Abstract—DQN (Deep Q-Network) is a method to perform Q-learning for reinforcement learning using deep neural networks. DQNs require a large buffer and batch processing for an experience replay and rely on a backpropagation based iterative optimization, making them difficult to be implemented on resource-limited edge devices. In this paper, we propose a lightweight on-device reinforcement learning approach for low-cost FPGA devices. It exploits a recently proposed neural-network based on-device learning approach that does not rely on the backpropagation method but uses OS-ELM (Online Sequential Extreme Learning Machine) based training algorithm. In addition, we propose a combination of L2 regularization and spectral normalization for the on-device reinforcement learning so that output values of the neural network can be fit into a certain range and the reinforcement learning becomes stable. The proposed reinforcement learning approach is designed for PYNQ-Z1 board as a low-cost FPGA platform. The evaluation results using OpenAI Gym demonstrate that the proposed algorithm and its FPGA implementation complete a CartPole-v0 task 29.77x and 89.40x faster than a conventional DQN-based approach when the number of hidden-layer nodes is 64.

I. INTRODUCTION

Reinforcement learning differs from a typical deep learning in that agents themselves explore their environment and learn appropriate actions. This means that it learns correct actions while creating a dataset. In DQN (Deep Q-Network) [1], Q-learning for reinforcement learning is replaced with deep neural networks so that it can acquire a high generalization capability by the deep neural networks. In this case, continuous input values can be used as inputs. Also, to reduce a dependence on a sequence of input data, an experience replay technique [2], in which past experiences including states, actions, and rewards are recorded in a buffer and then randomly picked up for training, is typically used for DQNs. However, such DQNs are costly for resource-limited edge devices and a standalone execution on edge devices is not feasible, because they rely on a backpropagation based training algorithm that iteratively optimizes their weight parameters and the convergence is sometimes time-consuming.

In this paper, we propose a lightweight on-device reinforcement learning approach for resource-limited FPGA devices. It exploits a recently proposed neural-network based on-device learning approach [3] that does not rely on the backpropagation methods but uses OS-ELM (Online Sequential Extreme Learning Machine) based training algorithm [4]. Computational cost for this training algorithm is quite low, because its weight parameters are analytically solved in a one-shot manner without the backpropagation based iterative optimization. In theory, it has been demonstrated that it can satisfy the universal approximation theorem [5] as in deep learning.

However, since the training algorithm of OS-ELM assumes single hidden-layer neural networks, their output values tend to be unstable in some cases, e.g., when they are overfit to some specific inputs and/or when unknown patterns are fed. In the case of reinforcement learning, one of crucial issues is that an action acquisition with Q-learning becomes unstable. To address this issue, this paper proposes a combination of L2 regularization and spectral normalization [6] so that output values of the proposed OS-ELM Q-Network can be fit into a certain range and the reinforcement learning becomes stable. This enables us to implement the reinforcement learning on small-sized FPGA devices for standalone execution on resource-limited edge devices. In this paper, the proposed reinforcement learning approach is designed for PYNQ-Z1 board. The evaluation results using OpenAI Gym show that the proposed algorithm and its FPGA implementation complete a CartPole task 29.77x and 89.40x faster than a conventional DQN when the number of hidden-layer nodes is 64.

The rest of this paper is organized as follows. Section II introduces basic technologies behind our proposal. Section III proposes the lightweight on-device reinforcement learning approach and illustrates an FPGA implementation. In Section IV, it is evaluated in terms of training curve and execution time to complete a CartPole task. Section V summarizes this paper.

II. PRELIMINARIES

This section introduces (1) ELM (Extreme Learning Machine), (2) OS-ELM (Online Sequential ELM), (3) ReOS-ELM (Regularized OS-ELM), and (4) DQN (Deep Q-Network).

A. ELM

ELM [7] is a batch training algorithm for single hidden-layer neural networks. In this case, the network consists of input layer, hidden layer, and output layer (see Figure 1 in a few pages later). The numbers of their nodes are n, N, and m, respectively.

Assuming an n-dimensional input chunk $x \in \mathbb{R}^{k \times n}$ with batch size $k$ is given, an m-dimensional output chunk $y \in \mathbb{R}^{k \times m}$ is computed as follows,

$$y = G(x \cdot \alpha + b)\beta,$$  \hspace{1cm} (1)

where $G$ is an activation function, $\alpha \in \mathbb{R}^{n \times N}$ is an input weight matrix between input and hidden layers, $\beta \in \mathbb{R}^{N \times m}$ is an output weight matrix between hidden and output layers, and $b \in \mathbb{R}^{N}$ is a bias vector of the hidden layer.
Assuming this neural network approximates an \( m \)-dimensional target chunk (i.e., teacher data) \( t \in \mathbb{R}^{k \times m} \) with zero error, the following equation is satisfied.
\[
G(x \cdot \alpha + b)\beta = t
\]  
(2)
Here, the hidden layer matrix is defined as \( H \equiv G(x \cdot \alpha + b) \). The optimal output weight matrix \( \beta \) is computed as follows.
\[
\hat{\beta} = H^\dagger t,
\]  
(3)
where \( H^\dagger \) is a pseudo inverse matrix of \( H \), which can be computed with matrix decomposition algorithms, such as SVD and QRD (QR Decomposition).

In ELM algorithm, the input weight matrix \( \alpha \) is initialized with random values and not changed thereafter. The optimization is thus performed only for the output weight matrix \( \beta \); thus, it is quite simple compared with backpropagation based neural networks that optimize both \( \alpha \) and \( \beta \). In addition, the training algorithm of ELM is not iterative; it analytically computes the optimal weight matrix \( \beta \) for a given input chunk in one shot, as shown in Equation 3. That is, it can always obtain the optimal \( \beta \) in one shot, unlike a typical gradient descent method that iteratively tunes the parameters toward the optimal solution.

Please note that ELM is a batch training algorithm and it becomes costly when the training data size grows sequentially. This means that, when a new training data arrives, the whole dataset including the new data must be retrained to update the model. This issue is a limiting factor for reinforcement learning, which can be addressed by OS-ELM.

B. OS-ELM

OS-ELM [4] is an online sequential version of ELM, which can update the model sequentially using an arbitrary batch size. Assuming that the \( i \)-th training chunk \( \{ x_i \in \mathbb{R}^{k_i \times n}, t_i \in \mathbb{R}^{k_i \times m} \} \) with batch size \( k_i \) is given, we need to compute an output weight matrix \( \beta \), that can minimize the following error.
\[
\begin{pmatrix}
H_0 \\
\vdots \\
H_i
\end{pmatrix}
\begin{pmatrix}
\beta_1 \\
\vdots \\
\beta_i
\end{pmatrix}^2
= \begin{pmatrix}
t_0 \\
\vdots \\
t_i
\end{pmatrix},
\]  
(4)
where \( H_i \) is defined as \( H_i \equiv G(x_i \cdot \alpha + b) \).

Assuming \( P_i \equiv \begin{pmatrix}
H_0^\top \\
\vdots \\
H_i^\top
\end{pmatrix} \begin{pmatrix}
H_0 \\
\vdots \\
H_i
\end{pmatrix}^{-1} \) (\( i \geq 0 \)), the optimal output weight matrix is computed as follows.
\[
P_i = P_{i-1} - P_{i-1}H_i^\top (I + H_i^\top P_{i-1}H_i^\top)^{-1} H_i P_{i-1}
\]  
(5)
\[
\beta_i = \beta_{i-1} + P_i H_i^\top (t_i - H_i \beta_{i-1})
\]

In particular, the initial values \( P_0 \) and \( \beta_0 \) are precomputed as follows. This computation is called initial training.
\[
P_0 = (H_0^\top H_0)^{-1}
\]  
(6)
\[
\beta_0 = P_0 H_0^\top t_0
\]

As shown in Equation 5, the output weight matrix \( \beta_i \) and its intermediate result \( P_i \) are computed from the previous training results \( \beta_{i-1} \) and \( P_{i-1} \). Thus, OS-ELM can sequentially update the model with a newly-arrived target chunk in one shot, and there is no need to retrain all the past data unlike ELM.

In this approach, the major bottleneck is the pseudo inverse operation \( (I + H_i P_{i-1} H_i^\top)^{-1} \) in Equation 5. As proposed in [3], the batch size \( k \) is fixed at 1 in this paper so that the pseudo inverse operation of \( k \times k \) matrix for the sequential training is replaced with a simple reciprocal operation; thus, we can eliminate SVD or QRD computation from Equation 5.

C. ReOS-ELM

ReOS-ELM [8] is an OS-ELM variant where an L2 regularization is applied to the output weight matrix \( \beta \) so that it can mitigate an overfitting issue of OS-ELM and improve its generalization capability. The training algorithm of ReOS-ELM is same as that of OS-ELM, except that the initial training of \( P_0 \) and \( \beta_0 \) is changed as follows.
\[
P_0 = (H_0^\top H_0 + \delta I)^{-1}
\]  
(7)
\[
\beta_0 = P_0 H_0^\top t_0,
\]
where \( \delta \) is a regularization parameter that controls an importance of the regularization term.

D. Reinforcement Learning and DQN

In DQNs, deep neural networks are used for Q-learning which is a typical reinforcement learning algorithm. In time step \( t \), \( Q_{\theta_1}(s_t, a_t) \) represents a value for taking action \( a_t \) in state \( s_t \), predicted with a set of neural network parameters \( \theta_1 \). In this case, \( \theta_1 \) is trained so that the value \( Q_{\theta_1}(s_t, a_t) \) can be predicted accurately by the neural network. However, if \( \theta_1 \) is trained for each time step \( t \), it is continuously changed and the Q-learning will not be stable. To address this issue, DQNs use a fixed target Q-network technique [9], in which another neural network with a set of parameters \( \theta_2 \) is used for stabilizing the Q-learning, in addition to that with \( \theta_1 \). More specifically, \( \theta_2 \) is used but fixed for a while, and it is updated with \( \theta_1 \) at a predefined interval.

In DQNs, an optimization target is computed as follows.
\[
f(r_t, s_{t+1}, d_t) = r_t + (1 - d_t) \gamma \max_{a \in A} Q_{\theta_2}(s_{t+1}, a),
\]  
(8)
where \( \gamma \in [0, 1] \) is a discount rate that controls an importance of the next step, \( r_t \) is a current reward given by an environment, and \( d_t \) indicates if the current episode \(^1\) is finished or not. If \( d_t \) is equal to 1, the current episode is finished and a new episode is started. As shown in Equation 8, the sum of the reward and the maximum Q-value among all the possible actions \( A \) in one step ahead is regarded as the optimization target. As mentioned above, \( \theta_2 \) is periodically updated with \( \theta_1 \) by using the fixed target Q-network technique. Specifically, the loss value for \( \theta_1 \) is denoted as follows [10].
\[
L(\theta_1) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}, d_t) \sim D}
\]  
(9)
\[
\left[ (Q_{\theta_1}(s_t, a_t) - f(r_t, s_{t+1}, d_t))^2 \right],
\]
\(^1\)In this paper, an episode is defined as a complete sequence of states, actions, and rewards.
where $D$ is a buffer for the experience replay technique [1], which is used to suppress impacts of temporal dependence on input data for training. In this case, past experiences (e.g., $s_t$, $a_t$, $r_t$, $s_{t+1}$, and $d_t$ in Equation 9) are stored in the buffer $D$. Then, they are randomly picked up from the buffer to form a batch which will be used for updating the weight parameters of the neural network.

E. Spectral Regularization and Spectral Normalization

To stabilize an action acquisition with Q-learning, we focus on regularization methods used in deep learning. Specifically, for reinforcement learning, a range of neural network outputs should be within a constant multiplication of their input for the stability. Such a property is referred to as Lipschitz continuity. More specifically, assuming an input value is changed from $x_1$ to $x_2$, their output values $f(x_1)$ and $f(x_2)$ should satisfy the following constraint:

$$\forall x_1, x_2, \|f(x_1) - f(x_2)\| \leq K \|x_1 - x_2\|,$$

where $K \in \mathbb{R}$ is a constant value called Lipschitz constant. Lipschitz constant of a neural network is derived by partial products of Lipschitz constants of all the layers, each of which is equal to a product of Lipschitz constant of a weight matrix (i.e., its largest singular value) and that of an activation function (i.e., $\leq 1$ for ReLU and tanh). It should be suppressed for the stable Q-learning. A spectral regularization [11] can be used to suppress the Lipschitz constant of a neural network, in which the sum of the largest singular value in each weight matrix is added to the loss function as a penalty term.

In practice, well-known extensions of spectral regularization is spectral normalization [6], in which an output of a neural network is computed based on partial products of input data and each weight matrix divided by its largest singular value. In this case, the Lipschitz constant is limited to $\leq 1$. Since 1-Lipschitz continuity is required for GANs (Generative Adversarial Networks), it is widely used in these applications. In this paper, we use this approach for stabilizing the OS-ELM based reinforcement learning.

III. ON-DEVICE REINFORCEMENT LEARNING APPROACH

In Q-learning, the value $Q_\theta(s_t,a_t)$ is approximated with a neural network. Toward the standalone reinforcement learning on resource-limited edge devices, in this paper we propose to use OS-ELM for this purpose.

A. Baseline OS-ELM Q-Network

Algorithm 1 shows the proposed OS-ELM Q-Network. It consists of four states: Determine, Observe, Store, and Update.

- In Determine state (lines 10-13), a current action $a_t$ is determined based on the current state $s_t$. More specifically, an action that maximizes the Q-value (line 11) or randomly-selected one (line 13) is selected as $a_t$.
- In Observe state (lines 14-16), based on an interaction using $a_t$ with the environment, the next state $s_{t+1}$, reward $r_t$, and flag $d_t$ are observed.
- In Store state (line 17), these observed values, action $a_t$, and state $s_t$ are stored in buffer $D$ so that they can be used in Update state.

Algorithm 1: OS-ELM Q-Network

```plaintext
1 Initialize parameters $\theta_1 = \{\alpha_0, \beta_0\}$ using random values $\mathbb{R} \in [0,1]$
2 $\sigma_{max}(\alpha_0) \leftarrow \text{SVD}(\alpha_0)$
3 $\alpha_0 \leftarrow \alpha_0 / \sigma_{max}(\alpha_0)$ // Initialize $\alpha_0$
4 Initialize parameters $\theta_2$ as $\theta_2 \leftarrow \theta_1$
5 Initialize buffer $D$
6 Initialize global step $t$
7 for episode $\in 1 \ldots$ do
8   for step $\in 1 \ldots$ do
9     $t \leftarrow t + 1$
10    // Determine
11    if random value $r_1 < \varepsilon_1$ then
12      $a_t \leftarrow \text{arg max}_{a \in A} Q_{\theta_1}(s_t, a)$
13    else
14      $a_t \leftarrow \text{random action value}$
15    // Observe
16    Observe $(s_{t+1}, r_t, d_t)$ from environment
17    if $d_t == 1$ then
18      Break
19    // Store
20    Store $(s_t, a_t, r_t, s_{t+1}, d_t)$ in buffer $D$
21 for episode $\%$ UPDATE_STEP $== 0$ then
22    $\theta_2 \leftarrow \theta_1$
```

- In Update state (lines 18-23), $\beta$ is initialized or updated, depending on the global step $t$. More specifically, it is initially trained with stored values in $D$ (line 20) based on Equation 6 when the number of experiences in $D$ is same as $\tilde{N}$ (i.e., $t == \tilde{N}$). Or, it is sequentially updated with the latest experience (line 23) based on Equation 5 when $t > \tilde{N}$. The former is referred as an initial training and the latter is referred as a sequential training.

Please note that the buffer $D$ is used for the initial training only and it is not used in subsequent sequential training in the case of OS-ELM Q-Network.

a) Fixed Target Q-Network: OS-ELM Q-Network uses the fixed target Q-network technique as well as DQNs. At first, two sets of neural network parameters $\theta_1$ and $\theta_2$ are initialized in lines 1 and 4, $\theta_1$ is updated more frequently (lines 20 and 23) and $\theta_2$ is synchronized with $\theta_1$ at a certain interval.
(lines 24-25). Please note that a straightforward algorithm that approximates $Q(s_t, a_t)$ with OS-ELM is unstable and cannot complete a reinforcement learning task in this paper. We thus introduce some techniques below in order to improve OS-ELM Q-Network.

b) Simplified Output Model: In DQNs, the $i$-th node of an output layer is Q-value of the $i$-th action, and they are trained so that the $i$-th node can predict $Q(s, a_i)$ accurately. In this case, their input and output sizes are equal to the numbers of state variables and actions, respectively. The left hand side of Figure 2 shows an example of such a network when the numbers of state variables and actions are 4 and 2, respectively. Since an action value is fed to the model in this case, Q-value is calculated for all the possible actions.

In Update state of DQNs, a loss value computed with Equations 8 and 9 is used for the backpropagation based iterative optimization. In OS-ELM, on the other hand, teacher data $t \in \mathbb{R}^m$ is required to update $\beta$ when the batch size $k$ is 1, as shown in Equation 5. To directly use $(r_t + (1 - d_t)) \gamma \max_{a \in A} Q_{\theta_2}(s_{t+1}, a)$ in Equation 8 to update $\beta$, in this paper we employ a simplified output model, which is illustrated in the right hand side of Figure 2. In this model, a set of state variables and an action value (e.g., -0.5 for action $a_0$ and 0.5 for action $a_1$) is given as an input and its corresponding Q-value is an output, which is scalar (i.e., $m = 1$). Thus, $(r_t + (1 - d_t)) \gamma \max_{a \in A} Q_{\theta_2}(s_{t+1}, a)$ can be directly used as a teacher data when updating $\beta$ in the simplified output model (lines 20 and 23).

c) Q-Value Clipping: OS-ELM Q-Network tends to be unstable especially when unseen inputs are fed to the network, and its output values become anomaly in such cases. Such outliers hinder the reinforcement learning, because these values are significantly large and exceed a range of normal reward values. In a typical setting for the reinforcement learning, the maximum reward given by the environment is 1 and the minimum reward is -1. Thus, as shown in lines 20 and 23, output values of OS-ELM Q-Network are clipped so that they are fit into the range of $-1 \leq r_t + (1 - d_t) \gamma \max_{a \in A} Q_{\theta_2}(s_{t+1}, a) \leq 1$. Such a Q-value clipping suppresses outliers and enables a stable reinforcement learning with OS-ELM Q-Network.

d) Random Update: DQNs typically train their neural network parameters in a batch manner and use the experience replay technique to form a batch randomly so that it can mitigate a dependence on a sequence of input data. On the other hand, OS-ELM is a sequential training algorithm that can update its neural network parameters sequentially with a small batch size $k$. As mentioned in Section II-B, the major bottleneck of OS-ELM when implemented for resource-limited FPGA devices is the pseudo inverse matrix operation that may require an SVD or QRD core. In [3], the pseudo inverse matrix operation is eliminated by fixing $k$ to 1 for enabling the neural network based on-device learning. In this paper, to reduce the dependence on a sequence of input data while keeping the small batch size $k$ to 1, we adopt a method of randomly determining whether or not to update the neural network parameters for each step, as shown in lines 22-23. More specifically, depending on a random value $r_2$, the latest experience (i.e., a set of observed values, action $a_t$, and state $s_t$) is sequentially trained so that the batch size is fixed to 1 and the pseudo inverse matrix operation can be eliminated. Assuming that the first initial training is done by software and all the subsequent sequential training is computed by the FPGA device (see Figure 3), we can eliminate the buffer $D$ in the FPGA part. Thus, a combination of the random update with OS-ELM whose batch size is set to 1 [3] can reduce both computational cost and memory usage.

B. OS-ELM Q-Network with Regularization/Normalization

In Q-learning, a neural network is updated based on comparisons of an expected value of the reward with the next state; thus, it can be expected that Q-values in successive states are basically close to recent ones. As mentioned in Section II-E, the spectral regularization and normalization would be effective in reinforcement learning for improving the generalization capability. As discussed below, our recommendation is that the spectral normalization and the L2 regularization are applied to weight parameters $\alpha$ (lines 2-3) and $\beta$ (line 20), respectively.

a) Spectral Normalization for $\beta$: Let us start with the spectral normalization for the weight parameter $\beta$ of OS-ELM Q-Network. Let $\sigma_{\max}(\beta_i)$ is the largest singular value in $\beta$ at step $i$. In this case, $\beta_i$ is divided by $\sigma_{\max}(\beta_i)$ for every feedforward operation. To obtain $\sigma_{\max}(\beta_i)$, SVD is typically applied to $\beta$ for every time, which is a costly operation; so, we do not use the spectral normalization for $\beta$.

2 This approach can mitigate temporal dependency, but the sampling efficiency is reduced compared to the experience replay.
original network without regularization/normalization at step \( i \) function, Lipschitz constant of a neural network is derived as a partial product (Step 2c), Steps 4a, 4b, and 4c are continuously executed as the CPU part (i.e., Cortex-A9 processor). After the initial train on the Z1 platform, while the initial train in Step 2c is executed by a programmable logic part (denoted as FPGA part).

As mentioned in Section II-E, when ReLU or tanh is used as an activation function, Lipschitz constant of a neural network is derived as a partial product of Lipschitz constant of each layer. In this case, Lipschitz constant of the original network without regularization/normalization at step \( i \) is derived as \( \sigma_{\text{max}}(\alpha)\sigma_{\text{max}}(\beta_i) \).

### C. FPGA Implementation

Table I shows the target platform in this paper. Figure 3 shows the design overview of FPGA that consists of CPU and FPGA parts. The predict and sequential train modules in Steps 4a and 4c are designed with Xilinx Vivado and implemented in a programmable logic part (denoted as FPGA part) of PYNQ-Z1 platform, while the initial train in Step 2c is executed by the CPU part (i.e., Cortex-A9 processor). After the initial train (Step 2c), Steps 4a, 4b, and 4c are continuously executed as a main loop. We assume that the interactions with environment (Steps 2b and 4b) are handled by the CPU part.

A low-cost OS-ELM core optimized to batch size 1 was proposed in [3]. In this paper, we redesigned a further optimized core that consists both the predict and sequential train modules (i.e., Steps 4a and 4c) in Verilog HDL, and it is implemented for the same FPGA platform as in [3]. The target FPGA device is Xilinx XC7Z020-1CLG400C. Operating frequency of the programmable logic part is 100MHz, while the CPU is running at 650MHz. Xilinx Vivado v2017.4 is used for the implementation.

As shown in the right hand side of Figure 2, in the OS-ELM Q-Network core, its input size (i.e., the number of input-layer nodes) is equal to the sum of the numbers of state variables and a single action variable, which is five in the CartPole-v0 task. The output size is 1, which is a scalar. The number of hidden-layer nodes is varied from 32 to 256 in the evaluations. The predict and sequential train modules can be implemented with matrix add, mult, and div operations. SVD or QRD core is not needed as in [3]. For these matrix operations, only a single set of add, mult, and div units is implemented in this design for minimizing the area, but a parallel execution using multiple arithmetic units is also possible. We use 32-bit Q20 numbers as a fixed-point number format. Input data, weight parameters \( \alpha \) and \( \beta \), and intermediate computation results are stored in on-chip BRAMs. As mentioned in Section III-A, since the fixed target Q-network technique is used, two sets of neural network parameters \( \theta_1 \) and \( \theta_2 \) are needed. Specifically, the same \( \alpha \) is used for both \( \theta_1 \) and \( \theta_2 \), while different \( \beta \) is needed for \( \theta_1 \) and \( \theta_2 \); thus, two sets of \( \beta \) are implemented in the BRAMs.

Table II shows FPGA resource utilization of the OS-ELM Q-Network core that consists of the predict and sequential train modules when the number of hidden-layer nodes \( N \) is changed from 32 to 256. The largest design with 256 hidden-layer nodes cannot be implemented for PYNQ-Z1 board due to an excessive BRAM requirement. The other designs can be fit into the FPGA part.

### IV. Evaluations

The proposed OS-ELM Q-Network is evaluated in terms of the execution time to complete a reinforcement learning task. Its variants with and without the spectral normalization and L2 regularization techniques are compared to a typical DQN.
A. Evaluation Environment

As a reinforcement learning task in this experiment, we use OpenAI Gym CartPole-v0 that tries to make an inverted pendulum stand longer. As simulation parameters, Cart position, Cart velocity, Pole angle, and Pole velocity at tip are set to -2.4 to 2.4, -∞ to ∞, -41.8° to 41.8°, and -∞ to ∞, respectively. The numbers of state variables and actions are 4 and 2, respectively.

The following designs are compared in terms of (i) training curve and (ii) average execution time to complete the reinforcement learning task. The proposed FPGA design is evaluated in terms of FPGA resource utilization.

1) OS-ELM: The proposed OS-ELM Q-Network with the fixed target Q-network, simplified output model, Q-value clipping, and random update techniques (i.e., the L2 regularization and spectral normalization are not included)

2) OS-ELM-L2: The above OS-ELM with the L2 regularization for β

3) OS-ELM-Lipschitz: The above OS-ELM with the spectral normalization for α

4) OS-ELM-L2-Lipschitz: The above OS-ELM with the spectral normalization for α and L2 regularization for β

5) DQN: A three-layer DQN with the fixed target Q-network and experience replay

6) ELM: The above DQN replaced with ELM with the simplified output model and Q-value clipping

7) FPGA: Same as OS-ELM-L2-Lipschitz but its prediction and sequential training parts are implemented in programmable logic using fixed-point numbers as described in Section III-C

We use ReLU as an activation function. As reinforcement learning parameters, we use the following setting: $\varepsilon_1 = 0.7$, $\varepsilon_2 = 0.5$, and $\text{UPDATE}_{STEP} = 2$. As the L2 regularization parameter, $\delta$ is set to 1 and 0.5 for OS-ELM-L2 and OS-ELM-L2-Lipschitz, respectively. In DQN, $\varepsilon_2$ is not used, the buffer depth for the experience replay is set to 10,000, the batch size is set to 32, Adam [12] is used as an optimizer, the learning rate is set to 0.01, and Huber function [13] is used as a loss function.

B. Training Curve

In this section, algorithm-level evaluations for the reinforcement learning task are conducted. Among the seven designs listed in Section IV-A, ELM, OS-ELM, OS-ELM-L2, OS-ELM-Lipschitz, OS-ELM-L2-Lipschitz, and DQN are compared. They are executed as a software on a 650MHz Cortex-A9 processor of the PYNQ-Z1 board. NumPy version 1.17.2 and Pytorch version 1.3.0 are used for DQN and the ELM/OS-ELM based approaches, respectively. In the designs other than DQN, because their dependence on initial weight parameters are high, unpromising weight parameters are reset when a given condition is met. Specifically, the ELM/OS-ELM based approaches are reset if they did not complete the reinforcement learning task after 300 episodes elapsed.

C. Execution Time to Complete

We evaluate the seven designs in terms of execution times to complete the CartPole-v0 task when the number of hidden-layer nodes $N$ is varied from 32 to 192. X-axis shows the number of episodes elapsed, and Y-axis shows the number of continuous steps that the inverted pendulum is standing (higher is better). There are two line types for each design. Light-colored lines show the number of steps for continuously standing in each episode, and highly-colored lines show the moving average over the last 100 episodes. In these graphs, a representative result is picked up for each design for illustration purpose. Average execution time to complete the task is evaluated in Section IV-C.

The upper left graph shows the results when the number of hidden-layer nodes is 32. In this case, in addition to the baseline DQN, the proposed OS-ELM Q-Networks with regularization and/or normalization techniques (OS-ELM-L2 and OS-ELM-L2-Lipschitz) acquire better actions that can make the inverted pendulum stand longer. In the case of OS-ELM, on the other hand, as the number of episodes increases, the number of steps for continuously standing is getting worse. This result demonstrates that the Q-value clipping technique is not sufficient for the stable reinforcement learning and the regularization and/or normalization techniques are required.

The reinforcement learning is stable in OS-ELM-L2-Lipschitz that uses both the L2 regularization and spectral normalization. In this case, a generalization capability is improved by the L2 regularization and an output range is limited by the spectral normalization. That is, the L2 regularization works directly on weight parameters $\beta$ which are updated at each step. The spectral normalization affects $\alpha$ so that an output value range of OS-ELM-L2-Lipschitz is less than or equal to $\sigma_{max}(\beta)$; thus, outliers due to $\alpha$ values can be suppressed by the spectral normalization. Please note that even if rewards of OS-ELM-L2-Lipschitz are declined once, it can recover the situation and then get right actions.

The upper right graph shows the results when the number of hidden-layer nodes is 64. A similar tendency mentioned above is observed in this case too, but ELM also acquires correct actions, because it is expected that this configuration ($N = 64$) is best suited for ELM.

The lower two graphs show the results when the numbers of hidden-layer nodes are 128 and 192. These results are similar. Only DQN and the proposed OS-ELM-L2-Lipschitz can acquire correct actions. OS-ELM-L2 and OS-ELM-Lipschitz fail to learn correct actions, indicating that using either the L2 regularization or the spectral normalization is not sufficient. In summary, OS-ELM-L2-Lipschitz can avoid the overfitting situation and acquire correct actions thanks to the constraints on both $\alpha$ and $\beta$.
DQN, and FPGA. DQN is separated in the graph since its execution time is quite large compared to the others. Table III shows detailed breakdown of the proposed FPGA design.

In these graphs, each bar shows the execution time breakdown of each operation: train_seq, predict_seq, train_init, predict_init, train_DQN, predict_1, and predict_32.

- In the OS-ELM based approaches except for FPGA, train_init and train_seq indicate their initial training and sequential training, respectively. predict_init and predict_seq are their predictions before and after their initial training is completed, respectively. All these operations are done by the CPU part.

- In the proposed FPGA, before the initial training, train_init and predict_init (Steps 2a and 2c) are executed by the CPU part. After the initial training, train_seq and predict_seq (Steps 4a and 4c) are done by the FPGA part. PS and PL parts are connected via AXI bus and DMA transfer is used for their communication though not fully implemented in our design. We assume data transfer latency between the CPU and FPGA parts is 1 cycle per float32. This is an optimistic assumption, but we use this value for simplicity because it varies depending on an underlying hardware platform (e.g., DMA performance).

- In the baseline DQN, train_DQN is its training. predict_1 and predict_32 indicate its predictions when the batch sizes are 1 and 32, respectively. More specifically, predict_1 and predict_32 are called from Determine and Update states, respectively. All the operations are done by the CPU part.

Execution time for interactions with a given environment (Steps 2b and 4b) is not considered in this evaluation. train_init and predict_init exist but are negligible.

The breakdown of each operation is computed by (the number of executions of the operation) / (execution time of the single operation). train_seq is dominant compared to train_init because train_init is executed only once for each episode. These execution times are averaged over 150 trials.

When the number of hidden-layer nodes is 32, OS-ELM-L2, OS-ELM-L2-Lipschitz, DQN, and FPGA can acquire correct actions. Their execution times are 132.27sec, 55.02sec, 3232.54sec, and 9.39sec, respectively. When the number of hidden-layer nodes is 64, OS-ELM-L2, OS-ELM-L2-Lipschitz, DQN, and FPGA can acquire correct actions. Their executions times are 647.56sec, 74.20sec, 2208.90sec, and 24.71sec, respectively. The execution times of OS-ELM-L2, OS-ELM-L2-Lipschitz, and the proposed FPGA are increased compared to their previous result having 32 hidden-layer nodes because of a larger matrix size. In this case, OS-ELM-L2, OS-ELM-L2-Lipschitz, and the proposed FPGA are faster than DQN by 3.41x, 29.77x, and 89.40x, respectively.

![Fig. 4. Training curve (light-colored lines: # of steps for continuously standing in each episode; highly-colored lines: moving average over last 100 episodes)](image)

**Fig. 4.** Training curve (light-colored lines: # of steps for continuously standing in each episode; highly-colored lines: moving average over last 100 episodes)

| \(\hat{N}\) | train_seq | predict_seq | train_init | predict_init | Total |
|-----|------------|-------------|------------|--------------|-------|
| 32  | 7.847      | 1.466       | 0.023      | 0.053        | 9.389 |
| 64  | 22.458     | 2.135       | 0.047      | 0.067        | 24.707|
| 128 | 84.038     | 4.036       | 0.245      | 0.166        | 88.484|
| 192 | 218.258    | 7.005       | 0.685      | 0.281        | 226.230|

**TABLE III**

Execution time to complete (breakdown of FPGA) [sec]
These results demonstrate that FPGA is the fastest followed by OS-ELM-L2-Lipschitz and DQN, because update formula of the OS-ELM based approaches is simple as shown in Equations 5 and 7. Although FPGA and OS-ELM-L2-Lipschitz use the same algorithm, FPGA is faster, because train_seq and predict_seq are accelerated by dedicated circuits, as shown in Figure 3. Regarding the performance bottleneck, the OS-ELM based approaches spend most of time for train_seq, while DQN spends a certain time for train_DQN, predict_1, and predict_32. As mentioned above, the execution times tend to increase as the number of hidden-layer nodes is increased except for DQN. This is because the size of matrix products is denoted as $R^{N \times N}$, and the computation cost increases rapidly as the number of hidden-layer nodes is increased. Such matrix products can be accelerated efficiently by dedicated logic; thus, the proposed FPGA design is advantageous for the on-device reinforcement learning on resource-limited edge devices.

V. SUMMARY

To solve reinforcement learning tasks on resource-limited edge devices, in this paper, we proposed OS-ELM Q-Network as a lightweight reinforcement learning algorithm that do not rely on a backpropagation based iterative optimization. More specifically, the following techniques were proposed for OS-ELM Q-Network: (1) simplified output model, (2) Q-value clipping, (3) random update, and (4) combination of the spectral normalization for $\alpha$ and L2 regularization for $\beta$. Especially, thanks to (4), the Lipschitz constant of OS-ELM can be suppressed under $\sigma_{\max}(\beta)$ and further controlled by adjusting the parameter $\delta$.

OS-ELM Q-Network with all the above techniques was proposed OS-ELM-L2-Lipschitz and its FPGA implementation complete a CartPole-v0 task 29.77x and 89.40x faster than a conventional DQN-based approach when the number of hidden-layer nodes is 64. Also, they are robust against the number of hidden-layer nodes thanks to (4).

Acknowledgements This work was partially supported by JST CREST Grant Number JPMJCR20F2, Japan.

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” arXiv:1312.5602, Dec 2013.
[2] L.-J. Lin, “Reinforcement Learning for Robots Using Neural Networks,” Ph.D. dissertation, Carnegie Mellon University, USA, Jan 1993.
[3] M. Tsukada, M. Kondo, and H. Matsutani, “A Neural Network-Based On-Device Learning Anomaly Detector for Edge Devices,” IEEE Transactions on Computers, vol. 69, no. 7, pp. 1027–1044, Jul 2020.
[4] N.-Y. Liang, G.-B. Huang, P. Saratchandran, and N. Sundararajan, “A Fast and Accurate Online Sequential Learning Algorithm for Feedforward Networks,” IEEE Transactions on Neural Networks, vol. 17, no. 6, pp. 1411–1423, Nov 2006.
[5] K. Hornik, M. Stinchcombe, and H. White, “Multilayer Feedforward Networks are Universal Approximators,” Neural Networks, vol. 2, no. 5, pp. 359 – 366, Jul 1989.
[6] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, “Spectral Normalization for Generative Adversarial Networks,” in Proceedings of the International Conference on Learning Representations (ICLR’18), Feb 2018.
[7] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, “Extreme Learning Machine: A New Learning Scheme of Feedforward Neural Networks,” in Proceedings of the International Joint Conference on Neural Networks (IJCNN’04), Jul 2004, pp. 985–990.
[8] H. T. Hynh and Y. Won, “Regularized Online Sequential Learning Algorithm for Single-Hidden Layer Feedforward Neural Networks,” Pattern Recognition Letters, vol. 32, no. 14, pp. 1930 – 1935, Oct 2011.
[9] V. Mnih et al., “Human-Level Control through Deep Reinforcement Learning,” Nature, vol. 518, no. 7540, pp. 529–533, Feb 2015.
[10] J. Achiam, “Spinning Up in Deep Reinforcement Learning,” https://github.com/openai/spinningup, 2018.
[11] Y. Yoshida and T. Miyato, “Spectral Norm Regularization for Improving the Generalizability of Deep Learning,” arXiv:1705.10941, May 2017.
[12] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” in Proceedings of the International Conference on Learning Representations (ICLR’15), May 2015.
[13] P. J. Huber, “Robust Estimation of a Location Parameter,” Annals of Mathematical Statistics, vol. 35, no. 1, pp. 73–101, Mar 1964.