27th ITS World Congress, Hamburg, Germany, 11-15 October 2021

Paper ID 251

Allocation of container slots based on machine learning

Jan Niklas Sikorra[1], Dr. Wilfried Bohlken[1], Marlies Goes[1]

1. HPC Hamburg Port Consulting GmbH, Hamburg, Germany

Abstract

HPC Hamburg Port Consulting develops a software solution based on reinforcement learning for the automated allocation of container slots at terminals. The aim of the project is to use a two-step approach and develop first a demonstration model and then a prototype, which can then be used as an add-on module for Terminal Operating Systems (TOS).

Keywords: container stacking, stack optimisation, machine learning, reinforcement learning, TOS-add-on

Introduction

Today's slot allocation is based on complex rule-based algorithms that are optimized by manually setting geometries and parameters. These set parameters have to be manually readjusted on a regular basis. A self-learning slot allocation solution is both easy to customize and requires significantly less maintenance because it continuously optimizes itself. In addition, the simple design of the algorithm requires less initial effort and investment.

Container slot allocation, using a machine learning (ML) module, could offer advantages over current solutions not only in terms of operating cost savings and quality improvements. The full automation of yard allocation can greatly reduce the number of yard planners required. Simulation studies at HPC have shown that optimized stacking of containers can significantly reduce restacking in the container yard. Considering partial costs (operating only costs, without equipment) this results in significant potential annual savings. Additionally optimized container slot allocation reduces unnecessary restacking of containers and travel distances for port equipment, leading to significant energy savings. Furthermore, a more efficient storage system will lead to faster turnaround times for ships and trucks, reducing emissions during berth and waiting time, thereby enabling the green port of the future.

This paper describes a solution for container slot allocation in the yard of a container terminal, based on the semantics of OpenAI Gym for defining the solution space and using a deep Q-network. The implemented solution is validated on a modeled container yard, yet it has not been transferred to a real-world-sized container yard.

State of the art

When it comes to container stacking, there are two conflicting objectives to persue. On one hand, the

amount and density of the stored containers has to be maximized to use the available yard space efficiently. On the other hand, the number of unproductive shuffle moves should be as small as possible, which is harder to achieve with increasing storage density. To address this contradiction, containers have to be allocated to the slots in the yard as cleverly as possible by using the available information of the container (Kemme 2020).

One simple traditional strategy to approach the container stacking problem is to define categories for containers and stack items of the same category on top of each other. Another strategy is to only stack a container on top of ones with later departure times (Dekker et al. 2006). While these approaches appear promising in theory, they often fail in reality, due to poor data quality and too much computing effort to get the appropriate information on time (Kemme 2020).

Both issues can be addressed by using reinforcement learning (RL), a subdiscipline of machine learning, which simulates humans' and animals' learning behavior. The general idea of RL has been around for a while, but the combination of RL and neuronal networks only yielded an unstable learning success for a long time. With the introduction of the deep Q-network algorithm (DQN) in 2015, these instabilities have been circumvented by using a replay memory and RL came back to the focus of attention (Mnih et al. 2015). Since then, various milestones have been reached with the proposed algorithm, starting with AlphaGo, the program that defeated the (human) European Go champion in 2016 (Silver et al. 2016).

All the computer is given with RL is an environment, a possible set of actions it can choose from, and a reward function. Thus, after numerous trial-and-error attempts, the agent can learn from its own experience based on rewards given. This makes the implementation of complicated sets of rules, as used in traditional heuristics, obsolete. Additionally, the agent can also find new strategies to solve a problem (Kaelbling et al. 1996). AlphaGo for instance found known strategies for playing the game Go and discovered new ones to eventually beat the human players (Silver et al. 2016).

In the highly complex environment of a container terminal, it is not surprising that self-learning technologies like DQN are being evaluated. Fotuhi et al. for instance propose an algorithm to model crane operators as RL agents to optimize waiting times for trucks at the terminal (Fotuhi et al. 2013). Other work includes determining the best schedule for loading and unloading containers (Zeng et al. 2012) and how the yard layout should be designed to execute the loading or unloading in the right order (Hirashima et al. 2006; Verma et al. 2019).

In the present work, however, the focus is on the container stacking problem (CSP). Originally, the problem was proposed as the steel stacking problem by Rei et al. (2008): A set of steel slabs is stored in stacks in a warehouse between production and client delivery. This is an extension of the block relocation problem, where an initial state of the warehouse is given.

A problem related to the CSP is the container pre-marshalling problem, referred to as CPMP (Bortfeldt and Forster 2012). In contrast to the CSP, the CPMP starts with a given configuration of a container yard and aims to reshuffle the yard to a more optimal layout. There are approaches to solve CPMPs with RL, as Hirashima et al. described in 2006 for instance, where a desired layout could be reached faster with Q-

learning than with conventional methods.

However, with the CSP, the objective is to get an optimized arrangement while initially stacking the containers, so a reshuffling won't be necessary in the first place (Rei and Pedroso 2013). Fechter et al. use linearly approximated Q-learning with a "Sarsa(λ)" algorithm to solve the stacking problem in the context of steel slabs. Shuffling movements are being optimized and hence used as the negative reward. The authors use a small storage space with four stacks plus one delivery stack and insert and remove 12 to 27 items in each episode (Fechter et al. 2019). In the present paper, the approach is similar to the work of Fechter et al., although a larger block storage with more than one bay will be used to get closer to realistic scenarios. Additionally, with the implementation by Fechter et al. the list of stack in and out jobs stays the same in every episode. In the present work, the composition and sequence of jobs is randomly sampled to avoid overfitting.

A paper from 2020 addresses a similar problem in the context of a shipyard (Kim et al. 2020). Two learning agents are being combined to optimize the storage of components necessary for the shipbuilding process: One to conquer the reshuffling to achieve an optimized storage layout, and one to find the best stack for incoming components. Both agents are implemented with an asynchronous advantage actor-critic (A3C) approach.

Container alot allocation using reinforcement learning

Concept

This paper focuses on container slot allocation using reinforcement learning and only getting container attributes as an input. Furthermore, we wanted the solution to be adaptable with the advances of the standard frameworks for machine learning like tensorflow. This can be achieved by creating a training environment using standard semantics, which allow for easy use of already developed self-learning algorithms. To achieve the goal described above, we first separated the learning algorithm from the training space.

We wanted to ensure future compatibility for the learning environment, which could be achieved by building upon standard semantics for describing training environments for artificial intelligence (AI). OpenAI developed a relevant semantic for describing training environments for its OpenAI Gym (OpenAI 2016). While the idea of the OpenAI Gym originally was to create environments against which solution algorithms could be compared, a similar setup could be a viable solution to describe a container yard with its constraints as a learning space for self-learning agents. This learning environment can then present standard interfaces to a machine learning library that implement the self-learning algorithm.

Within the training environment created using the OpenAI semantics, self-learning agents should be trained. The agents, which are deep Q-agents, are described in more detail in the respective paragraph. Our goal was for the self-learning system to decouple the learning and optimization function from the inputs. Therefore we designed the system to take a container with an open number of attributes as input and optimize it against a cost function. For the specific case, we simplified the container attributes to the dwell time and the cost functions to the number of restackings of a container.

The core principles for implementing the algorithm for container slot allocation are the OpenAI semantics and deep Q-learning and will be described in the following paragraph.

OpenAI semantics

OpenAI Gym is a design framework used for testing reinforcement learning agents with different environments. By implementing a model by the standards of an OpenAI Gym, it is separated from any RL agent and can be improved independently.

First, the action and observation space need to be defined. These parameters represent the dimension and range of values of the actions the agent can choose from and the observation that describes the state of the environment. OpenAI Gym provides the class *spaces.Box* to declare these spaces uniformly. In the case of Stack Allocation, the actions are represented by discrete numbers from zero to *n-1,* with *n* being the number of available ground slots.

The observation space represents the state of the block storage at a specific time. For example, an empty slot is represented by -1.0. On the other hand, a container that is placed in the yard storage, is encoded as a floating-point number between 0.0 and 1.0. Therefore, the total value range of the observation space is -1.0 to 1.0. In addition to the containers currently in the storage, the one that is about to be stacked in needs to be represented in the observation as well. Accordingly, the dimension of the observation is *m+1* with *m* being the total capacity of the storage block.

Another function of the OpenAI Gym framework is *reset(): observation*. This function is called before every episode to reset the state of the storage block and in the case of the stack allocator, to get a new sample of jobs to perform the next episode. The reset function returns as an observation the state of the empty storage plus the encoded representation of the first container to stack in.

The *render(mode): void* function has been implemented as well. From this function, the visualization methods are being called to display the current state of the environment graphically. While many of the existing OpenAI Gyms have complex graphical visualizations, the stack allocator only prints the current filling heights for each slot to the console. Alternatively, the entire observation is printed when a corresponding mode is passed to the render function.

Core of the RL environment is the function *step(action): observation, reward, done, info*. After getting the desired action, the environment runs the corresponding steps (in this case it performs a stack in) and returns the new state of the storage block, the reward the move achieved, whether the episode is done with this move and additional meta information.

The two adaptions necessary for the framework to run the stack allocation problem both occur in the step function. First, the observation does not solely consists of the current state of the storage block, but is actually a tuple of two components: the state and an action mask to constraint the allowed actions at this specific point. The action mask is a vector of the size of the available ground slots and consists of zeros for allowed slots and ones for forbidden slots. This is necessary to prevent the agent from stacking the containers higher than allowed and to prevent a container from coming back to the same stack during

a shuffle move.

The second adaption made to the framework is due to the delay of the reward. The agent learns with packages of the state before a stack in, the state after the stack in, the action chosen, and the reward the move yielded. While the first three values are immediately available after a stack in, the reward depends on the shuffle moves the container participated in during its whole dwelltime on the block storage and can thus only be retrieved after the final stack out. This leads to a shift in the temporal course of the algorithm as described below (Figure 1).

Deep Q-learning

Regarding the slot allocation problem we are facing a situation where an agent performs an action (to stack a container in or out) in a certain state of an environment (the block storage), that produces a reward (the number of necessary restacks). This is the prerequisite for the meaningful use of reinforcement learning to discover a good sequence of stackings of containers with as few restacking operations as possible. Let S be the quantity of states of the block storage, then the decision for an action $a_t \in A(s_t)$ in a state $s_t \in S$, is independent of previous states, which meets the conditions of a Markov Decision Process. The goal of the agent is to maximize the future expected reward, which means to minimize the number of restackings:

$$R_t = \sum \gamma^k * r_{t+k+1} \quad \text{with } 0 \leq \gamma \leq 1 \text{ and } k \in \{0, \ldots, T\}, \tag{1}$$

where γ is the discount factor that weights future rewards. Although an episode of stack in and out operations of a block storage is in principle continuous without a defined end state ($T=\infty$), the episodes here are modelled as sequences of stack in and out operations of a constant length, so that the discount factor γ is set to 1. This ensures better transparency and control of the learning process.

The optimal Q-value of a state-action pair (s, a) is the sum of future rewards the agent can expect on average when it chooses action a (put container on top of one stack) in state s (dwelltimes of all containers in the block storage). This leads to the following recursive definition of the Q-function (Füßler et al. 2003):

$$Q(s, a) = r(s, a) + \gamma * max\ Q(s', a') \quad \text{where } s' \text{ is the following state of } s \text{ and } a' \in A(s'). \tag{2}$$

Once you have the Q-values for all (s, a) the optimal policy is trivial: in state s the agent chooses the action with the highest Q-value for (s, a). There are several Q-value iteration algorithms to determine Q-values, but these are only suitable for environments with a low number of states and actions. A realistic block storage at a terminal in Hamburg can have 600 slots on ground with a stack height of 4 containers. Assuming 100 different dwelltimes for the containers and an average utilization of 80%, this leads to approximately 100^{1900} states of the block storage. This is an astronomical number which illustrates the need for an approximate learning method. Deep Q-learning can be used in such complex environments to learn Q-values: here neural networks, which are called deep Q-networks (DQN) are used to approximate Q-learning. The basic Q-learning algorithm is given by

$$Q_{k+1}(s, a) \leftarrow (1-\alpha) * Q_k(s, a) + \alpha(r + \gamma * max\ Q_k(s', a')), \quad (3)$$

where α is the learning rate (Karim 2018).

For every episode a list of container stack in and out operations from real data from a HHLA terminal in Hamburg is created, ordered by their time stamps, for example: $op = \{c(1)_{IN,t1}, c(2)_{IN,t2}, c(3)_{IN,t3}, c(2)_{OUT,t4}, c(4)_{IN,t5}, c(5)_{IN,t6}, c(1)_{OUT,t7}, \ldots\}$ with containers $\{c(1),\ldots,c(n)\}$ and timestamps $t_1 \leq t_2 \leq t_3 \leq \ldots \leq t_n$. The dwelltime of a container $c(x)$ is computed by $c(x)_{dwelltime} = t_j(c(x)_{OUT}) - t_i(c(x)_{IN})$.

The DQN is trained with these episodes with the following algorithm (using epsilon-greedy, M is the number of episodes and T is the number of stack in and out operations in one episode):

```
for i=1 to M
    initialize environment
    for j=1 to T
        if operation == stack in c(x)                    with probability ε : randomly
            choose an action a∈A(s_t) at time t:         with probability 1 – ε : action a, which is
            execute action a                                 associated with max Q(a, s_t)
            save (s_t, a, s_t+1)                             from the DQN with input s_t
        else operation == stack out c(x)
            if there are containers {c(y_1),…,c(y_n)} above c(x)   // these are restackers
                insert these as stack in operations at the beg. of the list: op = {c(y_1)_IN,t1,…,c(y_n)_IN,tn,…}
            else
                compute reward r = – (number of restackings of c(x))
                put (s_t, a, s_t+1, r) to the replay memory with (s_t, a, s_t+1) from stack in op. of c(x)
        observe rewards
```

Figure 1: DQN-Algorithm

The *experiences* (s_t, a_t, s_{t+1}, r_t) are stored in the replay memory (capacity: 100,000). In the step *observe rewards* step a mini batch (size: 64) of randomly selected experiences from the replay memory is used to train the DQN:

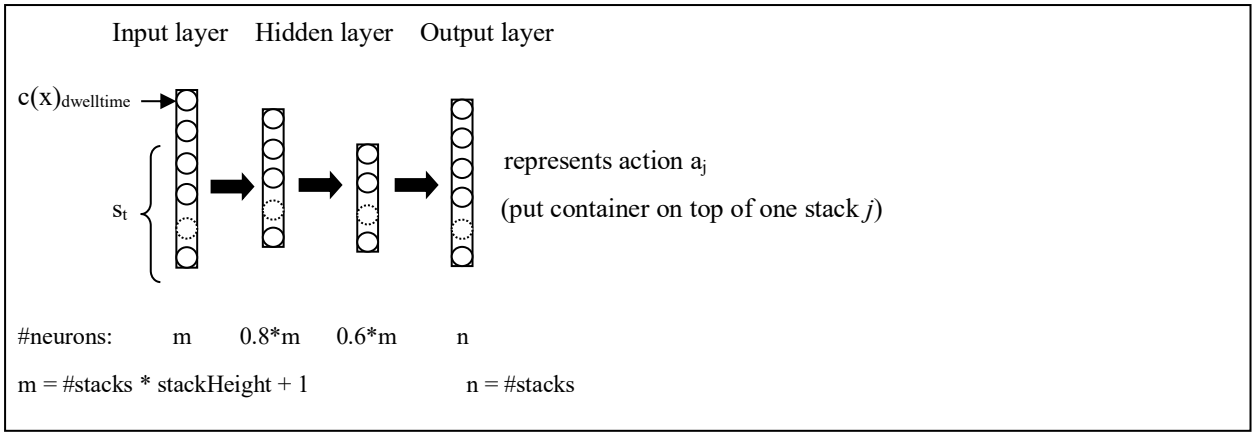

Figure 2: Structure of the DQN

The number of neurons for the input layer is defined by the number of stacks times the stack height + 1 (for the dwelltime of the container to stack in). Two hidden layers with 0.8*m respectively 0.6*m neurons and the ReLU activation function are used. The output layer (with softmax function) has as many neurons as available stacks, each representing the action a_j to put the container on top of the according stack.

Validation

To evaluate the designed model and DQN agent, and get first results quickly, a layout of five bays, two rows and four tiers was chosen. This results in a total stack amount of ten with the capacity to hold at most 40 containers.

The DQN is initialized with a batch size of 64 and a buffer size of 100,000. The gamma value is fixed with 0.99, the learning rate with 0.001. The exploration rate (epsilon) stays at 0.05 during the whole experiment, so five percent of the decisions are made randomly, in the other 95% of the cases, the agent predicts the best possible action.

A dynamic exploration rate (i.e., decreasing from 1.0 to 0.05 with a certain factor in each episode) has shown to be contra productive in this specific case. The agent can choose between ten stacks, most of which are empty in the beginning. If the action is randomly chosen, the results will be relatively good, since every stack is chosen with the same probability, and the storage block is filled evenly. If the agent predicts most of the actions, it will create a less balanced storage layout, because it tries to repeat actions that previously resulted in a good reward. This leads to an unevenly filled storage. Hereof the agent can learn better what not to do, than in an evenly filled storage. Therefore, the focus is on exploitation rather than exploration right from the beginning.

The agent is relatively free in choosing the stack for the next container, only two constraints are being made. Firstly, the agent is not allowed to choose a stack that is already full. In the current setting 'full' means there are already four containers in the stack. Stacking on a full stack must not occur in real life and is therefore also forbidden in the model. Secondly, when doing a relocation move, the agent can't put the container back on the same stack it removed it from. Without the second constraint, the agent spends a lot of time stacking items in and out of one and the same stack. Due to the delayed gratification, it can't see instantly, that this isn't a smart move. Eventually, it learns to avoid relocating to the same stack, but it takes time that can be avoided by restricting the choice of actions.

In this configuration, 40 containers have been stacked in and out of the storage. The information about in- and out-times were sampled from real historical data from the Container Terminal Altenwerder in Hamburg. The sampled set differs in every episode to avoid overfitting.

Another test run with 100 container stacks and 800 containers also yielded a promising result (Figure 3). To gain further insights, additional metadata of the experiment have been plotted. As before, the rewards simply consist of the negative number of shuffle moves ('Restackers') per episode. The left graph in the second row of Figure 3 depicts the distance of the portal cranes in each episode, which decreases similar to the amount of restackers. Another key figure that has been monitored here is the storage density. This parameter basically describes how densely the containers are being stacked into the storage. In this setup, this parameter doesn't seem to follow any trend over the played episodes.
In the last row on the left of Figure 3, the amount of shuffle moves of each container was divided by the time it spends in the storage. Lastly, the loss is shown in the right graph of the third row.

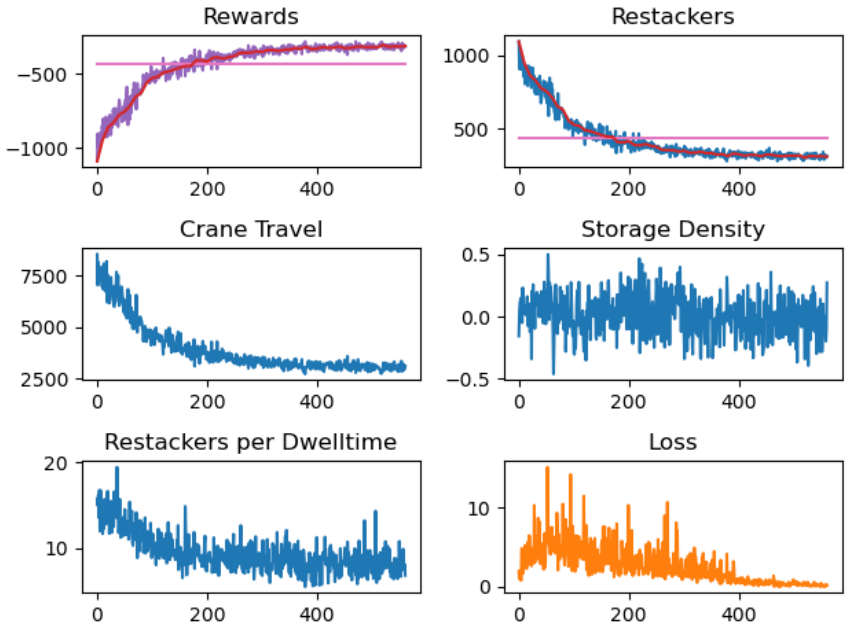

Figure 3: Experiment with 800 containers and 100 stacks

Summary and further work

In this paper, an algorithm to perform slot allocation at a container terminal with deep reinforcement learning has been proposed. The developed strenght of the proposed solution ist, that it builds on OpenAI Gym as a standard semantic to describe a real world problem. The opportunities and limits of the usage of the framework OpenAI Gym as well as the mechanics behind the DQN algorithm were described. The agent learns steadily in a small storage block. Future work will include the scaling to more realistic dimensions with more than 600 ground slots. For this matter, a connection with a simulation software for real size container terminals is in progress.

Also, further reward parameters such as crane travel distance or density of the storage are being evaluated. The ultimate goal is to build a self learning real world container yard management add-on for terminal operating systems (TOS).

References

1. Bortfeldt, Andreas; Forster, Florian (2012): A tree search procedure for the container pre-marshalling problem. In *European Journal of Operational Research* 217 (3), pp. 531–540. DOI: 10.1016/j.ejor.2011.10.005.
2. Dekker, R., Voogd, P. & van Asperen, E. Advanced methods for container stacking. *OR Spectrum* 28, 563–586 (2006). https://doi.org/10.1007/s00291-006-0038-3
3. Fechter, Judith; Beham, Andreas; Wagner, Stefan; Affenzeller, Michael (2019): Approximate Q-Learning for Stacking Problems with Continuous Production and Retrieval. In *Applied Artificial Intelligence* 33 (1), pp. 68–86. DOI: 10.1080/08839514.2018.1525852.
4. Fotuhi, Fateme; Huynh, Nathan; Vidal, Jose M.; Xie, Yuanchang (2013): Modeling yard crane operators as reinforcement learning agents. In *Research in Transportation Economics* 42 (1), pp. 3–12. DOI: 10.1016/j.retrec.2012.11.001.
5. Füßler, H., J. Widmer, M. Käsemann, M. Mauve, H. Hartenstein (2003). Contention-based forwarding for mobile ad hoc networks, *Ad Hoc Networks*, vol.1, pp.351-369
6. Géron, A. (2017). *Hand-On Machine Learning with Scikit-Learn & TensorFlow.* Sebastopol: O'Reilly Media Inc., pp. 321-322
7. Hirashima, Y.; Ishikawa, N.; Takeda, K. (2006): A New Reinforcement Learning for Group-Based Marshaling Plan Considering Desired Layout of Containers in Port Terminals. In : 2006 IEEE International Conference on Networking, Sensing and Control. 2006 IEEE International Conference on Networking, Sensing and Control. Ft. Lauderdale, FL, USA, 23-25 April 2006: IEEE, pp. 670–675.
8. Kaelbling, Leslie Pack; Littman, Michael L.; Moore, Andrew W. (1996): Reinforcement learning: A survey. In *Journal of artificial intelligence research* 4, pp. 237–285.
9. Karim, M. R. (2018). *Java Deep Learning.* Birmingham: Packt Publishing Ltd., pp. 459-461

10. Kemme N. (2020) State-of-the-Art Yard Crane Scheduling and Stacking. In: Böse J.W. (eds) Handbook of Terminal Planning. Operations Research/Computer Science Interfaces Series. Springer, Cham. https://doi.org/10.1007/978-3-030-39990-0_17
11. Kim, Byeongseop; Jeong, Yongkuk; Shin, Jong Gye (2020): Spatial arrangement using deep reinforcement learning to minimise rearrangement in ship block stockyards. In *International Journal of Production Research* 58 (16), pp. 5062–5076. DOI: 10.1080/00207543.2020.1748247.
12. Mnih, Volodymyr; Kavukcuoglu, Koray; Silver, David; Rusu, Andrei A.; Veness, Joel; Bellemare, Marc G. et al. (2015): Human-level control through deep reinforcement learning. In *Nature* 518 (7540), pp. 529–533. DOI: 10.1038/nature14236.
13. OpenAI (2016) Background: Why Gym? *Gym.* Retrieved from https://gym.openai.com/docs/
14. Rei, Rui; Pedroso, João Pedro (2013): Tree search for the stacking problem. In *Ann Oper Res* 203 (1), pp. 371–388. DOI: 10.1007/s10479-012-1186-2.
15. Rei, Rui Jorge; Kubo, Mikio; Pedroso, João Pedro (2008): Simulation-based optimization for steel stacking. In : International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences. Springer, pp. 254–263.
16. Silver, David; Huang, Aja; Maddison, Chris J.; Guez, Arthur; Sifre, Laurent; van den Driessche, George et al. (2016): Mastering the game of Go with deep neural networks and tree search. In *Nature* 529 (7587), pp. 484–489. DOI: 10.1038/nature16961.
17. Sutton, Richard S.; Barto, Andrew G. (2018): Reinforcement learning: An introduction: MIT press.
18. Tiecheng Jiang; Bo Zeng; Yong Wang; Wei Yan (2021): A New Heuristic Reinforcement Learning for Container Relocation Problem. In *Journal of Physics: Conference Series* 1873 (1), p. 12050. DOI: 10.1088/1742-6596/1873/1/012050.
19. Verma, Richa; Saikia, Sarmimala; Khadilkar, Harshad; Agarwal, Puneet; Shroff, Gautam; Srinivasan, Ashwin (2019): A reinforcement learning framework for container selection and ship load sequencing in ports. In : Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, pp. 2250–2252.
20. Zeng, Qingcheng; Yang, Zhongzhen; Hu, Xiangpei (2012): A method integrating simulation and reinforcement learning for operation scheduling in container terminals. In *TRANSPORT* 26 (4), pp. 383–393. DOI: 10.3846/16484142.2011.638022.