An FPGA-Based On-Device Reinforcement Learning Approach using Online Sequential Learning

Hirohisa Watanabe*, Mineto Tsukada*, Hiroki Matsutani*
*Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan 223-8522
Email: {watanabe,tsukada,matutani}@arc.ics.keio.ac.jp

***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, $\tilde{N}$, and m, respectively.

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

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

where G is an activation function, $\boldsymbol{\alpha} \in \mathbb{R}^{n \times \tilde{N}}$ is an input weight matrix between input and hidden layers, $\boldsymbol{\beta} \in \mathbb{R}^{\tilde{N} \times m}$ is an output weight matrix between hidden and output layers, and $\boldsymbol{b} \in \mathbb{R}^{\tilde{N}}$ is a bias vector of the hidden layer.

Assuming this neural network approximates an m-dimensional target chunk (i.e., teacher data) $\boldsymbol{t} \in \mathbb{R}^{k \times m}$ with zero error, the following equation is satisfied.

$$G(\boldsymbol{x} \cdot \boldsymbol{\alpha} + \boldsymbol{b})\boldsymbol{\beta} = \boldsymbol{t} \tag{2}$$

Here, the hidden layer matrix is defined as $\boldsymbol{H} \equiv G(\boldsymbol{x} \cdot \boldsymbol{\alpha} + \boldsymbol{b})$. The optimal output weight matrix $\hat{\boldsymbol{\beta}}$ is computed as follows.

$$\hat{\boldsymbol{\beta}} = \boldsymbol{H}^{\dagger}\boldsymbol{t}, \tag{3}$$

where $\boldsymbol{H}^{\dagger}$ is a pseudo inverse matrix of $\boldsymbol{H}$, which can be computed with matrix decomposition algorithms, such as SVD and QRD (QR Decomposition).

In ELM algorithm, the input weight matrix $\boldsymbol{\alpha}$ is initialized with random values and not changed thereafter. The optimization is thus performed only for the output weight matrix $\boldsymbol{\beta}$; thus, it is quite simple compared with backpropagation based neural networks that optimize both $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. In addition, the training algorithm of ELM is not iterative; it analytically computes the optimal weight matrix $\boldsymbol{\beta}$ for a given input chunk in one shot, as shown in Equation 3. That is, it can always obtain the optimal $\boldsymbol{\beta}$ 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 $\{\boldsymbol{x}_i \in \mathbb{R}^{k_i \times n}, \boldsymbol{t}_i \in \mathbb{R}^{k_i \times m}\}$ with batch size k_i is given, we need to compute an output weight matrix $\boldsymbol{\beta}_i$ that can minimize the following error.

$$\left(\begin{bmatrix} \boldsymbol{H}_0 \\ \vdots \\ \boldsymbol{H}_i \end{bmatrix} \boldsymbol{\beta}_i - \begin{bmatrix} \boldsymbol{t}_0 \\ \vdots \\ \boldsymbol{t}_i \end{bmatrix} \right)^2, \tag{4}$$

where $\boldsymbol{H}_i$ is defined as $\boldsymbol{H}_i \equiv G(\boldsymbol{x}_i \cdot \boldsymbol{\alpha} + \boldsymbol{b})$.

Assuming $\boldsymbol{P_i} \equiv \left(\begin{bmatrix} \boldsymbol{H_0} \\ \vdots \\ \boldsymbol{H_i} \end{bmatrix}^{\top} \begin{bmatrix} \boldsymbol{H_0} \\ \vdots \\ \boldsymbol{H_i} \end{bmatrix} \right)^{-1}$ $(i \geq 0)$, the optimal output weight matrix is computed as follows.

$$\begin{aligned} \boldsymbol{P}_i &= \boldsymbol{P}_{i-1} - \boldsymbol{P}_{i-1}\boldsymbol{H}_i^{\top}\left(\boldsymbol{I} + \boldsymbol{H}_i\boldsymbol{P}_{i-1}\boldsymbol{H}_i^{\top}\right)^{-1}\boldsymbol{H}_i\boldsymbol{P}_{i-1} \\ \boldsymbol{\beta}_i &= \boldsymbol{\beta}_{i-1} + \boldsymbol{P}_i\boldsymbol{H}_i^{\top}\left(\boldsymbol{t}_i - \boldsymbol{H}_i\boldsymbol{\beta}_{i-1}\right) \end{aligned} \tag{5}$$

In particular, the initial values $\boldsymbol{P}_0$ and $\boldsymbol{\beta}_0$ are precomputed as follows. This computation is called initial training.

$$\begin{aligned} \boldsymbol{P}_0 &= \left(\boldsymbol{H}_0^{\top}\boldsymbol{H}_0\right)^{-1} \\ \boldsymbol{\beta}_0 &= \boldsymbol{P}_0\boldsymbol{H}_0^{\top}\boldsymbol{t}_0 \end{aligned} \tag{6}$$

As shown in Equation 5, the output weight matrix $\boldsymbol{\beta}_i$ and its intermediate result $\boldsymbol{P}_i$ are computed from the previous training results $\boldsymbol{\beta}_{i-1}$ and $\boldsymbol{P}_{i-1}$. 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 $\left(\boldsymbol{I} + \boldsymbol{H_i}\boldsymbol{P_{i-1}}\boldsymbol{H_i}^{\top}\right)^{-1}$ 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 $\boldsymbol{\beta}$ 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 $\boldsymbol{P}_0$ and $\boldsymbol{\beta}_0$ is changed as follows.

$$\begin{aligned} \boldsymbol{P}_0 &= \left(\boldsymbol{H}_0^{\top}\boldsymbol{H}_0 + \delta\boldsymbol{I}\right)^{-1} \\ \boldsymbol{\beta}_0 &= \boldsymbol{P}_0\boldsymbol{H}_0^{\top}\boldsymbol{t}_0, \end{aligned} \tag{7}$$

where δ 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 θ_1. In this case, θ_1 is trained so that the value $Q_{\theta_1}(s_t, a_t)$ can be predicted accurately by the neural network. However, if θ_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 θ_2 is used for stabilizing the Q-learning, in addition to that with θ_1. More specifically, θ_2 is used but fixed for a while, and it is updated with θ_1 at a predefined interval.

In DQNs, an optimization target is computed as follows.

$$f(r_t, s_{t+1}, d_t) = r_t + (1 - d_t)\gamma \max_{a \in A} Q_{\theta_2}(s_{t+1}, a), \tag{8}$$

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, θ_2 is periodically updated with θ_1 by using the fixed target Q-network technique. Specifically, the loss value for θ_1 is denoted as follows [10].

$$L(\theta_1) = \mathop{\mathbb{E}}_{(s_t, a_t, r_t, s_{t+1}, d_t) \sim D} \left[\left(Q_{\theta_1}(s_t, a_t) - f(r_t, s_{t+1}, d_t) \right)^2 \right], \tag{9}$$

[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.

$$\forall x_1, x_2,\ \|f(x_1) - f(x_2)\| \leq K\|x_1 - x_2\|, \tag{10}$$

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., ≤ 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, a well-known extension of the 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 ≤ 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

```
1  Initialize parameters θ1 = {α0, β0} using random
     values ℝ ∈ [0, 1]
2  σmax(α0) ← SVD(α0)
3  α0 ← α0/σmax(α0) // Initialize α0
4  Initialize parameters θ2 as θ2 ← θ1
5  Initialize buffer D
6  Initialize global step t
7  for episode ∈ 1... do
8      for step ∈ 1... do
9          t ← t + 1
           // Determine
10         if random value r1 < ε1 then
11             at ← arg max_{a∈A} Qθ1(st, a)
12         else
13             at ← random action value
           // Observe
14         Observe (st+1, rt, dt) from environment
15         if dt == 1 then
16             Break
           // Store
17         Store (st, at, rt, st+1, dt) in buffer D
           // Update
18         if t == Ñ then
19             Retrieve ∀i ∈ [1, Ñ], (si, ai, ri, si+1, di)
                from buffer D
20             Update ∀i ∈ [1, Ñ], Qθ1(si, ai) to clip(−1,
                   ri + (1 − di)γ max_{a∈A} Qθ2(si+1, a), 1)
                   // Initialize βt
21         else if t > Ñ then
22             if random value r2 < ε2 then
23                 Update Qθ1(st, at) to clip(−1, rt + (1 −
                       dt)γ max_{a∈A} Qθ2(st+1, a), 1)
                       // Update βt
24     if episode % UPDATE_STEP == 0 then
25         θ2 ← θ1
```

- In Update state (lines 18-23), $\boldsymbol{\beta}$ 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 θ_1 and θ_2 are initialized in lines 1 and 4. θ_1 is updated more frequently (lines 20 and 23) and θ_2 is synchronized with θ_1 at a certain interval

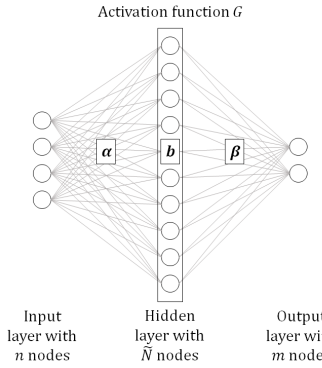


Fig. 1. Extreme Learning Machine

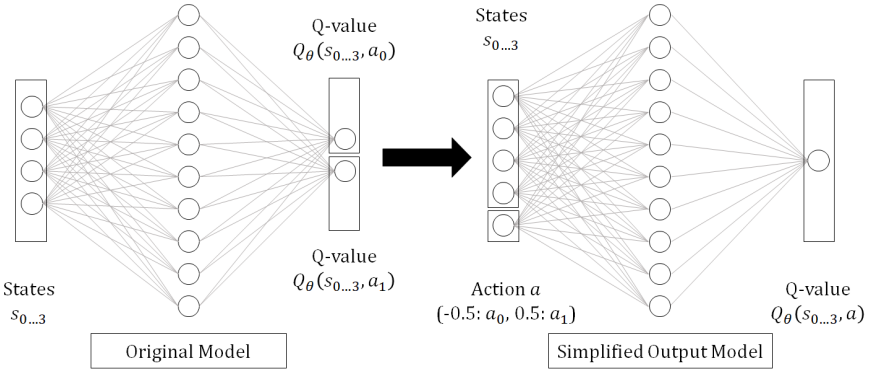


Fig. 2. Simplified output model (numbers of state variables and actions are 4 and 2 in this example)

(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 $\boldsymbol{t} \in \mathbb{R}^m$ is required to update $\boldsymbol{\beta}$ 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 $\boldsymbol{\beta}$, 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 $\boldsymbol{\beta}$ 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 [2].

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 $\boldsymbol{\alpha}$ (lines 2-3) and $\boldsymbol{\beta}$ (line 20), respectively.

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

[2]This approach can mitigate temporal dependency, but the sampling efficiency is reduced compared to the experience replay.

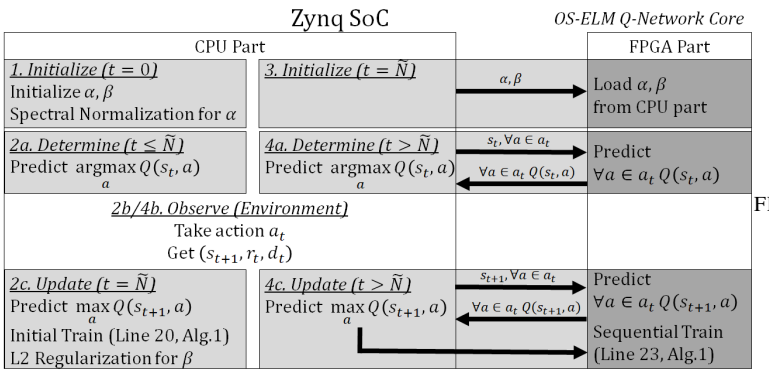


Fig. 3. On-device reinforcement learning on PYNQ-Z1 platform (Steps 4a and 4c are OS-ELM Q-Network core implemented in FPGA part)

b) L2 Regularization for $\boldsymbol{\beta}$: In this paper, we thus use the L2 regularization for $\boldsymbol{\beta}$ as an alternative to the spectral normalization for $\boldsymbol{\beta}$. In this case, the initial training of Equation 6, which is called from line 20 of Algorithm 1, is replaced with Equation 7. This approach is validated below. Assuming $\boldsymbol{A}$ is a general matrix, the following relation is satisfied.

$$\|\boldsymbol{A}\|_2^2 = \sigma_{max}^2(\boldsymbol{A}) \leq \|\boldsymbol{A}\|_F^2 = \sum_i \sigma_i^2(\boldsymbol{A}), \qquad (11)$$

where $\|\cdot\|_2$ and $\|\cdot\|_F$ denote a spectral norm and an L2 norm, respectively. As shown in Relation 11, the L2 norm introduces a stronger constraint than the spectral norm [11]. This means that the L2 regularization for $\boldsymbol{\beta}_i$ of OS-ELM can introduce the same or stronger effect of the spectral regularization.

c) Spectral Normalization for $\boldsymbol{\alpha}$: Different from $\boldsymbol{\beta}$, weight parameter $\boldsymbol{\alpha}$ of OS-ELM is randomly generated at the initialization step and not changed at runtime. Since the initial values of $\boldsymbol{\alpha}$ can be computed at offline (e.g., by software), the spectral normalization can be easily applied to $\boldsymbol{\alpha}$, as shown in lines 2-3 of Algorithm 1. By applying the spectral normalization for $\boldsymbol{\alpha}$, the Lipschitz constant depending on $\boldsymbol{\alpha}$ is suppressed within 1 or less; thus, the Lipschitz constant of OS-ELM is $\sigma_{max}(\boldsymbol{\beta}_i)$ or less. More specifically, it depends on $\boldsymbol{\beta}_i$ and the L2 regularization parameter δ, which means that the Lipschitz constant can be controlled by these parameters. As a result, by a combination of the spectral normalization for $\boldsymbol{\alpha}$ and the L2 regularization for $\boldsymbol{\beta}$, the Lipschitz constant of OS-ELM can be kept under $\sigma_{max}(\boldsymbol{\beta}_i)$. [3]

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.

[3]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 $\sigma_{max}(\boldsymbol{\alpha})\sigma_{max}(\boldsymbol{\beta}_i)$.

TABLE I
SPECIFICATION OF TARGET PLATFORM

OS	PYNQ Linux based on Ubuntu 18.04
CPU	Cortex-A9 processor (650MHz)
RAM	DDR3 SDRAM (512MB)
FPGA	Zynq XC7Z020-1CLG400C (100MHz)

TABLE II
FPGA RESOURCE UTILIZATION OF OS-ELM Q-NETWORK CORE

$\tilde{N}$	BRAM [%]	DSP [%]	FF [%]	LUT [%]
32	2.86	1.82	1.49	3.52
64	11.43	1.82	2.47	5.00
128	45.71	1.82	4.50	7.93
192	91.43	1.82	6.44	11.01

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 includes 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 $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, 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 θ_1 and θ_2 are needed. Specifically, the same $\boldsymbol{\alpha}$ is used for both θ_1 and θ_2, while different $\boldsymbol{\beta}$ is needed for θ_1 and θ_2; thus, two sets of $\boldsymbol{\beta}$ 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 $\tilde{N}$ 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 device. The BRAM utilization is thus a limiting factor, and those of the other resources are not high.

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, $-\infty$ to ∞, -41.8°to 41.8°, and $-\infty$ 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 $\boldsymbol{\beta}$
3) **OS-ELM-Lipschitz**: The above **OS-ELM** with the spectral normalization for $\boldsymbol{\alpha}$
4) **OS-ELM-L2-Lipschitz**: The above **OS-ELM** with the spectral normalization for $\boldsymbol{\alpha}$ and L2 regularization for $\boldsymbol{\beta}$
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 $UPDATE\_STEP = 2$. As the L2 regularization parameter, δ is set to 1 and 0.5 for **OS-ELM-L2** and **OS-ELM-L2-Lipschitz**, respectively. In **DQN**, ε_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 [4]. 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.

[4]Here, **OS-ELM-L2-Lipschitz** is corresponding to **FPGA**. Their difference is that **FPGA** uses 32-bit Q20 fixed-point numbers, but the negative impact was not significant in this experiment.

Figure 4 illustrates training curves of the six designs when the number of hidden-layer nodes $\tilde{N}$ 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 $\boldsymbol{\beta}$ which are updated at each step. The spectral normalization affects $\boldsymbol{\alpha}$ so that an output value range of **OS-ELM-L2-Lipschitz** is less than or equal to $\sigma_{max}(\boldsymbol{\beta})$; thus, outliers due to $\boldsymbol{\alpha}$ 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 ($\tilde{N} = 64$) 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 $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.

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 $\tilde{N}$ is varied from 32 to 192. In this evaluation, an execution was terminated as "impossible" if it could not complete the task after 50,000 episodes. As a result, **OS-ELM** and **OS-ELM-Lipschitz** could not complete the task in our evaluation. Also, **ELM** was not stable. Figure 5 shows the execution times of **OS-ELM-L2**, **OS-ELM-L2-Lipschitz**,

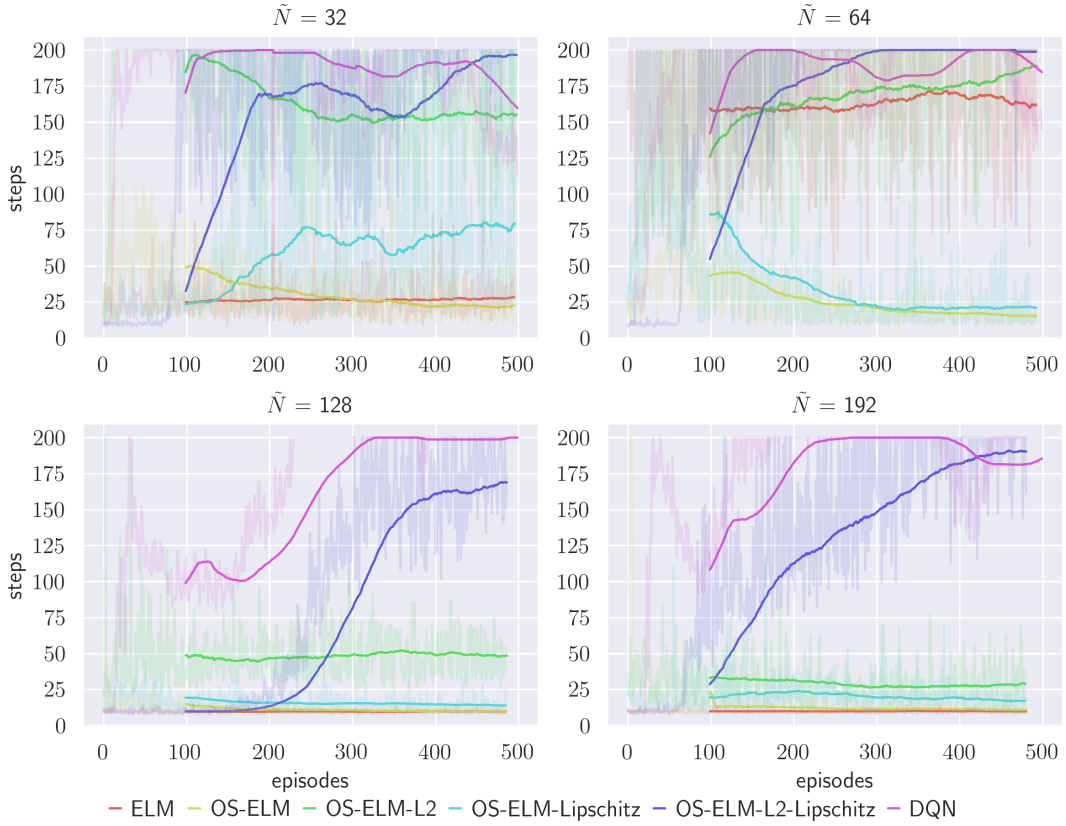


Fig. 4. Training curve (light-colored lines: # of steps for continuously standing in each episode; highly-colored lines: moving average over last 100 episodes)

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.

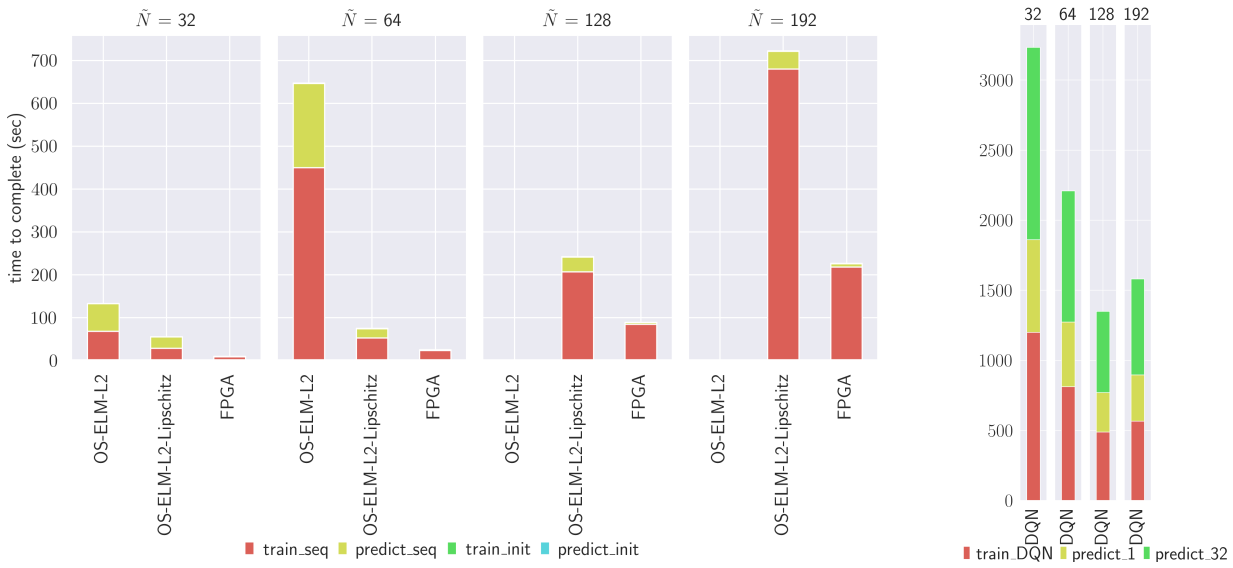
TABLE III
EXECUTION TIME TO COMPLETE (BREAKDOWN OF **FPGA**) [SEC]

$\tilde{N}$	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

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) $\times$ (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 **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.



(a) OS-ELM based approaches (b) **DQN**

Fig. 5. Execution time to complete [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 $\mathbb{R}^{\tilde{N}\times\tilde{N}} \cdot \mathbb{R}^{\tilde{N}\times\tilde{N}}$, 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 $\boldsymbol{\alpha}$ and L2 regularization for $\boldsymbol{\beta}$. Especially, thanks to (4), the Lipschitz constant of OS-ELM can be suppressed under $\sigma_{max}(\boldsymbol{\beta})$ and further controlled by adjusting the parameter δ.

OS-ELM Q-Network with all the above techniques was designed for PYNQ-Z1 board as a low-cost FPGA platform by extending an existing on-device learning core [3]. Prediction and sequential training in most of Determine and Update states (i.e., predict_seq and train_seq) are accelerated by the FPGA part, and the others are executed by the CPU part. The evaluation results using OpenAI Gym demonstrated that the 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. Huynh 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.