An efficient hardware implementation of reinforcement learning
The q-learning algorithm

Spanò, Sergio; Cardarilli, Gian Carlo; Di Nunzio, Luca; Fazzolari, Rocco; Giardino, Daniele; Matta, Marco; Nannarelli, Alberto; Re, Marco

Published in:
IEEE Access

Link to article, DOI:
10.1109/ACCESS.2019.2961174

Publication date:
2019

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
An Efficient Hardware Implementation of Reinforcement Learning: The Q-Learning Algorithm

SERGIO SPANÒ, GIAN CARLO CARDARILLI, LUCA DI NUNZIO, ROCCO FAZZOLARI, DANIELE GIARDINO, MARCO MATTA, ALBERTO NANNARELLI, AND MARCO RE

1Department of Electronic Engineering, University of Rome "Tor Vergata," 00133 Rome, Italy
2Department of Applied Mathematics and Computer Science, Danmarks Tekniske Universitet, 2800 Kgs. Lyngby, Denmark

Corresponding author: Sergio Spanò (spano@ing.uniroma2.it)

ABSTRACT In this paper we propose an efficient hardware architecture that implements the Q-Learning algorithm, suitable for real-time applications. Its main features are low-power, high throughput and limited hardware resources. We also propose a technique based on approximated multipliers to reduce the hardware complexity of the algorithm. We implemented the design on a Xilinx Zynq Ultrascale+ MPSoC ZCU106 Evaluation Kit. The implementation results are evaluated in terms of hardware resources, throughput and power consumption. The architecture is compared to the state of the art of Q-Learning hardware accelerators presented in the literature obtaining better results in speed, power and hardware resources. Experiments using different sizes for the Q-Matrix and different wordlengths for the fixed point arithmetic are presented. With a Q-Matrix of size $8 \times 4$ (8 bit data) we achieved a throughput of 222 MSPS (Mega Samples Per Second) and a dynamic power consumption of 37 mW, while with a Q-Matrix of size $256 \times 16$ (32 bit data) we achieved a throughput of 93 MSPS and a power consumption 611 mW. Due to the small amount of hardware resources required by the accelerator, our system is suitable for multi-agent IoT applications. Moreover, the architecture can be used to implement the SARSA (State-Action-Reward-State-Action) Reinforcement Learning algorithm with minor modifications.

INDEX TERMS Artificial intelligence, hardware accelerator, machine learning, Q-learning, reinforcement learning, SARSA, FPGA, ASIC, IoT, multi-agent.

I. INTRODUCTION

Reinforcement Learning (RL) is a Machine Learning (ML) approach used to train an entity, called agent, to accomplish a certain task [1]. Unlike the classic supervised and unsupervised ML techniques [2], RL does not require two separated training and inference phases being based on a trial & error approach. This concept is very close to the human learning.

As depicted in Fig. 1, the agent “lives” in an environment where it performs some actions. These actions may affect the environment which is time-variant and can be modelled as a Markovian Decision Process (MDP) [1]. An interpreter observes the scenario returning to the agent the state of the environment and a reward. The reward (or reinforcement) is a quality figure for the last action performed by the agent and it is represented as a positive or negative number. Through this iterative process, the agent learns an optimal action-selection policy to accomplish its task. This policy indicates which is the best action the agent should perform when the environment is in a certain state. Eventually, the interpreter may be integrated into the agent that becomes self-critic.

Thanks to this approach, RL represents a very powerful tool to solve problems where the operating scenario is unknown or changes over time.

Recently, the applications of RL have become increasingly popular in various fields such as robotics [3]–[5], Internet of Things (IoT) [6], power management [7], financial trading [8] and telecommunications [9], [10]. Another research area in RL is multi-agent and swarm systems [11]–[14].
This kind of applications require powerful computing platforms able to process very large amount of data as fast as possible and with limited power consumption. For these reasons, software-based implementations performance is now the main limitation in further development of such systems and the use of hardware accelerators based on FPGAs or ASICs can represent an efficient solution for implementing RL algorithms.

The main contribution of this work is a flexible and efficient hardware accelerator for the Q-Learning algorithm. The system is not constrained to any specific application, RL policy or environment. Moreover, for IoT target devices, a low-power version of the architecture based on approximated multipliers is presented.

The paper is organized as follows.

- Section I is a brief survey on Reinforcement Learning and its applications. Q-Learning algorithm, and the related work in the literature are presented.
- Section II describes the proposed hardware architecture, detailing its functional blocks. A technique to reduce the hardware complexity of the arithmetic operations is also proposed.
- Section III presents the implementation results and the comparisons with the state of the art.
- In sec. IV final considerations and future developments are given.
- Appendix shows how the architecture can be exploited to implement the SARSA (State-Action-Reward-State-Action) RL algorithm [15] with minor modifications.

### A. Q-LEARNING ALGORITHM

Q-Learning [16] is one of the most known and employed RL algorithms [17] and belongs to the class of off-policy methods since its convergence is guaranteed for any agent’s policy. It is based on the concept of Quality Matrix, also known as Q-Matrix. The size of this matrix is \( N \times Z \) where \( N \) is the number of the possible agent’s states to sense the environment and \( Z \) is the number of possible actions that the agent can perform. This means that Q-Learning operates in a discrete state-action space \( S \times A \). Considering a row of the Q-Matrix that represents a particular state, the best action to be performed is selected by computing the maximum value in the row.

At the beginning of the training process, the Q-Matrix is initialized with random or zero values, and it is updated by using (1).

\[
Q_{\text{new}}(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha \left( r_t + \gamma \max_a Q(s_{t+1}, a) \right)
\]

(1)

The variables in (1) refer to:

- \( s_t \) and \( s_{t+1} \): current and next state of the environment.
- \( a_t \) and \( a_{t+1} \): current and next action chosen by the agent (according to its policy).
- \( \gamma \): discount factor, \( \gamma \in [0, 1] \). It defines how much the agent has to take into account long-run rewards instead of immediate ones.
- \( \alpha \): learning rate, \( \alpha \in [0, 1] \). It determines how much the newest piece of knowledge has to replace the older one.
- \( r_t \): current reward value.

In [16] it is proved that the knowledge of the Q-Matrix suffices to extract the optimal action-selection policy for a RL agent.

### B. RELATED WORK

Despite the growing interest for RL and the need for systems capable to process large amount of data in very short time, just a few works can be found in the literature about the hardware implementation of RL algorithms. Moreover, the comparison is hard due to the lack of implementation details and homogeneous benchmarks. In this section we show the most prominent researches in this field.

In 2005, Hwang et al. [18] proposed a hardware accelerator for the “Flexible Adaptable Size Topology” (FAST) algorithm [19]. The system was implemented on a Xilinx XCV800 FPGA and was validated using the cart-pole problem [20]. The architecture is well described but few details about the implementation are given.

In 2007, Shao et al. [21] proposed a smart power management application for embedded systems based on the SARSAR algorithm [15]. The system was implemented on a Xilinx Spartan-II FPGA. Although the authors proved its functionality, neither the architecture nor the implementation details are given.

One of the most relevant work in the field is [22] by Gankidi et al. that, in 2017, proposed a RL accelerator for space rovers. The authors implemented the Deep Q-Learning technique [23] on a Xilinx Virtex-7 FPGA. They obtained a throughput of 2.34 MSPS (Mega Samples Per Second) for a \( 4 \times 2 \) state-action space.

Also in 2017, Su et al. [24] proposed another Deep Q-Learning hardware implementation based on an Intel Arria-10 FPGA. The architecture was compared to an Intel i7-930 CPU and a Nvidia GTX-760 GPU implementation. They achieved a throughput of 25 KSPS with 32 bit fixed point representation for a \( 27 \times 5 \) state-action space.

In 2018, Shao et al. [21] proposed a hardware accelerator for robotic applications based on “Trust Region Policy Optimization” (TRPO) [25]. The architecture was implemented
on different devices: FPGA (Intel Stratix-V), CPU (Intel i7-5930K) and GPU (Nvidia Tesla-C2070). With respect to the CPU, the authors obtained a speed-up factor of 4.14× and 19.29× for the GPU and the FPGA implementation, respectively.

The most recent works (published in 2019) include Cho et al. [26]. They propose a hardware accelerator for the “Asynchronous Advantage Actor-Critic” (AC3) algorithm [27], describing an implementation based on a Xilinx VCU1525 FPGA. The system was validated using 6 Atari-2600 videogames.

In the work by Li et al. [28] another Deep Q-Learning network was implemented on a Digilent Pynq development board for the cart-pole problem. The system is meant only for inference mode and, consequently, cannot be used for real-time learning.

One of the most advanced hardware accelerators for Q-Learning was proposed by Da Silva et al. [29]. The authors presented an implementation based on a Xilinx Virtex-6 FPGA. Moreover, they performed a fixed-point analysis to confirm the convergence of the algorithm. Different comparisons with state of the art implementations were made. Since this is one of best performing Q-Learning accelerators at today, we provide an extensive comparison with our architecture (sec. III-B).

II. PROPOSED ARCHITECTURE

The Q-Learning agent shown in Fig. 2 is composed by two main blocks: the Policy Generator (PG) and the Q-Learning accelerator.

![Figure 2. High level architecture of the Q-Learning agent.](image)

The agent receives the state \( s_{t+1} \) and the reward \( r_{t+1} \) from the observer, while the next action is generated by the PG according to the values of the Q-Matrix stored into the Q-Learning accelerator.

Note that \( s_t, a_t \) and \( r_t \) are obtained by delaying \( s_{t+1}, a_{t+1} \) and \( r_{t+1} \) by means of registers. \( s_t \) and \( a_t \) represent the indices of the rows and columns of the Q-Matrix, respectively. These delays do not affect the convergence of the Q-Learning algorithm, as proved in [30].

With the aim to design a general purpose hardware accelerator, we do not provide a particular implementation for the PG since it is application-defined. The PG has been included only in the experiments for the comparison with the state of the art (sec. III-B).

Figure 3 shows the Q-Learning accelerator.

The Q-Matrix is stored into \( Z \) Dual-Port RAMs, named Action RAMs. Consequently, we have one memory block per action. Each RAM contains an entire column of the Q-Matrix and the number of memory locations corresponds to the number of states \( N \). The read address is the next state \( s_{t+1} \), while the write address is the current state \( s_t \). The enable signals for the Action RAMs, generated by a decoder driven by the current action \( a_t \), select the value \( Q(s_t, a_t) \) to be updated. The Action RAMs outputs correspond to a row of the Q-Matrix \( Q(s_{t+1}, A) \).

The signal \( Q(s_t, a_t) \) is obtained by delaying the output of the memory blocks and then selecting the Action RAM through a multiplexer driven by \( a_t \). A MAX block fed by the output of the Action RAMs generates \( \max_a Q(s_{t+1}, A) \).

The Q-Updater (Q-Upd) block implements the Q-Matrix update equation (1) generating \( Q_{\text{new}}(s_t, a_t) \) to be stored into the corresponding Action RAM.

The accelerator can be also used for Deep Q-Learning [23] applications if the Action RAMs are replaced with Neural Network-based approximators.

A. MAX BLOCK

An extensive study about this block has been proposed in [30]. In the paper, the authors proved that the propagation delay of this block is the main limitation for the speed of Q-Learning accelerators when a large number of actions is required. Consequently, they propose an implementation based on a tree of binary comparators (\( M \)-stages) that is a good trade-off in area and speed [31].

This architecture is employed by the Q-Learning accelerators presented in [22], [29] and has also been used in our architecture (Fig. 4).

Moreover, in [30] it is proved that, when pipelining is used to speed up the MAX block, the latency does not affect the convergence of the Q-Learning algorithm. This means that, when an application requires a very high throughput, it is possible to use pipelining.

B. Q-UPDATER BLOCK

Equation (1) can be rearranged as

\[
Q_{\text{new}}(s_t, a_t) = Q(s_t, a_t) + \alpha \left( r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right)
\]

(2)

to obtain an efficient implementation. Equation (2) is computed by using 2 multipliers, while (1) requires 3 multipliers.

The Q-Updater block in Fig. 5 is used to compute (2), generating \( Q_{\text{new}}(s_t, a_t) \).

The critical path consists in 2 multipliers and 2 adders. In the next section (II-B1) a method to reduce the hardware complexity for the multipliers is illustrated.
1) APPROXIMATED MULTIPLIERS

The main speed limitation in the updater block is the propagation delay of the multipliers. Using a similar approach to [32], it is possible to replace the full multipliers shown in Fig. 5 with approximated multipliers based on barrel shifters [33]. In this way, we are approximating \( \alpha \) and \( \gamma \) with a number equal to their nearest power of two (single shifter), or to the nearest sum of powers of two (two or more shifters). Due to the fact that \( \alpha, \gamma \in [0, 1] \), only right shifts have been used.

Considering a number \( x \leq 1 \), its binary representation using \( M \) bits for the fractional part is:

\[
x = x_0 2^0 + x_{-1} 2^{-1} + x_{-2} 2^{-2} + \ldots + x_{-M} 2^{-M}
\]  

(3)

where \( x_0, \ldots, x_{-M} \) are the binary digits. Let \( i, j, k \) be the positions of the first, second and third ‘1’ in the binary representation of \( x \) starting from the most significant bit. Moreover, we define \( < x >_{OP} \) the approximation of \( x \) with the \( n \) most significant powers of two in the \( M + 1 \) bits representation. That is

\[
< x >_{OP_1} = 2^{-i} \\
< x >_{OP_2} = 2^{-i} + 2^{-j} \\
< x >_{OP_3} = 2^{-i} + 2^{-j} + 2^{-k}
\]

(4)

for the approximation with one, two and three powers of two. The concept can be extended to more power of two terms.

For example, \( x = 0.101101(2) = 0.703125 \) can be approximated as:

\[
< 0.101101(2) >_{OP_1} = 2^{-1} = 0.5 \\
< 0.101101(2) >_{OP_2} = 2^{-1} + 2^{-3} = 0.625 \\
< 0.101101(2) >_{OP_3} = 2^{-1} + 2^{-3} + 2^{-4} = 0.6875.
\]

(5)

Some examples of the approximated values for different powers of two are presented in Fig. 6 (\( x \leq 1 \)). Consequently, the product \( z = x \cdot y \) can be approximated as:

\[
< z >_{OP_1} = 2^{-i} \cdot y \\
< z >_{OP_2} = 2^{-i} \cdot y + 2^{-j} \cdot y \\
< z >_{OP_3} = 2^{-i} \cdot y + 2^{-j} \cdot y + 2^{-k} \cdot y.
\]

(6)

The approximated multipliers are implemented by one or more barrel shifters in the Q-Updater block, depending on the approximation, as shown in Fig. 7 and 8.
III. IMPLEMENTATION EXPERIMENTS

In order to validate the proposed architecture, we implemented different versions of the Q-Learning accelerator. In the experiments, we used a Xilinx Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit featuring the XCZU7EV-2FFVC1156 FPGA. All the results in this section were obtained using the Vivado 2019.1 EDA tool with default implementation parameters and setting a timing constraint of 2 ns. The system was coded in VHDL.

The design exploration was implemented for the following range of parameters:

- Number of bits for the Q-Matrix values: 8, 16 and 32 bit.
- Number of states $N$: 8, 16, 32, 64, 128 and 256.
- Number of actions $Z$: 4, 8 and 16.

We focused the implementation analysis on the following resources [37]:

- Look Up Tables (LUT);
- Look Up Tables used as RAM (LUTRAM);
- Flip-Flops (FF);
- Digital Signal Processing slices (DSP);

For every resource of the device, we also provide the percent usage respect to the total available.

The performances were measured in terms of maximum clock frequency (CLK) and dynamic power consumption (PWR). The latter was evaluated using Vivado after the Place&Route considering the maximum clock frequency and a worst case scenario with a 0.5 activity factor on the circuit nodes [38].

All the implementation examples in this section do not make use of pipelining in the MAX block (sec. II-A). Unless otherwise stated, no approximated multipliers are used.

Tables 1 to 9 show the implementation results for different number of states, actions and data-width for the Q-Matrix values (tables header color: blue 8-bit, red 16-bit, green 32-bit data-widths).

<table>
<thead>
<tr>
<th>$N$</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>DSP</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>193</td>
<td>0.08%</td>
<td>32</td>
<td>0.03%</td>
<td>154</td>
<td>0.03%</td>
</tr>
<tr>
<td>16</td>
<td>192</td>
<td>0.08%</td>
<td>32</td>
<td>0.03%</td>
<td>156</td>
<td>0.03%</td>
</tr>
<tr>
<td>32</td>
<td>193</td>
<td>0.08%</td>
<td>32</td>
<td>0.03%</td>
<td>158</td>
<td>0.03%</td>
</tr>
<tr>
<td>64</td>
<td>214</td>
<td>0.09%</td>
<td>64</td>
<td>0.06%</td>
<td>160</td>
<td>0.03%</td>
</tr>
<tr>
<td>128</td>
<td>271</td>
<td>0.12%</td>
<td>80</td>
<td>0.08%</td>
<td>162</td>
<td>0.04%</td>
</tr>
<tr>
<td>256</td>
<td>362</td>
<td>0.16%</td>
<td>160</td>
<td>0.16%</td>
<td>164</td>
<td>0.04%</td>
</tr>
</tbody>
</table>

The first consideration is related to the number of DSPs. Since only one Q-Matrix element is updated per clock cycle, the only parameter that affects the number of required DSPs is the bit-width. For a Q-Matrix with 8-bit data, we obtain the fastest implementations that do not require any DSP slice. For 16-bit and 32-bit data, 3 DSPs and 5 DSPs are required respectively.

Another consideration comes with the maximum clock frequency (that corresponds exactly to the throughput of the
TABLE 2. Implementation results for Q-Matrices with 8 bit data and $Z = 8$.

<table>
<thead>
<tr>
<th>N</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>DSP</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>290</td>
<td>64</td>
<td>252</td>
<td>0</td>
<td>0.00%</td>
<td>190</td>
</tr>
<tr>
<td>16</td>
<td>293</td>
<td>64</td>
<td>254</td>
<td>0</td>
<td>0.00%</td>
<td>168</td>
</tr>
<tr>
<td>32</td>
<td>294</td>
<td>64</td>
<td>256</td>
<td>0</td>
<td>0.00%</td>
<td>187</td>
</tr>
<tr>
<td>64</td>
<td>343</td>
<td>128</td>
<td>258</td>
<td>0</td>
<td>0.00%</td>
<td>183</td>
</tr>
<tr>
<td>128</td>
<td>451</td>
<td>160</td>
<td>260</td>
<td>0</td>
<td>0.00%</td>
<td>210</td>
</tr>
<tr>
<td>256</td>
<td>652</td>
<td>320</td>
<td>268</td>
<td>0</td>
<td>0.00%</td>
<td>188</td>
</tr>
</tbody>
</table>

TABLE 3. Implementation results for Q-Matrices with 8 bit data and $Z = 16$.

<table>
<thead>
<tr>
<th>N</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>DSP</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>497</td>
<td>128</td>
<td>318</td>
<td>0</td>
<td>0.00%</td>
<td>147</td>
</tr>
<tr>
<td>16</td>
<td>498</td>
<td>128</td>
<td>320</td>
<td>0</td>
<td>0.00%</td>
<td>130</td>
</tr>
<tr>
<td>32</td>
<td>499</td>
<td>128</td>
<td>322</td>
<td>0</td>
<td>0.00%</td>
<td>141</td>
</tr>
<tr>
<td>64</td>
<td>587</td>
<td>256</td>
<td>330</td>
<td>0</td>
<td>0.00%</td>
<td>149</td>
</tr>
<tr>
<td>128</td>
<td>717</td>
<td>320</td>
<td>344</td>
<td>0</td>
<td>0.00%</td>
<td>160</td>
</tr>
<tr>
<td>256</td>
<td>1192</td>
<td>640</td>
<td>346</td>
<td>0</td>
<td>0.00%</td>
<td>167</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>N</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>DSP</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>179</td>
<td>64</td>
<td>250</td>
<td>3</td>
<td>0.17%</td>
<td>152</td>
</tr>
<tr>
<td>16</td>
<td>178</td>
<td>64</td>
<td>252</td>
<td>3</td>
<td>0.17%</td>
<td>152</td>
</tr>
<tr>
<td>32</td>
<td>180</td>
<td>64</td>
<td>254</td>
<td>3</td>
<td>0.17%</td>
<td>149</td>
</tr>
<tr>
<td>64</td>
<td>204</td>
<td>96</td>
<td>256</td>
<td>3</td>
<td>0.17%</td>
<td>153</td>
</tr>
<tr>
<td>128</td>
<td>333</td>
<td>160</td>
<td>258</td>
<td>3</td>
<td>0.17%</td>
<td>158</td>
</tr>
<tr>
<td>256</td>
<td>513</td>
<td>320</td>
<td>272</td>
<td>3</td>
<td>0.17%</td>
<td>156</td>
</tr>
</tbody>
</table>

TABLE 5. Implementation results for Q-Matrices with 16 bit data and $Z = 8$.

<table>
<thead>
<tr>
<th>N</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>DSP</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>382</td>
<td>128</td>
<td>444</td>
<td>3</td>
<td>0.17%</td>
<td>129</td>
</tr>
<tr>
<td>16</td>
<td>383</td>
<td>128</td>
<td>446</td>
<td>3</td>
<td>0.17%</td>
<td>127</td>
</tr>
<tr>
<td>32</td>
<td>380</td>
<td>128</td>
<td>446</td>
<td>3</td>
<td>0.17%</td>
<td>132</td>
</tr>
<tr>
<td>64</td>
<td>417</td>
<td>192</td>
<td>450</td>
<td>3</td>
<td>0.17%</td>
<td>127</td>
</tr>
<tr>
<td>128</td>
<td>691</td>
<td>320</td>
<td>458</td>
<td>3</td>
<td>0.17%</td>
<td>135</td>
</tr>
<tr>
<td>256</td>
<td>1048</td>
<td>640</td>
<td>478</td>
<td>3</td>
<td>0.17%</td>
<td>136</td>
</tr>
</tbody>
</table>

TABLE 6. Implementation results for Q-Matrices with 16 bit data and $Z = 16$.

<table>
<thead>
<tr>
<th>N</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>DSP</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>759</td>
<td>256</td>
<td>577</td>
<td>3</td>
<td>0.17%</td>
<td>117</td>
</tr>
<tr>
<td>16</td>
<td>759</td>
<td>256</td>
<td>583</td>
<td>3</td>
<td>0.17%</td>
<td>117</td>
</tr>
<tr>
<td>32</td>
<td>761</td>
<td>256</td>
<td>583</td>
<td>3</td>
<td>0.17%</td>
<td>117</td>
</tr>
<tr>
<td>64</td>
<td>828</td>
<td>384</td>
<td>592</td>
<td>3</td>
<td>0.17%</td>
<td>115</td>
</tr>
<tr>
<td>128</td>
<td>1386</td>
<td>640</td>
<td>606</td>
<td>3</td>
<td>0.17%</td>
<td>121</td>
</tr>
<tr>
<td>256</td>
<td>2101</td>
<td>1280</td>
<td>632</td>
<td>3</td>
<td>0.17%</td>
<td>115</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>N</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>DSP</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>383</td>
<td>96</td>
<td>586</td>
<td>5</td>
<td>0.29%</td>
<td>136</td>
</tr>
<tr>
<td>16</td>
<td>383</td>
<td>96</td>
<td>583</td>
<td>5</td>
<td>0.29%</td>
<td>125</td>
</tr>
<tr>
<td>32</td>
<td>382</td>
<td>96</td>
<td>590</td>
<td>5</td>
<td>0.29%</td>
<td>106</td>
</tr>
<tr>
<td>64</td>
<td>417</td>
<td>160</td>
<td>592</td>
<td>5</td>
<td>0.29%</td>
<td>116</td>
</tr>
<tr>
<td>128</td>
<td>682</td>
<td>320</td>
<td>606</td>
<td>5</td>
<td>0.29%</td>
<td>112</td>
</tr>
<tr>
<td>256</td>
<td>1035</td>
<td>640</td>
<td>620</td>
<td>5</td>
<td>0.29%</td>
<td>110</td>
</tr>
</tbody>
</table>

As expected, the power consumption is proportional to the number of required LUTs (considering architectures with the same parameters). The trend can be observed in Figs. 10 and 11.

Even for the largest implementation considered ($N = 256$, $Z = 16$, 32-bit Q-Matrix values), the required FPGA resources are moderate. This suggests that the architecture can be easily employed in applications requiring a large number of states or actions and applications where multiple agents must be implemented on the same device.

The main result of the design exploration shows that we can implement fast Q-Learning accelerators with small amount of resources and low power consumption.
TABLE 8. Implementation results for Q-Matrices with 32 bit data and $Z = 8$.

<table>
<thead>
<tr>
<th>N</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>DSP</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>752</td>
<td>0.33%</td>
<td>192</td>
<td>0.19%</td>
<td>940</td>
<td>0.20%</td>
</tr>
<tr>
<td>16</td>
<td>759</td>
<td>0.33%</td>
<td>192</td>
<td>0.19%</td>
<td>942</td>
<td>0.20%</td>
</tr>
<tr>
<td>32</td>
<td>763</td>
<td>0.33%</td>
<td>192</td>
<td>0.19%</td>
<td>944</td>
<td>0.20%</td>
</tr>
<tr>
<td>64</td>
<td>845</td>
<td>0.37%</td>
<td>320</td>
<td>0.31%</td>
<td>958</td>
<td>0.21%</td>
</tr>
<tr>
<td>128</td>
<td>1387</td>
<td>0.60%</td>
<td>640</td>
<td>0.63%</td>
<td>972</td>
<td>0.21%</td>
</tr>
<tr>
<td>256</td>
<td>2095</td>
<td>0.91%</td>
<td>1280</td>
<td>1.26%</td>
<td>1094</td>
<td>0.22%</td>
</tr>
</tbody>
</table>

TABLE 9. Implementation results for Q-Matrices with 32 bit data and $Z = 16$.

<table>
<thead>
<tr>
<th>N</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>DSP</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1366</td>
<td>0.59%</td>
<td>384</td>
<td>0.38%</td>
<td>1092</td>
<td>0.24%</td>
</tr>
<tr>
<td>16</td>
<td>1362</td>
<td>0.59%</td>
<td>384</td>
<td>0.38%</td>
<td>1096</td>
<td>0.24%</td>
</tr>
<tr>
<td>32</td>
<td>1365</td>
<td>0.59%</td>
<td>384</td>
<td>0.38%</td>
<td>1100</td>
<td>0.24%</td>
</tr>
<tr>
<td>64</td>
<td>1328</td>
<td>0.66%</td>
<td>640</td>
<td>0.63%</td>
<td>1116</td>
<td>0.24%</td>
</tr>
<tr>
<td>128</td>
<td>2357</td>
<td>1.12%</td>
<td>1280</td>
<td>1.26%</td>
<td>1210</td>
<td>0.25%</td>
</tr>
<tr>
<td>256</td>
<td>4017</td>
<td>1.74%</td>
<td>2560</td>
<td>2.52%</td>
<td>1210</td>
<td>0.26%</td>
</tr>
</tbody>
</table>

A. Q-UPDATER BASED ON APPROXIMATED MULTIPLIERS

As discussed in sec. II-B1, to allow the use of IoT devices, the hardware complexity of the Q-Updater block can be reduced by replacing the full multipliers with approximated multipliers based on barrel shifters.

In order to evaluate the benefits of such approach, we implemented the multipliers by using the single power-of-two approach (1 barrel shifter per multiplier) and the more precise approach based on the linear combination of two powers-of-two (two barrel shifters per multiplier), as depicted in Figs. 7 and 8. We considered 8, 16 and 32 bit operands.

FIGURE 9. Average clock frequency for different Q-Matrix data bit-width vs number of actions.

Since the dynamic power consumption is directly proportional to the clock frequency [38], for a fair comparison we provide the energy required to update one Q-Matrix element and the percentage of energy saved respect to the traditional implementation. Tables 10, 11 and 12 show the comparison between the implementations of approximated and full multipliers. Note that for the 8 bit architectures the power dissipation was too low to be accurately estimated. In the traditional implementation, we forced the Vivado synthesizer not to use any DSP block.

FIGURE 10. Number of LUTs for different implementations.

FIGURE 11. Dynamic power consumption for different implementations.

TABLE 10. Implementation comparisons: approximated and full multipliers with 8 bit operands.

<table>
<thead>
<tr>
<th>Impl. type</th>
<th>LUT</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
<th>Energy (nJ)</th>
<th>Energy saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>35</td>
<td>0.01%</td>
<td>795</td>
<td>1×</td>
<td>0.002516</td>
</tr>
<tr>
<td>1 Shift</td>
<td>11</td>
<td>&lt;0.01%</td>
<td>1644</td>
<td>&lt;1</td>
<td>&lt;0.000608</td>
</tr>
<tr>
<td>2 Shift</td>
<td>27</td>
<td>0.01%</td>
<td>864</td>
<td>&lt;1</td>
<td>&lt;0.004116</td>
</tr>
</tbody>
</table>

TABLE 11. Implementation comparisons: approximated and full multipliers with 16 bit operands.

TABLE 12. Implementation comparisons: approximated and full multipliers with 32 bit operands.
The barrel shifter-based architectures do not require any DSP slice, they use less hardware resources, they are faster and more power-efficient than their full multiplier-based counterparts, especially for the 16 and 32 bit implementations. For these reasons, they are suitable for Q-Learning applications on very small and low-power IoT devices at the cost of a reduced set of possible $\alpha$ and $\gamma$ values.

B. STATE OF THE ART ARCHITECTURE COMPARISON

The architecture proposed in this paper has been compared with one of best performing Q-Learning hardware accelerators at today [29].

In their paper, Da Silva et al. proposed a parallel implementation based on the number of states $N$, while in our work the parallelization is based on the number of actions $Z$. Since in most of the RL applications $Z \ll N$ (see examples in sec. 1), our approach results in a smaller architecture.

Another important difference consists in the earlier selection of the Q-matrix value to be updated. This allows to implement a single block for the computations of $Q_{new}(s_t, a_t)$, while in [29] $N \times Z$ blocks are required. Moreover, in case of FPGA implementations, our architecture allows to employ distributed RAM or embedded block-RAM. This gives an additional degree of freedom compared to [29] where only registers are considered for storing the Q-Matrix values.

To obtain a fair comparison:

- We implemented the same RL environment of [29] and stored the reward values in a Look-Up Table.
- We implemented a random PG as described in [29].
- We considered 16-bit Q-Matrix values.
- We implemented the architectures on the same Virtex-6 FPGA ML605 Evaluation Kit (using the ISE 14.7 Xilinx suite).

The experimental results are shown in Tables 13, 14, 15 and 16. We can only make comparisons with $Z = 4$ and $Z = 8$ since they are the only values implemented in [29].

The implementation results are given in terms of [39]:

- DSP blocks (DSP)
- Slice Registers (REG)
- Slice LUT (LUT)
- Maximum clock frequency (CLK)
- Power consumption (PWR)
- Energy required to update one Q-Matrix element (Energy)

As expected, our architecture employs a constant number of DSP slices, while in [29] this number is proportional

### B. STATE OF THE ART ARCHITECTURE COMPARISON

The architecture proposed in this paper has been compared with one of best performing Q-Learning hardware accelerators at today [29].

In their paper, Da Silva et al. proposed a parallel implementation based on the number of states $N$, while in our work the parallelization is based on the number of actions $Z$. Since in most of the RL applications $Z \ll N$ (see examples in sec. 1), our approach results in a smaller architecture.

Another important difference consists in the earlier selection of the Q-matrix value to be updated. This allows to implement a single block for the computations of $Q_{new}(s_t, a_t)$, while in [29] $N \times Z$ blocks are required. Moreover, in case of FPGA implementations, our architecture allows to employ distributed RAM or embedded block-RAM. This gives an additional degree of freedom compared to [29] where only registers are considered for storing the Q-Matrix values.

To obtain a fair comparison:

- We implemented the same RL environment of [29] and stored the reward values in a Look-Up Table.
- We implemented a random PG as described in [29].
- We considered 16-bit Q-Matrix values.
- We implemented the architectures on the same Virtex-6 FPGA ML605 Evaluation Kit (using the ISE 14.7 Xilinx suite).
TABLE 16. Proposed implementation results for 16 bit Q-Matrix values and $Z = 8$.

<table>
<thead>
<tr>
<th>N</th>
<th>DSP</th>
<th>REG</th>
<th>LUT</th>
<th>CLK (MHz)</th>
<th>PWR (mW)</th>
<th>Energy (nJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>0.39%</td>
<td>316</td>
<td>316</td>
<td>62</td>
<td>20</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>0.39%</td>
<td>318</td>
<td>324</td>
<td>62</td>
<td>18</td>
</tr>
<tr>
<td>32</td>
<td>3</td>
<td>0.39%</td>
<td>320</td>
<td>320</td>
<td>63</td>
<td>18</td>
</tr>
<tr>
<td>64</td>
<td>3</td>
<td>0.39%</td>
<td>322</td>
<td>423</td>
<td>64</td>
<td>22</td>
</tr>
<tr>
<td>128</td>
<td>3</td>
<td>0.39%</td>
<td>324</td>
<td>655</td>
<td>63</td>
<td>24</td>
</tr>
<tr>
<td>256</td>
<td>3</td>
<td>0.39%</td>
<td>326</td>
<td>1064</td>
<td>59</td>
<td>35</td>
</tr>
</tbody>
</table>

The number of Slice Registers required by our implementations remains almost unaltered when the number of states increases, while in [29] it grows with $N$.

Figure 12 compares the maximum clock frequency for different number of states and actions. Our system is more than 3 times faster and the speed is almost independent to the Q-Matrix number of states.

Figure 13 compares the energy required to update a single Q-Matrix element for different number of states and actions. Also in this case, our architecture, except for the $N = 6$ $Z = 4$ case, presents a better energy efficiency which remains almost unaltered increasing the number of states.

It is important to highlight that the most evident difference between the proposed architecture and [29] is its independence from the environment and agent’s policy. This happens because the system in [29] cannot be used as a general-purpose hardware accelerator since the RL environment is mapped on the FPGA. Our system does not have such limitation.

IV. CONCLUSION

In this paper we proposed an efficient hardware implementation for the Reinforcement Learning algorithm called Q-Learning. Our architecture exploits the learning formula by a-priori selecting the required element of the Q-Matrix to be updated. This approach made possible to minimize the hardware resources.

We also presented an alternative method for reducing the computational complexity of the algorithm by employing approximated multipliers instead of full multipliers. This technique is an effective solution to implement the accelerator on small ultra low-power FPGAs for IoT applications.

Our architecture has been compared to the state of the art in the literature, showing that our solution requires a smaller amount of hardware resources, is faster and dissipates less power. Moreover, our system can be used as a general-purpose hardware accelerator for the Q-Learning algorithm, not being related to a particular RL environment or agent’s policy.

With little effort, the proposed approach can be also exploited to implement the on-policy version of the Q-Learning algorithm: SARSA. This aspect is further explored in Appendix.

For the above reasons, our architecture is suitable for high-throughput and low-power applications. Due to the small amount of required resources, it also allows the implementation of multiple Q-Learning agents on the same device, both on FPGA or ASIC.

APPENDIX

SARSA ACCELERATOR ARCHITECTURE

The proposed architecture for the acceleration of the Q-Learning algorithm can be easily exploited to implement the SARSA (State-Action-Reward-State-Action) [15] algorithm. Equation (7) shows the SARSA update formula for the Q-Matrix.

$$Q_{\text{new}}(s_t, a_t) = Q(s_t, a_t) + \alpha (r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

(7)

Comparing (2) to (7), it is straightforward to note the similarities between the two equations. Since the update of the Q-Matrix depends on the agent’s next action $a_{t+1}$, SARSA algorithm is the on-policy version of the Q-Learning algorithm (which is off-policy).

The resulting architecture is presented in Fig. 14. The main difference between the Q-Learning implementation in Fig. 3...
The analysis about the Q-Learning architecture can also be extended to the SARSA accelerator.

ACKNOWLEDGMENT

The authors would like to thank Xilinx Inc., for providing FPGA hardware and software tools by Xilinx University Program.

REFERENCES


FIGURE 14. SARSA accelerator architecture.

consists in the replacement of the MAX block with a multiplexer driven by the next action $a_{t+1}$.

$Q(s_t, a_t) ightarrow Q_{new}(s_t, a_t)$

$Q(s_{t+1}, a_{t+1}) ightarrow Q_{Upd}$

$z^1 \rightarrow z^1$

$\vdots$

$\vdots$

$S_t \rightarrow S_{t+1}$

$A_{t} \rightarrow A_{t+1}$

$EN_2 \rightarrow EN_2$
LUCA DI NUNZIO received the master’s degree (summa cum laude) in electronics engineering and the Ph.D. degree in systems and technologies for the space from the University of Rome “Tor Vergata,” in 2006 and 2010, respectively. He has a working history with several companies in the fields of electronics and communications. He is currently an Adjunct Professor with the Digital Electronics Laboratory, University of Rome “Tor Vergata” and an Adjunct Professor of digital electronics with University Guglielmo Marconi. His research activities are in the fields of reconfigurable computing, communication circuits, digital signal processing, and machine learning.

ROCCO FAZZOLARI received the master’s degree in electronic engineering and the Ph.D. degree in space systems and technologies from the University of Rome Tor Vergata, Italy, in May 2009 and in June 2013, respectively. He is currently a Postdoctoral Fellow and an Assistant Professor with the Department of Electronic Engineering, University of Rome “Tor Vergata”. He works on hardware implementation of high-speed systems for digital signals processing, machine learning, array of wireless sensor networks, and systems for data analysis of acoustic emission (AE) sensors (based on ultrasonic waves).

DANIELE GIARDINO received the B.S. and M.S. degrees in electronic engineering from the University of Rome “Tor Vergata,” Italy, in 2015 and 2017, respectively, where he is currently pursuing the Ph.D. degree in electronic engineering and is a member of the DSPVLSI research group. He has industrial experiences in the space and telecommunications field. His interests include digital signal processing, machine learning, telecommunications, and ASIC/FPGA hardware design. His current research topics related to machine learning hardware implementations for embedded and low-power systems.

MARCO MATTA was born in Cagliari, Italy, in 1989. He received the B.S. and M.S. degrees in electronic engineering from the University of Rome “Tor Vergata”, Italy, in 2014 and 2017, respectively. He is currently pursuing the Ph.D. degree in electronic engineering. Since 2017, he has been a member of the DSPVLSI research group, University of Rome “Tor Vergata”. His research interests include the development of hardware platforms, and low-power accelerators aimed at machine learning algorithms and telecommunications. In particular, he is currently focused on the implementation of reinforcement learning models on FPGA.

GIAN CARLO CARDARILLI (S’79–M’81) was born in Rome, Italy. He received the Laurea degree (summa cum laude) from the University of Rome “La Sapienza,” in 1981. From 1992 to 1994, he was with the University of L’Aquila. From 1987 to 1988, he was with the Circuits and Systems team, EPFL, Lausanne, Switzerland. He has been with the University of Rome “Tor Vergata,” since 1984. He is currently a Full Professor of digital electronics and electronics for communication systems with the University of Rome “Tor Vergata.” He has regular cooperation with companies, such as Alcatel Alenia Space, Italy, STM, Agnate Brianza, Italy, Micron, Italy, and Selex S.I., Italy. He works in the field of computer arithmetic and its application to the design of fast signal digital processor. His interests are in the area of VLSI architectures for signal processing and IC design. In this field, he published over than 160 articles in international journals and conferences. His scientific interest concerns the design of special architectures for signal processing.

SERGIO SPANO received the B.S. degree in electronic engineering and the M.S. degree (summa cum laude) in electronic engineering from the University of Rome “La Sapienza,” Rome, Italy, in 2015 and 2018, respectively, where he is currently pursuing the Ph.D. degree in electronic engineering, as a member of the DSPVLSI research group. He has industrial experiences in the space and telecommunications field. His interests include digital signal processing, machine learning, telecommunications, and ASIC/FPGA hardware design. His current research topics related to machine learning hardware implementations for embedded and low-power systems.

IEEE Access
ALBERTO NANNARELLI (S’94–M’99–SM’13) graduated in electrical engineering from the University of Roma “La Sapienza,” Roma, Italy, in 1988, and received the M.S. and Ph.D. degrees in electrical and computer engineering from the University of California at Irvine, CA, USA, in 1995 and 1999, respectively. He worked for SGS-Thomson Microelectronics and Ericsson Telecom as a Design Engineer and for Rockwell Semiconductor Systems as a summer Intern. From 1999 to 2003, he was with the Department of Electrical Engineering, University of Roma “Tor Vergata,” Italy, as a Postdoctoral Researcher. He is currently an Associate Professor with the Technical University of Denmark, Lyngby, Denmark. His research interests include computer arithmetic, computer architecture, and VLSI design. He is a Senior Member of the IEEE Computer Society.

MARCO RE (M’92) received the Ph.D. degree in microelectronics from the University of Rome “Tor Vergata.” He is currently an Associate Professor with the University of Rome “Tor Vergata,” where he teaches digital electronics and hardware architectures for DSP. He is the Director of a master in audio engineering with the Department of Electronic Engineering, University of Rome Tor Vergata. He was awarded with two NATO fellowships at the University of California at Berkeley, working as a Visiting Scientist with Cadence Berkeley Laboratories. He has been awarded with the Otto Moensted fellowship as a Visiting Professor with the Technical University of Denmark. He collaborates in many research projects with different companies in the field of DSP architectures and algorithms. He is author of about 200 articles on international journals and conferences. His main scientific interests are in the field of low power DSP algorithms’ architectures, hardware–software codesign, fuzzy logic and neural hardware architectures, low power digital implementations based on non-traditional number systems, computer arithmetic, and CAD tools for DSP. He is member of Audio Engineering Society (AES).