Neural Computing and Applications
https://doi.org/10.1007/s00521-021-05848-4

ORIGINAL ARTICLE

Reinforcement learning methods based on GPU accelerated industrial control hardware

Alexander Schmidt[1] · Florian Schellroth[1] · Marc Fischer[1] · Lukas Allimant[1] · Oliver Riedel[1]

Received: 20 July 2020 / Accepted: 19 February 2021
© The Author(s) 2021

Abstract
Reinforcement learning is a promising approach for manufacturing processes. Process knowledge can be gained automatically, and autonomous tuning of control is possible. However, the use of reinforcement learning in a production environment imposes specific requirements that must be met for a successful application. This article defines those requirements and evaluates three reinforcement learning methods to explore their applicability. The results show that convolutional neural networks are computationally heavy and violate the real-time execution requirements. A new architecture is presented and validated that allows using GPU-based hardware acceleration while meeting the real-time execution requirements.

Keywords Reinforcement learning · PLC · Real-time · Industrial control · Manufacturing · GPU

1 Introduction

Reinforcement learning is a promising approach for manufacturing processes. Process knowledge can be gained automatically, and autonomous tuning of control is possible. These advantages can be used for complex nonlinear, time-variant or highly dynamic processes where a complex control program is needed. Typical industrial fields are, for example, welding, milling, and finishing.

Nowadays, the parameters of the control program are still optimized manually or an inaccurate model of the process is used. The development of such control programs is a time-consuming activity that requires expert knowledge. Likewise, even small changes to the production process can make repeated optimization necessary, again involving a great deal of time and effort.

To increase productivity, there is a need for approaches that can automatically create and optimize control programs. Therefore, reinforcement learning is evolving as a new research field to overcome manual work by applying autonomous black-box optimization.

Different approaches exist for enabling the use of reinforcement learning for manufacturing processes. However, to apply to industrial control hardware in a real manufacturing environment, they must apply the three following criteria. First, the learning approach must be efficient, meaning that the number of required experiments must be low. Each experiment is time- and material-consuming; therefore, the economically feasible amount of experiments is limited. Second, the learning approach must be able to cope with nonlinear models. Since the learning methods are applied to real industrial processes, the underlying models must be nonlinear to properly account for mechanical effects like friction, elastic materials, or gear backlash. External influences like temperature or humidity also have nonlinear effects and must be accounted for properly. Third, the resulting control programs trained by reinforcement learning methods must fulfill real-time (RT) constraints to run on industrial control hardware.

This article evaluates those requirements and is structured as follows. In Sect. 2, the required terminology for RT systems and reinforcement learning is introduced and they are mapped to the context of manufacturing processes. In Sect. 3, the related work is presented that shows how reinforcement learning is currently used in the

✉ Alexander Schmidt
alexander.schmidt@isw.uni-stuttgart.de

[1] Institute for Control Engineering of Machine Tools and Manufacturing Units, University of Stuttgart, Stuttgart, Germany

manufacturing industry. Furthermore, it is shown that deficits exist in terms of applicability of reinforcement learning on control hardware. In Sect. 4, three different reinforcement learning methods are evaluated to explore the space of requirements in the manufacturing domain. First, the NEAT algorithm is tested, which is suitable for low-dimensional problems and requires no prior knowledge but a large number of experiments. Second, an algorithm using Bayesian optimization (BO) with Gaussian processes (GP) is tested, which is also suitable for low-dimensional problems and requires only a small number of experiments but also requires prior knowledge of the process. Third, a convolutional neural network (CNN) is analyzed, which is suitable for high-dimensional problems. The results show that CNNs are computationally heavy and violate the RT execution requirements when applied on commercial off the shelf control hardware. Therefore, in Sect. 5 a new hardware architecture is presented that allows using hardware acceleration while meeting the RT execution requirements. The proposed architecture is validated in Sect. 6 by executing CNNs on a GPU and measuring the RT behavior. This novel architecture allows to run computational heavy neural networks, like CNNs, on industrial control hardware with hardware acceleration under RT constraints, and therefore enables the application on complex manufacturing processes, like welding.

2 Background

This section gives a brief definition of reinforcement learning and its requirements in the context of manufacturing. Therefore, RT is introduced in the context of programmable logic controllers (PLCs). Thereupon, the GPU as a common hardware acceleration method is introduced. A definition of neural networks is given, and their use as agents trained by reinforcement learning is described. The section ends with an architecture for reinforcement learning in PLCs.

2.1 Real-time systems

For understanding RT in the context of manufacturing, a brief introduction into the basics of RT systems is given according to Kopetz [32]. An RT system produces a result within strict temporal requirements. Therefore, the correctness of a result depends both on the logical correctness and on the correct time of delivery. The parameters latency τ and jitter J are used to describe temporal requirements. The latency is the time between the rise of an event and the rise of the related result. Due to inaccuracies like noise, the latency varies. The difference between the maximum and minimum value of the latency is the jitter.

2.2 Programmable logic controller

A PLC is a standardized and often used embedded system for controlling machines and processes in manufacturing. The functionality can be described from a system theoretical perspective. A system consists of inputs, outputs, and functions that describe the relationship between inputs and outputs. In manufacturing, a system can be a machine or a process, the inputs are sensors, the outputs are actuators and the functions are the control program in the PLC [28].

In manufacturing, many processes need strict RT constraints to guarantee the correct behavior of the process and to prevent damage of a workpiece or harming of the user. Thus, a PLC has RT constraints with typical latencies between 1 ms and 100 ms [50].

2.3 GPU

A GPU is hardware that consists of many individual processing units. Due to this hardware structure, the GPU can process many operations in parallel. Therefore, computations can be accelerated if they are parallelizable. Originally, GPUs were designed to accelerate graphics computations. Nowadays, GPUs and their programming are opened to many different kinds of computations like linear algebra and image processing. Two types of GPUs can be distinguished: on-board and dedicated. An on-board GPU resides on the same hardware board as the CPU and often shares the same memory. A dedicated GPU is connected via Peripheral Component Interconnect Express (PCIe) bus and has its own memory [47].

As the hardware design of a GPU differs from a central processing unit (CPU), a special compiler is needed which creates an executable program for a GPU. Furthermore, a GPU cannot run standalone but requires a CPU that offloads work. Therefore, tight coupling between the two types of processors is needed. To enable fast and easy development of GPU applications, several development kits exist like OpenCL,[1] Vulkan[2], OpenGL[3], and CUDA.[4] They provide a high-level interface for the CPU-GPU communication and a special compiler which can compile CPU and GPU code simultaneously.

2.4 Reinforcement learning

In reinforcement learning, a learning method in the machine learning complex, agents are developed and enhanced for a decision-making problem through

[1] https://www.khronos.org/opencl/.

[2] https://www.khronos.org/vulkan/.

[3] https://www.khronos.org/opengl/.

[4] https://developer.nvidia.com/cuda-zone.

rewarding. Hereby, the agent is an autonomous program interacting with the environment, aiming to maximize the final reward $R(r_{1:T})$ by taking the optimal action a_t regarding the current environmental state s_t [62]. The relation between agent, environment, and the state-action-reward principle is shown in Fig. 1a.

The relation between state, action, and reward is formulated as a finite, discrete-time Markov decision process (MDP) as described by Watkins [67], where decision making is modeled as a stochastic process. This stochastic process possesses the Markov property, defined as the conditional independence of a future state s' regarding all previous states $s_{i<t}$ and actions $a_{i<t}$. Consequential a state transition only depends on the current state s_t and action a_t with the conditional probability

$$P(s'|s,a) = P(s_{t+1} = s'|s_t = s, a_t = a) \tag{1}$$

of the state transition $(s \rightarrow s')$, with the respective reward $r = R(s, s')$ [16, p. 9-20], [62, p. 48] as shown in Fig. 1b.

Concluding, the objective of reinforcement learning is to find an agent's policy $\pi(a|s)$ which maximizes the cumulative future reward by choosing the optimal action for every given state. The iterative search for an optimal policy is based on the policy and cumulative reward of one or many past agents. The upgrade of an agent with an improved policy is commonly performed after a full execution of an agent, i.e., the cumulative reward of the previous policy is calculated.

2.5 Neural networks

To approximate a nonlinear system model, a neural network links single neurons to a layered network. In the following, a generalized introduction to feedforward neural networks is presented based on Duda et al. [14].

In a feedforward neural network, three types of neurons are present. The input layer consists of linear input neurons, which receive the input and normalize it if necessary. Following are one or many hidden layers consisting of nonlinear hidden neurons. A hidden neuron is composed of a linear net activation $u = \text{net}(\boldsymbol{\omega}, \boldsymbol{x})$ with the weight vector $\boldsymbol{\omega}$ and the vector of inputs $\boldsymbol{x}$ and the nonlinear activation function $\phi(\cdot)$ forming the neuron's output $z = \phi(u)$. The final layer provides the output of the neural network. If the neural network is fully connected, i.e., all neurons of a layer are connected with each neuron in the subsequent layer, we define the NN as a multilayer perceptron (MLP). This common definition does not satisfy the strict definition of a perceptron, which uses a threshold activation function for binary classification [54].

2.6 Convolutional neural networks

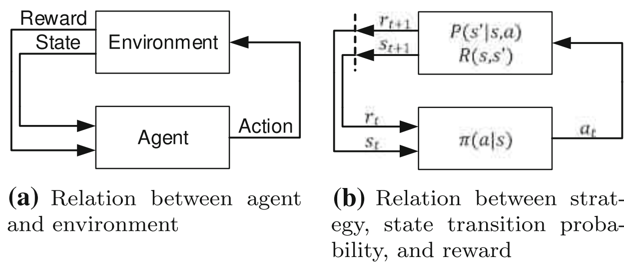
Since MLPs, as introduced in Sect. 2.5, are using one-dimensional arrays as inputs, they cannot recognize features spreading over multiple spatial dimensions [33]. Furthermore, MLPs are not translation invariant, i.e., the movement of an object through the picture is critical and the MLP needs to be trained for every possible object position. Thus, a major challenge in camera-based object detection is the treatment of correlation between neighboring pixels in multiple spatial dimensions, forming important features like edges of an object. This problem is addressed by the CNN [33]. Contrary to MLPs, the CNN can process multidimensional tensors as input. For that, the first layers in a CNN consist of convolutional neurons, each filtering areas of inputs by convolving with a kernel function. Thus, the raw image input is transformed into a multidimensional feature map, which is processed in an activation function to determine the presence of a feature in a certain pixel. Since common CNNs use padding for convolving the edges of an image, the image resolution is preserved. Therefore, pooling layers are commonly scattered between convolutional layers to decrease complexity and computation time [20]. After the last convolutional layer, multiple layers of fully connected perceptrons are used for object classification. If CNNs are used for object detection, bounding boxes are set around the detected objects. A comprehensive variety of algorithms for scaling and positioning of bounding boxes exists [20].

CNNs are commonly pretrained on large datasets and distributed through libraries or model-files. When applying a pretrained CNN on a specialized environment, additional training is necessary to enable object classification and detection for new classes of objects [20].

2.7 Architecture for reinforcement learning in manufacturing processes

In order to apply reinforcement learning in manufacturing processes with RT constraints, Schmidt et al. [55] present an architecture for reinforcement learning on an industrial controller using non-real-time (NRT) training and RT agents. Therefore, four modules are introduced as shown in Fig. 2. In summary, those four components are:

- *Learning Framework* containing the learning algorithm and agent training. Common frameworks can be used for learning, like Tensorflow, SciKit Learn, PyTorch.
- *Agent Exchange* converting raw agents of the Learning Framework to deterministic IEC-61131 [26] compliant code, which is executable on PLCs.



(a) Relation between agent and environment

(b) Relation between strategy, state transition probability, and reward

Fig. 1 Schematics of reinforcement learning based on [62, p. 38]

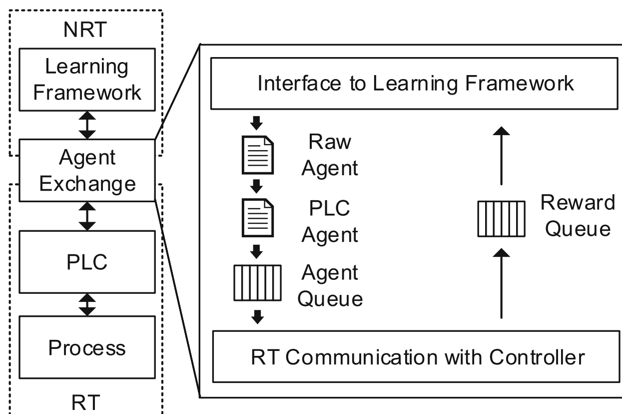


Fig. 2 Architecture for reinforcement learning on an industrial controller

- *PLC* controlling the actuators of an attached process. The converted agent is executed in the PLC similar to other IEC-61131 conform Function Blocks (FB) [26].
- *Process* to be controlled by reinforcement learning.

Since policy search and generation of new agents in a reinforcement learning framework are non-deterministic, the *Learning Framework* is executed in a NRT environment. To execute the agent in the RT PLC, the *Agent Exchange* bridges between the NRT- and RT-environment by using an RT communication channel to the PLC. A comprehensive introduction to the architecture's properties is found in [55].

3 Related work

This section shows how reinforcement learning is currently used in the manufacturing industry and clarifies the deficits existing in terms of the application of reinforcement learning on control hardware. Building upon, approaches for using GPUs in RT computations and integrating GPUs in control hardware are shown.

3.1 Reinforcement learning in manufacturing processes

Reinforcement learning is used in manufacturing processes that are hard to control or insufficiently describable. To implement reinforcement learning for controlling manufacturing processes, three different approaches are described in the related work. Those approaches are discussed below and compared in Table 1 regarding their suitability in manufacturing processes.

In the first approach (Sim-to-Real), learning is carried out in a simulation prior to the commissioning of the manufacturing process. After the completion of the learning phase, the resulting agent is manually transferred to the real controller. The advantage of this approach is the scalable and resource-efficient learning phase since no real industrial components are used. If the real system is modeled with sufficient accuracy, the agents learned in the simulation achieve good results in the real controller, as shown in [29] and [17, p. 295]. However, for dynamic industrial systems with multiple nonlinearities like Stribeck effects, sensor noise, adhesive friction, or vibration, there often is insufficient system knowledge to create an accurate model, as shown in [13].

In the second approach (supervisory task), learning is carried out directly in the real environment, but the agent's actions are limited to supervisory tasks outside of the control's RT environment. The interaction between agent and control is mostly realized via NRT-Communication. A common task for this approach is creating a sparsely sampled trajectory, where fine sampling is done separately in the RT-control [41, 57]. If RT is not crucial, this approach can also be used to directly control the actuators of a process, as shown in [35] and [27] for robots. Those works illustrate the capabilities of reinforcement learning in NRT robot control, which can be transferred to RT processes using the proposed architecture in this work.

In the third approach (control task), learning is carried out directly in the real environment as in the second approach, but the agent is capable of RT execution. Thus, usage of the agent in the RT control is enabled and the agent can directly control the process. For PLC applications, a proprietary machine-learning module is presented by Siemens and discussed in [60]. This module is designed for deep reinforcement learning but lacks computational power to enable PLC-RT inference of deep neural networks and is non-standardized. Equally, the architecture presented in [55] is limited to computationally non-complex reinforcement learning agents like shallow neural networks. Both works are limited to the experimental proof of concept of an architecture for reinforcement learning without providing a discussion on the suitability of

Table 1 Comparison of reinforcement learning approaches in manufacturing processes

	Sim-to-Real	Supervisory Task	Control Task
Efficient learning	●	◑	◑
Effectiveness on nonlinear models	◔	●	◕
Usable for RT PLC-Control	◕	○	◕

Legend: ● = fulfilled . . . ○ = not fulfilled

Table 2 Real-time scheduling approaches for GPUs

	Adapted driver	Persistent thread	Available features
Multiple GPU applications	●	●	◕
Library use	●	○	◑
Low development effort	○	◑	●

Legend: ● = fulfilled . . . ○ = not fulfilled

different reinforcement learning methods for manufacturing processes. This discussion is provided in 4. Contrary to PLC control, the third approach is widely discussed for robot control with the robot operating system (ROS), since the second version enables RT-control components to be created by the user. Furthermore, deep reinforcement learning is already applied in robot control, as shown in a survey by Arulkumaran et al. [5].

The related work using this approach demonstrates the capabilities of agent-based control using camera input to directly control the robot's actuators, but discussions about RT constraints are limited to inference times [42, 49]. Thus, those works cannot be transferred directly to RT-critical manufacturing tasks, where RT execution needs to be investigated. Another problem arising in the latter work is the need of hardware acceleration to achieve faster inference times for deep reinforcement learning with CNNs. Therefore, Morrison et al. [42] use a GPU for CNN inference, which can be transferred to RT manufacturing processes if enabling RT inference on a GPU. As a first step to achieving RT inference on GPUs, the RT-capability of GPUs needs to be discussed.

3.2 Real-time capability of GPUs

Accelerating complex computations with specialized hardware such as field-programmable gate arrays (FPGA), GPUs, or digital signal processors (DSP) is a common approach. Thereby, the GPUs are the easiest to use and the least development effort is needed [10]. Accelerating computations with GPUs is common and examples are given below. However, using a GPU acceleration in RT systems is a current field of research since GPUs are not specifically designed to comply with RT constraints.

When designing RT systems, a fundamental understanding of the system is necessary to guarantee predictable timings but GPUs are proprietary and the internals are not well documented. Therefore, no information on scheduling and memory access is available and the behavior can change without notice [43, 69].

To guarantee a predictable timing behavior, memory, driver, and scheduling must be considered. Several works exist with different approaches. In the following, an overview of these approaches is presented.

Considering **memory** is important if the CPU and GPU share a common memory. Exhaustive use of memory by either the CPU or GPU must be prevented, so that the memory access times are not badly influenced on the counterpart. Works like [1, 9] present mechanisms to regulate the memory usage and therefore guarantee memory timings. Since our tests are executed on a dedicated GPU, we do not need to consider memory.

The **driver** with the device interrupt handling is only mentioned in [15, 31, 56] where the authors present concepts to enable RT by modifying the driver. All other works do not consider the driver. Since modifying the driver is more complex and must be continuously developed for each GPU generation, we do not consider the driver approach.

Three types of **scheduling** approaches can be identified. The first approach is a customized driver with an RT scheduling [31, 56]. The second approach is called persistent thread style and uses scheduling on the application level by creating a persistent thread on the GPU. Inside of the persistent thread, a customized RT scheduling is implemented [11, 21, 36, 68]. Due to the lack of information on the GPU internals, there is research to reverse engineer the detailed behavior of GPUs [3, 48]. Based on this information, the third approach implements an RT scheduling by using only the available features like priorities of streams and preemption of the development kit.

Especially in the last years, more useful features for RT scheduling were introduced in the GPU architecture and development kits [19, 22, 34, 64, 65].

These scheduling approaches are compared in Table 2, and the criteria are explained in the following. When starting multiple applications using the same GPU, the driver approach may guarantee the required timings when all applications are considered in the RT scheduling. When using the persistent thread approach, the timings of the applications developed with the persistent thread style can always guarantee timing requirements, even if other NRT applications are executed. With the approach based on the development kit feature, time guarantees can only be given if all applications use the right settings.

Distincting the different scheduling approaches is important if using pre-built libraries. When developing algorithms for GPUs, high effort is required to gain good performance. Therefore, pre-built libraries exist like Cublass[5] for linear algebra or TensorRT[6] for neuronal network inference with high performance. These libraries are closed source. Therefore, the influence on the kernels and scheduling is limited to some features of the GPU development kits and the driver. Implementing the persistent thread style is not possible. Thus, RT guarantees can only be made when limiting access to the GPU. Works like [51] use pre-built libraries and analyze the execution time of the GPU application but do not mention any details about the RT-capability of the whole system.

Besides the works focusing on the RT behavior of the GPU, many works can be found which analyze GPUs as acceleration for time-critical workloads in autonomous driving, vision, and robotics like [12, 24, 43, 46, 51, 66]. These works focus on the execution time of the accelerated workload. There is no consideration of the RT-capability of the whole CPU-GPU system. If measurements are made, no pre-conditions are mentioned, in which the measured timings can be achieved. Only in [63] a detailed view on the system and measurement is given, but the paper considers a heterogeneous CPU-GPU scheduling to reduce the overall execution time in ROS. Measurements show 3000 ms for the execution of the GPU application, which exceeds the target time of 4 ms in Sect. 2.7.

In conclusion, it can be said that many approaches were presented in the last decade to overcome the unavailable RT-capability and the lack of information on GPU internals. Therefore, the integration of GPUs into RT systems is possible. Nevertheless, works focusing on the application of GPUs mostly do not consider the whole system of GPU and CPU but only show the acceleration of workloads.

[5] https://developer.nvidia.com/cublas.

[6] https://developer.nvidia.com/tensorrt.

3.3 GPU integration into a PLC

In the previous section, the RT-capability of GPUs is analyzed and shows possible approaches, but the integration into PLCs is not shown. To the best of our knowledge, no works can be found integrating a GPU into the RT part of a PLC. Only two works can be found, dealing with the integration of GPUs into control in general.

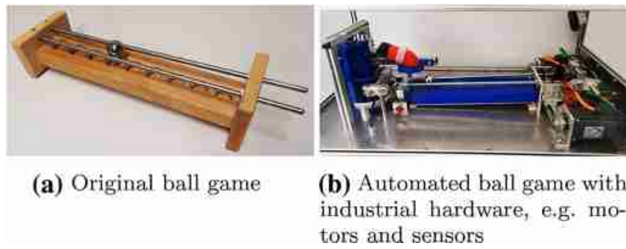
Maceina et al. [39] analyze the applicability of GPUs in RT control of fusion research. They use the multithreaded application real-time executor (MARTe) framework which provides a development framework for RT applications on different RT operating systems. They show matrix computation in 2.7 ms and jitter of 50 µs whereby the GPU is used exclusively. Furthermore, they analyze vision-based computations with a Sobel filter and measure time below 1 ms. Therefore, they present measurements in the lower millisecond range for a whole CPU-GPU RT system, but they give no details on the connection to the controller.

Bamakhrama et al. [6] use a co-simulation for the waver production to predict and mitigate temperature influences. The demand for computing resources is high. Different techniques are used to gain the required timings. The paper is targeting 70 ms and lower for big matrix multiplication requiring high bandwidth between CPU and GPU. There are no details about the connection to the control system. Instead, only the computational aspects of offloading the matrix multiplication to the GPU are considered.

In summary, the following insights can be gained in the related work chapter. For manufacturing processes, three approaches of reinforcement learning are developed; however, no work applied reinforcement learning for a nonlinear control process where the agent is executed in the RT environment of a PLC. This is partially due to the computational limitations of the PLC, which can be overcome by integrating GPUs into the RT part of PLCs. Since no work can be found, we analyze the applicability of the RT approaches for GPUs on PLCs in Sect. 5.3. Therefore the following chapter introduces three methods of reinforcement learning and investigates their applicability on a testbed (see Sect. 4.1) with industrial RT criteria.

4 Evaluation of reinforcement learning methods on industrial hardware

In the heterogeneous landscape of manufacturing processes, multiple process properties need to be considered to find a well-suited reinforcement learning method. Therefore, this section discusses three methods of reinforcement learning aiming to provide a survey suitable for a broad spectrum of manufacturing processes. Firstly, the



(a) Original ball game (b) Automated ball game with industrial hardware, e.g. motors and sensors

Fig. 3 Testbed for reinforcement learning methods

reinforcement learning algorithm NEAT is introduced. NEAT enables policy search without prior system knowledge. The respective NEAT agent performs well on a PLC. Secondly, the optimization algorithm BO is introduced and tested on the testbed. Contrary to NEAT, BO requires prior system knowledge but thereby achieves shorter learning times. Concluding the section of BO is a discussion on model-based and model-free reinforcement learning algorithms for different industrial applications. Thirdly, CNNs are introduced, which are suitable as agents for deep reinforcement learning methods with a high number of inputs or outputs. Because of the complexity of CNNs, RT constraints cannot be met using commercial off-the-shelf hardware. Thus, a new architecture with hardware acceleration is proposed in Sect. 5 for RT CNN inference.

4.1 Testbed

To validate the architecture and its enhancements, a testbed has been introduced in [55]. This testbed needs to fulfill three criteria to be suitable for validation of reinforcement learning approaches:

1. Complex nonlinear system model to validate the architecture for manufacturing processes which are hard to simulate.
2. Fast and autonomous restart of the experiment to enable reinforcement learning with a large number of iterations.
3. Intuitive setup to grasp the challenge of control design in the model.

A testbed fulfilling all three requirements is found in the ball game shown in Fig. 3a. Originally designed for children and persons with motor disorders, the game's objective is to move a steel ball uphill by actuating two metal rods, which are installed with a gradient. Since the movement of the ball depends on a complex nonlinear friction model between the ball and rods, this game is hard to simulate. The game is automated for validation of reinforcement learning using industrial components as shown in Fig. 3b. The cycle time for the RT PLC controlling the drives for rod positioning is set to 4 ms. To repeat the experiment autonomously, an actuated screw lift is

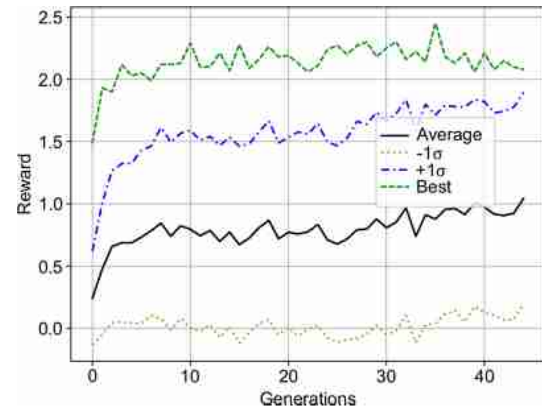


Fig. 4 Rewards for each generation of NEAT

installed, which moves the steel ball to the initial position on the rods. The reward of an agent is defined as the distance between the initial position on the lower end of the rods and the position where the ball falls through the rods. This distance is measured by two industrial laser sensors.

4.2 NEAT

In industrial applications without sufficient knowledge about the system behavior, reinforcement learning methods are suitable which independently develop an optimized strategy regarding the maximum cumulative reward [45]. One challenge is to limit the strategy space Π without preventing complex system models. Agents that are based on a neural network are suitable for this, since adding more hidden neurons will scale the model's complexity. Adding hidden neurons is dependent on the algorithm, and one prominent approach is to iteratively enlarge the neuronal network by evolution as in the NEAT algorithm [61]. In this approach, agents with an NN as phenotype are generated in generations, whereby the best agents are selected for reproduction. The corresponding genotypes are then recombined for the next generation to transfer the strong genes of the parent agents to the child agent. The resulting optimized neural networks are enlarged by random mutation, resulting in more complex system models as the generation increases.

NEAT is successfully tested in our prior work using the proposed architecture in Fig. 2 [55]. Using the automated ball game, NEAT is capable of evolving a policy that proactively uses the nonlinear friction of the rods to propel the steel ball toward the upper end, thus maximizing the reward. The resulting reward per generation is shown in Fig. 4 with a total of 6750 agents used in 45 generations. In the first generations, a comparatively steep learning process is apparent, which is due to the randomized initialization of the primal population. Furthermore, the exploitation of a local optimum toward the end can be recognized by the

Table 3 Pseudocode for Bayesian optimization

1:	Create a prior GP with prior knowledge about the system
2:	**while** $n \leq$ N **do**
3:	Select the next $x_n = \mathrm{argmax}_x a(x)$ by maximizing the acquisition function $a(x)$. This x_n corresponds to the next combination of control parameters to be tested on the system
4:	Query objective function at point x_n to obtain $y_n = f(x_n) + \epsilon_{noise_n}$. This corresponds to an experiment on the system with the control parameters x_n and receiving the noisy reward y_n
5:	Update the GP posterior
6:	Increment n
7:	**end while**
8:	Return a solution: either an evaluated point with the largest y_i, or the point with the largest posterior mean $\mu(x_i)$ [18]

narrowing of the distribution, i.e., the gap between maximum, standard deviation, and mean.

4.3 Bayesian optimization with Gaussian processes

Since the NEAT algorithm requires a large number of evaluations, it is unsuitable for applications where experiments are expensive and time-consuming. For this reason, the following section will consider BO as another approach. First, we briefly introduce the BO algorithm. Following this, using the automated ball game, we demonstrate experimentally that the algorithm can be implemented with the architecture presented in Sect. 2.7 to optimize control programs.

4.3.1 Motivation

BO is a highly data-efficient optimization method from the field of machine learning. BO derives its data efficiency by avoiding local optima, using all evaluations, and explicitly modeling noisy observations. Especially, the last point is an advantage when using data, obtained by experiments on real systems.

BO is already successfully applied to real technical systems, where optimization takes place directly through evaluations on the real system. For example, gait learning of a dynamic bipedal walker is done in [8] using BO. In [37] the authors perform gait optimization of a four-legged Sony AIBO ERS-7 robot, thereby showing that BO outperforms state-of-the-art local gradient approaches. In [44] BO is used for automatic learning of optimal throttle valve controller parameters from experimental data. In these papers, it is experimentally proven that BO can optimize parameters in different scenarios with only a few experiments and outperforms other approaches like manual tuning, gradient, grid, or random search. Thus, BO is a promising approach for the automated optimization of control programs for manufacturing processes.

4.3.2 Fundamentals of BO

A good overview of BO can be found in [4, 7, 18, 44, 58, 59]. A detailed description of the GPs for machine learning is given in [52]. Based on these sources, a summary is given below.

BO is a sequential, model-based approach to optimize an unknown black-box objective function $f(x)$ over a compact set $\mathcal{A}$:

$$x^* = \underset{x \in \mathcal{A} \subset \mathbb{R}^d}{\mathrm{argmax}}\, f(x). \tag{2}$$

The term black box function indicates that there is no evaluable mathematical representation of this function and its derivatives are unknown. The function f can only be queried at single points x_n thereby obtaining noisy observations $y_n = f(x_n) + \epsilon_{noise_n}$ with $\epsilon_{noise_n} \sim \mathcal{N}(0, \sigma^2_{noise})$.

Herein, the objective function is a mapping of the control parameters to the quality of the process, which is quantified by a reward value. By maximizing the objective function $f(x)$, we want to find the control parameters x^* that achieve the highest reward which means that the quality of the process is optimal. A query of the objective function at point x_n and obtaining the noisy observation y_n corresponds to an experiment on the system with the control parameters x_n and noisy observation of the reward y_n. To minimize the number of evaluations required, BO uses a combination of prior knowledge about the system and experimental data.

The resulting BO algorithm is shown in Table 3 as pseudocode. The algorithm is based on two key components, the GP and the acquisition function, which are described below.

A GP is a probabilistic framework which is used for nonparametric regression on the unknown objective function. We first prescribe a prior belief about the possible objective functions called prior by defining a prior mean

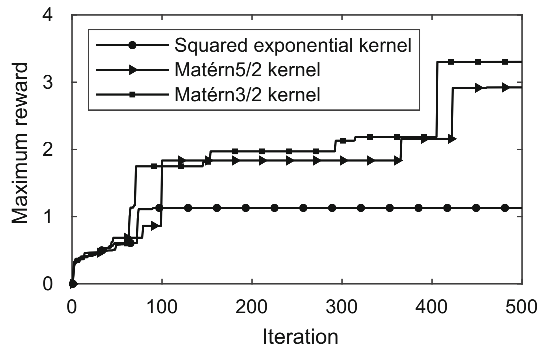


Fig. 5 Maximum reward over iterations for BO

and a covariance function. Whereas the prior mean represents the expected value of the objective function *f*, the covariance function, which is also called kernel, encodes prior assumptions about smoothness and rate of change of *f*. The prior is sequentially updated after querying and receiving new experimental data in order to produce a more informative posterior distribution over the possible objective functions. The posterior describes possible values of $f(x)$ at each point x and represents the current probabilistic estimate of the objective function we are optimizing. That is, the posterior predicts for each observed and unobserved point x a mean value $\mu(x)$ and a variance $\sigma^2(x)$.

Based on the current posterior GP, we can establish an acquisition function $a(x)$ to systematically select the next experiment. The acquisition function calculates the utility of candidate points x for the next evaluation of *f*. For this, the acquisition function trades off exploration and exploitation. For exploitation, the acquisition function is large at points with a large mean value and for exploration, the acquisition function is large at points with large variance and thus high uncertainty according to the current posterior GP. The acquisition function is maximized in order to select the next combination of control parameters x_n to try on the system, $x_n = \text{argmax}_x a(x)$.

4.3.3 Experiment and results

In the following, we want to experimentally examine whether BO can optimize a complex nonlinear process using the architecture presented in Sect. 2.7.

The physical setup is the automated ball game from Sect. 4.1 which is already used for validation of the NEAT algorithm in Sect. 4.2. Since BO is an optimization algorithm, the structure of the control function has to be manually specified in advance, in contrast to NEAT. To define the structure of this control function and its parameters, prior knowledge about the system is required. The specification of a fixed structure reduces the complexity of the optimization problem, and therefore, fewer experiments are necessary. Nevertheless, a design choice is made at this point and there may be solutions to the problem that are not covered by the selected control function. Based on the functioning of the ball game, we choose a sine function, as we expect that the ball can be moved upwards by cyclically opening and closing the rods. This control function directly determines the position of the rods as a function of time and can be adjusted via four control parameters, which we want to optimize using BO. For this, we implement the BO algorithm on the controller of the automated ball game based on the architecture from Sect. 2.7. To validate the architecture for the execution of the BO algorithm we perform 3 optimization runs of 500 iterations each with the three kernels squared exponential, Matérn5/2, and Matérn3/2 [40] in combination with the acquisition function Expected Improvement [30] with fixed hyperparameters.

Figure 5 shows the reward of the best experiment so far over 500 iterations and thus shows the learning progress. Using the Matérn5/2 and Matérn3/2 kernel within 500 iterations a reward of about 3.0 is achieved and the ball game is successfully optimized. Only the squared exponential kernel is not able to optimize the system and needs further investigation. Nevertheless, these results demonstrate that BO can be effectively implemented with the architecture from Sect. 2.7.

4.4 Evaluation of NEAT and Bayesian optimization

The advantage of the NEAT algorithm is that no model, and therefore no prior knowledge, of the system is necessary. On the other hand, to use BO, prior knowledge must be introduced in the form of the structure of the control function, the kernel, the acquisition function, and the hyperparameters. For this reason, the NEAT algorithm is better suited for complex problems where no prior process knowledge is available and thus the creation of a probabilistic model is difficult.

Using BO, only a few experiments are necessary to obtain a competitive control program. In the presented application of the ball game, only 500 experiments are necessary to optimize the system successfully. In contrast, NEAT requires 6750 experiments to create a comparable good control program. Due to its data efficiency, the BO approach is particularly suitable for the optimization of systems that require a long time for their evaluation, where data collection is expensive, or where only a few evaluations are available.

Based on the architecture proposed in Sect. 2.7, both algorithms can be executed directly on the local controller in an automated way. This requires only off-the-shelf hardware. Both approaches, NEAT and BO, can create or

optimize the control program of nonlinear and non-deterministic mechanics and are therefore applicable for the optimization of manufacturing processes.

4.5 CNNs for deep reinforcement learning

The preceding sections discussed methods for reinforcement learning which successfully developed strategies to control the rods in the testbed using a limited amount of input and output parameters. In this case, the agents directly used the ball's position as measured by the laser sensor. Using one or a few sensor values to determine a process variable is a common strategy in manufacturing processes. However, modern industrial applications commence using camera-based methods as in bin picking with robots [42]. Here, the position of chaotically arranged objects in a bin is determined by object detection and used as input for the robot control. Due to their precision in object detection, CNNs are frequently used for static bin-picking in NRT. For RT control of dynamic bin picking with moving objects, the position of those objects must be determined in RT. This is also true for the detection of the moving ball in the testbed.

To decide which CNNs should be tested for industrial RT suitability, two criteria are defined. Firstly, the classification accuracy should be comparatively high, since classification failure in manufacturing processes can result in cost or damage. Secondly, the inference should be comparatively fast to satisfy the time constraints in Sect. 2.7.

Both accuracy and inference time are already benchmarked for a vast variety of object classification CNNs. The selection is based on Almeida et al. [2] who benchmarked 26 different CNNs regarding inference times on different hardware. In this benchmark, ShuffleNet and ResNet 18 show a good trade-off between accuracy and inference time and are therefore tested in this work for object classification in their newest version (ShuffleNet-v2.1 [38] and ResNet 18 V2 [23]).

For object detection, we use you only look once (YOLO) v2 [53] based on the benchmark in Zhao et al. [70]. Furthermore, TinyYOLO v2 [53] is used as a lightweight version of YOLO v2 for faster inference times.

5 Architectures for PLC RT workload integration

For implementing reinforcement learning in manufacturing processes, a deep integration into the RT part of a PLC is necessary. Implementing reinforcement learning into a commercial off-the-shelf PLC is not possible without further customization of the control architecture. Schmidt et al. [55] propose an architecture which uses C/C++ and external libraries to integrate reinforcement learning into a PLC. This architecture requires less effort compared to just using IEC 61131 languages and the provided functionalities of a PLC environment. In addition, for using acceleration hardware like GPUs as needed for CNNs, access to the hardware driver is required. Therefore, the development kit provided by the manufacturer must be used to access the acceleration hardware which is currently not directly possible in a PLC environment. For these reasons, a detailed view of the programming of PLCs is needed. This section provides an overview of PLC programming architectures and identifies possible solutions within these architectures to integrate special RT workloads into a PLC. In this section, we only consider the programming of applications executed in the RT part of a PLC. NRT programming is not considered.

The PLC is an encapsulated environment to ensure the RT constraints, easy development, and maintenance for the whole life cycle of a machine [25]. Therefore, drivers, libraries, or binaries cannot be used directly in the PLC environment. The IEC 61131 PLC standard defines the requirements for the supported IEC 61133 programming languages. By extending the standard, manufacturers can extend the functionality of their PLC.

For a few years, many PLC manufacturers like Siemens, Phoenix, and Beckhoff have been opening their PLC environment to integrate C/C++ code, Matlab code, and others. There is no standardization of this functionality which results in different possibilities.

5.1 Architectures for PLC programming

We identified four general architectural types for programming in a PLC environment, which are presented in the following.

5.1.1 Architecture 1: IEC 61131

The first architecture is the classical programming in the IEC 61131 languages. The programming is restricted to the provided functionality of the PLC environment and to the IEC 61131 languages. Moreover, access to operating system functionalities is restricted. The scheduling of the developed applications is managed by the PLC environment and can be configured by the user. The PLC environment controls execution and RT performance. The user only has to develop the functionality of the control application. All provided functionalities of the PLC environment work out of the box but are limited.

Table 4 Comparison of the architectures

	Advantage	Disadvantage
Arch. 1	Scheduling is done by PLC environment	Only IEC 61131 libraries can be used
Arch. 2	Existing C/C++ libraries can be used or adapted. Scheduling is done by PLC environment	Only unrestricted common compilers enable the use of drivers and libraries
Arch. 3	Existing C/C++ libraries can be used or adapted	Scheduling must be user defined
Arch. 4	Any language and library can be used or adapted	Scheduling must be user defined. High communication latencies

5.1.2 Architecture 2: integrated high level language

The programming is done in a high-level language. Mostly C/C++ is used due to the RT-capability and wide distribution. The code is either compiled via a specific compiler provided by the PLC environment or by any common compiler supported by the platform and operating system. The integration of the control application into the PLC environment is realized via an interface provided by the PLC environment and must be implemented by the control application. Using a specific compiler there can be restrictions to the supported language features and to the inclusion of libraries and operating system functionalities. Using a common compiler no restrictions are made. The PLC environment handles the scheduling of the application and thus the RT performance. As the user can use high-level languages, reuse of code in high-level languages and libraries is possible.

5.1.3 Architecture 3: separate high level language

The programming of the control application is done in any language which supports the communication to the PLC environment and which is RT-capable. To communicate with the PLC environment an RT-capable communication mechanism must be used which is supported by the PLC environment like shared memory, sockets, or a proprietary communication mechanism. The control application is split into two parts. One part resides in the PLC environment and handles the sensors and actors. The other part is separated outside of the PLC environment. That is, the other part runs in a separate process on the same machine. Every library, binary, or driver supported by the operating system can be used. The user must handle the scheduling of the outside part himself.

5.1.4 Architecture 4: external system

The control application is developed for an external system like a cloud which is connected to the PLC via a field bus. Therefore, any language supported by the external system can be used. Similar to architecture 3 the control application has two parts, one in the PLC environment and one in the external system. The user must ensure the RT constraints for the execution of the application on the external system and the communication via the field bus. Higher latencies occur for the execution of the separated part due to the use of a field bus.

5.2 Comparison of the architectures

In Table 4, we compare the advantages and disadvantages of the architectures. Also, note that each PLC manufacturer offers different support for the presented architectures. The architecture proposed in Schmidt et al. [55] corresponds to architecture 2 with a restricted compiler.

5.3 Proposal for GPU usage in PLCs

In this section, we present an easy and directly usable proposal with a state-of-the-art PLC to enable the use of GPUs in RT critical areas of manufacturing. As stated in the introduction of Sect. 5, the use of acceleration hardware requires access to the driver and the development framework which is often provided by a library. Therefore, only architecture 2, 3, or 4 can be used. Within architecture 2 only unrestricted common compilers enable the use of drivers and libraries. In the architecture of the proposed solution in Schmidt et al. [55], a Siemens CPU 1515SP F PLC is used. This PLC only supports architecture 1, 2 with a restricted compiler, or 4. Since architectures 1 and 2 are not suitable for this application for the reasons already mentioned, only architecture 4 can be used with this PLC, which has higher development effort and latencies than the other architectures. To simplify the development, another

PLC is used for this work. We integrate the GPU into the Codesys Control for Linux SL,[7] which is available for evaluation for free. Moreover, the Codesys Control for Linux SL can be installed on any hardware. Therefore, hardware supporting dedicated GPUs via PCIe can be used and memory must not be considered as explained in Sect. 3.2. Codesys supports all four architectures. We use architecture 3 to have two separate and independent processes. As explained in Sect. 5.1.3, a communication between the PLC and the separate RT process is necessary. This communication is realized with the shared memory extension of Codesys.[8] To avoid jitter, two lock-free single-producer single-consumer queues are implemented based on the shared memory to communicate via messages. Moreover, busy waiting is used to minimize the communication latency between PLC and external RT processes to a minimum. The PLC process can enqueue message with the input data at any time. After enqueueing, the PLC process busy waits for an answer to the external RT process. Equally, the external RT process side busy waits for a new message. When a message is received, the GPU computation is executed regarding the input data. The result is enqueued as a message and sent back to the PLC which continues the control program. To conclude, the PLC offloads a task and waits for the result.

6 Experimental validation for GPU accelerated CNNs in PLCs

To demonstrate the applicability of GPUs in the RT context of PLCs, we use the proposed architecture in Sect. 5.3 and the presented CNNs in Sect. 4.5.

- ResNet 18-v2: Input 1x3x244x244
- ShuffleNet-v2.1: Input 1x3x244x244
- TinyYOLO-v2: Input 1x3x416x416
- YOLO-v2: Input 1x3x416x416

This demonstration validates the usage of CNNs as reinforcement learning agents in a production environment as the analysis in Sect. 4.5 states good performance. The performance of deep reinforcement learning with CNNs in a real manufacturing process must be shown in further works.

In time-critical environments, one challenge is the RT inference of CNNs. The inference of CNNs requires computational effort which can be accelerated by GPUs because of the parallelizability of the convolution. To achieve even shorter execution times on the GPU, tools like TensorRT are developed to speed up the inference by optimizing the network structure regarding GPU inference without losing accuracy. Furthermore, TensorRT brings an optimized execution engine for the inference of CNNs.

In our experiments, we use TensorRT for the optimization and inference of CNNs, as it is widely used. TensorRT is a pre-built library; hence, we must ensure exclusive access to the GPU and no other RT scheduling approaches presented in Sect. 3.2 can be used.

6.1 Measurement and setup

For reproducibility of the measurements, we give a detailed description of our test setup. The computer has the following specification.

- CPU: Intel core i5-6400 @2.7GHz (4 cores)
- RAM: 16GB DDR4
- OS: Linux - Ubuntu 18.04
- Kernel: 5.4.26 with PREEMPRT-RT patch rt17
- GPU: Dedicated GTX 980

To get good RT performance with low jitter, we isolate two cores and disabled typical sources of jitter. To get detailed information about the timings in the whole system, we measured three different times:

1. Communication between the PLC and the GPU process
2. Inference of the CNNs
3. Execution of the PLC task with the inference on the GPU

Furthermore, we measure both in idle and heavily loaded mode. To generate load on the computer, we use stress-ng by the command “stress-ng -a 2”. This is an important check for the RT-capability of the GPU driver as the system load can influence the GPU driver.

6.2 Communication between the PLC and the GPU process

In Fig. 6, the cumulative distribution function (CDF) of the timings for the communication between Codesys and the external RT process is shown. The round trip time as shown in Table 5 is used. In the external process, no work is executed and a response is directly committed to the PLC, so that only the communication timings are measured. The external process is pinned to an isolated core, and the SCHED_FIFO scheduling with priority 99 is used. The communication overhead with 4.5 µs and additional jitter of 9.7 µs is far below the typical cycle times of 1 to 100 ms. Therefore, our proposed architecture for integrating GPUs by a separate RT process in a commercial of the shelf PLC is a promising possibility.

[7] https://store.codesys.com/codesys-control-for-linux-sl.html.

[8] https://store.codesys.com/shared-memory-communication.html.

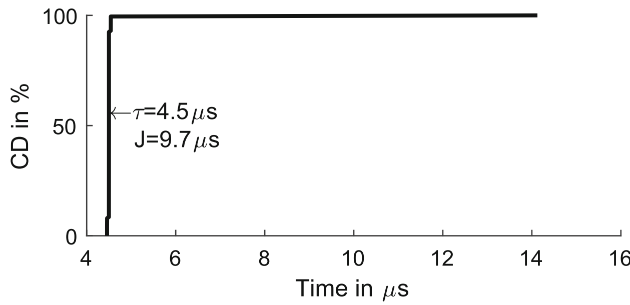


Fig. 6 CDF of PLC-GPU communication

Table 5 Time measurement for PLC-GPU communication

1:	SysTimeGetNs(start)
2:	Send input data to external RT process
3:	Busy waiting for response
4:	SysTimeGetNs(stop)
5:	duration = stop - start

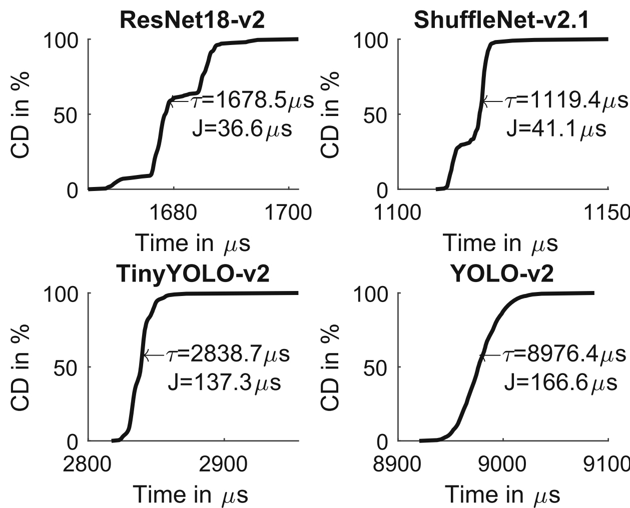


Fig. 7 CDF of CNN inference (no load)

6.3 Inference of the CNNs

In Figs. 7 and 8, the CDF of measurements for the inference of the CNNs with TensorRT is shown. For each measurement 5000 iterations are used. The process is pinned to an isolated core and the SCHED_FIFO with priority 99 is used. The timings are measured according to Table 6. When comparing the two figures with and without load, jitter differs in the range of 20 to 80 µs and latency in the range of less than 10 µs. These differences can have multiple causes like unknown behavior of the driver or

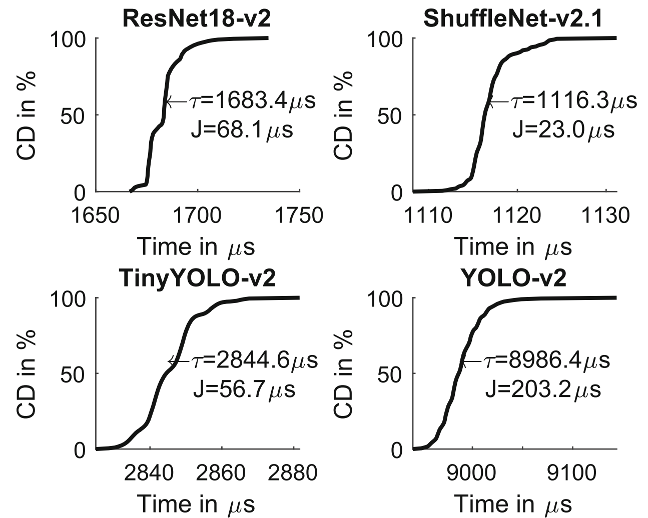


Fig. 8 CDF of CNN inference (with load)

Table 6 Time measurement for CNN inference

1:	Load optimized CNN to TensorRT and initialize memory
2:	Warm up as described in TensorRT documentation
3:	Clock_gettime(CLOCK_MONOTONIC, start)
4:	Copy input data to GPU
5:	Inference of the CNN
6:	Copy output data from GPU to main memory
7:	Clock_gettime(CLOCK_MONOTONIC, stop)
8:	duration = stop - start

TensorRT. Due to the acceptable influence, we do not investigate further.

6.4 Execution of the PLC with the inference on the GPU

For the timing measurements of the PLC task including the CNN inference, we use the TinyYOLO-v2 CNN. Similar code corresponding to Table 5 is used. In each PLC task cycle, the CNN is inferred with PLC data in the external RT process and the output is copied back to the PLC task. The PLC task is moved to the first isolated core and the external RT process to the second isolated core. We do a long-term measurement of 30 minutes and use the task statistics of Codesys. The statistics are available in the engineering tool of Codesys which are transmitted from the PLC to the engineering tool via the network. In the heavily loaded measurement, the network connection gets lost after one minute due to the high load and we can only get statistics for the first minute of the measurement. The statistics are presented in Table 7.

Table 7 Codesys timings with TinyYOLO-v2 inference

Property	No load in μs	With load in μs
Min. cycle time	3022	3017
Max. cycle time	3222	3361
Avg. cycle time	3050	3084
Jitter task	47	44
Min. jitter task	–23	–22
Max. jitter task	24	22

7 Conclusion

This work investigates the implementation of different reinforcement learning methods on industrial control hardware with respect to the applicability in manufacturing processes. Reinforcement learning algorithms can learn process knowledge automatically and are therefore a promising approach for the automated creation and optimization of control programs and the processing of data. While the creation and optimization of the reinforcement learning agents are not dependent on RT constraints, many control engineering applications require the execution of reinforcement learning agents with RT constraints to guarantee the correct process flow.

Therefore, an architecture was proposed by Schmidt et al. [55] where learning is carried out directly in the real environment. The experimental validation of this architecture is extended in this work by executing BO on the testbed to allow comparability between model-based and model-free algorithms. Overall, both NEAT and BO were capable of optimizing agents for solving the control task of a complex nonlinear process. However, the results show that model-based algorithms such as BO require significantly fewer experiments on the real process than model-free algorithms such as NEAT, but require prior knowledge of the process. In terms of performance, the agents of both algorithms are executable on the PLC in a cycle time of 4 ms.

To execute more complex reinforcement learning agents like CNNs for image recognition and object detection in PLC RT, the use of hardware acceleration is crucial. There are already numerous studies that investigate GPUs as acceleration for time-critical workloads in autonomous driving, vision, and robotics, making GPUs a promising approach for PLCs as well. However, most of the studies do not cover the entire CPU-GPU system but only show the acceleration of the workloads. Furthermore, to our knowledge, there are no studies that describe the integration of a GPU into the RT part of a PLC in detail.

In this paper, the connection of a GPU to a PLC is described in detail for the first time and examined regarding RT. A new way of RT workload integration is presented which uses a separate external RT process to execute the inference of the CNNs on the GPU. This external RT process communicates with the PLC via shared memory and overlaid messaging.

We experimentally validated that this architecture can meet industrial RT constraints when running different CNNs in the areas of image recognition and object detection. To optimize and execute the inference of the CNNs, we used the TensorRT tool. To get detailed information about the timings in the whole system, we measured different times in idle and heavy loaded mode. The communication time between PLC and GPU process is only a few µs and thus far below the cycle time of 1 ms to 100 ms typical in control engineering. Hence, the presented architecture is suitable for use in manufacturing processes.

The measurements of the inference duration of different CNNs on the GPU show that the object classification CNNs ShuffleNet-v2.1 and ResNet 18-v2 and the object detection architecture YOLO-v2 achieve time below 4 ms and thus become interesting for use in manufacturing processes. The inference times of the two object classification architectures even reach almost 1 ms and are therefore suitable for highly dynamic processes. The execution with or without load has only a small effect on latency and jitter. For the time measurement of the whole PLC-GPU system including communication and inference with the CNN TinyYOLO cycle times of about 3 ms were achieved. This result validates both the RT-capability of the whole PLC-GPU system and the concept of hardware acceleration of PLCs with a GPU with our proposed communication method.

We demonstrated how GPUs can be integrated into PLCs to accelerate complex reinforcement learning agents like CNNs. In future works, we want to examine the application of a CNN for camera based position tracking in RT on the testbed. Furthermore, we want to integrate the position tracking in a PLC, enabling a deep reinforcement learning agent to directly control the process based on the camera stream. The proposed architecture for GPU integration in PLCs has restrictions. Therefore, further work is needed to identify other possibilities enabling unrestricted use of GPUs and even less jitter. As TensorRT uses CUDA, we also want to analyze whether RT improvements can be made, to allow multiple GPU applications in parallel. As a first demonstration, we used a dedicated GPU in this paper which is not always available in manufacturing. Most PLCs come combined with hardware where onboard GPUs are available only. A deeper analysis of onboard GPUs is necessary. Here, different manufacturers and different tools must be analyzed.

Acknowledgements This research and development project was funded by the German Federal Ministry of Education and Research (BMBF) within the Innovations for Tomorrow's Production, Services, and Work Program (funding number 02K16C010) and implemented by the Project Management Agency Karlsruhe (PTKA). The authors are responsible for the content of this publication.

Funding Open Access funding enabled and organized by Projekt DEAL..

Compliance with ethical standards

Compliance with ethical standards The authors declare compliance with ethical standards.

Conflicts of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Ali W, Yun H (2018) Protecting real-time gpu kernels on integrated cpu-gpu soc platforms. In: 30th Euromicro conference on real-time systems, pp 19:1–19:22. https://doi.org/10.4230/LIPIcs.ECRTS.2018.19
2. Almeida M, Laskaridis S, Leontiadis I, Venieris SI, Lane ND (2019) Embench. In: The 3rd international workshop on deep learning for mobile systems and applications, pp 1–6. https://doi.org/10.1145/3325413.3329793
3. Amert T, Otterness N, Yang M, Anderson JH, Smith FD (2017) Gpu scheduling on the nvidia tx2: Hidden details revealed. In: IEEE real-time systems symposium, pp 104–115. https://doi.org/10.1109/RTSS.2017.00017
4. Archetti F, Candelieri A (2019) Bayesian optimization and data science, 1st edn. Mathematics and statistics. Springer eBooks, Springer, Cham
5. Arulkumaran K, Deisenroth MP, Brundage M, Bharath AA (2017) Deep reinforcement learning: a brief survey. IEEE Signal Process Mag 34(6):26–38. https://doi.org/10.1109/MSP.2017.2743240
6. Bamakhrama MA, Arrizabalaga A, Overman F, Smeets JP, van der Sommen K, van der Vossen R, Wagensveld J (2019) Gpu acceleration of real-time control loops. arXiv preprint arXiv:1902.08018
7. Brochu E, Cora VM, De Freitas N (2010) A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. arXiv preprint arXiv:1012.2599
8. Calandra R, Seyfarth A, Peters J, Deisenroth MP (2016) Bayesian optimization for learning gaits under uncertainty. Ann Math Artif Intell 76(1–2):5–23. https://doi.org/10.1007/s10472-015-9463-9
9. Capodieci N, Cavicchioli R, Valente P, Bertogna M (2017) Sigamma: Server based integrated gpu arbitration mechanism for memory accesses. In: 25th international conference on real-time networks and systems, pp 48–57. https://doi.org/10.1145/3139258.3139270
10. Che S, Li J, Sheaffer JW, Skadron K, Lach J (2008) Accelerating compute-intensive applications with gpus and fpgas. In: Symposium on application specific processors, pp 101–107. https://doi.org/10.1109/SASP.2008.4570793
11. Chen G, Zhao Y, Shen X, Zhou H (2017) Effisha: a software framework for enabling efficient preemptive scheduling of gpu. ACM Sigplan Not 52(8):3–16. https://doi.org/10.1145/3155284.3018748
12. Chen Z, Huang X, Ni Z, He H (2014) A gpu-based real-time traffic sign detection and recognition system. In: IEEE symposium on computational intelligence in vehicles and transportation systems, pp 1–5. https://doi.org/10.1109/CIVTS.2014.7009470
13. Dialami N, Chiumenti M, Cervera M, Agelet de Saracibar C (2017) Challenges in thermo-mechanical analysis of friction stir welding processes. Arch Comput Methods Eng 24(1):189–225. https://doi.org/10.1007/s11831-015-9163-y
14. Duda RO, Hart PE, Stork DG (2001) Pattern classification, 2nd edn. A Wiley-interscience publication. Wiley, New York
15. Elliott GA, Anderson JH (2011) Real-world constraints of gpus in real-time systems. In: 17th International conference on embedded and real-time computing systems and applications, pp 48–54. https://doi.org/10.1109/RTCSA.2011.46
16. Feller W (1971) An introduction to probability theory and its applications., 2nd edn. An introduction to probability theory and its applications, Wiley, New York
17. François-Lavet V, Henderson P, Islam R, Bellemare MG, Pineau J (2018) An introduction to deep reinforcement learning. Found Trends Mach Learn 11(3–4):219–354. https://doi.org/10.1561/2200000071
18. Frazier PI (2018) A tutorial on bayesian optimization. arXiv preprint arXiv:1807.02811
19. Golyanik V, Nasri M, Stricker D (2017) Towards scheduling hard real-time image processing tasks on a single gpu. In: IEEE international conference on image processing, pp 4382–4386. https://doi.org/10.1109/ICIP.2017.8297110
20. Gu J, Wang Z, Kuen J, Ma L, Shahroudy A, Shuai B, Liu T, Wang G, Cai J, Chen T (2018) Recent advances in convolutional neural networks. Pattern Recognit 77(C):354–377
21. Gupta K, Stuart JA, Owens JD (2012) A study of persistent threads style gpu programming for gpgpu workloads. In: 2012 innovative parallel computing (InPar), pp 1–14. https://doi.org/10.1109/InPar.2012.6339596
22. Hartmann C, Margull U (2019) Gpuart - an application-based limited preemptive gpu real-time scheduler for embedded systems. J Syst Archit 97:304–319. https://doi.org/10.1016/j.sysarc.2018.10.005
23. He K, Zhang X, Ren S, Sun J (2015) Deep residual learning for image recognition. In: IEEE conference on computer vision and pattern recognition, pp 770–778. https://doi.org/10.1109/CVPR.2016.90
24. Hermann A, Klemm S, Xue Z, Roennau A, Dillmann R (2013) Gpu-based real-time collision detection for motion execution in mobile manipulation planning. In: 16th international conference on advanced robotics, pp 1–7. https://doi.org/10.1109/ICAR.2013.6766473
25. IEC (2003) Programmable controllers—part 1: general information. Tech. Rep. IEC 61131-2003. International Electrotechnical Commission

26. IEC (2013) Programmable controllers - part 3: Programming languages. Tech. Rep. IEC 61131-2013. International Electrotechnical Commission
27. Inoue T, de Magistris G, Munawar A, Yokoya T, Tachibana R (2017) Deep reinforcement learning for high precision assembly tasks. In: IEEE/RSJ international conference on intelligent robots and systems, pp 819–825. https://doi.org/10.1109/IROS.2017.8202244
28. Isermann R (2005) Mechatronic systems: fundamentals. Springer, London
29. Jaensch F, Csiszar A, Scheifele C, Verl A (2018) Digital twins of manufacturing systems as a base for machine learning. In: 25th International conference on mechatronics and machine vision in practice, pp 1–6. https://doi.org/10.1109/M2VIP.2018.8600844
30. Jones DR, Schonlau M, Welch WJ (1998) Efficient global optimization of expensive black-box functions. J Glob Optim 13(4):455–492. https://doi.org/10.1023/A:1008306431147
31. Kato S, Lakshmanan K, Rajkumar R, Ishikawa Y (2011) Timegraph: Gpu scheduling for real-time multi-tasking environments. In: Proceedings of the 2011 USENIX conference on USENIX annual technical conference, p 2
32. Kopetz H (2011) Real-time systems: design principles for distributed embedded applications. Springer, Boston
33. Lawrence S, Giles CL, Chung Tsoi Ah, Back AD (1997) Face recognition: a convolutional neural-network approach. IEEE Trans Neural Netw 8(1):98–113. https://doi.org/10.1109/72.554195
34. Lee H, Faruque MAA (2014) Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform. In: Design, automation & test in Europe conference & exhibition, pp 1–6. https://doi.org/10.7873/DATE.2014.233
35. Levine S, Finn C, Darrell T, Abbeel P (2016) End-to-end training of deep visuomotor policies. J Mach Learn Res 17(1):1334–1373. https://doi.org/10.5555/2946645.2946684
36. Liang Y, Huynh HP, Rupnow K, Goh RSM, Chen D (2015) Efficient gpu spatial-temporal multitasking. IEEE Trans Parallel Distributed Syst 26(3):748–760. https://doi.org/10.1109/TPDS.2014.2313342
37. Lizotte DJ, Wang T, Bowling MH, Schuurmans D (2007) Automatic gait optimization with gaussian process regression. In: 20th international joint conference on Artifical intelligence, pp 944–949. https://doi.org/10.5555/1625275.1625428
38. Ma N, Zhang X, Zheng H, Sun J (2018) Shufflenet V2: practical guidelines for efficient CNN architecture design. Comput Vis - ECCV 2018:122–138. https://doi.org/10.1007/978-3-030-01264-9_8
39. Maceina TJ, Manduchi G (2017) Assessment of general purpose gpu systems in real-time control. IEEE Trans Nucl Sci 64(6):1455–1460. https://doi.org/10.1109/TNS.2017.2691061
40. Matérn B (1986) Spatial variation: Zugl.: Stockholm, Univ., Diss, Lecture notes in statistics, vol 36, 2nd edn. Springer, Berlin
41. Meyes R, Tercan H, Roggendorf S, Thiele T, Büscher C, Obdenbusch M, Brecher C, Jeschke S, Meisen T (2017) Motion planning for industrial robots using reinforcement learning. Procedia CIRP 63:107–112. https://doi.org/10.1016/j.procir.2017.03.095
42. Morrison D, Corke P, Leitner J (2018) Closing the loop for robotic grasping: a real-time, generative grasp synthesis approach. arXiv preprint arXiv:1804.05172v2
43. N Otterness, M Yang, S Rust, E Park, J H Anderson, F D Smith, A Berg, S Wang (2017) An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In: IEEE real-time and embedded technology and applications symposium, pp 353–364. https://doi.org/10.1109/RTAS.2017.3
44. Neumann-Brosig M, Marco A, Schwarzmann D, Trimpe S (2020) Data-efficient autotuning with bayesian optimization: an industrial control study. IEEE Trans Control Syst Technol 28(3):730–740. https://doi.org/10.1109/TCST.2018.2886159
45. Ng AY (2003) Shaping and policy search in reinforcement learning. PhD thesis, University of California, Berkeley
46. Nguyen VD, Nguyen TT, Nguyen DD, Jeon JW (2011) Real-time vehicle detection design and implementation on gpu. In: 11th international conference on control, automation and systems, pp 1287–1292
47. Olmedo IS, Capodieci N, Cavicchioli R (2018) A perspective on safety and real-time issues for gpu accelerated adas. In: 44th annual conference of the IEEE industrial electronics society, pp 4071–4077. https://doi.org/10.1109/IECON.2018.8591540
48. Otterness N, Yang M, Amert T, Anderson J, Smith FD (2017) Inferring the scheduling policies of an embedded cuda gpu. In: 13th workshop on operating systems platforms for embedded real time systems applications
49. Pane YP, Nageshrao SP, Kober J, Babuška R (2019) Reinforcement learning based compensation methods for robot manipulators. Eng Appl Artifi Intell 78:236–247. https://doi.org/10.1016/j.engappai.2018.11.006
50. Pritschow G (2006) Einführung in die Steuerungstechnik. Hanser, München
51. Ramwala OA, Paunwala CN, Paunwala MC (2019) Optimizing driver assistance systems for real-time performance on resource constrained gpus. In: IEEE Conference on information and communication technology, pp 1–4. https://doi.org/10.1109/CICT48419.2019.9066239
52. Rasmussen CE, Williams CKI (2006) Gaussian processes for machine learning. Adaptive computation and machine learning. MIT Press, Cambridge
53. Redmon J, Farhadi A (2016) YOLO9000: better, faster, stronger. In: IEEE conference on computer vision and pattern recognition, pp 6517–6525. https://doi.org/10.1109/CVPR.2017.690
54. Rosenblatt F (1958) The perceptron: a probabilistic model for information storage and organization in the brain. Psychol Rev 65(6):386–408. https://doi.org/10.1037/h0042519
55. Schmidt A, Schellroth F, Riedel O (2020) Control architecture for embedding reinforcement learning frameworks on industrial control hardware. In: 3rd International conference on applications of intelligent systems, pp 1–6. https://doi.org/10.1145/3378184.3378198
56. Schnitzer S (2019) Real-time scheduling for 3d rendering on automotive embedded systems. Dissertation, University of Stuttgart, Stuttgart
57. Schwung D, Csaplar F, Schwung A, Ding SX (2017) An application of reinforcement learning algorithms to industrial multi-robot stations for cooperative handling operation. In: IEEE 15th international conference on industrial informatics, pp 194–199. https://doi.org/10.1109/INDIN.2017.8104770
58. Shahriari B, Swersky K, Wang Z, Adams RP, de Freitas N (2016) Taking the human out of the loop: a review of bayesian optimization. Proc IEEE 104(1):148–175. https://doi.org/10.1109/JPROC.2015.2494218
59. Snoek J, Larochelle H, Adams RP (2012) Practical bayesian optimization of machine learning algorithms. In: 25th International conference on neural information processing systems, pp 2951–2959.https://doi.org/10.5555/2999325.2999464
60. Solowjow E, Ugalde I, Shahapurkar Y, Aparicio J, Mahler J, Satish V, Goldberg K, Claussen H (2020) Industrial robot grasping with deep learning using a programmable logic controller (plc). arXiv preprint arXiv:2004.10251v1
61. Stanley KO, Miikkulainen R (2002) Evolving neural networks through augmenting topologies. Evolut Comput 10(2):99–127. https://doi.org/10.1162/106365602320169811

62. Sutton RS, Barto A (2018) Reinforcement learning: an introduction, second, edition. Adaptive computation and machine learning. The MIT Press, Cambridge
63. Suzuki Y, Azumi T, Kato S, Nishio N (2018) Real-time ros extension on transparent cpu/gpu coordination mechanism. In: IEEE international symposium on real-time distributed computing, pp 184–192. https://doi.org/10.1109/ISORC.2018.00035
64. Tekin R, Zahaf HE, Lipari G (2019) Pruda: An api for time and space predictible programming in nvdia gpus using cuda. In: Junior workshop: JRWRTC—real-time net- works and systems 2019
65. Tsog N, Becker M, Bruhn F, Behnam M, Sjödin M (2019) Static allocation of parallel tasks to improve schedulability in cpu-gpu heterogeneous real-time systems. In: 45th Annual conference of the IEEE industrial electronics society, pp 4516–4522. https://doi.org/10.1109/IECON.2019.8926767
66. Venugopal V, Kannan S (2013) Accelerating real-time lidar data processing using gpus. In: IEEE 56th international midwest symposium on circuits and systems, pp 1168–1171. https://doi.org/10.1109/MWSCAS.2013.6674861
67. Watkins CJCH (1989) Learning from delayed rewards. Dissertation, University of Cambridge
68. Wu B, Liu X, Zhou X, Jiang C (2017) Flep: Enabling flexible and efficient preemption on gpus. ACM Sigarch Comput Archit News 45(1):483–496. https://doi.org/10.1145/3093337.3037742
69. Yang M, Otterness N, Amert T, Bakita J, Anderson JH, Smith FD (2018) Avoiding pitfalls when using nvidia gpus for real-time tasks in autonomous systems. In: 30th Euromicro conference on real-time systems, vol 106, pp 20:1–20:21. https://doi.org/10.4230/LIPIcs.ECRTS.2018.20
70. Zhao Z, Zheng P, Xu S, Wu X (2018) Object detection with deep learning: a review. IEEE Trans Neural Netw Learn Syst 30(11):3212–3232. https://doi.org/10.1109/TNNLS.2018.2876865

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.