular configurations of the environments, as well as technical aspects, such as using OpenAI's gym interface, benchmark settings and heuristic approaches that have been used for comparison. The design of the DRL agent is then discussed in section 3, where, based on Sec. 1.3, a more detailed explanation of the reward signal used for training is given. The main findings of this paper are summarized in section 4, where all chosen production setups are presented, together with the respective performance of the DRL agents and heuristic approaches. Finally, in section 5, we conclude with a short overall summary and point out possible paths for future investigations.

2 Construction of the environment

2.1 Production plant modeling

A production plant is modelled by a set of n_M modular workstations. Each workstation consists of an Input Buffer (IB), a Production Unit (PU) and an Output Buffer (OF). Both buffers have a capacity of n_{buf} and are running as FIFO queues (First In-First Out). In addition, we defined a Start of Line station ($Source$) and an End of Line station ($Sink$). The $Source$ can be seen as a station with only an OB and the $Sink$ as a station with only an IB.

Parts are defined by their part type PT. For each part type PT_j, a sequence of n_O operations is defined. This sequence represents the workstations that the parts of type PT_j must visit in the right order. The sequence always starts at the Source and ends at the Sink. The same workstation can be used multiple times in a sequence. Different part types do not necessarily use the same workstations. Parts are carried by n_V AGVs V with speed v_V. A vehicle can only carry 1 part at a time. The transfer of a part between a vehicle and a station, or between 2 units of a station, has a duration T_{transfer}.

A weighted multi-directional graph G, called "waypoint graph", is defined to represent the distances between the stations. The nodes are the waypoints in the production plant through which the vehicles can pass. Each node N_j is described by Cartesian coordinates x_j, y_j. The edges are the paths connecting these locations. Each edge is either unidirectional or bidirectional and has a weight corresponding to the euclidean distance between its nodes. Each buffer of each workstation is linked to a node of the waypoint graph. A node can only be assigned to a maximum of one buffer.

In our model, the release order of parts at the source is modelled by a uniform distribution over the part types. A source clock C_{source} defines when a new part is released to the system. Our goal is to maximize the production throughput by assigning jobs to the AGVs. A job consists of driving to a station S_a, picking a part P_j, driving to another station S_b, dropping the part P_j. Jobs must be assigned in such a way as to ensure that there are no deadlocks. This problem is defined as the Vehicle Management problem (VM), which is solved with deep learning (more precisely with proximal policy optimization) in [**?**].

2.2 Implementation details

2.2.1 Discrete-event simulation

The modular production plant is simulated in Python using the Simpy library [**?**], a framework to build Discrete-Event Simulations (DES) [**?**]. In a DES, events are triggered by the environment, each of them at a predefined instant in time. Between two consecutive events, the environment is static. We chose this approach because of its efficiency as compared to continuous simulations. Having a simulation running in Python has also been beneficial for the communication between the Agent and the Environment.

We performed a benchmark of our simulation tool in various cases to estimate the speed-up factor of our approach over simulations not relying on discrete event simulation. We defined the complexity of a configuration as the number of AGVs N_{agvs} times the number of Machines N_{machines}. Each production plant configuration has been simulated for 12 hours using the FIFO agent. Since our simulation is not computing each intermediate steps when an AGV moves, but instantly jumps to the next relevant event (e.g. AGV reaches destination), the speed-up factor shows a strong dependence on the parameters of the environment, such as distances, AGV speed and processing times. For typical environments used in training, it takes between 360 ms and 2.6 s to simulate 1h of production, depending on the complexity of the environment. The calculation for the benchmark has been conducted on a laptop computer using a single thread (Dell XPS 15, Intel core i7 2.60Hz). Lacking an obvious definition of complexity of the environment renders quantitative speed-up factors somewhat arbitrary. We thus refrain from providing quantitative data at this point, but emphasize, that our DES based approach shows a very clear advantage in simulation time over discretized time simulations, which is a very desirable property simulation environments used for Deep Reinforcement Learning should possess.

2.2.2 OpenAI Gym interface

We use OpenAI's library Gym [**?**] to implement the interface between the Simpy simulation and the decision making process. In particular, the library provides utilities to handle observations, actions and rewards. The main concepts handled by the library are:

- **Observation:** object that describes the state of the Simpy simulation at a given instant. In our case, the Observation is a human-readable dictionary of values. Gym library takes care of serializing this data into a vector that can be sent to the neural network.
- **Action:** object that describes the action taken by an Agent.
- **Reward:** float value describing the amount of reward for a given action. The goal of the training is always to maximize the expected, cumulative reward.
- **Environment:** the minimal acceptable environment is an object implementing a "step" method. This "step" method takes as input an action and outputs a tuple of four elements:
  - **Observation:** the state of the environment once the action has been performed.
  - **Reward:** amount of reward achieved by the previous action
  - **Done:** a boolean value set to true if the environment must be reset.
  - **Info:** any information useful for debugging. Not used in our case.

Each time the "step" method of the environment is called, a new Experience – a tuple (initial state, action taken, next state, reward, done) – is stored in the memory for the learning process of the agent.

We had two major benefits of using this framework:

- Gym contains utilities in order to serialize and deserialize the Observation and Action concepts from a human-readable object to a vector suitable for a neural network.
- Gym implements several well known problems [**?**]. These tasks proved useful for testing and debugging the implementation of our agent against environments already solved by Deep-Q agents.

2.2.3 Communication protocol

The "Agent" is the algorithm that makes the decisions. We defined a common interface for both, deterministic and parameterized agents, which allowed us to seamlessly plug them into the simulation for comparison. Deterministic agents are algorithms that are entirely hard-coded, while parameterized agents require a training phase before evaluation.

The Agent Interface uses two methods:

- **Act:** takes as input an Observation state describing the environment and returns the action to perform.
- **Step:** takes as input an Experience and returns nothing. This method is used for training the parameterized agents in order to improve their policy.

The Controller is an element of the Simpy environment that makes the connection between the AGVs and the Agent. In a production environment, this would be the Fleet Management System.

The communication process is the following:

1. The environment is initialized with all its elements.
2. The simulation starts.
3. When an AGV is available (it has finished its current task), it triggers a custom Simpy Event.
4. The Controller stops the simulation and computes an Observation state from the environment.
5. The Observation state is sent to the Agent.
6. The Agent takes an action that is communicated to the Controller.
7. The Controller deserializes the action and assigns the corresponding job to the waiting AGV. A job is a station from which an AGV must pick up a part.
   (a) If multiple AGVs are waiting at the same time, the Controller sends the job to the waiting AGV that will perform the action the fastest (i.e. the closest AGV for that job).
   (b) The Agent is then called immediately after to define another job for the other waiting AGVs until all AGVs have a job.
8. Once all the AGVs have a job, the simulation starts over at 2.

The Agent also has the possibility to choose not to assign any task. In that case, the waiting AGV will stay inactive until a new event is triggered.

2.3 Benchmark settings

2.3.1 Production plant settings

Several production plants have been configured in order to compare the Agents on different types of problems.

In the figures presented in the following paragraphs (figures 1, 2 and 3), blue dots stand for the nodes. Source and sink are respectively the leftmost and the rightmost nodes. AGVs can commute between nodes that are connected by black double-arrows. The numbers in red are the length of each path in millimeters. Finally, the gears sections represent the machines.

Mayer-Classen-Endisch

This configuration is defined in [**?**]. It has 2 machines, 1 AGV and 1 part type. We re-implemented the simulation, taking the same parameters to verify that our approach reproduces the results of [**?**].

1-machine-big

This environment has 1 machine, 2 AGVs and 1 part type. The particularity lies in the distances between the nodes of the waypoint graph (see figure 1). The pairs "source/machines input" and "machines output/sink" are very close, while the pairs "source/sink" and "machines input/machines output" are very far apart. Since there are 2 AGVs, the optimal solution is to have one vehicle doing circles between the source and the machine's input, while another vehicle does the same between the machine's output and the sink. We introduced this environment, expecting that the FIFO Agent would struggle with this setup.

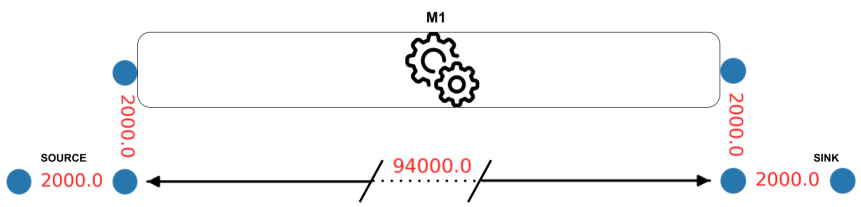

Fig. 1 1-machine-big configuration: optimizing the driving time is essential here.

3-machines-loop

We set up a configuration with 3 machines and a unique part type (figure 2). The particularity here is, that the sequence of operations to perform on a part has a loop, passing twice through the same machine. Each part P_j has to follow the sequence Source $\rightarrow M1 \rightarrow M2 \rightarrow M1 \rightarrow M3 \rightarrow$ Sink. This is a well

known type of problem in modular production that often leads to deadlocks when using static algorithms.

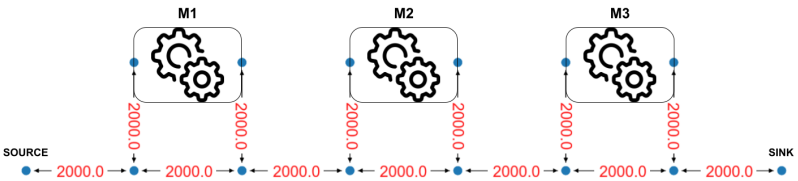

Fig. 2 3 machines configuration: the sequence of operations in this setup includes a loop.

6-machines-grid

This is a family of problems starting with the same production plant configuration. The goal is to incrementally increase the difficulty of the scenarios and monitor how it impacts the performance of the agent at each step. For all sub-problems, the production plant layout, shown in figure 3, is shared. It has 6 machines distributed as a grid (2 rows of 3 machines), but depending on the scenario, not all of them are in use.

Complexity is increased in 2 ways:

- By increasing the number of machines to visit. Each sub-problem has 2 part types that follow similar paths in the production plant. 3 scenarios are considered, using either 2 machines ($M1 \rightarrow M2$ and $M2 \rightarrow M1$), 4 machines ($M1 \rightarrow M3 \rightarrow M2 \rightarrow M4$ and $M2 \rightarrow M4 \rightarrow M1 \rightarrow M3$) or 6 machines ($M1 \rightarrow M3 \rightarrow M5 \rightarrow M2 \rightarrow M4 \rightarrow M6$ and $M2 \rightarrow M4 \rightarrow M6 \rightarrow M1 \rightarrow M3 \rightarrow M5$).
- By increasing the number of AGVs (from 1 to 4). With only 1 vehicle, the limiting factor is usually the speed of the vehicle, whereas with a larger number of vehicles it is possible to reach a 100% workload on the machines.

2.3.2 Deterministic Agents

Deterministic agents follow a hard-coded algorithm and are used as a baseline for result comparison. In this section, we call $\text{AGV}_{\text{longest}}$ the AGV that has been waiting the longest for a new assignment and P_{longest} the part that has been waiting the longest in the output buffer of a station (named S_{longest}).

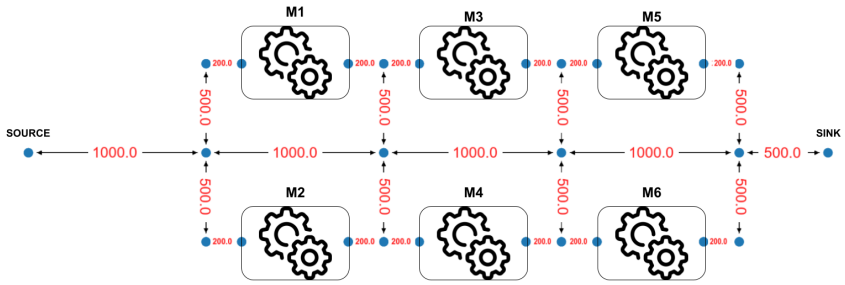

Fig. 3 6-machines grid configuration: depending on the scenario, not all machines are in use.

An AGV is considered as active, if it has a task assigned to it and inactive otherwise.

FIFO

The First-In First-Out (FIFO) algorithm consists of assigning S_{longest} to $\text{AGV}_{\text{longest}}$.

Nearest Neighbor

The Nearest Neighbor (NN) algorithm consists of selecting S_{longest} and compute for all the AGVs the time it will take to reach the station exit containing this part:

- If an AGV is inactive, it estimates the time it will take to drive to S_{longest}.
- If an AGV is active, it estimates the time it will take to finish the current task and then reach S_{longest}.

The Agent assigns the job to the closest AGV in terms of duration. If this AGV is already active, the Agent doesn't assign any task.

Cost Table

The Cost Table algorithm is an extension of the Nearest Neighbor approach. Instead of doing an estimation only for the station S_{longest}, it builds an array of size `[Number of inactive AGVs, Number of waiting stations]`, containing the time estimation for every AGV/Station pair (i.e., "costs"). The Agent solves the linear sum assignment problem on this Cost Table using the linear sum assignment algorithm [**?**] implemented in SciPy [**?**]. The solution of the problem is a list of AGV/Station pairs that minimizes the duration to start all the tasks. Since the Agent can only make one assignment at a time, it arbitrarily selects the task with the AGV that has been waiting the longest

among the AGVs of the solution. The Agent is immediately called again to re-compute the Cost Table and re-solve the optimization problem until all AGVs are active, or no parts are waiting in a station output.

2.3.3 Methodology

Our goal is to be able to compare a trained Agent against static algorithms in order to prove the ability of DRL to tackle the Vehicle Management problem in a modular production environment.

We found out that static algorithms are highly vulnerable to the input clock C_{source}. If C_{source} is too low, the AGVs will almost immediately overload the first machine with parts and run into a deadlock. On the contrary, if C_{source} is too high, the optimization problem doesn't make sense anymore because it is too easy to get an output clock C_{sink} equal to C_{source}.

In order to still be able to have baseline results to compare with, we first find the optimal C^*_{source} for each agent and each environment configuration. C^*_{source} is defined as the lowest input clock for which the simulation doesn't run into a deadlock situation. We use a dichotomic search algorithm to find this value, rounded to 1 second. By construction, C_{sink} is necessarily equal or greater than C^*_{source}. Once C^*_{source} is found, we run the simulation for 12 hours of production, gathering the metrics throuhout the run.

For the DRL Agent, C_{source} is set to 0 during both, train and test phases. It is assumed that it should learn how to avoid deadlocks by not overloading the first machine.

3 Design of the DRL Agent

In this section, we specify the deep-Q learning approach, briefly discussed in Sec. 1.3, to our specific situation and discuss some details of its implementation.

3.1 Observation, action, and reward design

3.1.1 Observation state

The observation state is the vector describing the Environment that is sent to the Agent. It contains all the information needed to choose the next action. We defined a state S_v for each AGV and a state S_{unit} for each station unit. The environment state S is the concatenation of all the S_v and S_{unit} states.

For each vehicle V_i, the state Sv_i consist of:

- `Action_state`: current action performed by the vehicle as a 1-hot encoded vector: `DRIVING`, `TRANSFERRING_IN`, `TRANSFERRING_OUT`, `WAITING_FOR_ORDER`, `WAITING_TO_DROPDOWN`, `WAITING_TO_PICKUP`
- `Carried part`: a representation of the part that is carried, if any (see below).
- `Current order target`: 1-hot encoded vector representing the destination of V_i, if driving. If V_i is not currently driving, it is a zero vector of appropriate dimension.

- `Last visited node`: 1-hot encoded vector representing the last node which V_i has visited.

For each unit U_i, the state Sunit_i consists of:

- `Action state`: current action performed by the unit as a 1-hot encoded vector: `PROCESSING`, `TRANSFERRING_IN`, `TRANSFERRING_OUT`, `WAITING_TO_DROPDOWN`, `WAITING_TO_PICKUP`
- `Carried parts`: a representation of each part that is in the unit (see below).

For each part is the production plant, either in a station unit or an AGV, a state representation is defined as consisting of:

- `Part_type`: 1-hot encoded vector representing the part type. The information is useful to determine what are the next operations to be performed on the parts.
- `Part_completion`: float value between 0 and 1. When a part is at the Source, the completion value is 0 while at the Sink it is set to 1. In between, the completion value is increased linearly each time an action is performed on the part (driving or processing).
- `Part_next_station`: 1-hot encoded vector representing the next station the part needs to visit (or Sink if processing is done). In principle, if only `part_type` and `part_completion` is provided, the Agent should be able to learn the next steps for a part by itself. However, we discovered by experience that providing redundant information helped to increase the overall performance.

3.1.2 Action

The set of all possible actions is equivalent to the set of stations where it is possible to pick-up a part (i.e., the Source and all the workstations). The Sink is not included in the set of actions, since it's only possible to drop off a part there. In contrast to [**?**], the set of all dropdown actions `A_drop` is not defined in our case, since the dropdown station is always implicit and only depends on the carried part. This is a limitation in the case of a production plant where the same operation can be performed by 2 different workstations, but it is beyond the scope of this paper.

In addition to this set of actions, we also allow the Agent to take a "do nothing" action, for which no task is assigned. The inactive vehicles stay inactive until another event triggers a request to the Agent.

3.1.3 Reward

The reward function has been one of the most difficult components to configure in this setup. We realized that the performance of the Agent and its capability to learn are highly dependent on the reward signal. In the end, we built a score function for each environment state S_t. From a state S_t and for an action A_t that resulted in the state S_{t+1}, we defined the reward $R(S_t, A_t)$ as $\text{Score}(S_{t+1}) - \text{Score}(S_t)$.

We built the score of a state as the sum of 4 components:

- Per-part reward S_{pp}: a positive reward for each completed part that reaches the Sink.
- Per-part completion reward $S_{\text{pp\%}}$: a positive reward each time a part is being processed. We define the completion of a part (between 0 and 1) as the percentage of processing steps that have been completed over all steps. When the part is at the Source, the completion is 0, while it is 1 at the Sink. This reward helps the Agent learn about the intermediate steps by providing shorter-term feedback.
- Per-assigned decision reward $S_{\text{decisions}}$: a positive reward for each decision taken by the Agent that is actually assigned to an AGV. In particular, the "do-nothing" action is not rewarded. We noticed a slight increase in performance when forcing the Agent into being less lazy.
- Per-second reward S_{time}: a positive reward for each second that has been simulated without encountering a deadlock.

We denote by $N_{\text{decisions}}(t)$ the number of assigned decisions since the beginning of the simulation, by $N_{\text{pp}}(t)$ the total number of completed parts, and by $N_{\text{pp\%}}(t)$ the sum of each part completion. In practice $N_{\text{pp\%}}(t) \geq N_{\text{pp}}(t)$ at all time.

A difficulty we had has been to balance the 4 components of the score function. Depending on the environment configuration, the expected number of assigned decisions and parts processed were highly variable, whereas the number of simulated seconds was always the same (12-hours when no deadlocks). To balance those differences, we defined 3 configurable values: E_{pp} (expected processed parts), $E_{\text{decisions}}$ (expected assigned decisions) and E_{seconds} (expected duration). The final score function is defined as a weighted average of the 4 components:

$$S(t) = \begin{cases} K \left[\frac{N_{\text{pp}}(t)+N_{pp\%}(t)}{E_{\text{pp}}} + \frac{N_{\text{decisions}}(t)}{E_{\text{decisions}}} \right], & \text{if deadlocked,} \\ K \left[\frac{N_{\text{pp}}(t)+N_{\text{pp\%}}(t)}{E_{\text{pp}}} + \frac{N_{\text{decisions}}(t)}{E_{\text{decisions}}} + \frac{t}{E_{\text{seconds}}} \right], & \text{otherwise.} \end{cases} \tag{2}$$

In practice, we found that $K = 4000$ worked best for us. E_{pp}, $E_{\text{decisions}}$ and E_{seconds} are estimated based on the Cost Table Agent performances.

3.2 DQN implementation

In this research, we tested the vanilla DQN agent described in [**?**] and implemented some additional promising extensions to improve the agent's performance. This section briefly describes the implemented extensions. The exact hyperparameters used for the performance can be found in 1. In [**?**], the authors showed that the vanilla DQN-agent performed better in 43 of 49 games as the best previously known reinforcement learning baselines, setting a new standard in the industry. However, further investigations of the DQN

Parameter	Value
Replay memory size	1e5
Batch size	64
Gamma	0.99
Learning rate	0.001
Target update rate	24
Update rate	4
Epsilon	1.0
Epsilon decay	0.9995
Epsilon min	0.01
NN nb layers	2
FC1 units	64
FC2 units	32

Table 1 DDQN training hyperparameters.

showed that it is prone to be overly optimistic when estimating Q-values when using Q-learning. Hence, it overestimates the Q-values. This over-optimism could hinder the convergence to the agent's optimum, or the agent might not even converge at all. In [**?**], the authors showed that the over-optimism could be counteracted by applying Double Q-learning rather than Q-learning. The resulting agent is called DDQN and requires almost no changes to the existing implementation.

Moreover, in DQN, the baseline approach to handle the exploration-exploitation dilemma is the straightforward epsilon-greedy strategy. This strategy relies on random permutations of the sequence of actions to explore the environment. However, in some environments, where the reward is very sparse, it can be challenging to explore the environment sufficiently using this strategy. Hence, we also tried to drive exploration using noisy networks as described in [**?**] by inducing noise when estimating the Q-values. Moreover, in this research, we also tested agents using a dueling network architecture, splitting the data stream in the neural network into two separate data streams as described in [**?**]. One data stream estimates the value of the state, while the other data stream estimates the advantage for picking an action in that state. Finally, the most significant extension that showed the most robust performance during all tests is the prioritized experience replay (PER) as described in [**?**]. The PER samples the experiences prioritized based on the temporal difference error rather than sampling uniformly.

4 Experimental results

4.1 Per-scenario results

The benchmark has been run on each scenario for 12 hours with 4 different agents: 3 static (FIFO, Nearest Neighbor, Cost Table) and 1 trained (DDQN). Before running the simulation, we determined the optimal source clock $C_{\text{source}}*$

Environment	Agent	1 AGV	2 AGVs	3 AGVs	4 AGVs
Mayer-Classen-Endisch	FIFO	50			
	Nearest Neighbor	50			
	Cost Table	50			
	DDQN	0			
1-machine-big	FIFO		83		
	Nearest Neighbor		48		
	Cost Table		11		
	DDQN		0		
3-machines-loop	FIFO	153	81	56	
	Nearest Neighbor	153	80	57	
	Cost Table	120	82	55	
	DDQN	0	0	0	
2-machines-grid	FIFO	31	25	25	25
	Nearest Neighbor	31	25	24	22
	Cost Table	20	25	24	22
	DDQN	0	0	0	0
4-machines-grid	FIFO	41	25	25	25
	Nearest Neighbor	41	25	25	25
	Cost Table	37	25	25	24
	DDQN	0	0	0	0
6-machines-grid	FIFO	53	29	25	25
	Nearest Neighbor	53	29	25	25
	Cost Table	48	28	25	25
	DDQN	0	0	0	0

Table 2 Optimal source clock

for each static agent. The main metric that we monitor is the overall throughput, defined in parts per hour (pph). We also measured the total amount of parts produced in 12 hours.

Optimal source clocks can be found in table 2, throughput results in table 3, and total parts produced results in table 4. Cells are kept empty when the experiment was not meaningful. In particular, Mayer-Classen-Endisch is reproduced with the exact same settings as in [**?**] (1 AGV) and 1-machine-big is designed for 2 AGVs.

4.1.1 Mayer-Classen-Endisch

The first environment we trained on has been the one described by [**?**]. Results are shown in figure 4. The trained agent reached a throughput of 71.8 parts/hour over 12 hours (863 parts). The maximum throughput in this case is 72 parts/hours (864 parts). This environment has been a good first example, since we have been able to reproduce 2 results described by [**?**]:

- The agent starts with a ramp-up process before reaching the optimal throughput. This explains the difference of 1 produced part observed over the 12 hours.

Environment	Agent	1 AGV	2 AGVs	3 AGVs	4 AGVs
`Mayer-Classen-Endisch`	`FIFO`	**71.8**			
	`Nearest Neighbor`	**71.8**			
	`Cost Table`	**71.8**			
	`DDQN`	**71.8**			
`1-machine-big`	`FIFO`		23.6		
	`Nearest Neighbor`		74.9		
	`Cost Table`		143.3		
	`DDQN`		**143.4**		
`3-machines-loop`	`FIFO`	23.4	44.2	63.8	
	`Nearest Neighbor`	23.4	44.8	62.8	
	`Cost Table`	**26.9**	43.7	**65.0**	
	`DDQN`	26.3	**45.4**	60.2	
`2-machines-grid`	`FIFO`	115.8	143.7	**143.8**	143.7
	`Nearest Neighbor`	115.8	**143.8**	**143.8**	143.7
	`Cost Table`	**123.1**	**143.8**	**143.8**	143.7
	`DDQN`	115.2	139.2	**143.8**	**143.8**
`4-machines-grid`	`FIFO`	87.4	**143.4**	143.4	**143.5**
	`Nearest Neighbor`	87.4	**143.4**	**143.5**	**143.5**
	`Cost Table`	**95.7**	**143.4**	**143.5**	143.3
	`DDQN`	88.8	128.9	138.9	134.1
`6-machines-grid`	`FIFO`	67.5	123.5	143.1	**143.2**
	`Nearest Neighbor`	67.5	123.5	**143.2**	**143.2**
	`Cost Table`	**74.1**	**127.8**	**143.2**	**143.2**
	`DDQN`	68.6	79.0	117.4	124.0

Table 3 Throughputs in parts per hour (pph)

- The final policy learned by the agent is the same, namely, the AGV loops through the production plant in a certain order: Source $\rightarrow M1_{\text{input}} \rightarrow M1_{\text{output}} \rightarrow M2_{\text{input}} \rightarrow M2_{\text{output}} \rightarrow$ Sink $\rightarrow$ Source $\rightarrow \ldots$

Finally, it is also worth mentioning that the baseline agents (FIFO, Nearest Neighbor and Cost Table) have also been able to achieve the same performance (862 parts produced). The main benefit of the DDQN agent is its robustness to the source clock.

4.1.2 1-machine-big

With this configuration, we expected the FIFO agent to be tricked, while a Cost Table approach was supposed to tackle the problem raised by the long distances in the production plant. We confirmed this intuition, since they reached 23.6 parts/hour and 143.4 parts/hour respectively, the optimal throughput in the equilibrium state being 144 parts/hour. The same performance has been achieved by the trained agent. Results can be compared in figure 4.

For both Cost Table and DDQN approaches, the final policy in the equilibrium state is to have an AGV looping between the Source and the Machine entry and the other AGV looping between the Machine exit and the Sink, thus avoiding the long drive between Machine entry and exit.

Environment	Agent	1 AGV	2 AGVs	3 AGVs	4 AGVs
Mayer-Classen-Endisch	FIFO	862			
	Nearest Neighbor	862			
	Cost Table	862			
	DDQN	**863**			
1-machine-big	FIFO		284		
	Nearest Neighbor		899		
	Cost Table		1720		
	DDQN		**1721**		
3-machines-loop	FIFO	281	531	768	
	Nearest Neighbor	281	538	755	
	Cost Table	**326**	525	**782**	
	DDQN	317	**547**	725	
2-machines-grid	FIFO	1392	**1726**	1726	1726
	Nearest Neighbor	1392	**1726**	1726	**1727**
	Cost Table	**1479**	**1726**	1726	**1727**
	DDQN	1383	1672	**1727**	**1727**
4-machines-grid	FIFO	1051	**1724**	**1724**	1724
	Nearest Neighbor	1051	**1724**	**1724**	1724
	Cost Table	**1151**	**1724**	**1724**	**1725**
	DDQN	1068	1551	1670	1611
6-machines-grid	FIFO	812	1485	**1722**	**1723**
	Nearest Neighbor	812	1485	**1722**	**1723**
	Cost Table	**892**	**1537**	**1722**	**1723**
	DDQN	828	951	1413	1491

Table 4 Total parts produced over 12h

It is also worth mentioning that the Nearest Neighbor approach gets intermediate results (74.9 parts/hour) which confirm that all static approaches are not equivalent even in small scenarios.

4.1.3 3-machines-loop

The difficulty in this scenario is to manage the AGV(s) in order not to overload the first machine. Results show that in the end, the different agents get on average similar results (see figure 5). In the details, Cost Table is slightly better than DDQN with 1 AGV (26.9 vs 26.3 pph), slightly worse with 2 AGVs (43.7 vs 45.4 pph) and significantly better with 3 AGVs (65.0 vs 60.2 pph). This shows the difficulties the DDQN agent has to manage multiple AGVs with our framework. We will discuss this aspect further in 4.2.3.

4.1.4 6-machines-grid

2-machines scenario

The 2-machines scenario seems quite easy to tackle for the static agents (see figure 6). The Cost Table approach performs better with 1 AGV but starting from 2 AGVs, the processing time is more limiting than transportation, which means that even a sub-optimal policy is able to obtain the best theoretical throughput (144 pph).

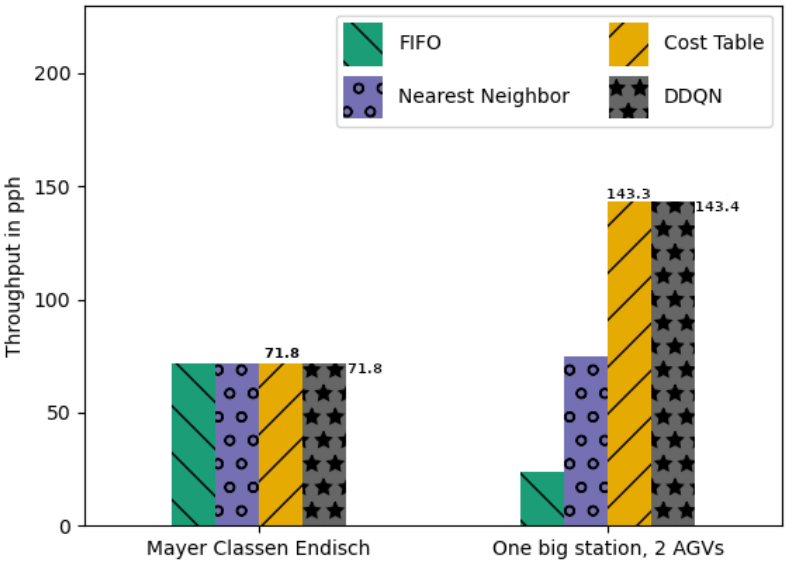

Fig. 4 Results on Mayer-Classen-Endisch and 1-machine-big environments.

The DDQN agent is able to reach FIFO/Nearest Neighbor performances with 1 AGV (115 pph) but is not optimal since Cost Table performs significantly better (123 pph). With 2 AGVs it is not optimal either, performing at 139 pph even though all static agents are already reaching 144 pph. Above that (3 and 4 AGVs), comparisons don't make sense, since all agents reach the limit.

4-machines scenario

The 4-machines scenario is similar to the previous one (see figure 7). Cost Table performs better than other approaches with 1 AGV (96 pph vs 89) and then all static agents reach the limit (144 pph).

However, this scenario starts to show the limitation of the DDQN agent. It is able to learn how to manage the AGVs to transport parts and still avoid deadlocks, but in a sub-optimal way. The best results are achieved with 3 AGVs (139 pph), a better performance than when a 4th AGV was available (134 pph).

6-machines scenario

In the 6 machines scenario, it is even more difficult for the DDQN to reach optimal performances -or at least similar to static agents- (see figure 8). It is worth mentioning that even if the DDQN performs worse than Cost Table in

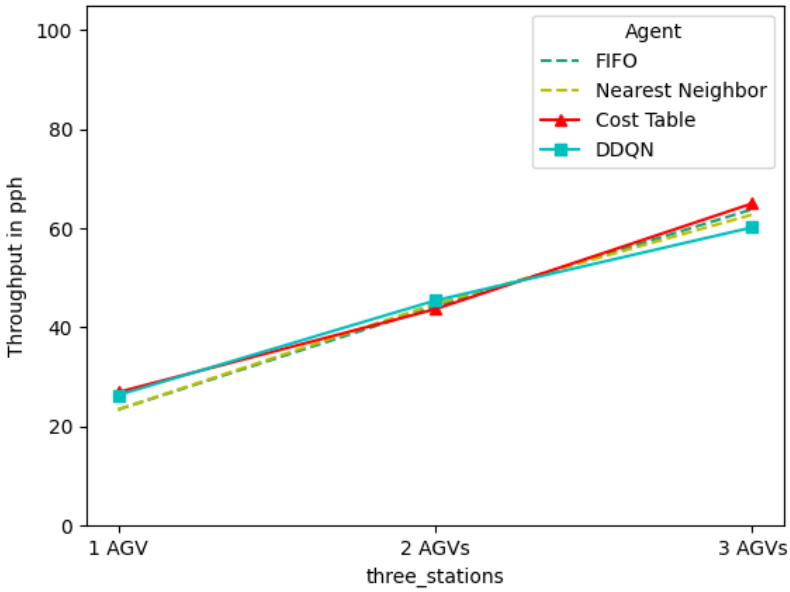

Fig. 5 Results on the 3-machines-loop environment.

all cases (69 vs 74 pph with 1 AGV, 79 vs 128 pph with 2, 117 vs 143 with 3 and 124 vs 143 with 4), it is still able to run the 12 hours without deadlocks.

4.2 Improvement perspectives

4.2.1 Improve the observation state

Our final observation state does not provide all the information needed by the agent to take the optimal decision. In particular, with our approach, the DDQN agent can't take advantage of the exact position of the AGVs and the transportation times. Our efforts to include this information in the observation state have not yet been successful. However, this information is available and used by the Cost Table agent, which can explain its better performances.

4.2.2 Improve reward signal and deadlock punishment

Another learning of our study is the sensitivity of the DDQN agent to the reward signal, especially to the deadlock punishment signal. In the end, our agent learned how to avoid running into a deadlock but at the cost of been too protective in some cases (i.e., it is more risky and not enough rewarded to start processing a new part as compared to wait for the production plant to be less full). This balance has been tough to configure and is still not entirely satisfying.

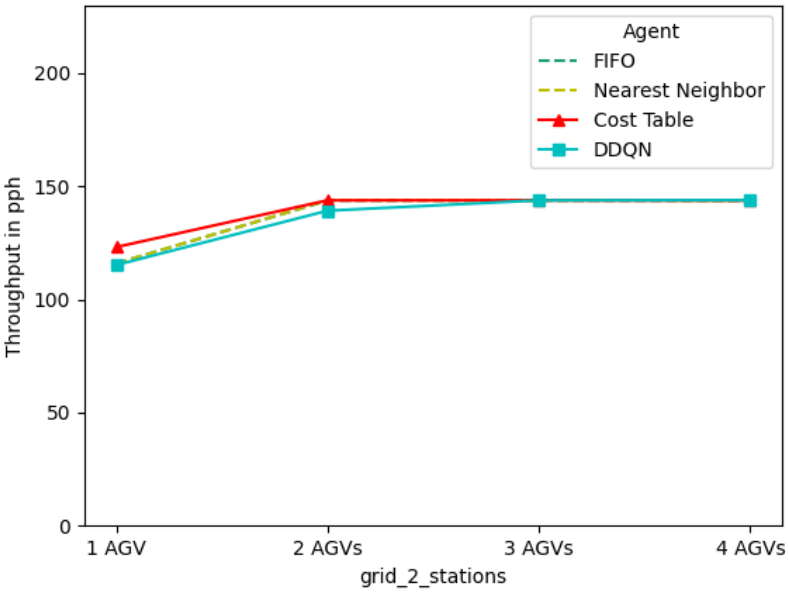

Fig. 6 Results on the grid-2-machines environment.

4.2.3 Improve management of several AGVs

The study has shown the limitation of our approach when it comes to managing several AGVs at the same time. We believe that there is room for improvement in the communication protocol between the Agent and the AGVs. Our intuition is that the current set of actions the Agent can take is too limited. We had difficulties into making the Agent choose both the Station and the AGV that must perform the action.

5 Conclusions and Outlook

In this paper, we investigated the deployment of Deep-Q based DRL agents to job-shop scheduling problems in modular production facilities using discrete event simulations for the environment. Here are the key findings of our study:

- We have developed a simulation that runs fast enough to train DRL agents. We have established a framework to configure environments of different complexity, on which we can additionally run a baseline of three static agents.
- For each environment, we trained a DDQN agent that is able to run the production plant deadlock-free. The trained agent has learned some complex strategies (beating FIFO in 1-machine-big and handling a loop in 3-machines-loop environment).

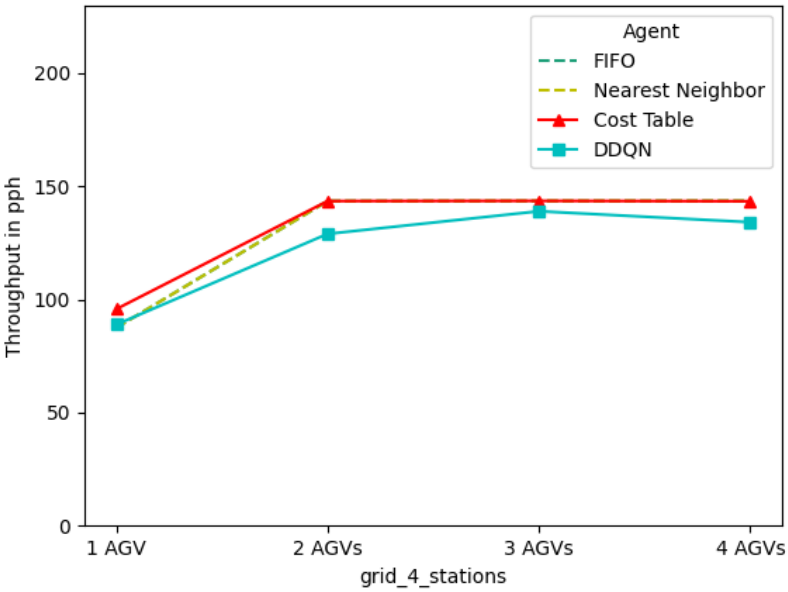

Fig. 7 Results on the `grid-4-machines` environment.

- We proved that it is possible to apply DRL to the vehicle management problem. From this, there is room for improvement, especially for managing complex scenarios with more AGVs and stations.
- Beyond the scope of this paper, there exist various avenues for further exploration. For instance, we hypothesize that a DRL approach increases stability in the system. This is suggested by the fact the DDQN agent is robust against the source clock, while static agents need additional precautions. We also performed simulations in which we introduced noise (in the source clock, the transports, the parts transfers and the processing). The trained agents tend to be more robust in those cases, avoiding deadlocks.

With these findings, we have laid out a foundation for future studies that can build upon these findings. Both, the agents, and the environment, can be improved to facilitate environments of a richer variety, and particular focus can be dedicated to further investigating the robustness of the DRL agents in comparison to the heuristic approaches. Also, the issue of dead-lock avoidance could be investigated more thoroughly. As for the environment and apart from increasing the complexity by means of adding stations, a more diverse work-flow, or more AGVs, one could furthermore envision an environment in which the AGVs themselves are controlled by DRL agents, considering additional factors such as battery status, nearest charging station, position of the other AGVs, etc. However, all these avenues are left unaddressed in this study,

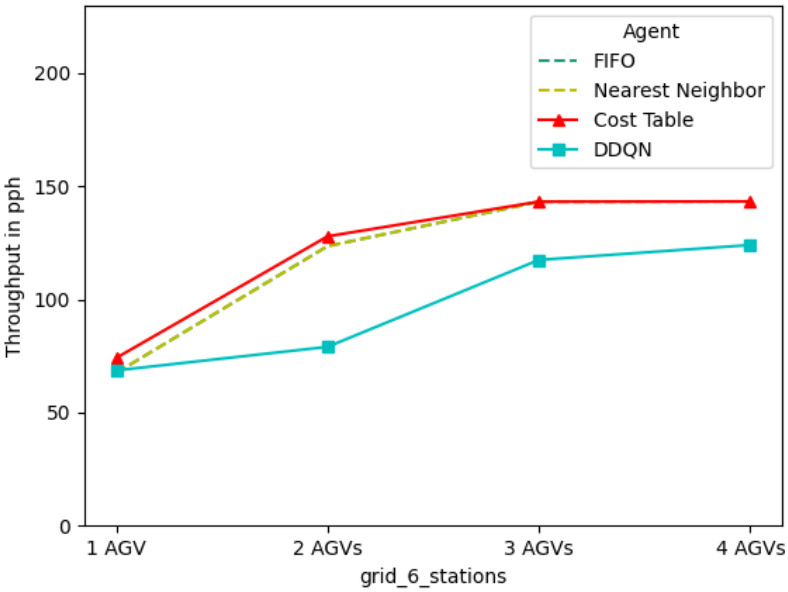

Fig. 8 Results on the `grid-6-machines` environment.

as the purpose of this paper is to establish a first, preliminary exploration of the feasibility of Deep-Q based DRL approaches in modular production environments.

6 Acknowledgements and references

Partial funding through the Austrian Research Promotion Agency FFG, Kleinprojekt Nr. 883243, is kindly acknowledged. Parts of the findings of this article are also given in [**?**].