IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 8, 2021, accepted January 12, 2022, date of publication January 26, 2022, date of current version February 7, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3146518

A Survey of Domain-Specific Architectures for Reinforcement Learning

MARC ROTHMANN AND MARIO PORRMANN, (Member, IEEE)
Institute of Computer Science, Osnabrueck University, 49090 Osnabrueck, Germany

Corresponding author: Marc Rothmann (mrothmann@uni-osnabrueck.de)

This work was supported by the European Union's Horizon 2020 Research and Innovation Program through the Very Efficient Deep Learning in IoT (VEDLIoT) Project under Grant 957197.

ABSTRACT Reinforcement learning algorithms have been very successful at solving sequential decision-making problems in many different problem domains. However, their training is often time-consuming, with training times ranging from multiple hours to weeks. The development of domain-specific architectures for reinforcement learning promises faster computation times, decreased experiment turn-around time, and improved energy efficiency. This paper presents a review of hardware architectures for the acceleration of reinforcement learning algorithms. FPGA-based implementations are the focus of this work, but GPU-based approaches are considered as well. Both tabular and deep reinforcement learning algorithms are included in this survey. The techniques employed in different implementations are highlighted and compared. Finally, possible areas for future work are suggested, based on the preceding discussion of existing architectures.

INDEX TERMS Domain-specific architectures, machine learning, deep learning, reinforcement learning, deep reinforcement learning, reconfigurable architectures, FPGA.

I. INTRODUCTION

Recent developments in reinforcement learning (RL) have shown promising results for the solution of sequential decision-making problems. With AlphaGo's success in the game of Go, the reinforcement learning approach has attracted much media attention [68]. While its capabilities have often been demonstrated by learning policies for video games, such as Atari games [52], Starcraft [80], and Dota [8], it can be applied to a wide variety of real-world scenarios. For example, RL has been used for scheduling in data centers [42] and for adaptive power management [12], [27], [44], [91]. Furthermore, reinforcement learning can also be applied to many problems in the domain of robotics, for example, navigation [9] or robotic manipulation [55]. Other interesting developments are agents with emergent tool use and cooperative behavior [3], [82].

Reinforcement learning often requires large amounts of time and computational resources. Hardware acceleration with GPUs and FPGAs can play a significant part in the continued progress of RL research and its application to real-world problems. GPUs are commonly used to accelerate deep learning processes. Hence, in Deep Reinforcement Learning (DRL), where neural networks are used as part of the learning algorithm, GPUs can easily be used to accelerate neural network computations.

However, the small batch sizes commonly used during DRL training can pose a problem for efficient GPU-based acceleration. Here, Domain-Specific Architectures (DSA) with specialized memory management and a degree of parallelism matching that of the problem domain can be advantageous. DSAs are often implemented on FPGAs due to their high degree of flexibility, faster development times, and lower cost when compared to ASICs. When applied to reinforcement learning, the possibility of fine-grained parallelism outside of the neural network updates and efficient use of on-chip memory in FPGA-based architectures promise further speed gains. Additionally, while FPGAs might not always outperform GPUs with respect to computation time, their utilization often improves energy efficiency.

This survey presents an overview of the state-of-the-art of reinforcement learning hardware acceleration. First, Section II introduces the theoretical background of reinforcement learning, followed by a brief examination of GPU-based

The associate editor coordinating the review of this manuscript and approving it for publication was Baker Mohammad.

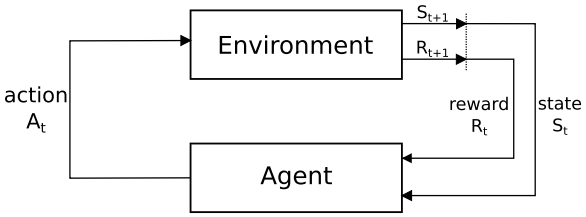


FIGURE 1. The basic reinforcement learning setup. An agent receives observations of the environment state and chooses actions to take in response. The environment rewards the agent based on its chosen action. The goal for the agent is to maximize the accumulated rewards [75].

acceleration of reinforcement learning in Section III. Then, Section IV presents existing hardware architectures for the acceleration of tabular and deep reinforcement learning. Subsequently, Section V highlights techniques used in state-of-the-art implementations, based on which directions for future work are proposed in Section VI. Finally, the survey ends with a conclusion in Section VII.

II. REINFORCEMENT LEARNING

In reinforcement learning, an agent learns to solve sequential decision-making problems. These problems can be modeled as Markov Decision Processes (MDP) [7]. At each timestep t, the agent receives an observation of the environment $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the state space, based on which it reacts by choosing an action $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t)$ is the set of all possible actions in state S_t. As a result of its action, the agent receives a reward R_{t+1}. This process is shown in Fig. 1. In an MDP, the state S_{t+1} and reward R_{t+1} depend only on the previous state S_t and action A_t, as described by the following function $p(s', r|s, a)$:

$$p(s', r|s, a) = Pr\{R_{t+1} = r, S_{t+1} = s'|S_t, A_t\} \tag{1}$$

While learning, the agent attempts to maximize the cumulative reward, represented by the expected discounted return G_t using a discount rate γ:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \quad 0 \leq \gamma \leq 1 \tag{2}$$

The agent follows a policy $\pi(a|s)$ which is a mapping from states to the probabilities of selecting each possible action. The value $v_\pi(s)$ of a state is equal to the expected discounted return while following the policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{3}$$

Similarly, the action-value function $q_\pi(s, a)$ is defined as the expected return starting from state s and taking action a:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{4}$$

The optimal action-value function is denoted as q_*:

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{5}$$

The Bellman optimality equation for the action-value function is:

$$q_*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \middle| S_t = s, A_t = a\right] \tag{6}$$

A similar Bellman optimality equation can be derived for the optimal value function v_*. Reinforcement learning algorithms use these optimality equations to improve the policy followed by the agent iteratively [75].

A. TABULAR REINFORCEMENT LEARNING

Classical reinforcement learning works on discrete state and action spaces, representing the action-value function as a table. The two most prominent tabular reinforcement learning algorithms are Q-Learning and SARSA. Q-Learning was first introduced in 1989 by Watkins [86], and its convergence was proven by Watkins and Dayan [85]. The following equation describes its update step:

$$\begin{aligned} &Q(S_t, A_t) \\ &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \end{aligned} \tag{7}$$

The quality $Q(a_t, s_t)$ of an action a_t in state s_t is updated by moving it closer to the sum of the received reward R_t and the expected future value $\gamma \max_a Q(s_{t+1}, a)$, assuming the agent would follow a greedy policy in the future. The action a_t is usually chosen by an ϵ-greedy policy. Many different variations of Q-Learning have since been introduced, as summarized in [34].

SARSA was introduced by [61] as a modification of the Q-Learning algorithm. While Q-Learning is an off-policy algorithm, SARSA is on-policy. An on-policy algorithm assumes the same policy used to select the current action will also be used to select future actions.

$$\begin{aligned} &Q(S_t, A_t) \\ &\leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \end{aligned} \tag{8}$$

That means, during the update, the value of the actual future action a_{t+1} is used instead of the maximum future value, as shown in (8).

B. DEEP REINFORCEMENT LEARNING

This section gives a short overview of a few model-free deep reinforcement learning (DRL) algorithms. Detailed surveys of state-of-the-art DRL methods can be found in [19], [46], and [40]. Classical reinforcement learning methods use tables to represent the value function. For large state and action spaces, as well as continuous state and action spaces, this approach is insufficient, as the tables become exceedingly large. In deep reinforcement learning, neural networks are used to approximate the value function to solve this problem. A major breakthrough in this domain were

Deep Q-Networks (DQN), which learned to play Atari games from pixel inputs at the level of a human expert [53]. In addition to replacing the Q-table with a Q-network, multiple improvements were made to enable stable neural network training. One problem of training neural networks in the context of sequential decision-making problems is the temporal correlation of the training data. In DQNs, this problem is solved by a technique called experience replay. Here, training samples (consisting of the state S_t, the action A_t that was performed, the reward R_t, and the subsequent state S_{t+1}) are stored in a replay buffer, and the Q-network is trained from randomly sampled mini-batches from this buffer. Additionally, the target network, a copy of the Q-network that is updated periodically, was introduced to hold the expected future value constant for some time [52]. Many small changes to the original DQN architecture have been proposed, a lot of which are combined in the Rainbow DQN algorithm [30].

DQN, as a value-based DRL algorithm, learns an action-value function from which it derives its policy. In contrast to that, policy gradient algorithms are a class of DRL algorithms that directly learn a policy that maps states to actions. With DQN, it is possible to apply reinforcement learning to continuous state and discrete action spaces. Policy gradient methods additionally extend reinforcement learning to continuous action spaces. A policy gradient algorithm learns a parameterized policy by estimating the policy gradient and using it to optimize the policy directly. The objective to be optimized, i.e., $J(\theta)$ can be defined as the value of a starting state s_0:

$$J(\theta) = v_{\pi_\theta}(s_0) \tag{9}$$

Here, π_θ is the policy characterized by a set of parameters θ. According to the policy gradient theorem, the gradient of this objective can be computed as follows:

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(a|s,\theta) \tag{10}$$

In this equation, the on-policy distribution $\mu(s)$ represents the fraction of time spent in state s. In deep reinforcement learning, the policy π and the value function $Q^\pi(s,a)$ are usually represented by neural networks. In these so-called actor-critic architectures, the policy network π is called the actor, and the value network $Q^\pi(s,a)$ is called the critic [75].

Based on the policy gradient theorem, many policy gradient algorithms have been developed. These algorithms often use stochastic policies and an actor-critic architecture. One such algorithm is Trust Region Policy Optimization (TRPO). It uses a KL-divergence constraint on the optimization problem to guarantee the monotonic improvement of its policy, but it is computationally expensive [63]. Actor-Critic using Kronecker-Factored Trust Region (ACKTR) [89] and Proximal Policy Optimization (PPO) [62] have been developed as more computationally efficient alternatives to TRPO.

Asynchronous Advantage Actor-Critic (A3C) is another policy gradient algorithm, which uses multiple parallel actors to speed up training [51]. All of the policy gradient algorithms mentioned above are model-free and on-policy and suffer from high sample complexity, i.e., a large amount of training samples is necessary for training. The Actor-Critic with Experience Replay (ACER) algorithm is an extension of A3C that can be trained off-policy and on discrete as well as continuous action spaces [83].

An example of an off-policy gradient method with a deterministic policy is the Deep Deterministic Policy Gradient (DDPG) algorithm [43]. It uses the deterministic policy gradient theorem to arrive at a similar update rule as stochastic algorithms but for deterministic policies. Finally, Soft Actor-Critic (SAC) is a policy gradient algorithm that combines off-policy updates with stochastic policy optimization and mitigates the problem of high sample complexity by introducing a maximum entropy term into the policy gradient objective [28].

In recent years, reinforcement learning has been applied to more and more complex problems, leading to increased computational demands, which can be met with GPUs or FPGA-based hardware accelerators. The following section gives a brief overview of GPU-accelerated reinforcement learning before the subsequent section presents FPGA-based hardware architectures for reinforcement learning.

III. GPU-ACCELERATED REINFORCEMENT LEARNING

GPUs are the hardware architecture most commonly used to accelerate machine learning algorithms and can be applied to reinforcement learning as well. GPU-based acceleration is usually applied to deep reinforcement learning rather than classical tabular reinforcement learning since the increased computational demands due to the introduction of neural networks into reinforcement learning make DRL more suitable for GPU-based acceleration. Table 1 summarizes the speed-ups achieved by GPU-based DRL implementations. However, the results of the individual publications are not necessarily comparable with each other since they implement different algorithms and attempt to solve different problems utilizing various hardware platforms. Furthermore, there is no common baseline for the speed-ups given by these publications. Therefore, the table serves as an overview rather than as a basis for comparisons. Most of the publications supply multiple performance comparisons. In that case, only the most relevant comparison was included in the table. The table also includes a column with selected GPU specifications for the platform used by the publication. If multiple identical GPUs were used, the specifications are given for a single GPU.

Multiple papers present general frameworks for the parallelization of DRL algorithms. Clemente *et al.* [14] proposed an algorithm agnostic framework for efficient parallelization of DRL, which can be efficiently implemented on a GPU. The training uses multiple instances of the environment to employ multiple actors synchronously on a single machine. Inference and training can be batched, which can be efficiently parallelized and leads to significant speed improvements.

TABLE 1. GPU-based implementations of deep reinforcement learning.

Reference	Algorithm(s)	Implementation Platform	Transistors / Processing Power / TDP [Billions] / [GFLOPS] / [W]	Reference Platform for Comparison	Speed-Up
[14]	PAAC	Nvidia GTX 980 Ti	8 / 6060.03 / 250	Intel i7-4790K	2×-3×
[73]	DQN, PPO, A3C	8 Nvidia P100 GPUs	15.3 / 10608.64 / 250	Nvidia Tesla P100	6×
[59]	Fitted Q-Iteration	Nvidia Tesla C2075	3 / 1030.4 / 225	Intel Xeon E5-2665	105×
[2]	A3C	Nvidia Titan X (Maxwell)	8 / 6691 / 250	Intel Xeon E5-2640v3	45×
[15]	UNREAL	Nvidia Titan X (Maxwell)	8 / 6691 / 250	Nvidia Titan X	9.5×
[18]	IMPALA/V-trace	8 Nvidia P100 GPUs	15.3 / 10608.64 / 250	Nvidia Tesla P100	7×
[17]	SEED	64 Google TPUs	– / 4000 / 450	Nvidia Tesla P100	80×
[57]	DQN	Nvidia Titan X (Pascal)	12 / 10974.21 / 250	Nvidia Titan X	2.14×

The system was tested with a modified A3C algorithm, called Parallel Advantage Actor-Critic (PAAC), and reduced the training time for the Atari domain significantly.

Stooke *et al.* [73] investigated the optimization of DRL algorithms for combinations of CPUs and GPUs. They developed a set of parallelization techniques for DRL, including synchronized sampling and synchronous, as well as asynchronous, Multi-GPU optimization. These techniques were tested with multiple DRL algorithms. They found that on a DGX-1 workstation with 8 GPUs, their techniques lead to a 6× speed-up compared to an implementation using a single GPU.

Other publications focus on efficient GPU implementations of a single DRL algorithm. Postma *et al.* [59] implemented Fitted Q-Iteration for GPUs. Nair *et al.* [54] proposed a massively distributed architecture for DRL called Gorila. It uses parallel actors and learners, a distributed neural network, and a distributed store of experience to implement the DQN algorithm. The GA3C architecture, introduced by Babaeizadeh *et al.* [2], is a GPU implementation of the A3C algorithm. Similarly, the GUNREAL architecture, introduced by Coppens *et al.* [15], accelerates UNREAL (UNsupervised REinforcement and Auxiliary Learning), an algorithm based on A3C, using techniques comparable to GA3C. Qt-Opt is a scalable DRL framework that was implemented on 10 Nvidia P100 GPUs [35]. The distributed DRL agent IMPALA (Importance Weighted Actor-Learner Architecture), proposed by Espeholt *et al.* [18], includes an off-policy actor-critic algorithm called V-trace and can scale to thousands of machines. In a subsequent publication, the SEED agent [17] was proposed, which is closely related to IMPALA. Other than an implementation based on the V-trace algorithm, an implementation based on Recurrent Replay Distributed DQN (R2D2) [36] was provided as well. Instead of a GPU, a Tensor Processing Unit (TPU) [23] was used for the implementation, but any other accelerator could be used in its place.

Furthermore, there are publications targeting the problems of experience replay in a setting with CPUs and GPUs. In [57], the possibility of storing the replay buffer completely in GPU memory instead of external RAM was explored. This approach speeds up the training process but is limited to domains where the replay buffer can fit inside the GPU memory. The Ape-X architecture uses multiple actors that contribute to the same share replay memory so that a single learner executed on a GPU can learn from them [31].

Besides speeding up the training itself, there are also efforts to speed up the simulation time of environments commonly used in RL. Liang *et al.* [41] use NVIDIA FleX [56], a GPU-based physics engine, to speed up the simulation of robotics locomotion tasks. Moreover, a CUDA port of the Atari Learning Environment [6] was implemented by Dalton *et al.* [16]

GPUs are very suitable to accelerate the training of deep neural networks. Thus, their application to deep reinforcement learning can lead to significant speed-ups as well. However, DRL algorithms tend to launch many GPU kernels with little computation, leading to increased kernel launch overhead for GPUs. As a promising alternative, FPGAs can be used to accelerate DRL algorithms, especially when focusing on energy efficiency. The price of high-end FPGA platforms, like a Xilinx Alveo U200, is similar to the price of high-end GPUs, like an Nvidia P100. Additionally, cloud-based FPGA solutions further increase the accessibility of FPGA hardware. Furthermore, high-level synthesis and OpenCL-based design flows mitigate the drawback of the increased development time when using traditional hardware description languages. Hence, the following sections are dedicated to the exploration of FPGA-based reinforcement learning accelerators.

IV. FPGA IMPLEMENTATIONS OF REINFORCEMENT LEARNING ALGORITHMS

FPGAs are integrated circuits that can be reconfigured after manufacturing. Their flexible, distributed on-chip memory resources – like distributed RAM built from the FPGA's Look-Up Tables (LUTs) and dedicated Block RAM – allow the design of domain-specific architectures that closely match the parallelism of the problem domain to achieve high computation speed and energy efficiency. A survey of FPGA architectures can be found, e.g., in [38]. Additionally, FPGAs allow flexible system integration, making it easy to connect to various external devices and communication protocols, which is an important requirement, especially for embedded systems. Other key capabilities of FPGAs are dynamic

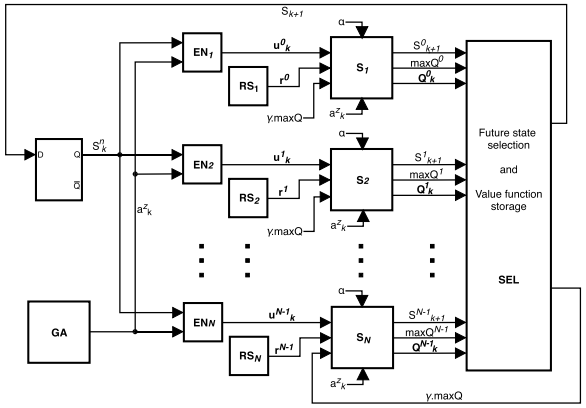


FIGURE 2. Q-Learning accelerator architecture implemented by Da Silva *et al.* [67].

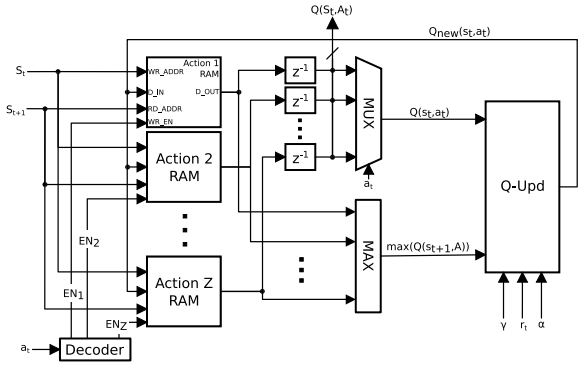


FIGURE 3. Q-Learning accelerator architecture with one on-chip memory for each action of the action space, implemented by Spanò *et al.* [72].

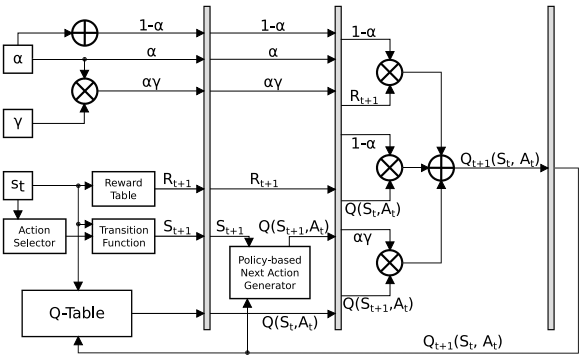


FIGURE 4. QTAccel architecture with four pipeline stages to increase the throughput of the accelerator, implemented by Meng *et al.* [48].

reconfiguration, i.e., reconfiguration while the FPGA is active, and partial reconfiguration. A survey of dynamic and partial reconfiguration of FPGAs can be found in, e.g., [81].

In the domain of deep learning, FPGAs show promising results for the acceleration of neural network inference, as summarized in [26]. This section examines FPGA-based implementations of reinforcement learning algorithms, starting with architectures for tabular reinforcement learning, followed by implementations of state-of-the-art deep reinforcement learning methods. Table 2 shows the speed-ups achieved by these implementations. As for the GPUs, this should be seen as an overview, but not as a comparison of the different implementations because they implement different algorithms, use different FPGAs for their implementations, and different hardware platforms as the reference for their comparisons. However, most FPGA implementations achieved a significant speed-up of an order of magnitude compared to CPU implementations and outperformed GPU-based reference implementations as well. When multiple comparisons are given in a publication, the most relevant one was included in the table. Additionally, the table includes the LUTs used by each implementation as a reference for the size of the implemented architecture.

A. TABULAR REINFORCEMENT LEARNING

The first implementation of reinforcement learning on FPGA was done by Prabha *et al.* [60]. A SARSA-based architecture was used to choose between different dynamic power management policies. Technically, this architecture is an implementation of a Multi-Armed Bandit, not of generic SARSA, because it is limited to a boolean state space and 4 actions.

A first complete implementation of an accelerator for a reinforcement learning algorithm was proposed by Da Silva *et al.* [67] The proposed architecture, shown in Fig. 2, is composed of five main module types. The GA (Generate Action) module selects random actions, the EN (Enable) modules decide which state-action pair should be updated, the RS (Reward Storage) modules store the reward function, the S (State) modules compute the Q-value, and the SEL (Selection of Future State) module selects the future state. It should be noted that the architecture contains an EN, RS, and S module for each state of the state space, including the Q-value computation. The Q-table is stored in registers in the S module. The state transitions and rewards of the environment are integrated into the learning system in the form of the RS and SEL modules. For a simple example problem with 6 possible states and 4 actions, the system achieves a throughput of 26.42 Million Samples per second (MSps) using a Xilinx Virtex-6 FPGA.

Spanò *et al.* [72] published an improved version of a comparable accelerator, shown in Fig. 3. Instead of integrating the environment into the learning system, states and rewards are seen as input to the hardware accelerator. The architecture contains one on-chip memory block per action in the action space, which stores the Q-values of the respective action for each state. For a given state, the Q-values for all actions can be read at the same time. A max-tree is used to find the maximum Q-value, as described by Liu *et al.* [93] in a theoretical paper about hardware accelerators for Q-Learning. Furthermore, the multiplications with the learning rate α and decay γ were found to be the limiting factor for computation speed and have been replaced with one or two barrel shifters. The possible values for γ and α were thus limited to powers of two or sums

TABLE 2. FPGA-based implementations of deep reinforcement learning.

Reference	Type	Algorithm	Platform	Reference Platform	LUTs	Speed-Up
[67]	tabular RL	Q-Learning	Xilinx Virtex-6 XC6VLX240T	Parsytec Powermouse	1086	708×
[72]	tabular RL	Q-Learning	Xilinx Virtex-6 XC6VLX240T	Xilinx Virtex-6 XC6VLX240T	148	2.8×
[48]	tabular RL	Q-Learning	Xilinx Virtex-7	Xilinx Virtex-6 XC6VLX240T	173	6.9×
[21]	deep RL	NNQL	Xilinx Virtex-7	6th Generation Intel i5 CPU	12253	43×
[74]	deep RL	DQN	Altera/Intel Arria 10	Intel i7-930	–	67×
[84]	deep RL	DQN	Xilinx Zynq XC7Z020	ARM Cortex A9 of Zynq SoC	1873	16.49×
[65, 66]	deep RL	TRPO	Intel Stratix-V 5SGSD8	Intel i7-5930K	140238	20×
[25]	deep RL	DDPG	Intel Stratix-V 5SGSD8	Intel i7-6700	143569	4.53×
[13]	deep RL	A3C	Xilinx UltrasScale+ VU9P	Nvidia Tesla P100	677300	1.3×
[47]	deep RL	PPO	Xilinx Alveo U200	Nvidia Titan Xp	501000	27.5×

TABLE 3. Comparison of tabular reinforcement learning implementations.

Reference	States	Reward Width	Resources LUTs / FFs / DSPs	TP[a] [MSps]	TP[a] per Power [MSps/mW]	TP[a] per LUT [MSps/LUT]
Da Silva et al. [67]	6	10	1086 / 367 / 34	24.42	8.14	0.022
Spanò et al. [72]	8	16	148 / 186 / 3	72	5.14	0.486
QTAccel [48]	64	16	172 / 346 / 4	189	2.36	1.094

[a] Throughput

of powers of two, respectively. The environment was implemented in hardware using the Xilinx System Generator [90]. An adaptation of the accelerator for the SARSA algorithm is shown as well. For an example environment with 8 states and 4 actions, the architecture achieves a throughput of 72 MSps using a Xilinx Virtex-6 FPGA.

QTAccel is a pipelined architecture for Q-Learning implemented by Meng *et al.* [48], shown in Fig. 4. It is a four-stage pipeline with the state transition and reward functions of the environment integrated into the first pipeline stage. In the first stage, Q-values and rewards are read from block RAMs, the action is selected, and the next states are computed based on the state transition function of the environment. In addition to the Q-table, QTAccel also uses a Q_{max}-table that stores the maximum Q-value for each state. While the additional Q_{max}-table increases the resource requirements of the design, its use enables QTAccel to avoid the more expensive computation of the maximum Q-value using, e.g., a max-tree, as implemented in the architecture by Spanò *et al.* In the second stage, the next action is chosen based on the update-policy (usually ϵ-greedy) and corresponding Q-value read from memory. The third stage calculates the updated Q-value, and the last stage stores this value in the Q-table. The architecture can be implemented for both Q-Learning and SARSA. The implementation was tested on larger state spaces than previous implementations and achieved a consistent throughput of around 180 MSps using a Virtex-7 FPGA.

A comparison of the throughput of the three implementations can be seen in Fig. 5(a). The implementation by Spanò *et al.* achieves higher throughput than the implementation by Da Silva *et al.* by using barrel shifters instead of multiplications, as well as other differences in the overall architecture. Due to the use of pipelining, QTAccel's throughput is even higher. Fig. 5(b) displays the utilization of DSP (Digital Signal Processing) blocks, i.e., FPGA resources used primarily for efficient multiplication, for each of the three Q-Learning implementations. The number of DSP blocks used by the implementation by Spanò *et al.* increases with the number of states because the Q-value is computed for all states regardless of the current state. The other implementations only compute the Q-value for the current state.

TABLE 4. Implementations of neural network Q-learning.

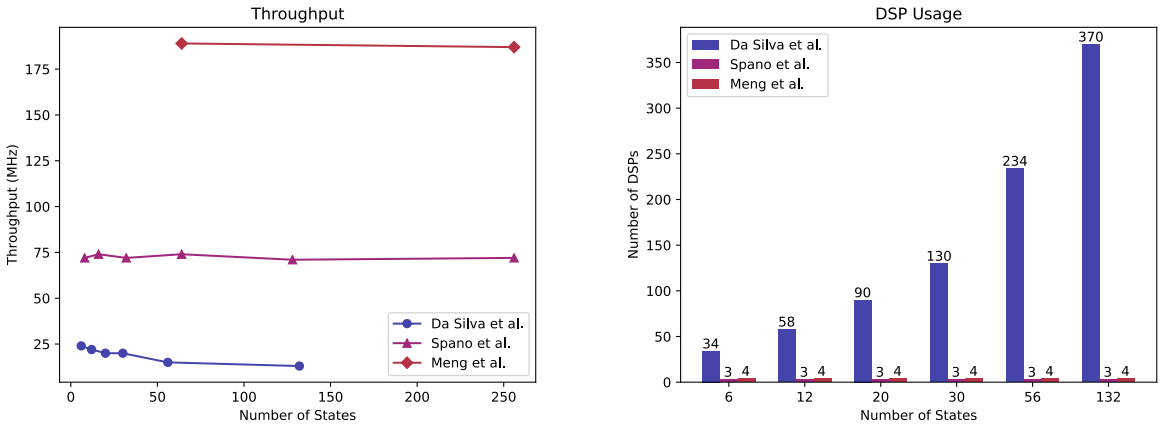
Reference	Inference	Training	Experience Replay
[39]	yes	no	no
[21]	yes	yes	no
[74]	yes	yes	yes (batch size = 1)
[84]	yes	yes	only update sometimes

As a further comparison, Table 3 shows the resource usage and throughput of the three implementations for similar state spaces and reward widths. For better comparisons, it also includes the throughput per power and throughput per LUT. Interestingly, QTAccel achieves the highest throughput per LUT, even though it implements the largest state space.

B. DEEP REINFORCEMENT LEARNING

1) VALUE-BASED DRL IMPLEMENTATIONS

Multiple publications implement Neural Network Q-Learning (NNQL) architectures where a neural network replaces the Q-table. However, they usually do not implement features like experience replay and target networks necessary



(a) Throughput (b) DSP Usage

FIGURE 5. Comparison of Q-Learning hardware implementations. Fig. 5(a) shows the throughput of the architectures for state spaces of increasing size. All implementations displayed in this figure were configured with 4 possible actions and 16 bit Q-values. The throughput differs dramatically for the three implementations, with QTAccel achieving the highest and Spanò *et al.* the second highest throughput. Figures based on data from [48], [67] and [72]. Fig. 5(b) shows the number of DSPs used by each of the architectures for state spaces of increasing size. It can be seen that the number of DSPs used by the implementation of Da Silva *et al.* increases with the number of states, while the DSP usage of the other implementations stays constant and low.

for a full implementation of DQN. In [39], the inference of the neural network of a reinforcement learning system was implemented on an FPGA. Due to the lack of data, this publication was not included in Table 2. In [21] and [22], a full MLP-based Q-Learning architecture was designed. The forward and backward passes of the neural network were both implemented on an FPGA. The neural networks used in this architecture are very small, with as little as one hidden layer with four neurons. A significant speed-up was achieved using this architecture when compared to an Intel i5 CPU.

Another implementation was done by Su *et al.* [74]. The proposed architecture includes backpropagation on the FPGA and an implementation of experience replay with a fixed batch size of one. The MLP training uses specially designed processing elements with three different modes for different parts of the forward and backward passes of the MLP. The replay buffer is stored in external memory, and the neural network weights are stored in on-chip BRAM. The evaluation shows that the implementation on an Arria 10 FPGA can handle up to 580 neurons, limited by available BRAM size, and achieves a significant speed-up compared to a GPU implementation on a GTX 760. The authors conclude that FPGA implementations of deep reinforcement learning algorithms can be advantageous compared to GPU-based implementations, especially for small neural networks.

One of the most recent implementations of a simple neural-network-based reinforcement learning architecture uses an extreme learning machine (ELM) [33] or online sequential ELM (OS-ELM) [32] to replace the neural network in a DQN [84]. The goal was to design a lightweight on-device reinforcement learning system for resource-limited FPGAs. By using ELM, the implementation does not need to rely on backpropagation, computing the neural network weights analytically instead. Multiple other changes were made to the DQN algorithm to stabilize the training, such as Q-Value clipping, spectral normalization, and L2 regularization. Instead of implementing experience replay, the proposed algorithm randomly determines whether or not to update at each time step to break the temporal correlation of training inputs. The system was implemented on a PYNQ-Z1 development board, utilizing a Xilinx Zynq XC7Z020 FPGA. Open AI Gym [10] was integrated as the reinforcement learning environment. A significant speed-up was achieved when compared to an implementation on the ARM core of the development board.

Table 4 summarizes the features implemented in the different NNQL or DQN implementations. Only three out of four designs implement the training of their reinforcement learning model in hardware. Furthermore, no design implements experience replay as described in the DQN papers [52], [53]. Instead, they either implement experience replay with a constant batch size of one, or they opt to implement simplified methods to break the temporal correlation of their training samples.

2) IMPLEMENTATIONS OF POLICY GRADIENT ALGORITHMS

Multiple hardware accelerators for other state-of-the-art deep reinforcement learning algorithms have been implemented. The first hardware implementation of a policy gradient algorithm, namely TRPO, was proposed by Shao *et al.* [65]. The most computationally intensive part of the TRPO algorithm is the computation of the Fisher Vector Product as part of the conjugate gradient. The proposed architecture implements this in hardware by employing a customized version of Pearlmutter Propagation [58], reducing the problem to dense matrix-vector multiplications, which can be implemented efficiently using blocked matrix-vector multiplications [24]. The overall architecture consists of a conjugate gradient solver written in C with the Fisher Vector Product

TABLE 5. Comparison of deep reinforcement learning implementations.

Reference	Algorithm	Resources LUTs / FFs / DSPs	BRAM [Kb]	$1/(T_{grad} \cdot \text{LUT})$ [1/(s·kLUT)]	IPS/LUT	(IPS/LUT) · P[a]
[65, 66]	TRPO	140238 / 269357 / 1368	37760	0.0371	n.a.	n.a.
[25]	DDPG	143569 / 304025 / 1913	38260	0.1416	n.a.	n.a.
[13]	A3C	677300 / 875700 / 2348	45612	n.a.	0.0038	2.575
[47]	PPO	501000 / 708000 / 3744	25056	n.a.	0.0175	0.999

[a] Number of neural network parameters in millions

TABLE 6. Comparison of CPU- and FPGA-based DRL implementations.

Reference	Category	Algorithm	Hardware Platform	Reference CPU	Speed-up	Throughput [IPS]
[14]	GPU	PAAC	Nvidia GTX 980 Ti (28nm)	i7-4790K (22nm)	2-3x	–
[2]	GPU	A3C	Nvidia Titan X (28nm)	Intel Xeon E5-2640-v3 (22nm)	45x	1361
[65, 66]	FPGA	TRPO	Intel Stratix-V 5SGSD8 (28nm)	i7-5930K (22nm)	20x	–
[25]	FPGA	DDPG	Intel Stratix-V 5SGSD8 (28nm)	i7-6700 (14nm)	4.52x	–
[13]	FPGA	A3C	Xilinx UltrasScale+ VU9P (16nm)	2× Xeon E5-2630 (14nm)	5x	2550
[47]	FPGA	PPO	Xilinx Alveo U200 (16nm)	Intel Xeon 5120 (14nm)	30.5x	32000

computed on an FPGA. The system was evaluated on an Intel Stratix-V FPGA using two MuJoCo benchmarks [78] from OpenAI Gym [10]. In a subsequent publication [66], the same authors explored the design space with respect to the loop unrolling factors of neural network computations, leading to a 4.65 times speed-up compared to a Tesla C2070 GPU. Furthermore, it applied the system to robotic control. Using the TRPO algorithm, a reinforcement learning agent was trained in simulation, accelerated by an FPGA, and then tested on a real robot arm, running on a CPU.

A similar approach to robotic control was taken by Guo *et al.* [25]. The proposed architecture is an accelerator for the DDPG algorithm. A CPU streams network parameters and state transitions to the FPGA, which computes the gradients and sends them back to the CPU to update the networks. As part of the proposed architecture, the method from [65] was adapted to matrix-matrix multiplications. The system achieved substantial acceleration compared to a CPU implementation, despite communication overhead.

The A3C algorithm was implemented by Cho *et al.* [13] in an architecture called FA3C. The architecture consists of a host CPU that writes data into a DRAM via PCIe DMA, which can then be accessed by the FPGA-based accelerator. A memory hierarchy was designed to efficiently supply the Compute Units in the design with data. The data stored in the off-chip DRAM, like training images and neural network parameters, is buffered in on-chip memory. Additionally, line buffers consisting of registers are employed, which prefetch elements from different locations of the on-chip buffer. The compute units, shown in Fig. 6, perform inference or training across all layers of the neural networks. Multiply-accumulates are employed as their basic Processing Elements (PEs), which are arranged in a one-dimensional array, and an RMSProp module is utilized to apply the computed gradients to the global parameters. Two compute units are used to balance off-chip data bandwidth. The system was evaluated on Atari games using the Arcade Learning Environment [6]. The evaluation has shown that the proposed architecture, using a Xilinx VCU1525 FPGA, surpasses state-of-the-art GPU-based implementations like GA3C [2], executed on an Nvidia Tesla P100 GPU, with respect to performance and energy efficiency. The authors argue that an FPGA-based implementation has several advantages over GPU-based implementations due to the small batch sizes commonly used for the algorithm and the kernel launch overhead of GPUs.

Another heterogeneous architecture was implemented by Meng *et al.* [47] for the PPO algorithm. It is composed of a host CPU doing the loss and advantage computations and an FPGA, doing the forward propagation, backward propagation, and weight update. They communicate via PCIe. Each of the two neural networks of the PPO algorithm has its own compute unit in the FPGA architecture. The compute unit is used for inference and training. Different training times between the networks are accounted for by a load balancing module that enables the compute units to be used for both networks as necessary. The compute units shown in Fig. 7 use 2D systolic arrays of PEs for matrix multiplications. Additionally, the compute units contain a weight update module and various buffers. The MLP weights are stored in on-chip memory. A special memory layout was designed to enable the architecture to read the weight matrices and their transposes quickly, which is necessary for the forward and backward pass of the gradient computation. The weight matrices are divided into blocks, which are saved in different BRAMs. The same set of BRAMs can store multiple weight matrices. The architecture was compared to a state-of-the-art GPU implementation using MuJoCo benchmarks. The system was implemented on an Intel Xeon 5120 CPU with a Xilinx Alveo

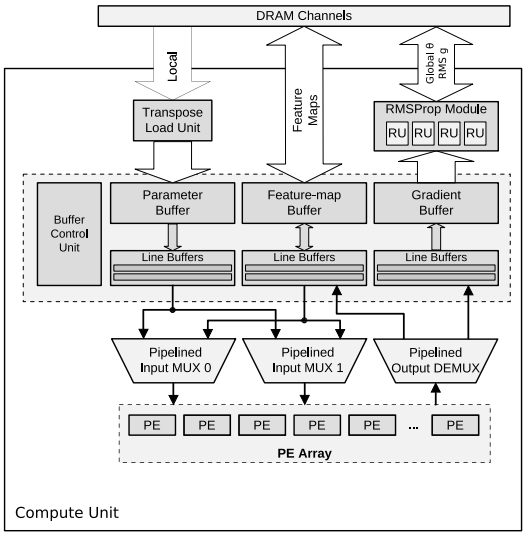


FIGURE 6. The compute unit of FA3C. Processing elements are arranged in an array. A buffer control unit handles buffers for network parameters, feature-maps and gradients [13].

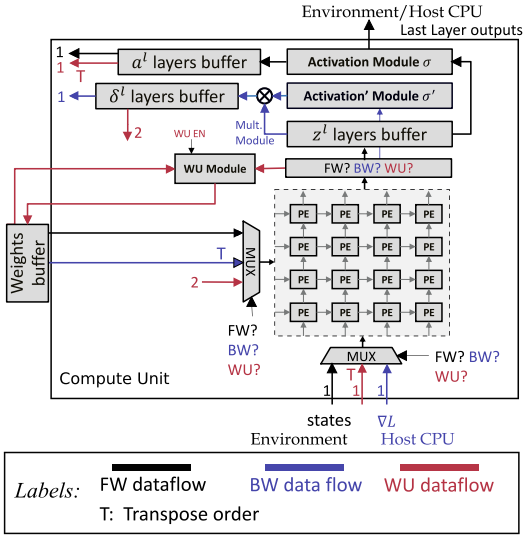


FIGURE 7. The compute unit of the PPO architecture. Processing elements are arranged in a matrix. Additionally, the compute unit contains buffers for the output and updates of neural network layers [47].

U200 FPGA accelerator card. It was compared to a Titan Xp GPU hosted by the same CPU. The proposed architecture had an up to 27.5 times higher throughput than the GPU baseline.

3) COMPARISON OF POLICY GRADIENT IMPLEMENTATIONS

To compare these implementations of policy gradient DRL algorithms, Table 5 summarizes the resource usage and efficiency reported in their publications. One major difference between these implementations is that the TRPO and DDPG designs only implement the gradient computations in hardware, while the A3C and PPO architectures implement the full DRL training on the FPGA. As expected, the resource usage of the A3C and PPO implementations is much higher.

As a measure of efficiency for the accelerators focused on gradient computation, the table includes the column $1/(T_{grad} \cdot \text{LUT})$, where T_{grad} is the time needed for one gradient computation. The DDPG accelerator achieves slightly higher efficiency than the TRPO architecture according to this measure. For the accelerators implementing full DRL training, the column IPS/LUT provides a point of comparison. IPS (Inferences per Second) is a speed metric commonly used in DRL, which is computed by dividing the number of samples collected during the inference phase by the time taken in the inference (including environment interactions) and training phase. Here, the PPO design appears to achieve slightly better efficiency than FA3C. However, FA3C uses a much larger neural network with 684K parameters, while the network used by the PPO design only has 57K parameters. The last column of Table 5 shows the IPS/LUT scaled by the number of network parameters. It shows that FA3C is more efficient than the PPO design if the network size is taken into account.

In addition to comparing FPGA architectures with each other, they can also be compared to GPU-based reinforcement learning. Of the existing publications, only Cho *et al.* [13] and Meng *et al.* [47] compare their results to GPU-based implementations. Cho *et al.* [13] compare their FA3C architecture to a cuDNN-based A3C implementation, as well as a TensorFlow implementation of GA3C [2]. A Xilinx VCU1525 FPGA is used for FA3C, and an Nvidia Tesla P100 GPU is used for the GPU-based approaches. The cuDNN-based A3C implementation is the fastest GPU-based implementation in their evaluation, only outperformed by FA3C itself. FA3C achieved 27.9% higher IPS than the fastest GPU-based implementation. Meng *et al.* [47] compare their FPGA implementation of PPO using a Xilinx Alveo U200 to the OpenAI baseline implementation of PPO on an Nvidia Titan Xp GPU. In all of their evaluations, the FPGA implementation achieves the highest throughput. Depending on the hyperparameters of the algorithm, the speed-up ranges from 2× up to 27.5×. The speed-up compared to the GPU increases as the minibatch size decreases and the number of parallel agents increases, as would be expected due to the kernel launch overhead of the GPU.

As an additional comparison, Table 6 shows the speed-up achieved by the GPU and FPGA implementations of policy gradient algorithms which compared their results to CPU implementations, as well as the throughput achieved by these architectures. Furthermore, the manufacturing process size of each of the hardware platforms used is given in parenthesis to enable fair comparisons. It can be seen that FPGA-based implementations achieve similar performance gains as achieved by GPU-based implementations.

C. NEURAL NETWORKS IN FPGA-BASED DRL IMPLEMENTATIONS

The implementation of efficient accelerators for the training of deep neural networks is an important prerequisite

for the implementation of DRL architectures since the neural network update is often a performance bottleneck. This includes the training of different types of layers, such as CNN (Convolutional Neural Network) or RNN (Recurrent Neural Network) layers. An overview of CNN inference accelerators on FPGA can be found, for example, in [1], while a survey of accelerators for recurrent neural networks, including LSTMs, can be found in [50]. In addition to publications that focus on the acceleration of DNN inference, some publications tackle the problem of implementing backpropagation for neural network training on FPGAs as well. For example, [92] and [29] implement frameworks for CNN training on FPGAs, and [76] explores the training of LSTM layers on FPGAs. With approaches like these, it would be possible to implement FPGA-based DRL architectures with models including CNN and LSTM layers.

Methods like quantization and model pruning are used to enable deep learning at the edge, and some publications have explored their use for deep reinforcement learning [37], [87]. Quantization can happen after training has finished or via quantization-aware training. Krishnan *et al.* [37] applied post-training quantization and quantization-aware training to a few different DRL algorithms, reducing training time for a pong environment by 50% and achieving 18× inference speed-up for a robot navigation policy by using weights quantized to 6-8 bits. A more detailed evaluation of the benefits of quantization was done by Guo [88] and concludes that 8 bit quantization reduces model size, increases throughput up to 16× while maintaining an accuracy within 1% of the floating-point model accuracy. Wu *et al.* [87] propose a pruned reinforcement learning method that reduces the worst-case latency of the learned policy by 32.5% – 68.6% over a policy without pruning.

While post-training quantization and model pruning is possible with all FPGA-based DRL accelerators, this does not differ from quantization or pruning based on models trained with other hardware. Quantization-aware training could be implemented on the FPGA but has not been implemented in practice. For quantization-aware training, both the floating-point and quantized model are needed, which leads to high resource requirements. The resulting quantized model could be employed on the same FPGA for its advantages in computation speed or on a different FPGA with more limited resources.

V. ANALYSIS OF CURRENT PRACTICE

This section explores the techniques used in existing publications and highlights important recurring implementation strategies.

A. CLASSICAL REINFORCEMENT LEARNING

Current publications focus on the implementation of Q-Learning but also suggest modifications to their proposed architectures to support SARSA [48], [67], [72]. The following paragraphs present similarities and differences between these architectures.

1) ENVIRONMENTS

The reinforcement learning environment is usually implemented in reconfigurable logic [67], [72] and is sometimes tightly integrated with the overall architecture. This approach introduces no significant communication overhead, thus allowing high throughput, but limits the flexibility of the architectures, as each new problem needs to be implemented in hardware, and popular collections of reinforcement learning environments cannot be used easily.

2) Q-TABLE

In tabular reinforcement learning, an important design decision is how to represent the Q-table and how to compute the maximum Q-value for the future state. It can be saved continuously in one large on-chip memory block, as implemented in [48]. In this case, a Q_{max}-table, storing the maximum Q-value for each state, is an efficient way to access these maximum values for the future state. Another approach is to the Q-table into one on-chip memory block per action in the action space, as demonstrated by [72]. This enables the system to read the Q-values of all actions simultaneously so that the maximum Q-value can be computed by a max-tree. While this approach may lead to lower throughput due to the max-tree, it also reduces the memory requirements as no additional Q_{max}-table is necessary. Independently of these design choices, the actual implementation of the Q-table can be described in a high-level fashion, leaving decisions such as the choice of memory resource (e.g., Block RAM or Distributed RAM utilizing the LUTs of the FPGA) to the synthesis tool.

3) HARDWARE ARCHITECTURE OPTIMIZATIONS

Multiplications are a significant part of the computations necessary for these classical reinforcement learning algorithms. Two of the existing architectures use DSPs to implement the multiplications [48], [67], while one architecture replaced the multiplications with barrel shifters to improve performance [72]. Another way to optimize the implementations is to introduce pipelining. Of the existing implementations, only [48] is a pipelined architecture, explaining its higher throughput compared to its predecessors and competitors.

B. DEEP REINFORCEMENT LEARNING

Accelerators for DRL target many different DRL algorithms. Some publications present architectures for a simple Q-Learning algorithm with a neural network to replace the Q-table. Others implement accelerators for state-of-the-art algorithms such as DQN, TRPO, DDPG, A3C, and PPO, often with higher resource requirements.

1) ENVIRONMENTS

Since DRL algorithms can solve more complex problems, which are typically implemented in software, their FPGA-based implementations usually rely on existing collections of reinforcement learning environments. This approach

introduces additional communication overhead, but since DRL algorithms are much more compute-intensive, significant speed-ups can still be achieved. The utilization of software environments eliminates the need to implement different environments in hardware and makes fair comparisons between CPU, GPU, and FPGA implementations easier.

2) NEURAL NETWORK TRAINING

All DRL architectures need to train neural networks, which is usually done by implementing backpropagation on the FPGA [13], [21], [47], [74]. Alternatively, some of the proposed architectures replace the neural network with another machine learning model, like ELM, that can be trained on the FPGA without backpropagation [84]. All current FPGA implementations are limited to neural networks without special layers, such as CNN or LSTM layers. In contrast to GPUs, which are commonly used for neural network training, the training process is expensive to implement on FPGAs. The inclusion of additional layer types would make the neural network training even more complex.

3) LOCATION OF NETWORK PARAMETERS

The utilization of neural networks in DRL introduces the architectural decision of where to store the network parameters: either on-chip on the FPGA or off-chip in external memory. The FPGA implementations that implement only gradient computations on the FPGA tend to store the network parameters off-chip since they are also needed for off-chip computations [25], [65]. When the complete DRL training is implemented on the FPGA, both storage locations can be viable when the networks are small enough. The FA3C architecture [13] uses neural networks with 684K parameters, stored in an off-chip DRAM on the FPGA side, and loads them into the FPGA on-demand. The FPGA implementation of PPO [47], on the other hand, uses smaller neural networks with just 57K parameters and stores all network parameters on-chip. Both of these implementations use multiple neural networks since they train multiple agents in parallel, which leads to large memory requirements if the parameters are stored on-chip. Thus, storing parameters on-chip is sometimes not feasible, even though it reduces communication overhead.

4) HARDWARE/SOFTWARE PARTITIONING

When designing heterogeneous systems with CPUs and FPGAs, one of the most important design decisions is which part to accelerate on the FPGA. The architectures proposed by the existing publications have explored different ways to partition the algorithm into hardware and software. For example, two accelerator architectures implement just the computation of the gradient on the FPGA [25], [66], while two other publications propose implementations of full DRL training on the FPGA [13], [47]. Implementing only parts of the DRL algorithm in hardware typically increases communication overhead since the intermediate results computed by the FPGA need to be sent back to the CPU for further processing. On the other hand, full FPGA implementations require larger FPGAs as their resource usage is much higher, as can be seen in Table 5.

5) EXPERIENCE REPLAY

Accelerators of algorithms that rely on experience replay currently avoid the implementation of experience replay in their architecture due to the memory requirements of the technique and the communication overhead introduced by storing the replay buffer in off-chip memory. In [74], a replay buffer was stored in external memory, but training only happened with a batch size of 1. Another implementation avoids experience replay by only updating its networks on some of the observed state transitions to break temporal correlation [84]. However, this significantly reduces sample efficiency.

6) MEMORY MANAGEMENT

The usage of on-chip memory resources, like Block RAM and Distributed RAM, with flexible, parallel memory access is a key feature of FPGA implementations, enabling high levels of parallelism in FPGA-based DSAs. To efficiently train DRL agents, implementations include memory management schemes that enable efficient use of off-chip RAM and different kinds of on-chip memory resources [13], [47].

VI. DIRECTIONS FOR FUTURE RESEARCH

Based on the analysis of current practice and current trends in deep reinforcement learning research, the following research challenges can be identified:

A. PERFORMANCE COMPARISON

In future work, an in-depth comparison between CPU-, GPU- and FPGA-based implementations of reinforcement learning, with respect to their energy consumption and throughput, would be beneficial. While some publications argue that the small batch sizes usually employed in deep reinforcement learning algorithms would favor FPGA implementations [13] due to the kernel launch overhead introduced by GPU implementations, others argue that these algorithms could be used with larger batch sizes as well to mitigate this drawback of GPUs [73].

B. IMPLEMENTATION OF ADDITIONAL ALGORITHMS

Many variations of the original DQN algorithm have been suggested in the RL literature [30]. Which of these would benefit from being implemented on FPGAs should be investigated as well. Some state-of-the-art deep reinforcement learning algorithms have not been implemented in hardware yet, for example, ACKTR [89], ACER [83], SAC [28], and TD3 [20]. While new implementations of all of these algorithms might generate new insights and open up new possibilities for faster and more efficient architectures, especially implementations for recently published DRL algorithms are needed to keep up with current developments in DRL research. New implementations could be inspired by new techniques introduced in recent hardware architectures

for DRL, like using OS-ELM instead of a neural network [84] or employing Pearlmutter propagation [65].

C. ADDITIONAL LAYER TYPES

While current DRL implementations are limited to fully connected layers in their neural networks, it would be useful for many problems to allow CNN and RNN layers. To keep up with current developments in DRL, future accelerators targeting DRL algorithms should aim to support these layer types as well. For added flexibility, future implementations could attempt to implement DRL accelerators with a modular design that can interface to loosely-coupled DNN accelerators on the same FPGA for their neural network updates. This way, the type of neural network could be changed more easily, and the DRL accelerator could be used for a larger variety of application scenarios. Additionally, this would also enable DRL hardware designs to benefit from newly developed DNN accelerators. However, efficient communication between the different modules would be important to ensure effective acceleration.

D. QUANTIZATION-AWARE TRAINING

Quantized neural network models are popular for resource-limited edge applications. Existing DRL accelerators use 32 bit floating-point numbers for training on the FPGA, producing models that are not quantized. The implementation of quantization-aware training of DNN models on FPGAs could be explored so that the resulting model can be employed on different hardware platforms with limited resources. Heterogeneous systems combining FPGAs with GPUs and CPUs might be beneficial for this use case, so the design can leverage the fast floating-point computation capabilities of a GPU while using the FPGA to accelerate other parts of the algorithm.

E. EXPERIENCE REPLAY

Most implementations of neural network Q-Learning do not implement experience replay as suggested by the original DQN algorithm [52]. However, experience replay is important to ensure successful training by breaking the temporal correlation of training data. Alternatives to full experience replay suggested by existing implementations [74], [84] are less sample efficient. The implementation of experience replay on FPGAs using on-chip memory to store the replay memory or as a buffer for a replay memory stored in external RAM could be explored.

F. HARDWARE/SOFTWARE INTEGRATION

The integration of reinforcement learning modules in a hardware/software environment needs further research to determine how an RL hardware module can most efficiently be connected to a software-based RL environment like, for example, OpenAI GYM [10], ALE [6], ELF [77], or DMLab [5]. Connecting to these commonly used environments would be highly useful since it enables fair comparisons to CPU and GPU implementations. However, efficient integration is a problem, especially for accelerators of classical reinforcement learning, since their low computational complexity does not allow much communication overhead.

An alternative or supplementary approach would be to implement a flexible set of RL environments in hardware to be able to benchmark new architectures independently of communication to CPUs. This enables significant reduction of the communication overhead, which is especially important for tabular reinforcement learning. However, this will likely not be feasible for more complex RL environments, such as simulations of robotic manipulators. High-level synthesis-based implementations can be a possible approach to bridge the gap between simple HDL-based designs and pure software solutions.

G. TOOLFLOW AND DESIGN PRODUCTIVITY

FPGA implementations can be built with different toolflows. RTL designs can be created with traditional hardware description languages like Verilog or VHDL. While this approach usually leads to the highest resource efficiency, it is very development-intensive and cannot be easily adapted to new requirements. However, other approaches like hardware design with high-level synthesis based on programming languages like C++, or hardware development with OpenCL promise faster development times and increased flexibility of the resulting designs. For example, high-level synthesis libraries can be developed in C++ to encourage code reuse and to allow easy reproduction of architectures by different developers. This approach is already used for neural network inference [79] and can be extended to reinforcement learning in the future.

H. NEAR- AND IN-MEMORY COMPUTING

Recently, near- and in-memory architectures are becoming increasingly popular for the acceleration of deep learning applications. In Near-Memory Computing (NMC) architectures, additional compute units are placed close to memory to reduce memory latencies and to increase effective memory bandwidth. A survey of near-memory computing can be found, e.g., in [69], [70]. Its viability in deep learning has been shown, for example, by Brown *et al.* [11], with a high-performance near-memory accelerator for CNNs.

One step further than that, In-Memory Computing (IMC) moves processing units directly into the memory itself. Shafiee *et al.* [64] implemented a neural network inference accelerator based on memristor crossbars to store weights and compute analog dot-products. Song *et al.* [71] propose a ReRAM-based accelerator for both training and inference of neural networks. While IMC reduces off-chip memory accesses, it also increases the volume of on-chip communication and communication latency. Mandal *et al.* [45] introduce a custom network-on-chip and scheduling method, which reduces the communication latency by 20%-80%. More detailed surveys of the application of IMC to deep learning can be found in [4] and [49]. The successful application of NMC and IMC in deep learning suggests that

it will also be useful for deep reinforcement learning in the future.

Overall, there are many exciting directions in which future research could develop. While the utilization of FPGAs is a promising endeavor, CPUs and GPUs have their advantages as well. Therefore, heterogeneous systems consisting of CPUs, GPUs, and FPGAs could be explored as well, to benefit from each of their advantages.

VII. CONCLUSION

Reinforcement learning has shown considerable potential in solving sequential decision-making problems, with applications in a wide range of domains. However, RL training is often time-consuming, with training times ranging from multiple hours to weeks. Domain-specific architectures can play an important role in the future of reinforcement learning by speeding up the training process and decreasing experiment turn-around time.

Some accelerators for classical and deep reinforcement learning already exist and have shown the capability to improve training time significantly. However, many opportunities for progress remain. Not all RL algorithms have been implemented in hardware, and new algorithms are developed frequently. New hardware implementations of these algorithms could open up new perspectives and possibilities. Commonly used techniques, like experience replay and multi-actor learning, need more research to be implemented efficiently. Finally, heterogeneous architectures, enabling efficient interplay of CPUs, GPUs, and FPGAs in the domain of Reinforcement Learning, should be further investigated.

ACKNOWLEDGMENT

The authors acknowledge support by Deutsche Forschungsgemeinschaft (DFG) and Open Access Publishing Fund of Osnabrück University.

REFERENCES

[1] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating CNN inference on FPGAs: A survey," 2018, *arXiv:1806.01683*.

[2] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, "Reinforcement learning through asynchronous advantage actor-critic on a GPU," in *Proc. ICLR*, 2017, pp. 1–12.

[3] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent tool use from multi-agent autocurricula," 2019, *arXiv:1909.07528*.

[4] S. Bavikadi, P. R. Sutradhar, K. N. Khasawneh, A. Ganguly, and S. M. P. Dinakarrao, "A review of in-memory computing architectures for machine learning applications," in *Proc. Great Lakes Symp. VLSI*, Sep. 2020, pp. 89–94.

[5] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, J. Schrittwieser, K. Anderson, S. York, M. Cant, A. Cain, A. Bolton, S. Gaffney, H. King, D. Hassabis, S. Legg, and S. Petersen, "DeepMind lab," 2016, *arXiv:1612.03801*.

[6] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *J. Artif. Intell. Res.*, vol. 47, pp. 253–279, Jun. 2013, doi: 10.1613/jair.3912.

[7] R. Bellman, "A Markovian decision process," *J. Math. Mech.*, vol. 6, no. 5, pp. 679–684, Apr. 1957. [Online]. Available: http://www.jstor.org/stable/24900506

[8] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. W. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," 2019, *arXiv:1912.06680*.

[9] A. Bernstein, E. Burnaev, and O. Kachan, "Reinforcement learning for computer vision and robot navigation," in *Proc. 14th Int. Conf., Mach. Learn. Data Mining Pattern Recognit.* New York, NY, USA: Springer, 2018, pp. 258–272, doi: 10.1007/978-3-319-96133-0_20.

[10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," 2016, *arXiv:1606.01540*.

[11] G. Brown, V. Tenace, and P.-E. Gaillardon, "NEMO-CNN: An efficient near-memory accelerator for convolutional neural networks," in *Proc. IEEE 32nd Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2021, pp. 57–60, doi: 10.1109/ASAP52443.2021.00016.

[12] Z. Chen and D. Marculescu, "Distributed reinforcement learning for power limited many-core system performance optimization," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2015, pp. 1521–1526.

[13] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "FA3C: FPGA-accelerated deep reinforcement learning," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: ACM, 2019, pp. 499–513, doi: 10.1145/3297858.3304058.

[14] A. V. Clemente, H. N. Castejón, and A. Chandra, "Efficient parallel methods for deep reinforcement learning," 2017, *arXiv:1705.04862*.

[15] Y. Coppens, K. Shirahata, T. Fukagai, Y. Tomita, and A. Ike, "GUNREAL: GPU-accelerated unsupervised reinforcement and auxiliary learning," in *Proc. 5th Int. Symp. Comput. Netw. (CANDAR)*, Nov. 2017, pp. 330–336, doi: 10.1109/CANDAR.2017.27.

[16] S. Dalton, I. Frosio, and M. Garland, "GPU-accelerated Atari emulation for reinforcement learning," 2019, *arXiv:1907.08467*.

[17] L. Espeholt, R. Marinier, P. Stanczyk, K. Wang, and M. Michalski, "SEED RL: Scalable and efficient deep-RL with accelerated central inference," 2019, *arXiv:1910.06591*.

[18] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu, "IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures," 2018, *arXiv:1802.01561*.

[19] Y. Fenjiro and H. Benbrahim, "Deep reinforcement learning overview of the state of the art," *J. Autom., Mobile Robot. Intell. Syst.*, vol. 12, no. 3, pp. 20–39, Dec. 2018, doi: 10.14313/JAMRIS_3-2018/15.

[20] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018, *arXiv:1802.09477*.

[21] P. R. Gankidi and J. Thangavelautham, "FPGA architecture for deep learning and its application to planetary robotics," in *Proc. IEEE Aerosp. Conf.*, Mar. 2017, pp. 1–9.

[22] P. R. Gankidi, "FPGA accelerator architecture for Q-learning and its applications in space exploration," M.S. thesis, Dept. Aerosp. Mech. Eng., Arizona State Univ., Tempe, ARI, USA, 2016.

[23] Google. (2020). *Tensor Processing Unit*. Accessed: Oct. 15, 2020. [Online]. Available: https://cloud.google.com/tpu/

[24] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*. Boston, MA, USA: Addison Wesley, 2003.

[25] C. Guo, W. Luk, S. Q. S. Loh, A. Warren, and J. Levine, "Customisable control policy learning for robotics," in *Proc. IEEE 30th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, vol. 2160, Jul. 2019, pp. 91–98.

[26] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[DL] a survey of FPGA-based neural network inference accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 1, pp. 1–26, Mar. 2019, doi: 10.1145/3289185.

[27] U. Gupta, S. K. Mandal, M. Mao, C. Chakrabarti, and U. Y. Ogras, "A deep Q-learning approach for dynamic management of heterogeneous processors," *IEEE Comput. Archit. Lett.*, vol. 18, no. 1, pp. 14–17, Jan./Jun. 2019, doi: 10.1109/LCA.2019.2892151.

[28] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," 2018, *arXiv:1812.05905*.

[29] X. Han, D. Zhou, S. Wang, and S. Kimura, "CNN-MERP: An FPGA-based memory-efficient reconfigurable processor for forward and backward propagation of convolutional neural networks," in *Proc. IEEE 34th Int. Conf. Comput. Design (ICCD)*, Oct. 2016, pp. 320–327.

[30] M. Hessel, J. Modayil, H. V. Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proc. AAAI*, 2018, pp. 1–14.
[31] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver, "Distributed prioritized experience replay," 2018, *arXiv:1803.00933*.
[32] G.-B. Huang, N. Liang, H.-J. Rong, P. Saratchandran, and N. Sundararajan, "On-line sequential extreme learning machine," in *Proc. IASTED Int. Conf. Comput. Intell.*, 2005, pp. 232–237.
[33] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: A new learning scheme of feedforward neural networks," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, vol. 2. Jul. 2004, pp. 985–990, doi: 10.1109/IJCNN.2004.1380068.
[34] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-learning algorithms: A comprehensive classification and applications," *IEEE Access*, vol. 7, pp. 133653–133667, 2019.
[35] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine, "QT-Opt: Scalable deep reinforcement learning for vision-based robotic manipulation," 2018, *arXiv:1806.10293*.
[36] S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney, "Recurrent experience replay in distributed reinforcement learning," in *Proc. ICLR*, 2019, pp. 1–12.
[37] S. Krishnan, M. Lam, S. Chitlangia, Z. Wan, G. Barth-Maron, A. Faust, and V. J. Reddi, "QuaRL: Quantization for sustainable reinforcement learning," 2019, *arXiv:1910.01055*.
[38] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," *Found. Trends Electron. Des. Autom.*, vol. 2, no. 2, pp. 135–253, Feb. 2008, doi: 10.1561/1000000005.
[39] M.-J. Li, A.-H. Li, Y.-J. Huang, and S.-I. Chu, "Implementation of deep reinforcement learning," in *Proc. 2nd Int. Conf. Inf. Sci. Syst.*, 2019, pp. 232–236, doi: 10.1145/3322645.3322693.
[40] Y. Li, "Deep reinforcement learning: An overview," 2017, *arXiv:1701.07274*.
[41] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox, "GPU-accelerated robotic simulation for distributed reinforcement learning," in *Proc. Conf. Robot Learn.*, 2018, pp. 270–282.
[42] S. Liang, Z. Yang, F. Jin, and Y. Chen, "Data centers job scheduling with deep reinforcement learning," in *Advances in Knowledge Discovery and Data Mining*, H. W. Lauw, R. C.-W. Wong, A. Ntoulas, E.-P. Lim, S.-K. Ng, and S. J. Pan, Eds. Cham, Switzerland: Springer, 2020, pp. 906–917.
[43] T. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, pp. 1–14, Sep. 2015.
[44] C.-T. Liu and R. C. Hsu, "Adaptive power management based on reinforcement learning for embedded system," in *New Frontiers in Applied Artificial Intelligence*, N. T. Nguyen, L. Borzemski, A. Grzech, and M. Ali, Eds. Berlin, Germany: Springer, 2008, pp. 513–522.
[45] S. K. Mandal, G. Krishnan, C. Chakrabarti, J.-S. Seo, Y. Cao, and U. Y. Ogras, "A latency-optimized reconfigurable NoC for in-memory acceleration of DNNs," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 10, no. 3, pp. 362–375, Sep. 2020, doi: 10.1109/JETCAS.2020.3015509.
[46] D. Mehta, "State-of-the-art reinforcement learning algorithms," *Int. J. Eng. Res.*, vol. V8, no. 12, pp. 717–722, Jan. 2020, doi: 10.17577/IJERTV8IS120332.
[47] Y. Meng, S. Kuppannagari, and V. Prasanna, "Accelerating proximal policy optimization on CPU-FPGA heterogeneous platforms," in *Proc. IEEE 28th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2020, pp. 19–27.
[48] Y. Meng, S. Kuppannagari, R. Rajat, A. Srivastava, R. Kannan, and V. Prasanna, "QTAccel: A generic FPGA based design for Q-table based reinforcement learning accelerators," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2020, pp. 107–114.
[49] S. Mittal, "A survey of ReRAM-based architectures for processing-in-memory and neural networks," *Mach. Learn. Knowl. Extraction*, vol. 1, no. 1, pp. 75–114, Apr. 2018, doi: 10.3390/make1010005.
[50] S. Mittal and S. Umesh, "A survey on hardware accelerators and optimization techniques for RNNs," *J. Syst. Archit.*, vol. 112, Jan. 2021, Art. no. 101839, doi: 10.1016/j.sysarc.2020.101839.
[51] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. ICML*, 2016, pp. 1928–1937.
[52] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.
[53] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. 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, no. 7540, pp. 529–533, 2019, doi: 10.1038/nature14236.
[54] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver, "Massively parallel methods for deep reinforcement learning," 2015, *arXiv:1507.04296*.
[55] H. Nguyen and H. La, "Review of deep reinforcement learning for robot manipulation," in *Proc. 3rd IEEE Int. Conf. Robotic Comput. (IRC)*, Feb. 2019, pp. 590–595.
[56] Nvidia. (2020). *NVIDIA FleX*. Accessed: Oct. 15, 2020. [Online]. Available: https://developer.nvidia.com/flex
[57] B. Parr, "Deep in-GPU experience replay," 2018, *arXiv:1801.03138*.
[58] B. A. Pearlmutter, "Fast exact multiplication by the Hessian," *Neural Comput.*, vol. 6, no. 1, pp. 147–160, Jan. 1994, doi: 10.1162/neco.1994.6.1.147.
[59] J. H. Postma, "Speeding up reinforcement learning with graphics processing units," M.S. thesis, Dept. Biomech. Eng., Delft Univ. Technol., Delft, The Netherlands, 2015.
[60] V. L. Prabha and E. C. Monie, "Hardware architecture of reinforcement learning scheme for dynamic power management in embedded systems," *EURASIP J. Embedded Syst.*, vol. 2007, pp. 1–6, Dec. 2007, doi: 10.1155/2007/65478.
[61] G. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," Dept. Eng., Univ. Cambridge, Cambridge, U.K., Tech. Rep., CUED/F-INFENG/TR 166, Nov. 1994.
[62] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.
[63] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization," in *Proc. 32nd Int. Conf. Mach. Learn.*, vol. 37, 2015, pp. 1889–1897.
[64] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with *in-situ* analog arithmetic in crossbars," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 14–26, 2016.
[65] S. Shao and W. Luk, "Customised pearlmutter propagation: A hardware architecture for trust region policy optimisation," in *Proc. 27th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2017, pp. 1–6.
[66] S. Shao, J. Tsai, M. Mysior, W. Luk, T. Chau, A. Warren, and B. Jeppesen, "Towards hardware accelerated reinforcement learning for application-specific robotic control," in *Proc. IEEE 29th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2018, pp. 1–8.
[67] L. M. D. Da Silva, M. F. Torquato, and M. A. C. Fernandes, "Parallel implementation of reinforcement learning Q-learning technique for FPGA," *IEEE Access*, vol. 7, pp. 2782–2798, 2019, doi: 10.1109/ACCESS.2018.2885950.
[68] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016, doi: 10.1038/nature16961.
[69] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "Near-memory computing: Past, present, and future," *Microprocessors Microsyst.*, vol. 71, Nov. 2019, Art. no. 102868, doi: 10.1016/j.micpro.2019.102868.
[70] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra, "A review of near-memory computing architectures: Opportunities and challenges," in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2018, pp. 608–617, doi: 10.1109/DSD.2018.00106.
[71] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 541–552, doi: 10.1109/HPCA.2017.55.
[72] S. Spano, G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, M. Matta, A. Nannarelli, and M. Re, "An efficient hardware implementation of reinforcement learning: The Q-learning algorithm," *IEEE Access*, vol. 7, pp. 186340–186351, 2019.

[73] A. Stooke and P. Abbeel, "Accelerated methods for deep reinforcement learning," 2018, *arXiv:1803.02811*.
[74] J. Su, J. Liu, D. B. Thomas, and P. Y. K. Cheung, "Neural network based reinforcement learning acceleration on FPGA platforms," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 4, pp. 68–73, Jan. 2017, doi: 10.1145/3039902.3039915.
[75] R. S. Sutton and A. G. Barto, *Reinforcement Learning-an Introduction* (Adaptive Computation and Machine Learning). Cambridge, MA, USA: MIT Press, 1998. [Online]. Available: http://www.worldcat.org/oclc/37293240
[76] R. Tavcar, J. Dedic, D. Bokal, and A. Zemva, "Transforming the LSTM training algorithm for efficient FPGA-based adaptive control of nonlinear dynamic systems," *Informacije Midem-J. Microelectron. Electron. Compon. Mater.*, vol. 43, no. 2, pp. 131–138, Jan. 2013.
[77] Y. Tian, Q. Gong, W. Shang, Y. Wu, and C. L. Zitnick, "ELF: An extensive, lightweight and flexible research platform for real-time strategy games," 2017, *arXiv:1707.01067*.
[78] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2012, pp. 5026–5033, doi: 10.1109/IROS.2012.6386109.
[79] Y. Umuroglu, N. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 65–74, doi: 10.1145/3020078.3021744.
[80] O. Vinyals, I. Babuschkin, W. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. Agapiou, M. Jaderberg, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, Nov. 2019, doi: 10.1038/s41586-019-1724-z.
[81] K. Vipin and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Comput. Surveys*, vol. 51, no. 4, pp. 1–39, Jul. 2018, doi: 10.1145/3193827.
[82] T. Wang, H. Dong, V. Lesser, and C. Zhang, "ROMA: Multi-agent reinforcement learning with emergent roles," 2020, *arXiv:2003.08039*.
[83] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," 2016, *arXiv:1611.01224*.
[84] H. Watanabe, M. Tsukada, and H. Matsutani, "An FPGA-based on-device reinforcement learning approach using online sequential learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, Jun. 2021, pp. 96–103.
[85] C. J. Watkins and P. Dayan, "Technical note: Q-learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 279–292, 1992.
[86] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, U.K., May 1989, doi: 10.1007/BF00992698.
[87] C. Wu, Y. Cui, C. Ji, T.-W. Kuo, and C. J. Xue, "Pruning deep reinforcement learning for dual user experience and storage lifetime improvement on mobile devices," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3993–4005, Nov. 2020, doi: 10.1109/TCAD.2020.3012804.
[88] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," 2020, *arXiv:2004.09602*.
[89] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation," in *Proc. NIPS*, 2017, pp. 5279–5288.
[90] Xilinx. (2020). *System Generator for DSP*. Accessed: Dec. 18, 2020. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html
[91] Q. Zhang, M. Lin, L. T. Yang, Z. Chen, S. U. Khan, and P. Li, "A double deep Q-learning model for energy-efficient edge scheduling," *IEEE Trans. Services Comput.*, vol. 12, no. 5, pp. 739–749, Sep. 2019, doi: 10.1109/TSC.2018.2867482.
[92] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-CNN: An FPGA-based framework for training convolutional neural networks," in *Proc. IEEE 27th Int. Conf. Appl.-Specific Syst., Archit. Processors (ASAP)*, Jul. 2016, pp. 107–114, doi: 10.1109/ASAP.2016.7760779.
[93] Z. Liu and I. Elhanany, "Large-scale tabular-form hardware architecture for Q-learning with delays," in *Proc. 50th Midwest Symp. Circuits Syst.*, Aug. 2007, pp. 827–830.

MARC ROTHMANN received the master's degree in computer science and intelligent systems from Bielefeld University, Germany, in 2019. He is currently pursuing the Ph.D. degree with the Computer Engineering Research Group, Osnabrück University. He is also working as a Research Assistant with the Computer Engineering Research Group, Osnabrück University. His master's thesis involved the implementation and evaluation of machine learning algorithms on embedded systems with limited resources. His research interest includes hardware acceleration of reinforcement learning algorithms.

MARIO PORRMANN (Member, IEEE) received the "Diplom-Ingenieur" degree in electrical engineering from the University of Dortmund, Germany, in 1994, and the Ph.D. degree in electrical engineering from the University of Paderborn, Germany, in 2001, for his work on "Performance Evaluation of Embedded Neurocomputer Systems." From 2001 to 2009, he was a "Akademischer Oberrat." From 2010 to March 2012, he was a Acting Professor with the Research Group System and Circuit Technology, Heinz Nixdorf Institute, University of Paderborn. He then joined the Research Group Cognitronics and Sensor Systems, Center of Excellence Cognitive Interaction Technology, at Bielefeld University, as an Academic Director. Since April 2019, he has been a Professor and the Head of the Computer Engineering Group, Osnabrück University, Germany. His research interests include resource-efficient computer architectures with a special emphasis on on-chip multiprocessor systems and dynamically reconfigurable hardware.