Deep Q-Learning versus Proximal Policy Optimization: Performance Comparison in a Material Sorting Task

Reuf Kozlica
Information Technologies and Digitalisation
Salzburg University of Applied Sciences
Salzburg, Austria
reuf.kozlica@fh-salzburg.ac.at

Stefan Wegenkittl
Information Technologies and Digitalisation
Salzburg University of Applied Sciences
Salzburg, Austria
stefan.wegenkittl@fh-salzburg.ac.at

Simon Hirländer
Artificial Intelligence and Human Interfaces
Paris Lodron University Salzburg
Salzburg, Austria
simon.hirlaender@plus.ac.at

***Abstract*—This paper presents a comparison between two well-known deep Reinforcement Learning (RL) algorithms: Deep Q-Learning (DQN) and Proximal Policy Optimization (PPO) in a simulated production system. We utilize a Petri Net (PN)-based simulation environment, which was previously proposed in related work. The performance of the two algorithms is compared based on several evaluation metrics, including average percentage of correctly assembled and sorted products, average episode length, and percentage of successful episodes. The results show that PPO outperforms DQN in terms of all evaluation metrics. The study highlights the advantages of policy-based algorithms in problems with high-dimensional state and action spaces. The study contributes to the field of deep RL in context of production systems by providing insights into the effectiveness of different algorithms and their suitability for different tasks.**

***Index Terms*—Reinforcement Learning, Material Flow System, Petri Nets, Deep Q-Learning, Proximal Policy Optimization**

I. Motivation

There is a plethora of simulation environments used for RL tasks. Among the most popular ones are MuJoCo, Gazebo, Webots, Gymnasium and PyBullets [1]. Most of the well-known environments rely on a physics engine, where the real world is being portrayed on a very low abstraction level. Designing such a simulation environment on this abstraction level can be a difficult and time consuming task. PNs represent a mathematical modelling language used to describe and simulate systems and processes. This way the simulation is being designed on a fairly high abstraction level, which helps to reduce the design complexity of such a system. In this work, we use PNs as a basis for solving RL tasks as shown in [2]–[4] and as been used in [5].

Reuf Kozlica and Simon Hirländer are supported by the Lab for Intelligent Data Analytics Salzburg (IDA Lab) funded by Land Salzburg (WISS 2025) under project number 20102-F1901166-KZP.

The main bottleneck in programming this kind of production tasks is the effectiveness of algorithms being used, thus this paper examines two different modern approaches on solving the RL problem: value approximation and policy approximation.

II. Related Work

Riedmann et al. conducted an experiment applying traditional Q-learning to a task using a PN model as a simulation environment. They showed that a PN simulation model can be a suitable basis for training a RL agent [3] and presented strategies for developing a supervisory controller using deep RL in a production context [4].

They designed a case study including a simple sorting task on a material flow production line shown in Fig. 2. The facility consisted of an entry point, rotary table, assembly station, storage and exit. At the entry point the lower and upper part of the product which is to be transported are placed on a transport carriage. Afterwards, the product is transported to the assembly station using conveyor belts and the rotary table. At the assembly station a rivet is being installed, which connects the lower part of the product with the upper part. The product is labeled either blue or green. The green products are transported to the storage and blue products are to be transported to the exit station.

They have shown that using traditional Q-learning, the agent has been able to correctly assembly and sort approximately 1.5 products of 3 given in each episode.

III. State of Research

This section provides a comprehensive review of the current literature on RL, DQN and PPO.

A. *Reinforcement Learning*

RL is a type of machine learning that seeks to emulate the way humans learn, particularly in their early years when exploration is crucial. It involves learning through interaction with an environment, making it a computational approach to learning [6].

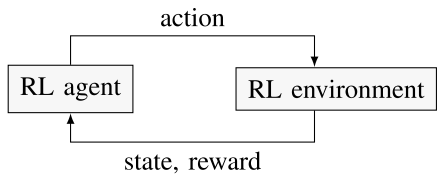

Fig. 1. Standard Reinforcement Learning Setup.

In a typical RL setup (as depicted in Fig. 1), there are two main components: the agent and the environment. The agent is in a particular state $s \in S$ and can perform actions $a \in A$, which may belong to either discrete or continuous sets and can be multi-dimensional [7]. The action performed by the agent changes the state of the environment, causing a transition and is evaluated by the environment, which sends a reward R_t to the agent after each step. The agent's objective is to maximize the accumulated reward over time. During the interaction with the environment, the agent tries to find a policy π that maps states to actions, and maximizes the overall return $G = \sum_{k=0}^{\infty} \gamma^k R_{k+1}$, where $\gamma < 1$ [8]. The policy can be either deterministic or probabilistic, with the former using the same action for a given state $a = \pi(s)$, while the latter maps a distribution over actions when in a specific state $a \sim \pi(s, a)$.

B. *Deep Q-Learning (DQN)*

The idea of Q-learning has been introduced in 1989 by Watkins and proven in detail by [9]. It is based on a so called state-action value function, also known as Q-function. This function of a given policy π, $Q^{\pi}(s, a)$, represents the expected return of a trajectory starting from a specific state s, taking a specific action a and following the policy π thereafter. The optimal policy $Q^*(s, a)$ is defined as the maximum return that can be achieved, when starting from state s, taking an action a and following the optimal policy thereafter. Using the Bellman optimality equation performing an iterative update of the Q-values, the optimal Q-function can be approximated. It has been shown that this way the $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$ [10].

In order to perform the iterative update of the Q-function, a Q-table, containing all pairs of s and a, must be used. Dependent on the application the state space and action space can become large, causing out-of-core situations, thus being impractical for usage. Instead of using a Q-table, Deep Q-learning trains a neural network to estimate the Q-values, which acts as a function approximator. This way the neural network with parameters θ serves as an approximate of the optimal function: $Q(s, a; \theta) \approx Q^*(s, a)$.

Using nonlinear function approximators in the RL domain is known to be unstable and can even diverge when such an approximator is used to represent the Q-function [11]. To tackle this problem, a biologically inspired mechanism named experience replay has been introduced. The agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored at each time-step t in a data set $D_t = \{e_1, ..., e_t\}$. In the training process, the Q-learning updates are performed using mini-batches of experience drawn uniformly randomly from the pool of samples named replay buffer [10].

Using Deep Q-learning, authors of [12] evaluated the performance of the same network for 49 different tasks on the Atari 2600 platform, which is designed to be difficult and engaging for human players. This agent outperformed other existing RL methods on 43 games and reached the level of professional human game testers on all 49 games.

C. *Proximal Policy Optimization (PPO)*

Another approach for solving RL problems using neural networks as function approximators have been policy gradient methods. PPO is trying to tackle the known issues which other methods like vanilla policy gradient method (VPG) and trust region policy optimization (TRPO) [13] are dealing with: Vanilla policy gradient methods struggle with data efficiency and robustness and TRPO is fairly computationally expensive, as well as not suited for architectures that include noise [14].

The second-order optimization used by TRPO makes it computationally complex and difficult to scale up for large scale problems [15]. PPO is trying to improve those methods by introducing an algorithm which retains the data efficiency and performance of TRPO but only using first-order optimization.

The main contribution of this novel approach is the clipped surrogate objective, loss function which is to be optimized:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{1}$$

In the above equation, an expectation of minimum of two terms is being computed. The first term inside the minimum operator $r_t(\theta)\hat{A}_t$ is the standard policy gradient objective. $r_t(\theta)$ is defined as probability ration between the action under the current policy and the same action under the old policy $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. The second term clips the probability ratio, removing the possibility of moving r_t outside the interval $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyperparameter. In case of a positive advantage $\hat{A}_t > 0$, meaning that selected action has a positive effect on the outcome, the loss function flattens out when $r_t(\theta)$ gets too large. The clipped objective prevents taking a too far step from the current policy. Moreover, when the advantage is negative $\hat{A}_t < 0$ the loss function flattens out when $r_t(\theta)$ converges to zero, meaning particular action is less likely on current policy.

D. *Petri Nets (PNs)*

PNs constitute a well-known paradigm for specification and operation of different systems, which is able to process multiple independent activities at the same time. This ability

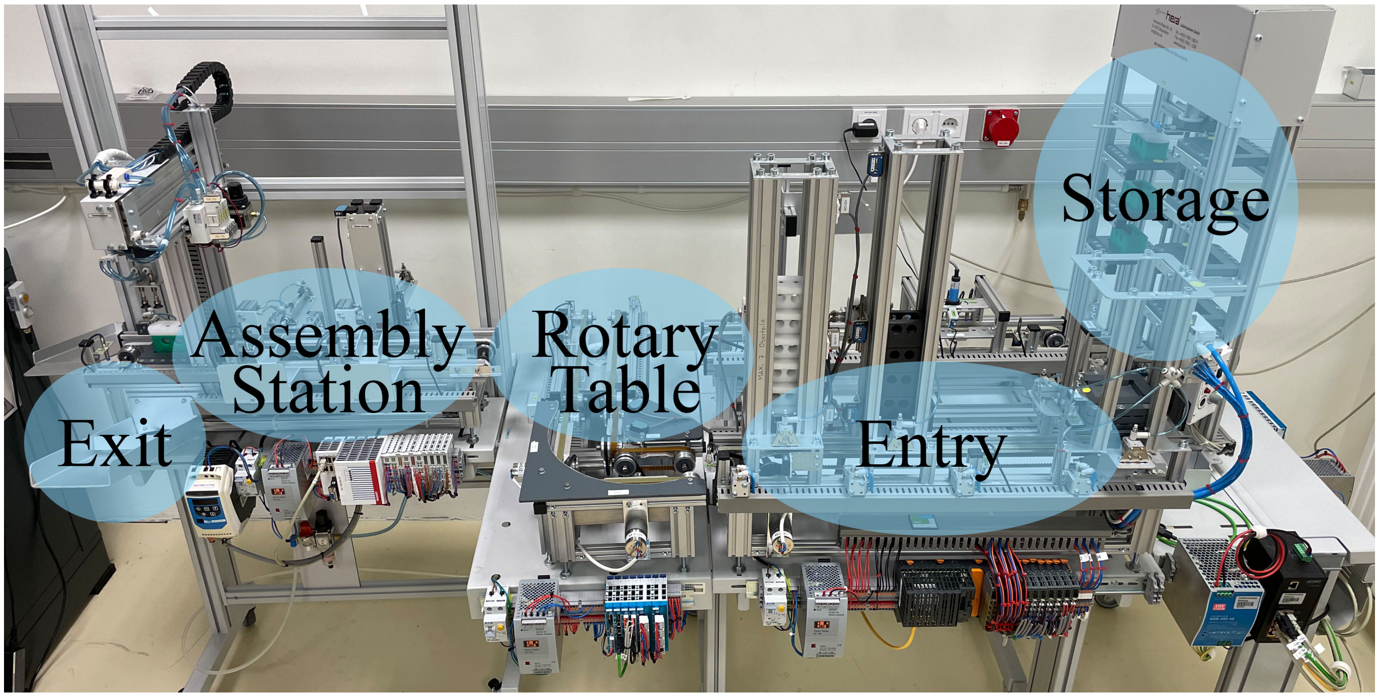

Fig. 2. Model factory placed in the Smart Factory Lab[1].

differentiates PNs from finite state machines, which are only able to operate one single state at a given moment. Petri nets thus are widely used as a tool for modelling, simulation, analysis and control of automated manufacturing systems [16]. Following the mathematical definition of PNs, a developed set of methodology and approaches is available in literature for theoretical and practical analysis of PNs [17].

A PN is a bipartite graph whose vertices consist of places (represented by circles) and transitions (represented by bars). Places and transitions are directly connected by arcs while tokens (usually represented by black dots) describe its state. Furthermore, two places are not allowed to be connected nor are two transitions allowed to be directly connected. A PN can formally be denoted as a quadruple $N = (P, T, Pre, Post)$, where: $P = p_1, ..., p_m$ represents a finite set of places, $T = t_1, ..., t_n$ is the finite set of transitions, and $Pre, Post : P \times T \rightarrow \mathbb{N}^{m \times n}$ are the pre- and post-incidence matrices that define direct arcs from places to transitions and from transitions to places, respectively [16]. Since 1962, when they have been proposed by Carl Adam Petri, PNs have evolved to meet needs of different systems, some of them include timed-, coloured-, interpreted-, stochastic-, and fuzzy models.

IV. Use Case and Simulation Environment

For this work the same simulation environment based on PNs proposed by [3] as described in Sect. II has been used. The original contribution of this paper is the comparison of two well-known algorithms in the world of RL. Deep Q-learning has been chosen as a representative of value based deep RL algorithms and PPO as a well-known representative of policy based deep RL algorithms.

The case study, as proposed by [3], has been created using a blue print of a model factory shown in Fig. 2. As introduced in Sect. II, this factory consists of 5 modules, namely: entry, rotary table, assembly station, storage and exit. The entry storage stores three different types of parts as illustrated in Fig. 3. The first part is the transport carriage, which is used for moving goods on the conveyor belts. The transport carriage is moving the other two parts which are stored in the entry storage: lower and upper part of the product. Moreover, the entry is also responsible for the assembly of the listed parts. After the parts have been assembled at the entry point, the goods are to be transported to the rotary table where they can be transported either to the storage point, to the assembly station or to the exit.

The model factory is designed to be representative for a real production system. Hence, it includes aspects like the transportation of goods using conveyor belts and a rotary table. Also the product modification is represented through the assembly station. Lastly, the storage of goods is found in both entry point and the main storage unit.

A. Task Definition

The task formulated in this use case is to transport and assemble the goods through this model factory following defined rules. Therefore, a specific number of goods $G = \{g_1, g_2, ..., g_{n_{max}}\}$ of a random color are being placed at the entry point during each simulation run. First, the goods are to be transported to the assembly station. At the assembly station two rivets are being installed into each product, as depicted in Fig. 3, connecting the upper part to the lower part of the product. After the rivets have been installed, the blue products must be transported to the exit, while the green products are to be transported to the storage.

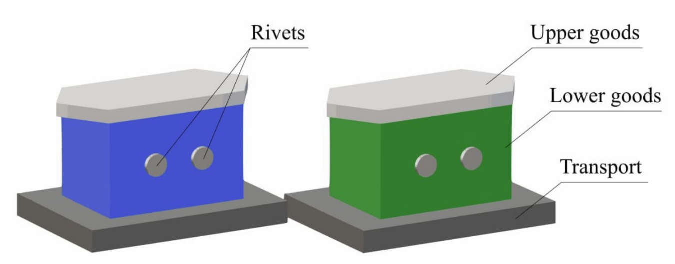

Fig. 3. Assembled products placed on the transport carriage.

B. *Simulation*

The PN based simulation proposed by [3] and extended in [4] has been used for the realization of this task. This model has been implemented in Python using the SNAKES toolbox [18] which has been slightly extended to facilitate the timed transitions. Thus, each transition is assigned a hidden place p_h enabling modelling the relative difference in execution times.

V. Implementation

This section details the implementation of the suggested case study introduced in Sect. IV.

A. *State and Action Space*

The action space of the agent is defined by the PN transitions to be controlled. Similar to [19], a dedicated action is being assigned to every PN transition T turning it on or off. Since there may be states where no transition should be fired, for example when firing a transition would result in a collision later, a non-action has been added. Respectively, the action space consists of 12 actions encoded as a vector $\vec{a} = [a_1, a_2, ..., a_{12}]^T$, $a_i \in \{0, 1\}$. The number $\#A$ of actions is thus depending on the number of actuators in the given production line.

The state space comprises all places P contained in the PN model. In order to translate the PN state to the state space of the agent, it is necessary to consider the similarities between places. Therefore, all storage places are encoded using a scalar value, since different tokens in storage places do not require separate treatment based on the color of the product. However, all other places are one-hot encoded, as tokens must be handled differently based on their color. One-hot vectors are generated with a number of components corresponding to the number of possible distinct token values for the respective place. Thus, the resource places number of components in the encoding is 2, meaning resource available or unavailable. Moreover, the number of components of all other places is either 4 or 6, depending on whether assembled tokens can reach these places. Additionally, the hidden places p_h are also one-hot encoded with 5 as a number of components and included within the state space. Hence, the state has been encoded as a vector $\vec{s} = [s_1, s_2, ..., s_{101}]^T$, $s_i \in \mathbb{N}_0$ consisting of 101 components, as shown in table I.

[1]https://its.fh-salzburg.ac.at/forschung/smart-factory-lab/

TABLE I
State Space Encoding

Parameter	Count	Number of components
Resource place	2	2
Storage place	4	1
Regular place	5	6
Regular short place	2	4
Hidden place	11	5

B. *Reward Design*

The goal of the RL agent is to autonomously learn to transport and assemble the goods according to the task formulation described in Sect. IV-A. For the purpose of this paper two slightly different reward functions R_1 and R_2 are defined and compared. Reward

$$R_1 = \begin{cases} -1 & \text{collision occurred} \\ -0.5 & \text{product incorrectly sorted} \\ -0.01 & \text{invalid transition fired} \\ +0 & \text{(non-)transition fired} \\ +1 & \text{goal reached} \end{cases}$$

only focuses on the task of correctly assembling and sorting the given products. Therefore, a simple transition is not being penalized $r_{transition} = 0$. A detected collision in the production system represents an event which shall not occur under any circumstances and results in a negative reward of $r_{collision} = -1$. Products which are incorrectly sorted lead to a negative reward of $r_{incorrect} = -0.5$. Furthermore, transitions which do not affect the production system, i.e. firing a transition of an unoccupied place, are penalized with a small negative reward $r_{invalid} = -0.01$. The successful completition of the task leads to a reward of $r_{successful} = 1$.

The reward function R_2 however is also considering the number of transitions needed for the completion of the task, hence defining the reward $r_{transition} = -0.001$. It is important to keep this negative reward sufficiently small, since a disproportional increase would result in the agent constantly choosing the non-action, which gets no reward, all the time. The reward values are chosen iteratively by comprehensive experiments.

$$R_2 = \begin{cases} -1 & \text{collision occurred} \\ -0.5 & \text{product incorrectly sorted} \\ -0.01 & \text{invalid transition fired} \\ -0.001 & \text{(non-)transition fired} \\ +1 & \text{goal reached} \end{cases}$$

VI. Evaluation

To show the applicability of the described modelling method to a wider range of RL algorithms, the model is implemented and tested using DQN and PPO algorithms described

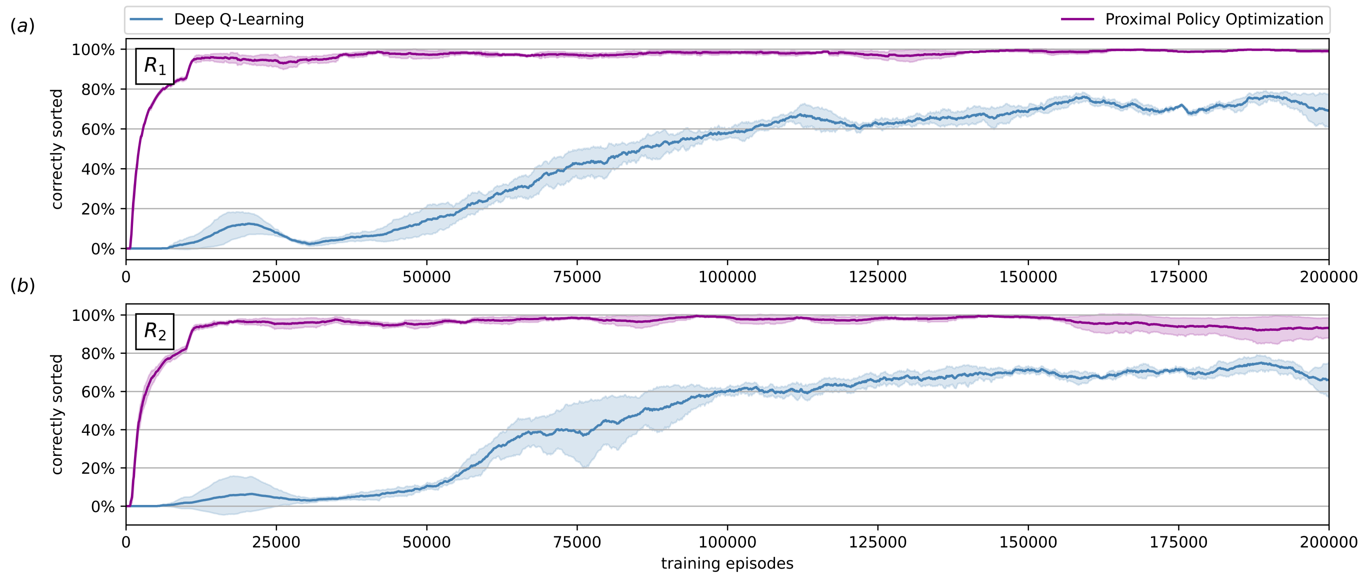

Fig. 4. Training process of Deep Q-Learning and Proximal Policy Optimization for rewards R_1 and R_2. After every 100 training episodes, the agent is evaluating the policy for 5 episodes. The evaluation results are smoothed for 200 data points.

in Sect. III. Both algorithms are specified to use a similar structure of the neural network. The dimensions of the approximation network are defined as following: input layer consisting of the state vector $\vec{s}$ with 101 components, two hidden layers with 200 nodes in the first layer and 100 nodes in the second layer and an output layer given by $\vec{a}$, consisting of 12 actions. Through an iterative process, the number of nodes within the neural network's layers was systematically reduced until the most efficient configuration capable of solving the given task was discovered, thereby optimizing the network's computational efficiency and minimizing its complexity.

The ϵ-greedy strategy, chosen as an exploration strategy for DQN, quantifies the probability that the agent will select a random action instead of choosing the action with the highest predicted value according to the current estimate of the Q-function (i.e., acting greedily). The exploration rate is initialized to 1 at the onset of training, and subsequently reduced linearly, in order to balance the agent's exploration and exploitation tendencies as it learns to navigate the environment. This progressive reduction in ensures that the agent transitions from a more exploratory behavior to a more exploitative one as its policy becomes more refined and robust, ultimately culminating in a minimum exploration rate of 0.1.

To ensure the comparability of the results both PPO and DQN are trained for 200,000 episodes having a maximum of 100 time steps for each episode. On an off-the-shelf hardware setup the full training process takes approximately 24 hours for the DQN and 40 hours for PPO. Each setup has been trained with 5 different seeds and all performance plots are showing the mean of all runs as a solid line, while the semi-transparent filled area shows the standard deviation.

A. *Analysis of Reward Design*

The analysis of reward design is a critical aspect of RL research. It refers to the process of selecting and designing the appropriate reward function that an algorithm uses to evaluate its performance and learn from it. The right reward function can significantly impact the performance and effectiveness of a reinforcement learning algorithm.

Fig. 4 shows the training process of both algorithms DQN and PPO for both rewards R_1 and R_2, described in Sect. V-B. The training process plot clearly shows that the PPO algorithm outperformed the DQN algorithm in terms of learning speed for both reward functions. This indicates that PPO was able to learn and adapt to the task more efficiently and effectively than DQN.

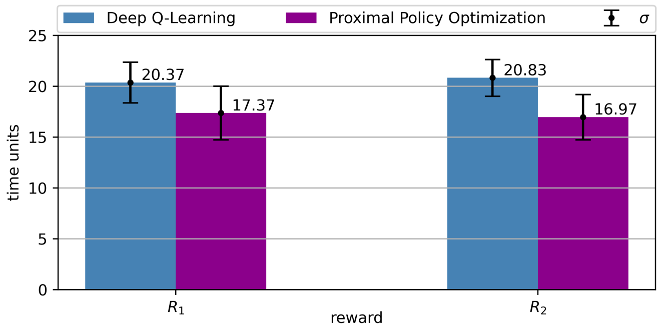

Fig. 5. Comparison of the learned policy using DQN and PPO: Time steps needed to complete the sorting task successfully.

Nevertheless, a direct comparison of the training performance of PPO in Fig. 4 (a) and (b) shows that even a small change in the reward design can cause instabilities of the learned policy. Especially when looking into the last 40,000 episodes in (b), where the policy is getting slightly worse. This could be caused by a phenomenon known as catastrophic forgetting, which can be overcome by not updating the model if the performance is getting worse.

Finally, all 5 trained policies for both algorithms in combination with R_1 and R_2 are evaluated for 100 episodes with a random sequence of products to be assembled and sorted. Fig. 5 shows the time the agent needs to correctly complete the task. For the PPO algorithm, the design of the R_2 reward function achieved to reduce the time needed to successfully accomplish the task. Specifically, the time for PPO was 0.4 time units faster. On the other hand, for the DQN algorithm, the same design did not have the same effect.

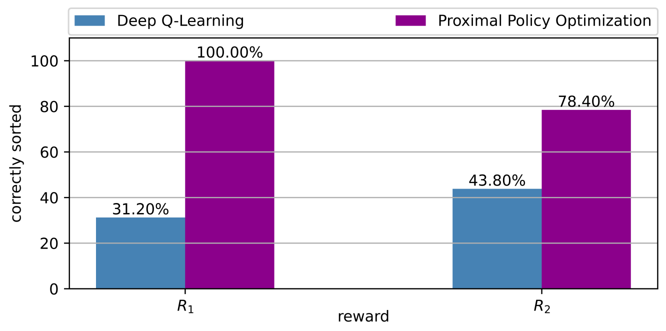

Fig. 6. Comparison of the policy trained using DQN and PPO: Percentage of episodes demonstrating successful sorting and assembling of all products by the RL agent for R_1 and R_2 reward designs.

The evaluation of trained policies in terms of the percentage of correctly sorted and assembled products during an episode is presented in Fig. 6. The results demonstrate a notable performance advantage of the PPO algorithm compared to the DQN algorithm for both reward designs.

However, further analysis of the results showed that the R_2 reward design actually managed to improve the performance of the DQN algorithm, while it had the opposite effect on PPO. These findings suggest that the impact of reward design on the performance of reinforcement learning algorithms is not always straightforward and can vary depending on the specific algorithm and task at hand.

VII. Conclusion and Future Work

The results highlight the importance of choosing the right algorithm for the task and the benefits of using more advanced reinforcement learning methods like PPO, removing the need of manual reward shaping presented by [4].

Nevertheless, the model has limitations that must be considered. The model assumes a fixed number of products in each simulation run, which may not reflect the real-world scenario accurately. Additionally, the model is limited to a maximum of 100 time steps, which may not capture the long-term effects of the manufacturing process. Moving forward, our research will focus on overcoming the aforementioned limitations to enhance the model's applicability in real-world settings. We will also explore the impacts of factory scaling by comparing conventional RL approaches with hierarchical methods.

References

[1] M. Körber, J. Lange, S. Rediske, S. Steinmann, and R. Glück, "Comparing popular simulation environments in the scope of robotics and reinforcement learning," 2021. [Online]. Available: https://arxiv.org/abs/2103.04616
[2] L. Hu, Z. Liu, W. Hu, Y. Wang, J. Tan, and F. Wu, "Petri-net-based dynamic scheduling of flexible manufacturing system via deep reinforcement learning with graph convolutional network," *Journal of Manufacturing Systems*, vol. 55, pp. 1–14, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0278612520300145
[3] S. Riedmann, J. Harb, and S. Hoher, "Timed coloured petri net simulation model for reinforcement learning in the context of production systems," in *Production at the Leading Edge of Technology*, B.-A. Behrens, A. Brosius, W.-G. Drossel, W. Hintze, S. Ihlenfeldt, and P. Nyhuis, Eds. Cham: Springer International Publishing, 2022, pp. 457–465.
[4] J. Harb, S. Riedmann, and S. Wegenkittl, "Strategies for developing a supervisory controller with deep reinforcement learning in a production context," in *2022 IEEE Conference on Control Technology and Applications (CCTA)*, 2022, pp. 869–874.
[5] G. Schäfer, R. Kozlica, S. Wegenkittl, and S. Huber, "An architecture for deploying reinforcement learning in industrial environments," in *Computer Aided Systems Theory – EUROCAST 2022*, R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, Eds. Cham: Springer Nature Switzerland, 2022, pp. 569–576.
[6] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, 2nd ed. MIT press, 2018.
[7] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.
[8] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, no. 1, pp. 237–285, 1996.
[9] C. J. C. H. Watkins and P. Dayan, "Technical note: Q-learning," *Mach. Learn.*, vol. 8, no. 3–4, p. 279–292, may 1992. [Online]. Available: https://doi.org/10.1007/BF00992698
[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. [Online]. Available: https://arxiv.org/abs/1312.5602
[11] J. Tsitsiklis and B. Van Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control*, vol. 42, no. 5, pp. 674–690, 1997.
[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.
[13] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," 2015. [Online]. Available: https://arxiv.org/abs/1502.05477
[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1707.06347
[15] Y. Wang, H. He, C. Wen, and X. Tan, "Truly proximal policy optimization," 2019. [Online]. Available: https://arxiv.org/abs/1903.07940
[16] C. Seatzu, "Modeling, analysis, and control of automated manufacturing systems using petri nets," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2019, pp. 27–30.
[17] I. Grobelna and A. Karatkevich, "Challenges in application of petri nets in manufacturing systems," *Electronics*, vol. 10, no. 18, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/18/2305
[18] F. Pommereau, "Snakes: A flexible high-level petri nets library (tool paper)," in *Application and Theory of Petri Nets and Concurrency*, R. Devillers and A. Valmari, Eds. Cham: Springer International Publishing, 2015, pp. 254–265.
[19] J. Zinn, B. Vogel-Heuser, F. Schuhmann, and L. A. Cruz Salazar, "Hierarchical reinforcement learning for waypoint-based exploration in robotic devices," in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, 2021, pp. 1–7.