Capacity Requirements Planning for Production Companies Using Deep Reinforcement Learning

Harald Schallner

Capacity Requirements Planning for Production Companies Using Deep Reinforcement Learning

Use Case for Deep Planning Methodology (DPM)

Harald Schallner[1][0000-0002-8214-4598]

[1] Jade Hochschule, Wilhelmshaven, Germany
harald.schallner@jade-hs.de

Abstract. In recent years, deep reinforcement learning has proven an impressive success in the area of games, without explicit knowledge about the rules and strategies of the games itself, like Backgammon, Checkers, Go, Atari video games, for instance [1]. Deep reinforcement learning combines reinforcement-learning algorithms with deep neural networks. In principle, reinforcement-learning applications learn an appropriate policy automatically, which maximizes an objective function in order to win a game. In this paper, a universal methodology is proposed on how to create a deep reinforcement learning application for a business planning process systematically, named Deep Planning Methodology (DPM). This methodology is applied to the business process domain of capacity requirements planning. Therefore, this planning process was designed as a Markov decision process [2]. The proposed deep neuronal network learns a policy choosing the best shift schedule, which provides the required capacity for producing orders in time, with high capacity utilization, minimized stock and a short throughput time. The deep learning framework TensorFlow™ [3] was used to implement the capacity requirements planning application for a production company.

Keywords: Artificial Intelligence Applications, Planning and Resource Management, Deep Learning Framework, Deep Planning Methodology (DPM).

1 Introduction

Many production companies have to plan the capacity requirements for their work center frequently. They have to decide how many working hours and shifts per day are needed for each production resource. Current standard software implementations of SAP® Enterprise Resource Planning (ERP) and Supply Chain Management (SCM) systems support the capacity requirements planning by reporting capacity utilization and by capacity levelling functions based on shift schedule [4]. Users have to maintain a feasible shift schedule before capacity requirements planning can be executed. SAP® ERP and SCM systems do not offer any planning functionality to optimize shift schedule automatically [5]. One main reason for this missing standard system functionality is the significant uncertainty about changes of available production capaci-

ties and customer requirements in the near future. Another reason is the exponential runtime complexity for a complete enumeration algorithm, which searches for the optimal solution within an exponential sized decision space. See *O( )* - formula (1) for complexity calculated by number of alternative shifts A (e.g. 8, 16 or 24 working hours per day), number of scheduled days T and number of capacities C:

$$O\left(A^{C^T}\right) \tag{1}$$

In order to achieve long-term goals for a sequence of inherent uncertain planning situations, reinforcement-learning approach is proposed in this work. Reinforcement-learning applications provide evaluative feedback to a learning agent interacting with an environment. This approach was chosen, because there were many reinforcement-learning implementations that could handle other decision problems with similar complexity and significant uncertainty successfully. Related work can be found in [1].

2 Methodology for Applying Deep Reinforcement Learning to Planning Processes

Subsequent sections describe all steps that are proposed to implement deep reinforcement learning application for a planning process, named Deep Planning Methodology (DPM).

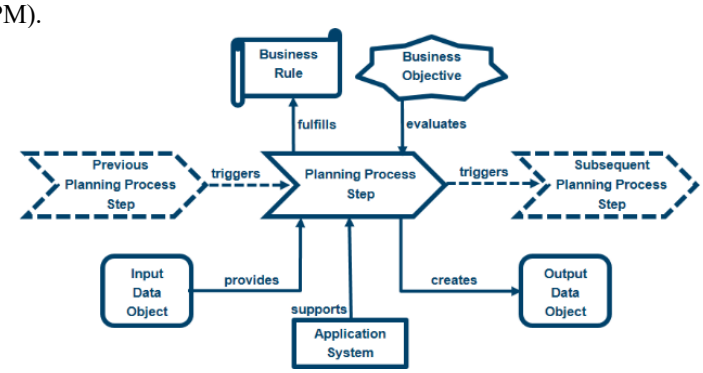

Fig. 1. Generic business process model

2.1 Step 1: Model Business Process

For the purpose of understanding the business process, data flow, used applications and the sequence of planning, process steps have to be modelled. The model has to comprise all relevant aspects of the business domain. Fig. 1 shows the graphical representation for all generic elements of the proposed business process model. For all planning process steps, the following business objects have to be modelled:

- Business rules:
  Business rules describe the organizational constraints, which have to be fulfilled by the planning result. Planning process has to comply with all business rules. Plan-

ning results are feasible, if all constrains are observed. Business rules have to be formulated by propositional logic or by procedures.

- Business objectives:
  Often, a huge number of different planning results are complying with business rules. In order to evaluate all feasible planning alternatives, an objective function measures the quality of planning results. Objective functions are based on key performance indicators (KPIs). Commonly, objective functions are defined by the sum of weighted key performance indicators, because trade-offs between different key performance indicators have to be balanced.
- Input and output data object:
  All data objects have to be specified by their attributes, which are provided for or created by a planning process step.
- Application systems:
  Every application system has to be modelled, that supports planning functions or provides data objects.

2.2 Step 2: Identify Relevant Planning Process Steps

Every planning process step can be analyzed according to its relevance for applying deep reinforcement learning based on the business rules, the planner performance and the planning problem complexity. The following checklist helps to decide which planning process step has high potential for implementing:

First check: If all business rules can be specified in detail, this means every organizational constraint can be formulated, then perform the second check. Else, go on with third check.

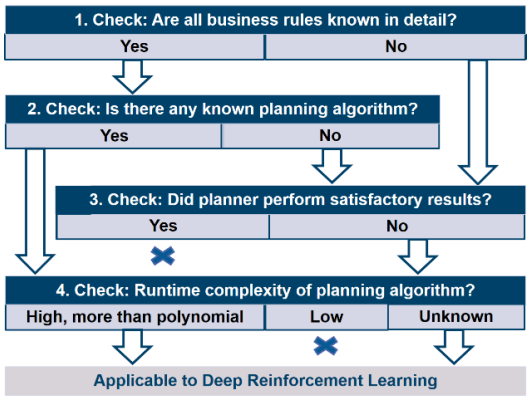

Fig. 2. Checklist for deep reinforcement learning applications

Second check: Is there any algorithm, heuristic or system transaction available, that computes a satisfactory planning result according to the objective function? If yes, the algorithm, heuristic or system transaction has to be analyzed in the fourth check. If there is no appropriate application available, then human planning skills have to be focused on in check three.

Third check: Find a human planner, who can create satisfactory planning results complying with implicit business rules manually. If there is nobody known with this competence, then go on with check four.

Fourth check: Analyze the computational complexity of the algorithm, heuristic or system transaction. If the runtime grows more than polynomially with the planning input size or the complexity is unknown, then deep reinforcement learning could be suitable for computing satisfactory planning results.

Generally, deep reinforcement learning applications are recommended, if the planning problem complexity is high or unknown or business rules cannot be specified completely and there is no planner performing satisfactory results, as shown in Fig. 2.

2.3 Step 3: Convert Planning Process into Markov Decision Process

In previous step two, planning process steps are identified that do not apply any algorithm with polynomial runtime. Fortunately, Papadimitriou and Tsitsiklis [2] proved that the Markov decision process for finite horizons is solvable in polynomial runtime by dynamic programming (so-called "P-complete"). Consequently, the planning process has to be converted into a Markov decision process.

In contrast to other machine learning, evolutionary or optimization approaches, the Markov decision process does not need exemplary supervision nor complete models of the business rules [1]. The task of the Markov decision process is to compute an appropriate policy within an uncertain environment in order to achieve long-term goals at the end of a given time horizon. A policy decides which action a_t has to be executed based on the current time t and state s_t. Thus, planning process has to be defined as a sequence of state dependent actions for each time of a finite horizon $T \in N$. The sequence of actions and states has to be modelled by time buckets $0 \leq t < T$, e.g. weeks, days or hours. In addition, a finite set of states S, an initial state s_0 for the first and for all other time bucket s_t have to be specified. The state s_t describes the decision relevant planning information for the current time bucket, provided by the planning environment. Furthermore, the action space has to be finite. Finally, a reward function $r(s_t, a_t, t) \in \mathbb{R}$ has to be specified. To sum up, the Markov decision process has to find an optimal policy function $\delta(s_t, t)$, that maximizes the following value function v, as shown in [2]:

$$v = \sum_{t=0}^{T} r(s_t, \delta(s_t, t), t) \tag{2}$$

2.4 Step 4: Develop Reinforcement Learning Environment

This methodology proposes the use of the deep learning framework TensorFlow that was developed as an open-source software library by the Google Brain Team and is available for everyone via GitHub [6, 7]. In recent years, TensorFlow has become very popular with over a million source code downloads [3]. In addition to Google, many companies have selected TensorFlow for their machine learning applications, e.g. Airbus, eBay, NVIDIA, Coca-Cola. Especially, SAP has integrated TensorFlow

into its Leonardo Machine Learning Foundation by providing the “Bing your own Model” (BYOM) web service [8].

Based on TensorFlow Kuhnle, Schaarschmidt, and Fricke [9] developed an open source library for applied deep reinforcement learning, named TensorForce. TensorForce offers a unified declarative interface to common reinforcement-learning algorithms [10]. TensorForce library provides an application-programming interface by the four following generic Python classes: `Environment, Runner, Agent` and `Model` [11].

The `Agent` class processes states, returns actions, stores past observations, loads and saves models. Each `Agent` class employs a `Model` class, that implements algorithms for calculating next action given the current state and for updating the model parameters from past experiences [11]. The `Runner` class implements the interaction between the `Environment` and the `Agent`. For every time step the `Runner` receives the current action from the `Agent`, executes this action in the `Environment` and passes the observation to the `Agent`. The `Runner` manages the duration and number of each learning episode and the cumulative rewards [11]. The planning horizon limits the number of time steps of each episode. The value function is implemented by cumulative rewards. Thus, the state sequence of one episode represents the planning process results for the time horizon. For each planning process step, following `Environment` methods and attributes have to be developed in Python [11]:

- Attribute `actions` returns the action space for all possible planning decisions.
- Attribute `states` returns the state space for all possible planning situations.
- Constructor method `_init_()` initializes the state based on planning input.
- Method `reset()` sets up a new learning episode with its initial state for the first time step.
- Method `execute(actions)` performs the selected actions and returns its reward, next state and a terminal indicator defining the end of the planning horizon.

TensorForce has no restriction on number and type of different states and actions [10]. To sum up, a planning specific Python class has to be developed, which overrides attributes and methods from the `Environment` superclass. A subclass has to simulate the consequences of each planning decisions to its domain. In order to handle the risk of unrealistic assumptions and oversimplifying, relevant business constrains have to be considered in the `execute(actions)` method.

2.5 Step 5: Configure Reinforcement Learning Agent

TensorForce library offers eight different pre-built `Agent` classes that implement state-of-the-art reinforcement-learning algorithms [10]:

- `DQNAgent`: Deep Q-Network agent applies the original Q-Learning algorithm [12].
- `NAFAgent`: Normalized Advantage Function agent uses continuous Q-learning algorithm [13].

- `DQFDAgent`: Double Q-learning from demonstration agent enriches expert knowledge [14].
- `VPGAgent`: Classic Vanilla policy gradient agent implements the classic policy gradients algorithm, known as REINFORCE [15].
- `TRPOAgent`: Trust Region Policy Optimization agent supports categorical, bounded and continuous action spaces [16].
- `PPOAgent`: Proximal Policy Optimization agent applies an alternative policy-based method [17].
- `DQNNstepAgent`: The n-step Q-Learning agent performs asynchronous gradient descent for optimization [18].
- `DDPGAgent`: Deep Deterministic Policy gradient agent supports continuous action domains [19].

Agent classes have to be configured at least by neuronal network layers, exploration, action space, state space, learning algorithm parameters and update methods. There are three update methods for network weights: episode based, batch based, time-step based. The selection of an appropriate agent class and the declarative configuration of hyper-parameters is required, when creating an instance in Python. The best way to choose a suitably configured class is to evaluate all eight classes experimentally, because each planning process step has different business rules, objective functions and planning data (see next step in following subsection 2.6).

2.6 Step 6: Evaluate Learning Results

The experimental evaluation of the configured agent classes interacting with the developed environment class requires representative input data. Therefore, data cleansing of real world planning input data is recommended, in order to create meaningful test scenarios.

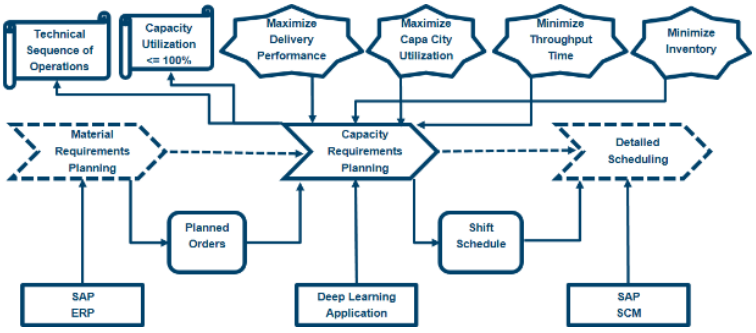

Fig. 3. Capacity Requirements Planning Process Model

Evaluation criteria are the convergence and level of the weighted objective function. Consequently, for each test scenario, each agent configuration and all training episodes the cumulative reward values have to be compared.

In addition, TensorBoard can be used as a dashboard for the deep learning models, implemented with TensorFlow [20]. TensorBoard is an open-source implementation

and available via GitHub [21]. It generates web-based visualizations of the computation dataflow graphs and training curves, which enable to fine-tune hyper-parameters.

3 Use Case for Deep Planning Methodology (DPM)

3.1 Standard Business Process Model for Production Companies

In this subsection, the described Deep Planning Methodology (DPM) is applied to the SAP® standard business process "plan-to-produce" for production companies. This standard planning process comprises the following consecutive planning steps [5]:

1. Demand Planning (DP)
2. Production Program Planning (PP)
3. Material Requirements Planning (MRP)
4. Capacity Requirements Planning (CRP)
5. Detailed Scheduling (DS)

Demand Planning consolidates the sales forecast, applies statistical methods and releases a demand plan to Production Program Planning. The goal of DP is to achieve a high forecast accuracy. Based on demand plan, sales orders and delivery schedule, PP calculates balanced production quantities for every finished product and period. PP's objective comprises the on-time delivery performance and the probability that demand is met from stock, called service-level. PP plans the requirement quantity and dates for finished products and releases independent demand to MRP.

Subsequently, MRP creates planned orders for each in-house production material, in order to meet the independent demands. A planned order schedules dependent demand for its components and a technical sequence of operations performed at capacities. These operations require adequate capacity availability for every scheduled operation. The duration of an operation defines the capacity requirement quantity. MRP assumes infinite capacities. This means, MRP does not check scheduled operations with regard to capacity availability. Thus, the availability of capacities has to be checked and ensured by the following Capacity Requirements Planning (CRP). Based on planned orders CRP has to plan feasible shift schedules for all production capacities. On the other hand, Detailed Scheduling (DS) has to consider the material availability of all components based on shift schedule and planned orders. DS schedules the sequence of operations performed at capacities and aims to minimize set-up efforts.

3.2 Capacity Requirements Planning Process

Fig. 3 visualizes the CRP process with its business rules, objectives, application systems and data objects. The shift schedule can be modeled by the decision variables $d_{i,j}$ (C *x* T – matrix) that defines the available working hours for all capacities $C \in \mathbb{N}$ and each day within a defined planning horizon $T \in \mathbb{N}$:

$$d \in \mathbb{R}^{C \times T}, \quad 0 \leq d_{i,j} \leq 24 \tag{3}$$

A planned order is specified by its operations i:

- Operation sequence: $i \twoheadrightarrow j \quad \Leftrightarrow$ i is predecessor of operation j
- Scheduled time bucket: $0 \leq \tau_i < T$
- Production duration: $\varrho_i \in \mathbb{N}$
- Required capacity: $\psi_i \in \mathbb{N}$

Capacity requirements *k* can be summed by all relevant operation durations:

$$k \in \mathbb{R}^{C \times T}, \; k_{c,t} = \sum_{\forall i:\, \tau_i = t \wedge \psi_i = c} \varrho_i \tag{4}$$

A planned shift schedule is feasible, if two business constrains are fulfilled: Firstly, finite capacity utilization is limited to 100% for all capacities *c* and time buckets *t*:

$$\forall\, c,t: \quad U_{c,t} = \frac{k_{c,t}}{d_{c,t}} \leq 100\% \tag{5}$$

Secondly, the technical sequence of operation is ensured, if succeeding operations have to be scheduled at a later time bucket:

$$\forall\, operation\, i \twoheadrightarrow j \;:\; \tau_i < \tau_j \tag{6}$$

The objective function f that measures the quality of a shift schedule comprises classic key performance indicators used in production companies, such as capacity utilization $U_{c,t}$, weighted by w_u, throughput time P_i, weighted by w_p, inventory R_i, weighted by w_r, and delivery performance P_j, weighted by w_p:

$$f = w_u \sum_{\forall\, c,t} U_{c,t} + w_p \sum_{\forall\, i,t} P_{i,t} + w_r \sum_{\forall\, i,t} R_{i,t} + w_u \sum_{\forall\, i,t} P_{i,t} \tag{7}$$

CRP was identified as a relevant planning step with high potential for deep reinforcement learning implementations, because the complete enumeration algorithm has exponential runtime complexity, see equation (1).

3.3 Markov Decision Process for Capacity Requirements Planning

CRP is converted into a Markov decision process by defining actions, states and rewards:

- A C-dimensional vector specifies the actions that decide how many working hours (here: 8,16 or 24) are available for each capacity and time bucket t:

$$a_t \in \{8, 16, 24\}^C \tag{8}$$

- The shift schedule *d* is calculated by the actions a_t performed for all capacities within one learning episode:

$$a_t = \begin{pmatrix} d_{0,t} \\ \vdots \\ d_{C-1,t} \end{pmatrix} \tag{9}$$

- State s_t is designed via capacity requirements based on a time interval with b buckets. This means, that the state slices the capacity requirements by a rolling time window:

$$s_t = \begin{pmatrix} k_{0,t} & \cdots & k_{0,t+b-1} \\ \vdots & \ddots & \vdots \\ k_{C-1,t} & \cdots & k_{C-1,t+b-1} \end{pmatrix} \tag{10}$$

- In orders to ensure, that for each learning episode the cumulative reward v is equal to the objective function value f, reward function r is calculated by:

$$r\,(s_t, a_t, t) = w_u \sum_{\forall\, c} U_{c,t} \; + \; w_p \sum_{\forall\, i} P_{i,t} \; + w_r \sum_{\forall\, i} R_{i,t} \; + w_u \sum_{\forall\, i} P_{i,t} \tag{11}$$

3.4 Learning Environment for Capacity Requirements Planning

The learning environment has to fulfill the two business constraints, which are described in subsection 3.2. In consequence, the `execute(actions)` method implements for each time step and each capacity that the planned shifts offer enough working hours to meet the capacity requirements. If capacity utilization is higher than 100%, operations are moved to the next time bucket. Else operations are pulled forward from subsequent to current time buckets, in order to achieve fully utilized capacities. Predecessors and successors of moved operations have to be rescheduled, too. This influences throughput time, inventory, delivery performance and consequently the reward. The implemented time window of states focuses on a rolling interval of two days for capacity requirements. This provides relevant information for the agent to find a satisfactory shift schedule.

3.5 Learning Agent for Shift Schedule

Thriving reinforcement learning often requires tuning the agent hyper-parameters. This subsection proposes some best practice for customizing of the selected `PPOAgent` class, which has proven its productivity in the domain of capacity requirements planning.

The hidden layers of deep neuronal network are specified by parameter `network`. State and action spaces define the structure of input and output layers implicitly. Explicitly, three hidden layers have to be configured to meet the complexity of this capacity-planning problem. The number of hidden units corresponds with the planning complexity, described in O – formula (1). This means high number of capacities, longer planning horizon and high number of alternative shifts require higher number of hidden units, called size of layer:

1. The first layer type is embedding, in order to handle all dimensions of the state space. This layer represents states in a continuous vector space where semantically similar states are mapped to nearby points, coded as indices. The number of embedding indices should be within the range of 32 to 1024 (here 100). The dimension of the vector space is defined by the size of the embedding layer, here 32

(range: 16 - 1024). This means, that the input layer is fully connected to the embedding layer. L1 and L2 regularization weights are not used in this layer, because overfitting is not an issue for learned policies.
2. The second layer is called flatten. This layer reshapes the first layer from multi-dimensional vector to a one-dimension vector.
3. The third dense layer with 32 hidden units (range: 32 - 1024) is fully connected to the second layer. The ReLU function [7] is chosen for activation.

Prominent results concerning the convergence of cumulative rewards to a satisfactory solution can be achieved by the following configuration of the `PPOAgent` class:

- The `update_mode` specifies how many episodes or time steps used for one iteration of a gradient descent update. The `frequency` defines the number of elapsed episodes or time steps that trigger the next update batch run. Both variables are set to one episode (range: 1 - 20 episodes).
- The `capacity` of the memory corresponds to the numbers of experiences collected before updating neuronal network weights. It has to be a multiple of the `update_mode` batch size and was set to 100000, to speed-up training.
- The learning rate specifies the strength of each gradient descent update step. Best results for Adam optimization were achieved with rate 0.001 (range: 0.00001 - 0.008). If scattering of the objective function values is too high or the rewards do not consistently increase for many episodes, the learning rate has to be decreased. On the other hand, if the learning rate is too low, then optimal solution is too hard to find for the agent.
- Parameter `entropy_regularization` controls the randomness rate of the policy. This parameter is very sensitive and should have the value 0.03 of interval 0.001 to 0.1. Other values have negative effects on convergence and level of cumulative rewards. Higher values create instable solutions for many periods. Lower values prevent finding satisfactory solutions.
- Parameter `likelihood_ratio_clipping` specifies the acceptable threshold of divergence between the new and old policies for gradient update step. High values correspond with a fast, but instable learning process. Best results are based on value 0.02 (range: 0.005 - 0.35).
- Parameter `discount` defines the factor of future rewards. The parameter value 0.99 ensures policies with high cumulative rewards at the end of an episode. Thus, discount factor should have a high value within the interval of 0.85 to 0.999, because the objective function of capacity requirements planning is calculated by the cumulative rewards for each time horizon.

3.6 Experimental Evaluation

Four of eight TensorForce agent classes are suitable for CRP: `DQNAgent`, `VPGAgent`, `PPOAgent` and `DQNNstepAgent`. The other agents were not applicable, because they are designed for continuous action space or additional expert knowledge. The qualified agents were evaluated according to ten different test scenarios. The test scenarios differ according to randomized planned orders, number of ca-

pacities (range: 6 - 24), number of operations (interval: 2.500 - 10.000) and planning horizon from 30 to 120 days. Learning algorithms were finished after 4000 episodes. All test scenarios converge to a satisfactory result, near global optimum. The mean of cumulative rewards constantly increased over periods. The scattering of the objective function values decreased over periods. The runtimes were reasonably short: from 0.3 to 1.2 seconds per episode. TensorFlow ran with the GPUs of NVIDIA Quadro M1200 with a compute capability of grade 5.0. The `PPOAgent` performed best results according to level and convergence of the objective function, as show in Fig. 4.

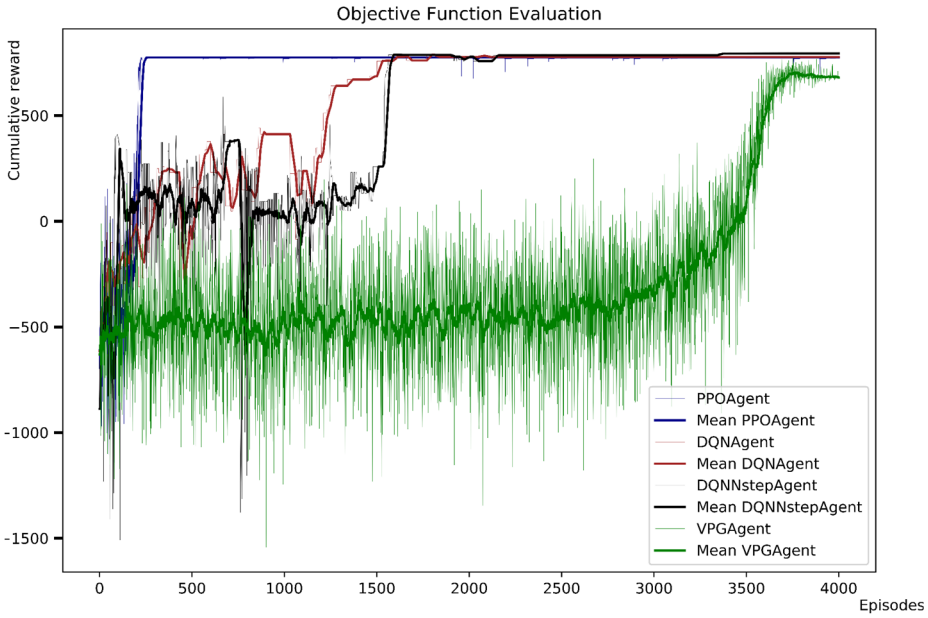

Fig. 4. : Evaluation of test scenario with six capacities, 2500 operations and 30 days horizon

4 Conclusion and Future Work

The proposed Deep Planning Methodology is an advanced implementation paradigm for planning processes in companies. It allows satisfactory planning results by developing a learning environment and customizing an agent without the need to implement all business rules and optimization algorithms explicitly. In the course of the CRP implementation a challenge emerged, how to tune hyper-parameter. Some hyper-parameters are sensitive concerning convergence and level of planning results. To meet the challenge many test scenarios are recommended to get a comprehensive insight, how to achieve reliable and reproducible results. To sum up, it is fascinating, how complex decision problems can be solved by reinforcement learning algorithms. In the future, I expect more Eureka moments for other planning tasks in companies, which have been unable to be successfully solved by current business applications until today.

References

1. Sutton, R.S., Barto A.G.: Reinforcement learning: an introduction. 2nd edn. MIT press, Cambridge (2018).
2. Papadimitriou, C.H., Tsitsiklis, J.N.: The complexity of markov decision processes. Mathematics of Operations Research Vol 12, No. 3, 441-450 (1987).
3. Abadi, M. et al.: TensorFlow: A System for Large-Scale Machine Learning. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16). USENIX Association, Savannah (2016). www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi, last accessed 2019/1/17.
4. Gulyássy, F.; Vithayathil, B..: Kapazitätsplanung mit SAP®. Galileo Press, Boston (2014).
5. Dickersbach, J.T.: Supply Chain Management with APO: Structures, Modelling Approaches and Implementation Pecularities. 3rd edn., Springer, Berlin, (2009).
6. TensorFlow Documentation. github.com/tensorflow (2017), last accessed 2019/1/17.
7. Géron, A.: Hands-On Machine Learning with Scikit-Learn and TensorFlow. O'Reilly Media, Sebastopol (2017).
8. Leukert B., Müller J., Noga M.: Das intelligente Unternehmen: Maschinelles Lernen mit SAP zielgerichtet einsetzen. In: Buxmann P., Schmidt H. (eds.) Künstliche Intelligenz, pp. 51-62. Springer Gabler, Berlin (2019). doi: 10.1007/978-3-662-57568-0_3
9. Kuhnle, A., Schaarschmidt, M., Fricke, K.: Tensorforce: a TensorFlow library for applied reinforcement learning. https://github.com/tensorforce/tensorforce (2017), last accessed 2019/1/8.
10. Schaarschmidt, M., Kuhnle, A., Ellis, B., Fricke, K., Gessert, F., Yoneki, E.: LIFT: Reinforcement Learning in Computer Systems by Learning from Demonstrations. arXiv preprint:abs/1808.07903v1 (2018).
11. TensorForce Documentation Release 0.3.3. media.readthedocs.org/pdf/tensorforce/latest/tensorforce.pdf (2018), last accessed 2019/1/17.
12. Mnih, V. et al.: Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). doi: 10.1038/nature14236
13. Gu, S., Lillicrap, T., Sutskever, I., Levine, S.: Continuous Deep Q-Learning with Model-based Acceleration. arXiv:1603.00748v1 (2016).
14. Van Hasselt, H., Guez, A., Silver, D.: Deep Reinforcement Learning with Double Q-learning. arXiv: cs.LG/1509.06461v3 (2015).
15. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning 8(3-4), 229–256 (1992).
16. Schulman, J., Levine, S., Abbeel, P., Jordan, M.I., Moritz, P.: Trust region policy optimization. In: Proceedings of the 32nd International Conference on Machine Learning (ICML-15). pp. 1889–1897, arXiv: 1502.05477v5 (2017).
17. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal Policy Optimization Algorithms. arXiv preprint:1707.06347v2 (2017).
18. Mnih, V. et al.: Asynchronous Methods for Deep Reinforcement Learning. arXiv: 1602.01783v2 (2016).
19. Lillicrap, T., et al.: Continuous control with deep reinforcement learning. arXiv: 1509.02971v5 (2016).
20. Kanit, W. et al.: Visualizing Dataflow Graphs of Learning Models in TensorFlow. IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, 24(1), 1-12 (2018).
21. The Tensor Board repository on GitHub. http://github.com/tensorflow/tensorboard, last accessed 2019/1/31.