REAL-TIME DECISION MAKING FOR A CAR MANUFACTURING PROCESS USING DEEP REINFORCEMENT LEARNING

Timo Philipp Gros
Joschka Groß
Verena Wolf

Saarland Informatics Campus
Saarland University
66123 Saarbrücken, Germany

ABSTRACT

Computer simulations of manufacturing processes are in widespread use for optimizing production planning and order processing. If unforeseeable events are common, real-time decisions are necessary to maximize the performance of the manufacturing process. Pre-trained AI-based decision support offers promising opportunities for such time-critical production processes.

Here, we explore the effectiveness of deep reinforcement learning for real-time decision making in a car manufacturing process. We combine a simulation model of a central production part, the after-paint line buffer, with deep reinforcement learning algorithms.

We simulate two different versions of the buffer, a single-agent and a multi-agent one, to generate large amounts of data and train neural networks to represent near-optimal strategies. Our results show that deep reinforcement learning performs extremely well and the resulting strategies provide near-optimal decisions in real-time, while alternative approaches are either slow or give strategies of poor quality.

1 INTRODUCTION

A major goal of computer simulations of manufacturing processes is the identification of bottlenecks and the analysis of what-if scenarios. In many cases, the goal is to improve the performance of the production process by ensuring a smooth production flow. For instance, unfavorable production sequences, machine breakdowns, or missing supply parts can lead to production stops in large parts of an automotive plant and induce high costs. Moreover, increasing customer orientation results in higher product diversification which then increases the variation in the workload at the different assembly stations. Many of such problems, in particular the order-sequence problem, can be formulated as sequential decision making problems and mathematically correspond to a Markov decision process (MDP), for which an optimal strategy needs to be found.

Here, we explore the effectiveness of deep reinforcement learning (DRL) for decision making in a car manufacturing process by combining a process simulation model of a central production part, the after-paint line buffer. We propose the use of deep reinforcement learning algorithms to enable decision making in real-time and thus, provide decision support even when unforeseen events occur.

We consider a concrete car plant where currently the re-ordering of cars at the after-paint line buffer is based on human decisions. Production stops because of overloaded assembly stations are common. We model the after-paint line buffer as an MDP and study in detail different approaches for solving this MDP with state-of-the-art deep reinforcement learning algorithms.

The idea of reinforcement learning goes back to the way animals and humans learn through interaction with their environment. Having no knowledge about the environment at all, they interact with it and

learn following the principle "do more of what was good and less of what was bad". The combination of reinforcement learning with deep neural networks has led to a major breakthrough in recent years in many challenging domains. Deep reinforcement learning has been successfully applied to Atari games (Mnih et al. 2015; Mnih et al. 2013) and the games Go and Chess (Silver et al. 2016; Silver et al. 2017; Silver et al. 2018). Successful results have also been obtained for problems of combinatorial optimization such as solving Rubic's cube (Agostinelli et al. 2019), vehicle routing (Nazari et al. 2018), or the traveling salesman problem (Kool et al. 2018). Deep reinforcement learning has also been applied successfully to scheduling problems, such as resource management (Mao et al. 2016; Chen et al. 2017) or global production scheduling (Waschneck et al. 2018). Here, we first investigate the effectiveness of deep Q-learning for optimal decision making at the line buffer. At the entry of the buffer, for each car a line has to be chosen. Similarly, at the buffer exit, the line from which the next car is transported to the final assembly unit has to be chosen.

In addition, we also apply a variant of Monte Carlo tree search (MCTS). In contrast to deep Q-learning, Monte Carlo tree search is not based on thousands of training episodes and an approximation of a value function, but on the idea to run several simulations starting from the current state, whenever a decision has to be made. It is well known to be efficient for sequential decision making (Sutton and Barto 2018). Further, MCTS has also successfully been used in general game playing (Finnsson and Björnsson 2008; Genesereth and Thielscher 2014). We improve MCTS by integrating pre-trained deep Q-learning networks as experts. We compare the performance of all DRL approaches to that of suitable heuristics and a look ahead search, which is based on the well-known planning algorithm A^*. To systematically test the performance of different approaches, we vary the size of the buffer and the size of the sequence window at the entry of the buffer. We also consider different multi-agent reinforcement learning approaches to cope with the problem that decisions are necessary at the exit and entry of the buffer. We find that sophisticated adaptations of DRL algorithms perform extremely well for the decision making problem at the line buffer. The quality of the resulting strategies is not only very close to that of the look ahead search, but DRL also provides near-optimal decisions in real-time whereas the look ahead search becomes slow when the complexity of the problem increases. Our implementation and all of our experiments can be found at https://mgit.cs.uni-saarland.de/timopgros/carmanufacturing.

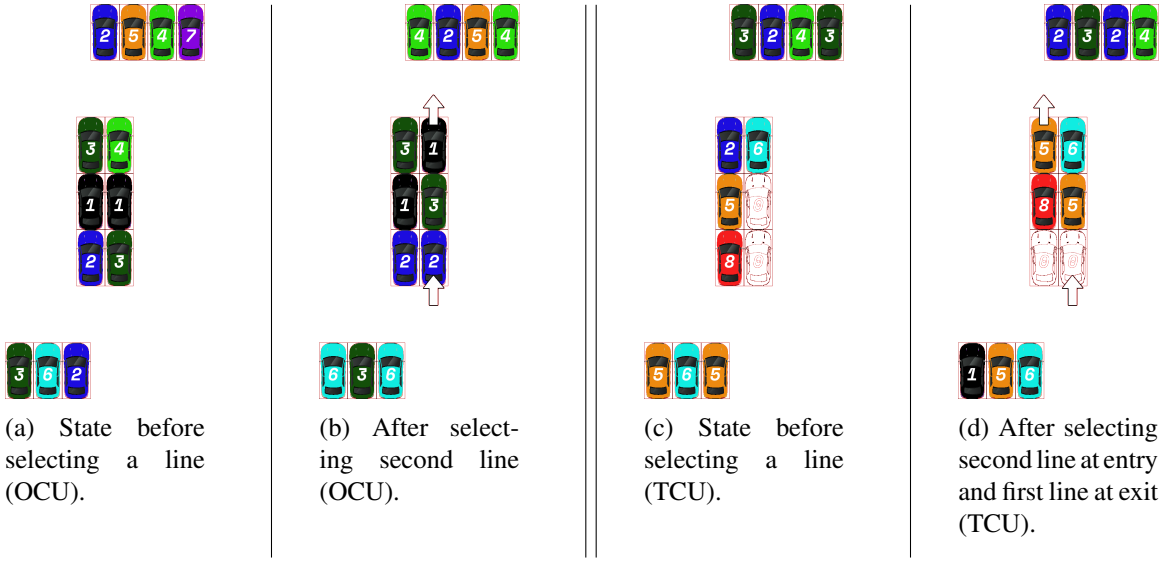
The remainder of the paper is structured as follows: We describe our model in Section 2 and propose two deep reinforcement learning approaches in Section 3. We introduce heuristics and the look ahead search in Section 4 and provide the results of a comparison between these and our approaches in Section 5. We finally draw a conclusion and give an outlook on future work in Section 6.

2 MODEL DESCRIPTION

2.1 Car Manufacturing Process

In this section we consider an important part of the decision-making process in a concrete German car plant, which is about 50 years old and currently mostly relies on human decisions for optimization. The production line starts from the chassis and body of the cars, continues with a paint unit, and ends with the final assembly. The after-paint buffer, which consists of multiple lines, connects the paint and final assembly unit. As the final assembly is a bottle neck of the production, the order of the cars leaving the buffer plays a crucial role for the global performance of the plant. A rearrangement of the order in real-time can significantly improve the throughput of the final assembly unit and prevent production stops triggered by unexpected time delays at assembly stations.

The after-paint buffer consists of n different lines. Hence, for each car leaving the paint unit, one of the n different buffer lines is chosen at the entry of the buffer. Within a line cars leave according to FCFS, i.e. a car can only be taken out from the end of a line and, conversely, can only be put into the beginning of a line. Hence, there exist n different possibilities (assuming all lines have space left) for choosing a line for a car entering the buffer and n possibilities (assuming that all lines are non-empty) for choosing the



(a) State before selecting a line (OCU).

(b) After selecting second line (OCU).

(c) State before selecting a line (TCU).

(d) After selecting second line at entry and first line at exit (TCU).

Figure 1: Example of both line buffer variants performing a transition. The left part (a,b) displays the OCU model and the right part (c,d) displays the TCU model.

next car that leaves the buffer. An illustration of the buffer is given in Figure 1, where the 6 slots (with 2 lines of capacity 3 each) in the center of each picture represent the buffer and the cars at the bottom are those that will enter the buffer next while the cars on the top are those that just left the buffer and will enter the final assembly unit next (in the respective order).

In order to evaluate the decisions at the entry and exit of the line buffer, a discrete-event simulation of the different assembly stations after the buffer is needed. Here, we concentrate on the decision-making part at the line buffer and use a set of rules to evaluate buffer decisions. These rules are similar to those that are currently used in the real plant and were derived based on a queuing model of the final assembly unit. They are based on certain car features and each rule determines how many cars with the same features are allowed in a certain window of the production sequence. If a rule is violated, delays or even stops at the final assembly will occur. Currently, in the plant the violation of only a small subset of these rules is checked by factory workers maintaining a tally list. The goal is to replace these human decisions by an automated decision system based on reinforcement learning that suggests near-optimal decisions at the entry and exit of the buffer.

It is important to note that decisions at the line buffer are needed in real-time, because cars that leave the paint unit and appear at the entry of the buffer arrive in an order that does not correspond to the original two-week-plan of the plant. The reason for this is that unforeseeable permutations of cars occur within the body and paint units.

2.2 Control Units

In the sequel, we will consider two different models. A simpler model with a single control unit, where only one line decision has to be made, and a more complex model, in which lines are chosen at the entry and exit of the buffer:

One Control Unit (OCU). In this variant of the model, we assume that the buffer is always filled and there are no empty spots. We restrict the decision making to a central control unit, selecting a line that is used for both, car removal and insertion. That is to say, if a car is put into the beginning of a line, the last car is automatically taken out (see Figure 1, (a) and (b)).

Two Control Units (TCU). With allowing distributed decision making, we assume one control unit at the entry and one at the exit of the buffer. Thus, for every step, two decisions have to be made, which

can be seen as decisions of two "agents": One agent has to decide which car is taken out at the exit of the buffer and another agent chooses a line at the entry of the buffer for the next car (see Figure 1, (c) and (d)).

2.3 Markov Decision Process

We model the decision-making process at the line buffer as a Markov decision process and consider T different types of cars. In the sequel, we sketch the different components of the MDP that describes the line buffer. For a formal and detailed definition of the transition probabilities of the MDP we refer to Appendix A.

States. To describe the current state of the system, we only monitor the type of each car in the following three parts of the system: (i) the input sequence of cars, (ii) the buffer, and (iii) the output sequence of cars.

The input and output sequence are one-dimensional lines of cars that have to be sorted into the buffer and cars that have already left the buffer. All of these lines in the buffer have the same capacity. If a line is empty, no car can be taken out and if a line is already filled, no car can be put in. The cars are always moved to the last free spot of a line, i.e. there are no empty spots between cars within one line but only in the beginning.

Transitions. A transition of the process corresponds to two consecutive steps in the real system: (1) the last car of the line chosen for output leaves the corresponding line in the buffer and appears at the front of the output sequence and the last car of the output sequence is dropped, (2) the car at the front of the input sequence enters the buffer line chosen for input; the input sequence is complemented with a new car (we choose uniformly among the T types).

The order of these two steps ensures that a car just chosen from the input sequence cannot be taken out in the same step as cars have to be physically moved from the entry of the buffer to the exit. However, the opposite, inserting a car into a line that is full in the beginning of the step and a car was just taken out of, is possible.

Rewards. The rules that ensure optimal throughput at the final assembly are basically distance rules, i.e. the lengths of the sequences until a car type is allowed to reappear in the output sequence. For each type, we choose a minimal distance and if in the output sequence a smaller distance appears for a certain car type, a negative reward is given. The value of this penalty is also car-type specific. After every transition, the output sequence is modified in the just specified way. With that altered output sequence, violation of the rules and therefore resulting penalties can be determined.

Variations. The length of the input (L_I) and output sequence (L_O), as well as number (n) and capacity (c) of the buffer lines are variables of our model and can be varied. Note that a larger window of the input and output sequence increases the information at the decision point and therefore theoretically allows to learn better strategies. We also allow to vary the number of car types (T) (i.e. distinguishable cars w.r.t. the given rules) and the initial ratio (i_R) of filled and empty spots in the buffer. Each of these parameters gives options to increase the complexity of the problem.

3 DEEP REINFORCEMENT LEARNING

Our goal is to use deep reinforcement learning to train one agent (OCU model) or two agents (TCU model) for decision making. As we want our agents to decide as optimal as possible, they aim to maximize the expected cumulative reward of the MDP's episodes. As (ideally) the manufacturing process runs nonstop 24 hours per day, the task is a continuing one (Sutton and Barto 2018) and the accumulated future reward, the so called *return*, of step t is therefore given by $G_t = \sum_{i=t}^{\infty} \gamma^i \cdot R_{i+1}$, where we assume that R_{i+1} is the reward obtained during the transition from the state S_i of the process at time i to state S_{i+1} for $i \in \{0,1,\ldots\}$ and γ is a discount factor with $\gamma \in [0,1]$ (Sutton and Barto 2018). We adapt two different deep reinforcement learning approaches to train decision making agents.

3.1 Deep Q-learning

Deep Q-learning is based on the idea to build a neural network that, for all states and available actions, approximates action-values, which can be used to find the best action for a given state and thus the optimal policy. For a fixed policy π, a state s, and an action a, the *action-value* $q_\pi(s,a)$ gives the expected return that is achieved by taking action a in state s and following the policy π afterwards, i.e.

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \,|\, S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right].$$

If the policy π is optimal, i.e. it maximizes the expected return, then we write $q_*(s,a)$ for the *optimal action-value*. Intuitively, the optimal action-value $q_*(s,a)$ is equal to the expected sum of the reward that we receive when taking action a from state s, and the (discounted) highest optimal action-value that we receive afterwards, which gives the Bellmann optimality equation (Mnih et al. 2015)

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \cdot \max_{a'} q_*\left(S_{t+1}, a'\right) | \, S_t = s, A_t = a\right],$$

where A_t is the action chosen in step t. In the following, the terms optimal action-value function, value function and Q-value will be used interchangeably.

The idea of *value-based* reinforcement learning methods is to find an estimate $Q(s,a)$ of the optimal action-value function. The simplest approach is to use the Bellmann equation to iteratively update the estimation. For a given observed state $S_{t+1} = s'$ and reward $R_{t+1} = r$, the expectation is then estimated by successively adjusting the current Q-value

$$Q_{i+1}(s,a) = Q_i(s,a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'} Q_i(s',a') - Q_i(s,a)\right),$$

where $\alpha \in (0,1)$ is the learning rate.

When the state space is too large for using tables that store $Q(s,a)$ and learning each individual $Q(s,a)$ takes far too long, function approximations provide a suitable representation of $Q(s,a)$. Artificial neural networks have become popular for function approximation since they can express complex non-linear relationships and are able to generalize. We consider a neural network with weights θ estimating the Q-value function as a deep Q-network (DQN) (Mnih et al. 2013). We denote this Q-value approximation by $Q(s,a;\theta)$ and optimize the network w.r.t. the target

$$y(s,a;\theta) = \mathbb{E}\left[R_{t+1} + \gamma \cdot \max_{a'} Q(S_{t+1}, a'; \theta) \mid S_t = s, A_t = a\right]. \tag{1}$$

Hence, the corresponding loss function in iteration i is

$$L(\theta_i) = \mathbb{E}\left[\left(y(S_t, A_t; \theta^-) - Q(S_t, A_t; \theta_i)\right)^2\right]. \tag{2}$$

where θ^- refers to the parameters from some previous iteration. We approximate $\nabla L(\theta_i)$ and optimize the loss function by stochastic gradient descent (Mnih et al. 2015). To avoid an unstable training procedure, we use a *fixed target* (Mnih et al. 2015), i.e. $y(S_t, A_t; \theta^-)$ does not depend on θ_i but corresponds to the weights that were stored C steps earlier in the iteration. Also, the target network is updated by performing a soft update, i.e. $\theta^- = (1-\tau)\cdot\theta + \tau \cdot \theta^- p$ with $\tau \in (0,1)$.

Another DQN improvement that we apply is *experience replay* (Mnih et al. 2015). The standard assumption for stochastic gradient descent is that the samples are independent and identically distributed. When learning from sequences (trajectories of the MDP), this assumption is violated as the consecutive states depend on former action choices. Therefore, instead of directly learning from observed behavior, we store all experience tuples $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ in a data set D. When adapting the network weights, we sample uniformly from that buffer to decorrelate the samples from which we learn, i.e. in Eq. (1) and (2) we estimate the expectation w.r.t. $S_t, A_t, R_{t+1}, S_{t+1} \sim \mathrm{Unif}(D)$. We remark also that experience replay has greater data efficiency.

We generate our experience tuples by exploring the state space epsilon-greedily, that is, with a chance of $1-\varepsilon$ during the Monte Carlo simulation we follow the policy that is implied by the current network weights and otherwise choose a random action (Mnih et al. 2015).

3.2 Deep Q-learning for the Line Buffer

When we combine Monte Carlo simulations of the line buffer MDP with deep Q-learning, several challenges arise. A suitable encoding for the MDP state has to be found as well as suitable layers for the neural network. Further challenges are related to the balance between exploration and exploitation and in the case of the TCU model, a training strategy for two cooperating agents has to be developed.

3.2.1 One Control Unit.

The OCU model relies on a single agent for decision making. Hence, a single neural network is trained which gets the state s and an action a of the MDP as an input. We considered different window sizes for the input sequence at the entry of the buffer and different buffer sizes. To encode s we use a one hot encoding, which is a popular boolean vector representation in the context of deep learning. For each slot in and around the buffer and each car type, the entry 1 at the corresponding position of the input vector indicates that there is a car of a certain type (otherwise the entry is 0). Additionally, we one hot encode whether there is a car at all. Note that integer values for encoding the car type are harder to process by the network than boolean values.

We use fully connected layers in our neural network because more sophisticated network structures are only needed for complex inputs such as images. After testing several different network structures we found that 4 hidden layers with 128 nodes each are most efficient for our case study in terms of training time and quality of the solution. As we varied the window sizes of the entry as well as the buffer size, the size of the input layer of the network depends on these parameters, while the size of the output layer is equal to the selected number of lines.

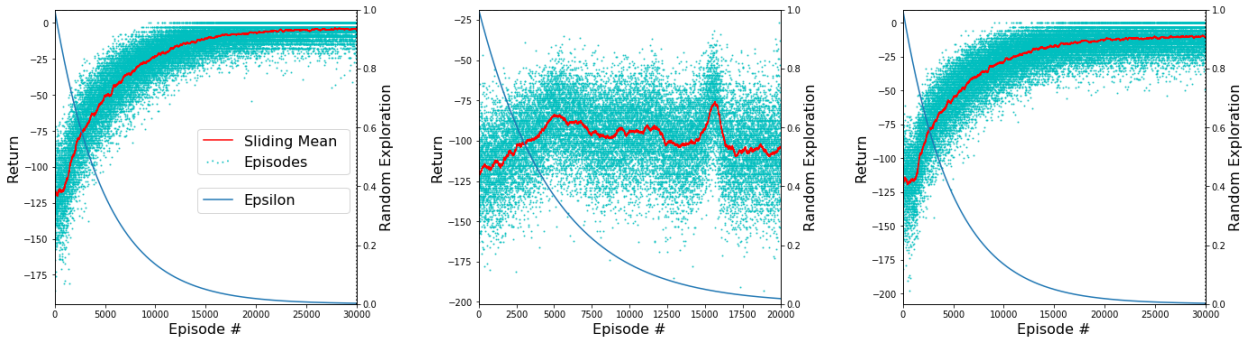
Although our manufacturing process technically is a continuing model, we turn it into an episodic task by organizing the training in episodes with 100 cars to be inserted in each episode. We trained our agents by playing $30,000$ episodes without prior knowledge. Therefore, at the beginning it is useful to act more exploratively, while during the training exploitation is growing in importance. Thus, we do not use an exploration constant ε, but decrease ε during training. In beginning of the training we set $\varepsilon_0 = \varepsilon_{start}$ and decrease it by a constant factor $\lambda < 1$ in every episode i until a threshold ε_{end} is reached, i.e. $\varepsilon_{i+1} = \max(\varepsilon_i \cdot \lambda, \varepsilon_{end})$. This results in a training process, where the focus lies on exploration in the beginning and on exploitation in the end. Our training with the specified network structure and number of episodes can be done within a few hours. We plot the training progress and the selected ε for a buffer with four lines and $L_I = 7$ in Figure 2 (a). Each point in the plot refers to one training episode.

3.2.2 Two Control Units

For the TCU model, we have to consider two control units for distributed decision making. Hence, we additionally need to handle training of two agents cooperating with each other. We present three different approaches to tackle this issue.

Vanilla Multi-Agent DQN. The easiest approach is to simply try to train two agents simultaneously. One of the agents is responsible for inserting into, the other agent for removing cars from the buffer. However, the DQN approaches that we use work well under the assumption that the environment does not change, i.e. the transition probabilities of the MDP do not change. But if another agent is involved, state transitions and rewards are affected and the environment evolves dynamically. Hence the agent must keep track of the other learning agents, possibly resulting in an ever-moving target (Busoniu et al. 2008; Schwartz 2014). For some training runs, e.g. the one displayed in Figure 2 (b), the plots suggest, that the agent is not learning anything at all.

Curriculum Reinforcement Learning (CC). To handle the former, we roughly follow the idea of *curriculum learning* (Bengio et al. 2009). In contrast to Gupta et al. (2017) we neither define the curriculum by increasing the number of agents, nor do we increase the complexity environment. Instead, we create the following *iterative curriculum*: to address the problem of changing environments, we alternate the training



(a) SCU. (b) TCU: Vanilla MA. (c) TCU: CP.

Figure 2: Training for $L_I = 7$ and $n = 4$ for both, OCU and TCU.

of the agent at the entry and exit of the buffer. When one agent is trained, the other one decides according to a fixed strategy. In the first iteration, we start by training the agent at the exit and use a random strategy for the agent at the entry. In following iterations, we use the policy that was just trained in the former iteration.

Cross Product Learning (CP). Instead of training two agents for distributed decision making, we can also train an agent that decides both, the line at the entry and the line at the exit. For this, we enlarge the input of the neural network. The number of actions increases to n^2, where n is the number of lines in the buffer. As DRL is known to not scale well with the number of available actions, we expect the performance of this approach to be good as long as the number of lines is small but to strongly decrease with an increasing number of lines. The achieved results in training look similar to the DQN for OCU (see Figure 2 (c)).

3.3 Monte Carlo Tree Search

MCTS is based on the idea to build a search tree, starting with the current state as root, and to run several simulations, whenever a decision is about to be made.

Balancing between exploration of the state space and exploitation of the return MCTS follows a certain branch of the tree. Every node of the tree consists of the visit count N, the average of the observed returns V, and the exploration/exploitation coefficient U. The approach is performed in four different steps (Sutton and Barto 2018). (i) Selection: starting in the current state, traverse the tree, i.e. choose actions and execute them, until a leaf of the node is reached. (ii) Expansion: if the leaf is not reached for the first time, expand the leaf by adding all possible children. Choose one of the new leaves. (iii) Simulation: from the selected leaf node run a simulation following a rollout policy until a terminal state is reached. (iv) Back propagation: traverse the tree backwards, updating the average return and the visit counts.

When the limit of the tree depths is reached, which can either be specified by visits or by time, MCTS chooses the action that was visited most often (and not, as may be expected, the one with the highest average value) (Silver et al. 2017). The child of the tree can be set as a new root for upcoming decisions to be made.

In order to keep the balance between exploration and exploitation, Schadd et al. (2008) adjust the standard UCT selection coefficient (Kocsis and Szepesvári 2006) such that it is more suitable for environments without adversaries and rewards within larger intervals other than $[-1,1]$. Building on their approach and making some changes on our own, we decided to use $U = V_i + R_i + 0.5 \cdot \sqrt{\log N / N_i}$, where N_i is the children's visit count ($N = \sum_i N_i$) and R_i the reward obtained reaching it.

3.4 MCTS for the Line Buffer

For the combination of the line buffer MDP and MCTS, we do not need to distinguish between the different control units. The tree-structure that is build to decide on the next action is capable of switching between insertion and removal.

For the expansion, we use the following strategy: When the selected input depth, that is equal to the length of the input sequence L_I, is reached, we stop expanding the tree. It follows that the tree is at most as deep as the window of the input sequence.

While classical MCTS then applies a rollout policy as soon as a leaf is visited for the first time, we instead make use of our already trained deep Q-networks. Whenever a leaf is reached, we do not perform a rollout but instead use the pre-trained DQN as an expert to estimate the expected return from this state s on, i.e. $\max_{a'} Q(s,a')$. This idea is similar to the AlphaGo Zero approach (Silver et al. 2017), except that our expert network was trained prior with deep Q-learning and does not change during MCTS. We propose two different versions for this DQN query.

MCTS-Expert (MCTS$_E$). While traversing the tree, we randomly choose a type for the next unknown car of the input sequence. Hence, the DQN-estimates that we use within the tree depend on different random (completions of the) input sequences.

MCTS-Expert$^+$ (MCTS$_E^+$). We fill the input sequence with the cars of the true input sequence of the current episode, i.e. we assume here that we have more information about the input sequence than the information given by the current MDP state. Hence, we expect MCTS$_E^+$ to perform better than used expert networks. In comparison to classical MCTS, our approach based on a DQN query saves time that can now be spent on exploring and building the tree instead of performing Monte Carlo simulations for rollout. However, its disadvantage is that poor decisions of the deep Q-network partially carry over to the MCTS algorithm.

We restrict MCTS to a maximal number of visits and a maximal computation time. Whenever one of both criteria is met, we stop the tree search and return the current result.

4 HEURISTICS AND LOOK AHEAD SEARCH

In this section, we present two alternative approaches for optimization. One is based on heuristics and the other one is based on a look ahead search and used to determine an accurate approximation of the optimal solution, which is time consuming but useful for a comparison with the DRL results.

4.1 Heuristics

We propose heuristics with the goal of providing a baseline (random heuristic) and an approach that is similar to human decision making. We therefore compare our results to the following heuristics.

Random Heuristic (RH). Randomly select a valid line of the buffer.

Greatest Distance Out Heuristic (GDH). Randomly select among the lines, where the to be removed car has the maximal distance to other cars with the same type in the output sequence.

Greatest Similarity In Heuristic (GSH). Randomly insert the car among the lines that have the highest number of cars with the same type.

Sorting Heuristic (SH). Insert the car according to a pre-defined ordering. If not possible, insert it into the next higher available line.

For OCU, we compare our results to RH and GDH. For the TCU setting, we use RH and GDH at the buffer entry and RH, GSH, and SH at the exit. We compare our DRL approaches to all 6 possible combinations.

4.2 Look Ahead Search (LAS)

Computing the optimal strategy for the MDP is difficult because standard approaches such as value iteration are infeasible. To approximate the optimal solution, we define a *look ahead search*. For a given length of the input sequence window, it computes the optimal decision as follows. It considers all possible action sequences (decisions at the buffer entry and exit) until the given input sequence is completely inserted into the buffer. It then selects the best action for the next step only. After inserting one car, the input line is filled again and new information is available. Therefore, the look ahead search can be applied again to select the next best decision given the current information.

Our look ahead search basically is a reimplementation of the well known planning algorithm A^*, tailored to our plant model without further heuristics. For the given input sequence, we create an optimal plan according to the reward function and return the first action of that plan. Needless to say that the runtime is growing exponentially with increasing number of lines and increasing size of input sequence.

5 EVALUATION

To evaluate our trained decision making agents, we sample $1,000$ episodes with a sequence length of 100 cars. For all experiments, we use $T = 8$ car types, a line capacity of $c = 3$, and an output sequence with length $L_O = 4$. We run experiments for $n = 2$ and $n = 4$ and vary the length of the input windows $L_I \in \{1,3,5,7,9\}$. All experiments were made for both variants of the model (OCU, TCU). For TCU we used $i_R = 0.6$. All measurements were made on a machine with an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz processor with 16 GB of RAM.

Additionally to comparing our agents to heuristics and LAS, we applied LAS with an increasing input window length to approximate the optimal strategy more accurate. We report the maximal possible window size (we write LAS_X for a maximal size X) until the computation timed out (0.5s per decision).

5.1 One Control Unit

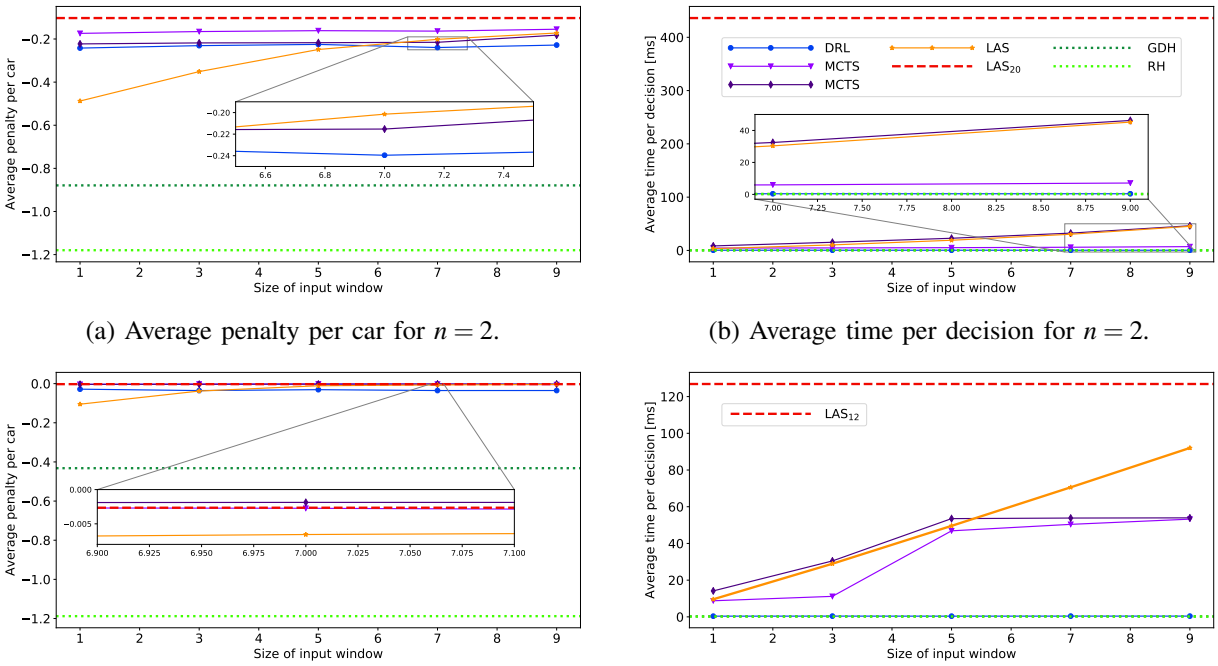
In Figure 3 we plot the penalty per car and the time per decision averaged over the $100,000$ cars of all episodes, where the decisions are based on a pre-trained neural network as explained in the previous section. We show results for increasing window sizes of the input sequence and consider $n = 2$ and $n = 4$.

DQN. For smaller window sizes, DQN performs better than LAS and only slightly worse otherwise, because DQN is trained on a model where the next car of the input sequence is chosen uniformly while LAS does not look any further than the input window. It clearly outperforms all heuristics, but taking only slightly more time while the runtime of LAS increases dramatically with the input window size.

MCTS_E. For $n = 2$ MCTS_E is performing similar to DQN, having a slightly increased performance, especially for $L_I \in \{7,9\}$. The runtime increases similarly to LAS. For $n = 4$, MCTS_E is not only constantly outperforming LAS, but even accomplishes better scores than the best possible LAS (LAS_{12}). The runtime increases until 50 ms and remains at this value afterwards, due to the time exploration limit. For small windows, it is further taking more time to decide than LAS. For some input sequence lengths, MCTS_E obtains the best results of all approaches.

MCTS_E^+. For $n = 2$, MCTS_E^+ is clearly performing best but does not reach the approximated maximal score of LAS_{20}. For $n = 4$, both MCTS_E^+ and MCTS_E perform better than LAS_{12}. Similar to MCTS_E, runtime is increasing with input length until reaching 50 ms.

Considering the runtime overall, we can see that computing LAS_{20} lasts several times the time that the other algorithms take. While the runtime of LAS increases with input sequence, MCTS_E and MCTS_E^+ have a runtime upper bound and DQN as well as the heuristics can decide nearly instantaneously.



(a) Average penalty per car for $n = 2$.

(b) Average time per decision for $n = 2$.

(c) Average penalty per car for $n = 4$.

(d) Average time per decision for $n = 4$.

Figure 3: Results for the OCU model.

5.2 Two Control Units

In the multi-agent setting of TCU, we see in Figure 4 that the approaches for training deep Q-networks show different performances.

Vanilla Multi-Agent DQN. Considering the easiest approach to train two agents simultaneously, we see that the training is very unstable (solid blue line DQN), which was already apparent from Figure 2. The agents partly even perform worse than the heuristics. From the return of the training episodes, we conclude that further training is necessary to reach a better performance, if so at all. After training, the runtime ist negligible compared to the heuristics.

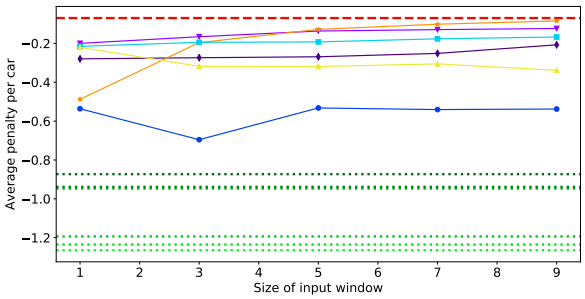
Curriculum Reinforcement Learning. Applying an iterative curriculum gave very good results. The agents clearly outperform all other DQNs and perform similarly to LAS with the same information depth for both examined number of lines. The decision time per car is roughly equal to vanilla DQN.

Cross Product Learning. Already for $n = 2$, the CP-approach does not exploit the additional information from the increasing window size as its performance decreases for larger windows. It performs better for $n = 4$ than for $n = 2$ since the task of reordering 8 different car types is much easier with 4 lines than with 2. Still, CP outperforms the heuristics by far and the runtime is slightly higher in comparison to vanilla DQN.

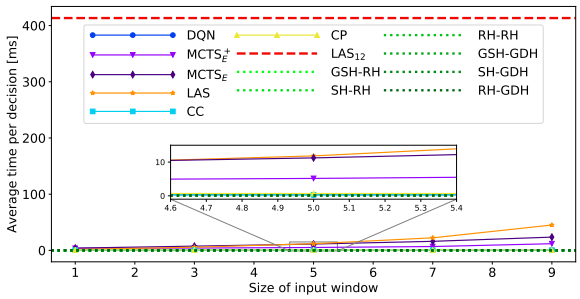
MCTS$_E$. For both MCTS approaches, we use the networks of curriculum learning as experts, as they were the best of the deep Q-networks. In contrast to OCU, for $n = 2$ MCTS$_E$ is not achieving better results than the CC-network. This is particularly surprising, as that same network is used as an expert. Still, the performance is better than CP trained agents.

MCTS$_E^+$. Just as for the OCU, MCTS$_E^+$ overall achieves the best results. For $n = 2$ it is in parts slightly worse than LAS, while for $n = 4$ it is again similar to our best approximation LAS$_7$.

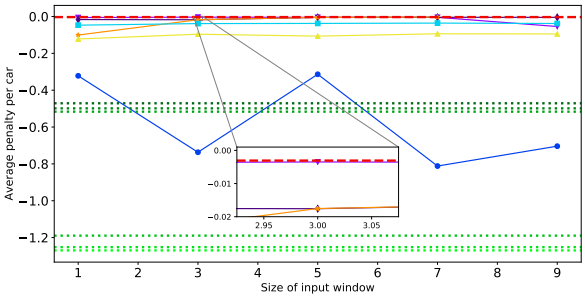
For both MCTS approaches, the runtime is increasing with input size. For small windows, it is further taking more time to decide than LAS. Due to the time exploration limit, time is not further increasing after reaching the upper bound.



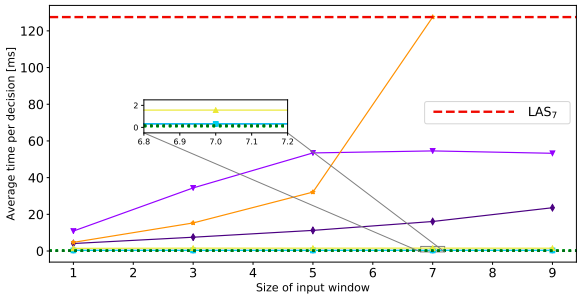
(a) Average penalty per car for $n = 2$.



(b) Average time per decision for $n = 2$.



(c) Average penalty per car for $n = 4$.



(d) Average time per decision for $n = 4$.

Figure 4: Results for the TCU model.

6 FUTURE WORK AND CONCLUSION

We investigated the usefulness of deep reinforcement learning approaches in the context of a car manufacturing process, for which an optimal sequence order is crucial. By simulating an MDP model of the after-paint buffer in order to train a deep Q-network, we obtained near-optimal decisions at the entry and exit of the buffer in real-time. For our comparison, we implemented a look ahead search, which is computationally expensive but yields an accurate approximation of the optimal strategy.

We also proposed a combination of deep Q-networks and Monte Carlo tree search, which is more costly than deep Q-learning, but yields strategies of even higher quality and can be seen as a method that lies between a costly planning procedure and pre-trained Q-networks.

We remark that our proposed reinforcement learning approaches are able to handle unexpected changes in the production process as long as the corresponding state is known from training.

As future work, we plan to systematically explore the performance of DRL for manufacturing processes with multiple agents for decision making arranged hierarchically or along the production line. We expect that clever training strategies of multiple agents using ideas from hierarchical reinforcement learning (Ahilan and Dayan 2019) will yield efficient learning procedures for such complex production systems.

Another line of future work is related to the integration of deep reinforcement learning approaches into complex discrete-event simulations such as a detailed simulation of other plant units (e.g. the final assembly unit). We believe that deep reinforcement learning is a powerful tool for complex sequential decision making problems emerging in computer simulations of real-world manufacturing applications that require real-time decisions.

REFERENCES

Agostinelli, F., S. McAleer, A. Shmakov, and P. Baldi. 2019. "Solving the Rubik's cube with deep reinforcement learning and search". *Nature Machine Intelligence*.

Ahilan, S., and P. Dayan. 2019. "Feudal multi-agent hierarchies for cooperative reinforcement learning". *arXiv preprint arXiv:1901.08492*.

Bengio, Y., J. Louradour, R. Collobert, and J. Weston. 2009. "Curriculum learning". In *Proceedings of the 26th annual international conference on machine learning*, 41–48.

Busoniu, L., R. Babuska, and B. De Schutter. 2008. "A Comprehensive Survey of Multiagent Reinforcement Learning". *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 38(2):156–172.

Chen, W., Y. Xu, and X. Wu. 2017. "Deep reinforcement learning for multi-resource multi-machine job scheduling". *arXiv preprint arXiv:1711.07440*.

Finnsson, H., and Y. Björnsson. 2008. "Simulation-based approach to general game playing". *Proceedings of the National Conference on Artificial Intelligence* 1:259–264.

Genesereth, M., and M. Thielscher. 2014. "General game playing". *Synthesis Lectures on Artificial Intelligence and Machine Learning* 8(2):1–229.

Gupta, J. K., M. Egorov, and M. Kochenderfer. 2017. "Cooperative multi-agent control using deep reinforcement learning". In *International Conference on Autonomous Agents and Multiagent Systems*, 66–83. Springer.

Kocsis, L., and C. Szepesvári. 2006. "Bandit based monte-carlo planning". In *European conference on machine learning*, 282–293. Springer.

Kool, W., H. Van Hoof, and M. Welling. 2018. "Attention, learn to solve routing problems!". *arXiv preprint arXiv:1803.08475*.

Mao, H., M. Alizadeh, I. Menache, and S. Kandula. 2016. "Resource management with deep reinforcement learning". In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 50–56.

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning". *CoRR* abs/1312.5602.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. 2015, feb. "Human-level control through deep reinforcement learning". *Nature* 518:529.

Nazari, M., A. Oroojlooy, L. Snyder, and M. Takác. 2018. "Reinforcement learning for solving the vehicle routing problem". In *Advances in Neural Information Processing Systems*, 9839–9849.

Schadd, M. P., M. H. Winands, H. J. Van Den Herik, G. M.-B. Chaslot, and J. W. Uiterwijk. 2008. "Single-player monte-carlo tree search". In *International Conference on Computers and Games*, 1–12. Springer.

Schwartz, H. M. 2014. *Multi-agent machine learning: A reinforcement approach*. John Wiley & Sons.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. 2016. "Mastering the Game of Go with Deep Neural Networks and Tree Search". *Nature* 529:484–503.

Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel et al. 2017. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm". *arXiv preprint arXiv:1712.01815*.

Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. 2018. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". *Science* 362(6419):1140–1144.

Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. 2017. "Mastering the game of Go without human knowledge". *Nature* 550:354–359.

Sutton, R. S., and A. G. Barto. 2018. *Reinforcement Learning: An Introduction*. Second ed. The MIT Press.

Waschneck, B., A. Reichstaller, L. Belzner, T. Altenmüller, T. Bauernhansl, A. Knapp, and A. Kyek. 2018. "Optimization of global production scheduling with deep reinforcement learning". *Procedia CIRP* 72(1):1264–1269.

AUTHOR BIOGRAPHIES

Timo Philipp Gros is a PhD student of computer science and works at the chair of Modeling and Simulation at Saarland University. His email address is timopgros@cs.uni-saarland.de.

Joschka Groß is a master student of computer science at Saarland University. His email address is s8jagros@stud.uni-saarland.de.

Verena Wolf is a full professor at Saarland University and head of the chair of Modeling and Simulation on the Saarland Informatics Campus. She received her PhD in Computer Science from the University of Mannheim and was a postdoc at the École Polytechnique Fédérale de Lausanne (EPFL) before joining Saarland University. Her email address is verena.wolf@uni-saarland.de.

A Technical Report

Definition 1 (Markov Decision Process) A Markov Decision Process (MDP) is a tuple $\mathscr{M} = \langle \mathscr{S}, \mathscr{T}, \mathscr{R}, \mathscr{A}, \gamma \rangle$ consisting of a finite set of *states* $\mathscr{S}$, a finite set of *actions* $\mathscr{A}$, the *transition probability function* $\mathscr{T} : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \mapsto [0,1]$, the *reward function* $\mathscr{R} : \mathscr{S} \times \mathscr{A} \times \mathscr{S} \mapsto \mathbb{R}$, and a discount factor $\gamma \in [0,1]$

An action $a \in \mathscr{A}$ is *applicable* in a state $s \in \mathscr{S}$ if there exists $s' \in \mathscr{S}$ such that $\mathscr{T}(s,a,s') > 0$. We denote by $\mathscr{A}(s) \subseteq \mathscr{A}$ the set of all actions that are applicable in s. For every state $s \in \mathscr{S}$ and applicable action $a \in \mathscr{A}(s)$, $\mathscr{T}(s,a,s')$ must induce a probability distribution over the states in $\mathscr{S}$, i.e., it must hold that $\sum_{s' \in \mathscr{S}} \mathscr{T}(s,a,s') = 1$.

A *policy* for $\mathscr{M}$ is a function $\pi : \mathscr{S} \mapsto \mathscr{A}$ where, for every $s \in \mathscr{S}$, $\pi(s) \in \mathscr{A}(s)$.

The manufactory model is then formally given as an MDP by

- $\mathscr{S} = \{(I,B,O)\}$ where,
  - $I = (i_1, \ldots, i_{L_I})$ with $\forall h \in \{1, \ldots L_I\} : i_h \in \{1,2,\ldots,T\}$
  - $B = (l_1, \ldots, l_n) = \big((b_{1,1}, \ldots, b_{c,1}), \ldots (b_{1,n}, \ldots, b_{c,n})\big)$ with $b_{k,l} \in \{\perp\} \cup \{1,2,\ldots,T\}$
  - $O = (o_1, \ldots o_{L_O})$ with $\forall m \in \{1, \ldots, L_O\} : o_m \in \{1,2 \ldots, T\}$

  and
- $\mathscr{A} = \{1, \ldots, n\} \times \{1, \ldots, n\}$
- $\mathscr{T}\big((I, (l_1, \ldots, l_n), O), (m,h), \big(I', \big(l_1, \ldots, l'_h, \ldots, l'_m, \ldots l_n\big), O'\big)\big) = \frac{1}{T}$ with
  - $I = (i_1, \ldots i_{L_I})$ and $I' = (x, i_1, \ldots i_{L_I - 1})$ with $x \in \{1, \ldots, T\}$
  - for $m \neq h$:
    - $l_m = \big(b_{1,m} \ldots, b_{c,h}\big)$ and $l'_m = \big(\perp, b_{1,m} \ldots, b_{c-1,h}\big)$
    - $l_h = \big(b_{1,h}, \ldots, b_{c,h}\big)$ and $l'_h = \big(b_{1,h}, \ldots, b_{z-1,h}, i_{L_I}, b_{z+1,h}, \ldots b_{c,h}\big)$ with $\forall z \in \{1, \ldots, c\} : \forall k \in \{1, \ldots, z\} : b_{1,k} = \perp$ and $\forall k' \in \{z+1, \ldots, c\} : b_{1,k} \neq \perp$
  - for $m = h$:
    - $l_h = \big(b_{1,h}, \ldots, b_{c,h}\big)$ and $l'_h = \big(b_{1,h}, \ldots, b_{z-1,h}, i_{L_I}, b_{z+1,h}, \ldots b_{c-1,h}\big)$ with $\forall z \in \{1, \ldots, c-1\} :$ and $\forall k \in \{1, \ldots, z-1\} : b_{1,k} = \perp$ and $\forall k' \in \{z+1, \ldots, c-1\} : b_{1,k} \neq \perp$ and $b_{1,c} \neq \perp$
  - $O = (o_1, \ldots o_{L_O})$ and $O' = (b_{c,m}, o_1, \ldots o_{L_O - 1})$ with $b_{c,m} \neq \perp$.
- $\mathscr{R}((I, (l_1, \ldots, l_n), O), (m,h), (I', (l_1, \ldots, l'_a, \ldots, l_n), O')) = \mathrm{Pen}(O')$ with $\mathrm{Pen}(O') = \sum_{i=2}^{D[O[1]]+1} \mathrm{ID}(O[1], O[i])$ and $\mathrm{ID}(a,b) = \begin{cases} P[a] & \text{if } a = b \\ 0 & \text{else} \end{cases}$
- $\gamma = 1$.

Table 1: Summary of the simulation parameters.

Parameter	Abbreviation	Explanation
Car Types	T	Number of different car types
Input Sequence Length	L_I	Length of input sequence at buffer entry
Output Sequence Length	L_O	Length of the output sequence after buffer exit
Number of Lines	n	Number of lines in the buffer
Capacity of Lines	c	Number of cars that can be stored per buffer line
Initial Ratio	i_R	Initial ratio of filled to empty buffer spots
Distances	D	Array of size T with required minimal distances for every car type in the output sequence
Penalties	P	Array of size T with penalties for every car type in case the corresponding distance rule is violated