Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Reinforcement Learning for Discrete Event Simulation of Production Lines

Thilo Krüger[a,*], Anna Mack[b]

[a]d:AI:mond GmbH, Campus E1 3, 66123 Saarbrücken, Germany
[b]Universität Stuttgart, Höchstleistungsrechenzentrum, Nobelstraße 19, 70569 Stuttgart, Germany

Abstract

In recent years, reinforcement learning has become incredibly popular as a method to find good solutions in complex situations, e.g. games like chess or go. Reinforcement learning uses simulations of the underlying systems to learn good decisions with the help of these simulations. In the case of games like chess or go, these simulations are computationally cheap, since the new situation on the board is easy to calculate once the player or the system has decided on a move. This SHAPE project attempts to use reinforcement learning to optimise decisions during production processes. The simulation of such processes needs much more computational effort, since it depends on many deterministic intermediate steps and also on many random actions that happen between two simulated decisions. This work uses discrete event simulations as a framework for these simulations and the well known reinforcement learning algorithm Deep Q-Learning to train an agent to find optimal decisions for production processes. The main challenge of this work was to find a setup that allows the training of reinforcement learning agents within a time interval of a few days at the most. This was done by setting up a workflow on the supercomputer of the High-Performance Computing Center Stuttgart Hawk that allowed the running of $n-1$ simulations asynchronously in parallel, and the running of the optimisation on one separate core, where n is the number of the available cores. It turned out that, in this way, combining simulations of production lines with reinforcement learning becomes very effective since it enhances the workload on the single nodes significantly.

1. Introduction

In industrial processes, optimisation of the different production units is always an interesting problem for scientific work as well as for commercial approaches to solve those problems. d:AI:mond started as a data science consulting company and worked in this context in several data science projects. To sharpen the profile of the company, optimisation of production lines by reinforcement learning and discrete event simulation was identified as an interesting use case to help a broad range of producing companies optimising their productions. A toy example of such optimisations was published [4] and with the described use case a pilot customer was found. However, since the performance of discrete event simulations is not always good, it was unclear whether it is possible to run such optimisations in a realistic time horizon. Therefore, the aim of this work is to explore if it is possible to parallelise those simulations in order to get optimisation results quickly.

In contrast to other optimisation methods, reinforcement learning has the advantage that the total optimisation time can be divided into two parts. Firstly, a neural network has to be trained with the help of a simulation model. Secondly, the trained model is fed with the current state of the production line and evaluated. The result of this evaluation is then used to find an optimal decision for the production. In this paper, we focus on the first part of this process, since this is the most time consuming part. Once, the training of the neural network is done, the second part of the optimisation is very quick and does not need any special prerequisites regarding the hardware.

The structure of the present paper is as follows: In Section 2, we describe briefly the discrete event simulation in general, as well as our pilot use case. In Sections 3 and 4, we give a short theoretical overview of reinforcement learning and especially our model. In Section 5, we show, how we optimised the concurrency of our solution.

*Corresponding author, e-mail: thilo@daimond.ai

2. Discrete event simulation of production lines

Most real world (sequences of) events can be modelled in a discrete event manner. The concept of discrete event simulation (DES) is simple yet effective. Given a system in a certain state and at a certain time, it is possible most of the time to define all events that lead to a change of state. The idea of DES is now to compute (or probabilistically sample) the durations until these events are triggered. Then these events are scheduled in a queue. When the model is simulated, the simulation can hop from one event to the next and update the event schedule as well as the system state in every step. This approach is explained in the following section in detail. It is also shown how d:AI:mond uses this technique to simulate complex production procedures. For further details on DES, see, e.g. [5, 6].

2.1. Concept of DES

To explain the concept of DES, a first step is the definition of all the relevant parts of a DES model. In the following, the main constructs that are necessary to build a DES model are described.

2.1.1. State

The most important part of a DES model is the state of the system. The state is a collection of variables of any type. The idea is that the variables exactly describe the most important properties of the system as well as all circumstances that have to be known to model the behavior of the system. It is not necessary and also not useful to use more variables than needed. As an example in this section a single queue at the checkout in a local supermarket is considered. The following variables are examples that are considered interesting:

- the number of waiting customers (a natural number)
- the number of articles in all shopping carts (a list of natural numbers)
- the employee who is currently working at the checkout (e.g. one out of the list [Alice, Bob, Charlie])
- the time that all employees need to process one article (a dictionary with the employees as keys and a floating point number as value)
- the actual time of day (with type time)

The number of waiting customers as well as the list of employees are computable from the other variables and therefore not explicitly needed. It is not helpful to model variables like the size of the supermarket or the colour of the customers' hair as further variables.

2.1.2. Simulation time

In order to simulate the model, we need the simulation time of the system. This is a floating point number and is different from the actual time of day and is needed to schedule the events that happen during simulation. In the example, the simulation time could start at 7:00 am real time with the value of 0. If we want to simulate the queue until the supermarket closes (e.g. at 10:00 pm), the model is executed until the simulation time reaches $15 \cdot 60$ minutes $= 900$ minutes.

2.1.3. Observables

It makes sense to distinguish between ordinary state variables and those variables that should be traced as observables during the simulation. The observables are always part of or are computable from the state variables and can be used for various applications, e.g. for an alerting system or for the optimisation of the system with regard to those observables. In the example, the observable may be the queue length. Whenever a certain length is reached in a certain time span, the manager of the supermarket could open a new checkout.

2.1.4. Event list

To simulate the future of a system with a DES model, we have to fill and execute an event list (another name of the event list is the event queue, we use event list here to prevent confusion between the event queue and the queue of waiting customers in our example). The event list is a list of 2-tuples sorted by time, where each tuple contains a (simulation) time point and an event that will be triggered at this time point. This event is modelled as a function that converts a state s_t to another state s_{t+1}. Furthermore, this function can also update the event list or trigger other events directly. In our example, an event list at simulation time 110 minutes could look as follows:

1. At time 115, a new customer will reach the queue with 12 articles in the trolley. The next customer who reaches the queue is determined and added to the event list.

2. At time 118, the current customer will leave the checkout. The next customer in the queue is served and the time of departure of that customer is computed and added to the event list.
3. At time 120, Bob replaces Alice at the checkout.

Here, the update functions are described in words, but they can also be described as mathematical expressions.

2.1.5. Initialisation

At the beginning of the simulation, the simulation time is set to 0 and the state of the system is in a certain initial state, that is, the queue is empty in our example and Alice could be the employee at the checkout. The event list will be initialised model based, therefore the model should also contain a description for the event list at (simulation) time 0. For example

1. At time 3, a new customer reaches the queue with 3 articles in the trolley.
2. At time 120, Bob replaces Alice at the checkout.

2.1.6. Implementation

Given these five components, the model of a real world system can be simulated accurately and the observables can be tracked. In order to implement such a simulation, all the actors of the system have to be realised. Furthermore, buffers are implemented to represent queues. All actors could have the possibility to change the buffers. Furthermore, the actors and the buffers can schedule events in the event list.

In the example, the actors are the workers at the checkouts, as well as the customers of the supermarket. The queue is implemented as a FIFO buffer. Whenever a customer reaches the queue, he is put into the buffer. Whenever the customer reaches the checkout, he/she leaves the buffer.

The Python library `SimPy` offers these functionalities and takes care of the event list by providing a simulation environment. The user just has to run processes (e.g. actors) in this environment. Whenever the actor waits for another action (e.g. the worker works on the cart of a customer), an event is inserted to the event list and the actor becomes idle. Then the simulation jumps to the next event in the event list. `SimPy` also provides a data type `Store`, which implements the buffer functionality. Whenever something is put to (or taken from) the buffer, an event is appended to the event list that triggers automatically whenever it is possible to put (or to get). Stores can have a fixed capacity.

2.2. DES of production lines

Many production lines can be modelled by a collection of actors and buffers. In those cases, discrete event simulation can be used to simulate such models. In this SHAPE project, the underlying production line is the manufacturing of tableware. In this process, a buffer with a fixed size exists, where each single piece of tableware is stored twice, once before and once after the burning process. We present here a reduced version of the model for clarity reasons. All the relevant parts of the model (for the optimisation) are presented in the following.

The article flow happens according to the arrows in the chart. The articles get their shape in N presses and are transported from there to the buffer. They have to stay at the buffer for two to six hours, before they are transported to the biscuit kiln to be burned for the first time. After a second stay in the buffer, the articles leave this part of the production process to become glazed.

Due to the limited capacity of the buffer, it happens that this buffer gets out of storage space and the production frequency has to be reduced in order to regain space in the buffer. Therefore, a DES model of the system to monitor this bottleneck was prepared.

2.2.1. State

The state of the model is a collection of the states of all parts of the model. That is, for a press, the current order and article ID, the number of already processed pieces of this order, and the total number of pieces that have to be processed. Other manufacturing stages are the setters that transport the articles from the central buffer to the biscuit kiln and vice versa, as well as the biscuit kiln itself.

The articles are collected on boards during the processes. To describe the total state completely, the central buffer and another buffer between the setters and the biscuit kiln are presented. Here, every single spot in the buffer that is occupied with an article board is described with article ID, order ID, number of articles on the plate, and a flag whether the articles already have been in the kiln.

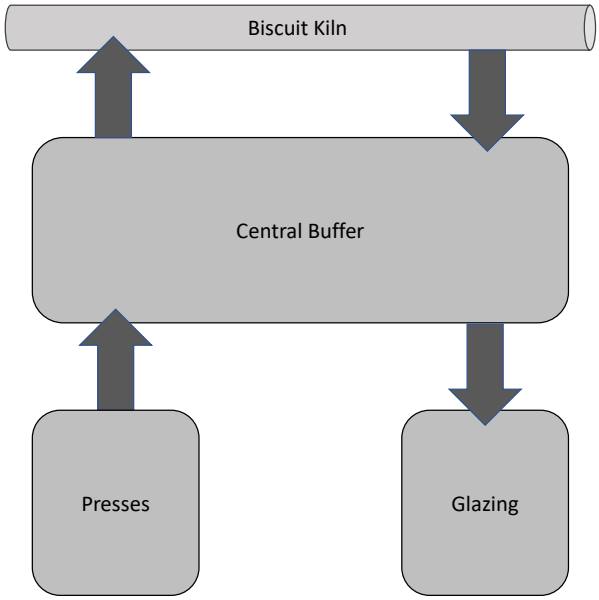

Figure 1. Material flow in the modelled real world system. Starting with pressing the articles in the lower left of the figure, the articles visit a central buffer twice, before leaving the system.

2.2.2. Observables

In the real world application of this simulation, we monitor main bottlenecks of the production dependence on the article mix. However, for the scope of this project, we focus on the number of empty spots in the central buffer as the main observable using a positive integer. A second important observable is a boolean value which is (True) for a certain point in time if and only if the state of the simulation is still valid, i.e. if there is still enough material for the biscuit kiln. This last observable only plays a minor role in the real world application, since many manual actions of the employees make sure that the real world production never runs into such an invalid state. However, since we deal with totally random initialisations in this project, we also have to consider this flag as an important observable.

2.2.3. Event list

The scope of the simulation comprises the following events that are scheduled to the event list:

- presses
  - start of the preparation of the presses
  - finishing of the preparation of the presses
  - finishing of filling the current article board
  - putting an article board into the central buffer
- setters
  - moving an article board to a setter
  - finishing of filling a board for the kiln
  - putting the kiln board to a second buffer
  - moving a kiln board back to the setter
  - finishing of filling the article board of burned articles
  - putting an article board back into the central buffer
- biscuit kiln

- queue up the kiln board in front of the kiln
- moving the kiln board into the kiln
- moving the kiln board out of the kiln
- moving the kiln board back to the second buffer

2.2.4. Initialisation

A key to the simulation tool is the initialisation of the DES with the data from the real systems. A major part of the work on the model was to mirror the real circumstances of the production. The production data had to be read and translated into the state variables. For initialising the setters we had to analyze two data sources to find out all the different article ids and remaining articles at all relevant spots in the production hall. However, for the scope of this paper, initialisation of the simulation with real data is not relevant. Instead, we initialised all parts of the model probabilistically. We started with a random list of articles that must be produced in the next time interval. Then, with the help of historical data, a list of articles that are currently pressed was created. The articles that are in the central buffer and in the kiln at time zero could again be probabilistically drawn from distributions that could be created with the help of the freshly created article list and historical data.

3. Reinforcement learning for DES

3.1. Reinforcement learning in a nutshell

In the last decade, reinforcement learning (RL) has become very popular to solve complex problems in several applications. The main idea of the applied version of RL (Q-learning) is to feed an agent with a simulation of a system and let the agent make all the decisions (i.e. actions) that are necessary to proceed the simulation. Hence, the agent implements a function that maps any combination of a state with an action that could be performed at this state to a real value (Q-value) and chooses the action with the best Q-value.

After the simulation ends, it sends back an evaluation of the simulation run. The agent now uses this evaluation to reinforce good decisions by matching the Q-value to the evaluation of the simulation. Therefore, in the following simulations, the agent will prefer decisions that have led to good results in previous simulation runs. The agent will be trained following this policy for several epochs, where each epoch consists of a constant number of episodes. At the end of the training, the agent is able to make locally optimal decisions.

In case that the system has a small number of possible states and a small number of possible actions, the function that the agent implements is a simple table, consisting a Q-value for each combination of state and action. Since neural networks have become much more performant in the last years, it is now possible to estimate the Q-values by encoding the state and the action as the input to a neural network, and then compute the Q-value as the output of the network. By minimising a loss between the Q-value and the actual (estimated) return of the simulation run, the neural network can be trained; this version of Q-learning is called Deep Q-learning (DQN). For further details on reinforcement learning in general and especially on Deep Q-Learning, see [1, 2, 3].

3.2. Some notations and maths

The core of a DQN model is a neural network that estimates the Q-value of a successor state, given the current state s of the system and an action a that can be performed in the state ($Q(s,a)$). The neural network is used to calculate the Q-values of all possible successor states and therefore to choose the action corresponding to the successor state with the best Q-value to proceed the simulation. This process (evaluating a certain neural network and choose the action according to the results) is called policy π and therefore

$$\pi(s) = \text{argmax}_a(Q_\pi(s,a)) \tag{1}$$

It is important to stress that the estimated Q-values only hold if the simulation follows the policy π for the rest of the simulation run. In DQN, the aim is to find the (globally) optimal policy π^*. By assuming that the neural network does an accurate estimation of the Q-values, we get

$$\pi^*(s) = \text{argmax}_a(Q_{\pi^*}(s,a)) \tag{2}$$

where π^* is at least locally optimal, since we always take the best action. Hence, in DQN, the training of the neural network minimises a loss between the Q-values computed by the neural network and an estimation of the Q-values

$$Q'(s_i,a) = r(a) + \gamma \cdot Q(s_{i+1}, \pi(s_{i+1})) \tag{3}$$

where $r(a)$ is the direct return for performing action a and s_{i+1} is the successor state of the system after performing action a. γ is a hyper parameter that is usually just slightly smaller than one.

In summary, DQN feeds a neural network with tuples from states, actions, and estimated Q-values for the successor states. This is done in parallel to the actual simulation. The actual actions taken during the simulation are derived from the outputs of the neural network. To get a better exploration of the state space of the simulated model, it is possible to choose some of the actions randomly at the beginning of the training.

4. Reinforcement learning for discrete event simulation of production lines

In the following, we will sketch how in general optimisation of production lines is feasible with the help of DES and DQN. Furthermore, we will show how we applied the framework to the production line described in 2.2.

4.1. Optimisation goal

With DQN it is possible to find optimal (or at least good) decisions for production lines. Depending on the type of the production line and the current production unit, those decisions could be e.g.

- choice of next article that will be processed by the unit
- size of the next produced article batch
- yes/no decisions like: should the unit get maintenance now

Such decisions are often made by domain experts with long lasting experience. Alternatively, heuristics are used to find good decisions, but also those heuristics depend on long lasting experience. Simulation of the production line in combination with DQN offers an alternative for such decisions. Whenever an expert can make a decision by looking at the current state of the system and the system can be simulated by a DES model, it is possible that the decision can be done by a neural network with at least the same quality. The above list is an example of use cases that we have seen in producing companies or that we discussed.

4.2. Application of DQN to existing DES models

Once a production line is modelled by a DES model and an optimisation goal is defined, DQN can be applied to the model. The simulation must proceed until the decision that has to be optimised is due. Then, the state s of the system at decision time must be vectorised. This is the trickiest part of the optimisation and must be done carefully and with a lot of manual optimisation. The vector that describes the state is fed into the neural network as described above. Then, there are mainly two possibilities. Firstly, action a_i can be also fed into the neural network. Consequently, the output is just one value that estimates the corresponding Q-value $Q_i(s,a_i)$, and the evaluation of the neural network happens n times, with n being the number of possible actions. Secondly, the action is not fed into the neural network, but the output vector of the neural network has length n. Then the i-th entry of the output vector is the corresponding Q-value $Q_i(s,a_i)$ (see Fig. 2). After the neural network was evaluated and the corresponding action was taken, the simulation must again proceed until the next decision is due. The described procedure is repeated, while the state s' that is fed into the neural network in the $(n+1)$-th step, is also the successor state of the state s and will therefore be used to compute the loss function to train the neural network according to Eq. 3.

4.3. Case study: Manufacturing of tableware

For the model of the production of tableware, the sketched application of DQN is straightforward. The simulation runs until a new article has to be sent to the biscuit kiln. In order to vectorise the state at this point, all articles that can occur in the simulation are clustered into N clusters regarding the different values that drive the simulation (mainly numbers at the different spots in the system). Then, the numbers of the articles of the different clusters at the different points in the simulation are tracked as the state vector. The neural network then computes N outputs that represent the estimated Q-values for the case that an article from the according cluster is chosen next. Then the article with the highest priority from the cluster is chosen as the next article. The rest of the optimisation works as described before.

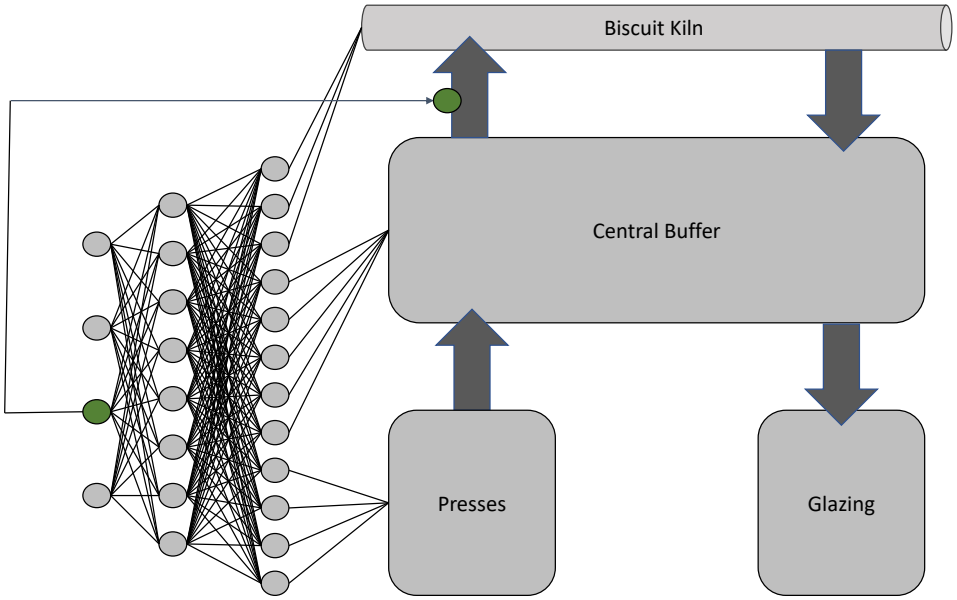

Figure 2. Visualisation of the structure of the training. The data of the presses, the buffer, the kiln, and of the glazing is vectorised and fed to the neural network. The result of the evaluation of the network is translated to an action and fed back into the process again.

5. Parallelisation strategy and workflow optimisation

It is well known that DQN and other RL algorithms perform well for problems that can be simulated efficiently. Many Atari games build base lines to compare different RL algorithms [1]. Since those games were implemented for ancient hardware, they can be simulated very quickly. The vectorisation of the state of the games usually works by vectorizing the screen image, and the neural networks that are used in such problems are known to work well for image processing. Another famous application of RL algorithms is chess, where the simulation of the system is even faster, since the positions of the pieces as well as the single moves are processed very easily. In those cases, the bottleneck of RL optimisations is always the RL algorithm itself. Optimising the algorithms for GPUs is a well known approach to solve such RL problems. However, in this project, the bottleneck is the simulation of the underlying system itself. The production process is very complex and takes a long time to simulate. Furthermore, it is not easily possible to simulate those processes on GPUs. Therefore, one of the main goals of this SHAPE project is to develop a smooth workflow for simulation and optimisation with DQN which allows an efficient usage of HLRSs resources.

As a base, several Python scripts were provided which held the essential functionalities of DES, DQN, as well as pre- and postprocessing. The scripts allowed a first workflow, but needed to be started manually and did not provide any parallelism. This code base was extended and adapted in order to get an automatic workflow. Furthermore, it was optimised to increase efficiency. The first step was to run a batch of simulations concurrently. The simulations run independently and each run uses its own randomised input data. This was implemented by using the Python framework `mpi4py`. It allows to create a message passing interface MPI environment with just a few code changes. In a second step, the environment was modified such that the simulations run asynchronously. That means a single core is able to start a subsequent simulation just after finishing the current simulation. Data is stored at the end of the simulation in separate directories on the Lustre file system. At the same time as the simulations run, one core is dedicated to do the optimisation and all remaining procedures like reading and storing of intermediate results separately. It gathers the data of simulation runs, runs the optimisation and also takes care of removing data which was already used. After finishing one optimisation it will immediately start over by collecting the data of the simulation runs that finished in the meantime. With this procedure it is not necessary to wait for simulations to finish before continuing to the next round of reinforcement learning (epoch). Idle times of the dedicated core for machine learning are reduced drastically.

In order to quantify the saved idle times using the asynchronous implementation we did some measurements on Hawk. As a representative test case we picked a 20 minute run. We counted the number of simulations we were

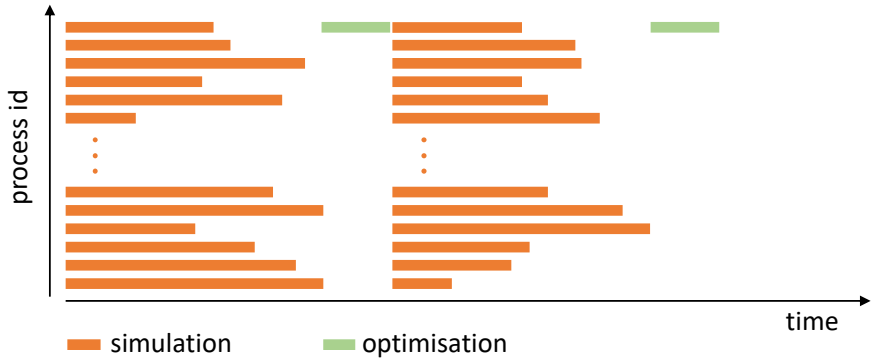

Figure 3. Visualisation of the core usage in a first approach. Here, each simulation was run as single process on one single core. After finishing all simulations, the optimisation was also run as process on one single core. This procedure would be repeated multiple times. Since there are many synchronising points we got a lot of idle times. Additionally, the simulations vary in the runtimes, which led to further idle cores.

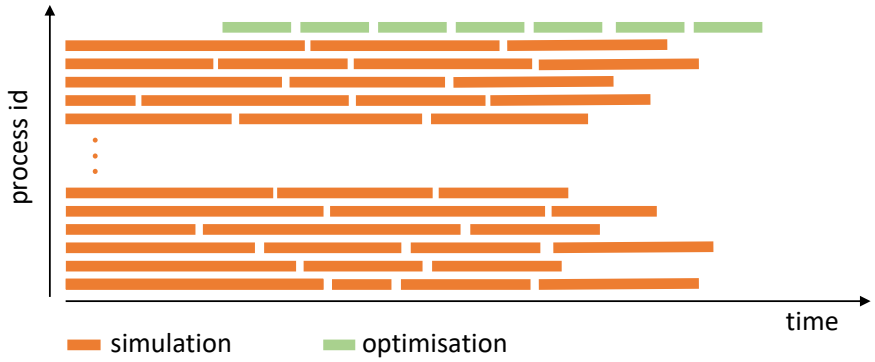

Figure 4. Visualisation of the core usage in the final approach. The improved workflow allows us to run simulations asynchronously and reduce the idle times. The optimisation runs independently on one separate core. The simulation processes do not have to wait during the optimisation phases.

able to run in this timeframe for a synchronous and asynchronous run. For the synchronous run (see Fig. 3) a batch of simulations was run on all available processes followed by an optimisation. Since the optimisation only runs on one core, all other cores stay unused for this period. Figure 4, visualises the asynchronous scheduling and shows the reduction of idle times clearly. Here, the core that does the optimisation is only idle at the very beginning of the process, while the other nodes are only idle as soon as a signal is sent to all nodes. After this signal, every node stops after the next simulation run and then has to wait for every other node to finish. In the 20 minute runs, 60% more simulation runs were finished in the asynchronous setting than in the synchronous setting. For bigger cases in terms of longer runtime (i.e. more simulation runs) we expect similar results as the amount of computation and data (per step) does not increase. The overhead of start and end routines will be approximately the same (absolutely) for cases with increased runtime, therefore the relative overhead will be smaller with increasing total simulation time.

6. Results and conclusion

Within the scope of this SHAPE project, we could make a proof of concept and show that it is feasible to optimise production lines with DQN in a practicable time horizon. To do so, we built a workflow on the Hawk supercomputer allowing the run of many simultaneous discrete event simulations of a tableware production on several cores and the reinforcement learning on one extra core. An important insight was that the 128 cores that we used in our experiments were enough to create more data than needed for the optimisation. With this workflow, the training of the neural network converged in less than one day. Since for many applications for the proposed solution, the training is only performed once, training times of multiple days would be fine. Therefore, purchasing a local machine to run such simulations is feasible, even for a small company like d:AI:mond.

On the downside of this project, we could not manage to find a globally optimal solution for the production line of our pilot customer. It turned out that further tuning of the hyperparameters is needed, to manage a generalisation of the trained neural networks. However, we could find decisions that were locally optimal for certain situations during the production, and therefore, we are very optimistic to find good solutions for our pilot customer, as well as for many other production lines.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra *et al.*, "Playing atari with deep reinforcement learning", arXiv preprint arXiv:1312.5602, 2013.

[2] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey", Journal of artificial intelligence research, vol 4, 1996.

[3] A Choudhary, "A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python", https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/, 2019.

[4] T. Gros, J. Groß, and V. Wolf, "Real-time decision making for a car manufacturing process using deep reinforcement learning", 2020 Winter Simulation Conference (WSC), 2020.

[5] S. Robinson, "Discrete-event simulation: from the pioneers to the present, what next?", Journal of the Operational Research Society, vol 56, 2005.

[6] N. Matlof, "Introduction to Discrete-Event Simulation and the SimPy Language", https://heather.cs.ucdavis.edu/ matloff/156/PLN/DESimIntro.pdf, 2008.

Acknowledgements

This work was supported by the PRACE project funded by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 823767. The work used the Hawk supercomputer at the HLRS, Stuttgart, Germany, under the PRACE Research Infrastructure resources.