Binarized P-Network: Deep Reinforcement Learning of Robot Control from Raw Images on FPGA

Yuki Kadokawa, Yoshihisa Tsurumine, and Takamitsu Matsubara

***Abstract*— This paper explores a Deep Reinforcement Learning (DRL) approach for designing image-based control for edge robots to be implemented on Field Programmable Gate Arrays (FPGAs). Although FPGAs are more power-efficient than CPUs and GPUs, a typical (DRL) method cannot be applied since they are composed of many Logic Blocks (LBs) for high-speed logical operations but low-speed real-number operations. To cope with this problem, we propose a novel DRL algorithm called Binarized P-Network (BPN), which learns image-input control policies using Binarized Convolutional Neural Networks (BCNNs). To alleviate the instability of reinforcement learning caused by a BCNN with low function approximation accuracy, our BPN adopts a robust value update scheme called Conservative Value Iteration, which is tolerant of function approximation errors. We confirmed the BPN's effectiveness through applications to a visual tracking task in simulation and real-robot experiments with FPGA.**

I. Introduction

A Field Programmable Gate Array (FPGA) is an integrated circuit that is designed to be programmable in proprietary optimizations by a customer or a designer after manufacturing. It is often described as *field-programmable*. By exploiting such field-programmable capability, FPGAs are often more power-efficient than CPUs and GPUs, which have a fixed number of available calculators that contain wasteful implementation. FPGAs are drawing much attention for such edge robots as flying and walking robots with limited battery capacity [1]–[4]. With this background, this paper focuses on designing a control for edge robots that can be implemented on FPGAs. We tackle the inability to calculate an image-input controller in real-time and discuss this issue below as a problem of conventional methods.

Deep Reinforcement Learning (DRL) is promising for automatically designing such a controller in a data-driven manner. DRLs can train a Neural Network (NN) to learn value functions for control policies that map from raw image observations to actions for task achievement. Their potential has been demonstrated in various fields, including arcade games and robot control [5, 6]. However, FPGAs' computational characteristics must be addressed to implement NNs learned from DRLs in FPGAs. Although Convolutional Neural Networks (CNNs) that can handle image input are typically used for learning value functions or control policies in DRL, FPGAs are mainly composed of Logic Blocks (LBs) that calculate logical operations at high-speed but real-number operations at low-speed.

This work was supported by JSPS KAKENHI Grant Number JP21H03522. (Corresponding author: Yuki Kadokawa.) All authors are with the Division of Information Science, Graduate School of Science and Technology, Nara Institute of Science and Technology, Japan: { kadokawa.yuki.kv3,tsurumine.yoshihisa, takam-m } @is.naist.jp

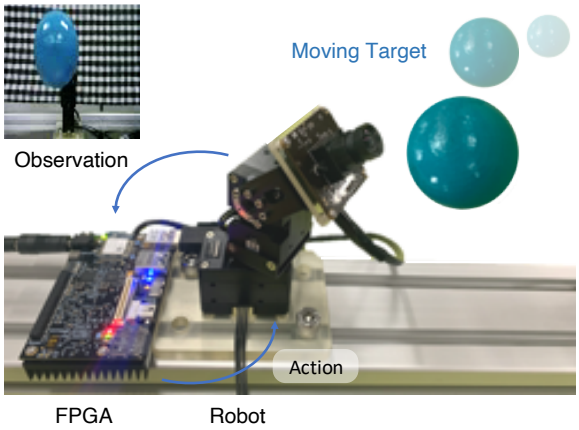


Fig. 1: Execution of visual tracking task using proposed method: In a system consisting only of an edge FPGA and a robot, we implement a real-time control policy for image input learned by our DRL method.

This paper proposes Binarized P-Network (BPN) as a novel DRL algorithm that can learn image-input control policies using a Binarized Neural Network (BNN) [7], which is suitable for FPGA implementation. BNNs are NN models and mainly consist of logical operations. BNN can be implemented and calculated at high speed by explicitly exploiting the FPGAs' LB to calculate the network's logical operations [8, 9]. However, since the approximation accuracy of the continuous function of BCNN is much lower than standard CNNs, accurately learning value functions in DRL is challenging [10]. To alleviate the instability of reinforcement learning caused by using a BCNN with low function approximation accuracy, our BPN adopts a robust value update scheme, Conservative Value Iteration, which is tolerant of function approximation errors [11]. The learning procedure loops two steps: (1) The FPGA executes the policy and collects datasets. (2) The server updates the policy. BPN's effectiveness is validated through an application to an arm-reaching task and a visual tracking task in simulation. Moreover, we applied BPN to an object tracking task in a real-robot experiment (Fig. 1) to learn the control policy in a real-robot environment using an FPGA.

The following are this paper's main contributions: 1) Proposed BPN, a new DRL method using a BCNN that is suitable for FPGA implementation; 2) Achieved real-time image-based robot control using BPNs.

II. Related Works

Repeatedly, FPGAs are mainly composed of LBs that calculate logical operations at high-speed but real-number operations at low-speed. To address this problem, a naive approach uses a remote server with rich computational resources to learn and execute policies. However, even in a stable communication environment between the server and the edge robot, there is a considerable latency in sending and receiving sensor information and control inputs. This latency cannot be ignored in real-time control. In addition, communication data loss may occur. Therefore, controlling an edge robot via a server is problematic from stability and speed of communication. Also, using edge-CPU may be thought helpful for executing policies fast in the edge-robot, but it is not suitable because the CNNs calculation by the edge-CPU is slow. Thus, previous studies have proposed the following two approaches.

Learning on FPGA & Inference on FPGA: Su et al. proposed implementing the entire flow of DRL algorithms on FPGAs, which can be applied as an approach that learns control policies through direct interaction between FPGAs and edge robots [12, 13]. In this approach, since the robot and FPGA can communicate without the network environment, the communication delay's influence is negligible. However, FPGA cannot quickly calculate the learning algorithm and control policies. Thus, such an approach is limited to small-scale NNs and is unavailable for CNNs with image input.

Learning on Server & Inference on FPGA: Shao et al. proposed a simulation environment on a server to learn control policies to offload the learned control policies to FPGA [14, 15]. Unfortunately, this proposal is again limited to small-scale NNs due to the slow calculation speed of the offloading policies in LBs on FPGAs. Learning performance is also likely to be poor due to modeling errors between the simulation and real-robot environments.

Based on the above, to realize a real-time controller with image input for edge robots on FPGAs, a system's server must remotely communicate with the robot to learn control policies, as in Shao et al. Moreover, a novel framework must be considered that can more effectively use LBs so that the learned policies can be computed in real-time on FPGAs. The BPN proposed in this paper addresses this challenge.

III. Preliminaries

A. Reinforcement Learning

Reinforcement learning (RL), which optimizes an agent's actions in an environmental model that follows the Markov Decision Process (MDP), has five components: $(\mathcal{S}, \mathcal{A}, \mathcal{T}, r, \gamma)$. $\mathcal{S}$ is the set of observations that can be obtained from the environment, and $\mathcal{A}$ is the set of selectable actions. $\mathcal{T}^a_{ss'}$ is the probability of transitioning to observation $s' \in \mathcal{S}$ when action $a \in \mathcal{A}$ is chosen in observation $s \in \mathcal{S}$. The reward for making the transition is represented by $r^a_{ss'}$, and $\gamma \in [0, 1)$ is the discount factor. Policy $\pi(a|s)$ is the probability of choosing action a in the case of observation s. State value function V^π is defined as Eq. (1) as the evaluation criterion for policy π at each observation s:

$$V^\pi(s) = \mathbb{E}_{\pi,T}\left[\sum_{t=0}^{\infty} \gamma^t r_{s_t} \middle| s_0 = s\right], \tag{1}$$

where $r_{s_t} = \sum_{\substack{a\in\mathcal{A}\\ s'\in\mathcal{S}}} \pi(a|s_t)\mathcal{T}^a_{s_t s'} r^a_{s_t s'}$. The RL goal is to find optimal policy π^* that satisfies the Bellman equation:

$$V^*(s) = \max_\pi \sum_{\substack{a\in\mathcal{A}\\ s'\in\mathcal{S}}} \pi(a|s)\mathcal{T}^a_{ss'}\left(r^a_{ss'} + \gamma V^*(s')\right), \tag{2}$$

where $V^*(s)$ is the optimal state value function. To evaluate policies based not only on observations s but also actions a, the optimal action value function is defined:

$$Q^*(s,a) = \max_\pi \sum_{s'\in\mathcal{S}} \mathcal{T}^a_{ss'}\left(r^a_{ss'} + \gamma \sum_{a'\in\mathcal{A}} \pi(a'|s') Q^*(s',a')\right), \tag{3}$$

where $Q^*(s)$ is an optimal Q function.

B. Conservative Value Iteration

Conservative Value Iteration (CVI) is an RL method based on a value function that is robust to function approximation errors [11]. CVI uses current policy π and baseline policy $\bar{\pi}$ and adds constraint $i^\pi_{\bar{\pi}}$ to the learning to maintain moderate policy updates. CVI's goal is to find policy π that satisfies the following modified Bellman equations:

$$V^*(s) = \max_\pi \sum_{\substack{a\in\mathcal{A}\\ s'\in\mathcal{S}}} \pi(a|s)\left[\mathcal{T}^a_{ss'}\left(r^a_{ss'} + \gamma V^*(s')\right) + i^\pi_{\bar{\pi}}(s)\right], \tag{4}$$

$$i^\pi_{\bar{\pi}}(s) = \sum_{a\in\mathcal{A}} \pi(a|s)\left[-\frac{1-\alpha}{\beta}\log\pi(a|s) - \frac{\alpha}{\beta}\log\frac{\pi(a|s)}{\bar{\pi}(a|s)}\right], \tag{5}$$

where $\alpha \in [0, 1]$ and $\beta \in (0, \infty)$ are hyperparameters. In contrast to the Q-function, the action preference function, denoted by P, is defined:

$$\begin{aligned} P^\pi(s,a) = \sum_{s'\in\mathcal{S}} \mathcal{T}^a_{ss'}(r^a_{ss'} + \gamma \sum_{a\in\mathcal{A}} \pi(a|s) V^\pi(s',a')) \\ + \frac{\alpha}{\beta}\log\pi(a|s). \end{aligned} \tag{6}$$

To find optimal policy π^* that maximizes Eq. (6), the update rule of action preference P is defined:

$$\begin{aligned} P_{k+1}(s,a) \leftarrow r^a_{ss'} + \gamma(m_\beta P_k)(s') + \mathcal{G}(s,a), \\ \mathcal{G}(s,a) = \alpha\Big(P_k(s,a) - (m_\beta P_k)(s)\Big), \end{aligned} \tag{7}$$

$$(m_\beta P)(s) = \frac{1}{\beta}\log\left(\frac{1}{|\mathcal{A}|}\sum_{a\in A}\exp\left(\beta P(s,a)\right)\right), \tag{8}$$

where $|\mathcal{A}|$ is the number of selectable actions. The policy is given as follows:

$$\pi_k(s,a) = \frac{\exp\left(\beta P_k(s,a)\right)}{\sum_{b\in A}\exp\left(\beta P_k(s,b)\right)}. \tag{9}$$

$\mathcal{G}(s,a)$ in Eq. (7) is the Gap Increasing Operator (GIO) [11] that amplifies the differences between the maximum

value and others. Therefore, it makes the resulting policy for choosing optimal action robust against function approximation errors [11]. We refer to α as the GIO coefficient. When the α is higher, the robustness to the function approximation errors is higher. Also, the β controls learning convergence. When the β is higher, the learning convergence is faster.

When the GIO coefficient is $\alpha = 1$, it is theoretically equivalent to Dynamix Policy Programming (DPP) [16], which has been used in previous studies to learn robot control policies and improved sample efficiency [17]. Moreover, CVI is nearly equivalent to Q-learning when the parameters are set as $\alpha = 0$ and $\beta = \infty$ [11]. The parameters mean that DQN updates the value function in greedy. Thus, the learning performance of DQN becomes degraded when the function approximation accuracy is low since DQN is sensitive to the function approximation errors. It means that CVI with high α and certain β is suitable for learning a policy calculated in low accuracy of function approximation.

C. Binarized Neural Network

This section briefly summarizes Binarized Neural Networks (BNNs), which are neural networks with binary weights and run-time activations. Assuming that the dimensions of the input and output vectors in each layer of the BNN are N and M, hierarchical functions output $\boldsymbol{y} \in \mathbb{R}^M$ from input $\boldsymbol{x} \in \mathbb{R}^N$. Each layer consists of a Fully-Connected Layer (FCL) and an activation function. Assuming that the number of BNN layers is L, the FCL output of the lth layer is $\boldsymbol{o}_l = [o_{l,1}, o_{l,2}, \ldots, o_{l,M}]^{\mathrm{T}} \in \mathbb{R}^M$, the output of the activation function of the lth layer is $\boldsymbol{x}_l = [x_{l,1}, x_{l,2}, \ldots, x_{l,M}]^{\mathrm{T}} \in \mathbb{R}^M$, and the BNN's network parameters of the lth layer are $\boldsymbol{W}_l = [\boldsymbol{W}_{l,1}\boldsymbol{W}_{l,2}\ldots\boldsymbol{W}_{l,M}] \in R^{N\times M}$, $\boldsymbol{W}_{l,m} = [w_{l,m,1}\ w_{l,m,2}\ldots w_{l,m,N}] \in \mathbb{R}^{N\times M}$ set to $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \boldsymbol{W}_2, \ldots, \boldsymbol{W}_L\}$.

As a key feature of BNN, binarized function $\mathrm{Sign} : \mathbb{R} \to \{-1, 1\}$ in Eq. (10) is included in FCL and in the activation functions of each layer:

$$z^b = \mathrm{Sign}(z) = \begin{cases} +1, & \text{if } z \geq 0, \\ -1, & \text{otherwise}, \end{cases} \tag{10}$$

where z is an arbitrary real number and the value after binarization is denoted as z^b. Assuming that BNN input $\boldsymbol{x}$ and output $\boldsymbol{y}$ are $\boldsymbol{x}_0$ and $\boldsymbol{o}_L$, the operation of each layer is given below:

$$o_{l,m} = \sum_{n=0}^{N} \mathrm{Sign}(w_{l,m,n}) x_{l-1,n}\,, \tag{11a}$$

$$x_{l,m} = \mathrm{Sign}(o_{l,m})\,. \tag{11b}$$

In Eq. (11a), since each value of parameter $\boldsymbol{W}_{l,m,n}$ can be converted to 1 and 0, each weight can represented by a single bit. Therefore, by storing each element of parameter $\boldsymbol{W}_l$ in 1 bit, the model size can be compressed to 1/32 compared to single-precision, floating-point numbers.

Moreover, by binarizing input $\boldsymbol{x}_{l-1}$ to each layer using the activation function in Eq. (11b), the multiplication-and-accumulation (MAC) operations in Eq. (11a) can be replaced with the *XNOR* and *popcount* operations:

$$o_{l,m} = \mathrm{Popcount}(\mathrm{XNOR}(w^b_{l,m,n}, x^b_{l-1,n})), \tag{12}$$

where weights $w^b_{l,m,n} \in \{-1, 1\}$ and input $x^b_{l-1,n} \in \{-1, 1\}$ are converted to $w^b_{l,m,n} \in \{0, 1\}$ or $x^b_{l-1,n} \in \{0, 1\}$. The XNOR operation corresponds to the product of $w^b_{l,m,n}$ and $x^b_{l-1,n}$, and the popcount operation corresponds to counting the number of output bits from the XNOR operation [8]. Therefore, FPGAs can calculate BNN at high speed since they can calculate such logical operations in LBs. Binarized Convolutional Neural Networks (BCNNs) [7] can also be constructed by simply including convolutional layers in the same way.

BNN updates parameter $\boldsymbol{\theta}$ by gradient descent, but the learning method is different from a standard NN. Updating NN parameters consists of forward- and back-propagation. In forward-propagation, BNN-output $\boldsymbol{y}$ is obtained using binarized weights $\boldsymbol{\theta}^b$. On the other hand, in back-propagation, $\boldsymbol{\theta}$ is updated by back-propagation using BNN-output $\boldsymbol{y}$ with non-binarized network parameter $\boldsymbol{\theta}$ to avoid unstable learning due to discontinuity in $\boldsymbol{\theta}^b$ [7].

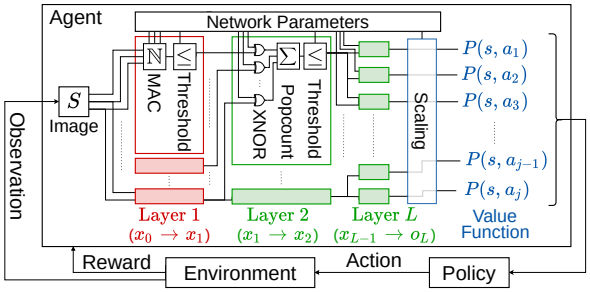


Fig. 2: BPN's network architecture: BPN's network calculates action preference $P(s, a)$ from an image as observation s in three steps: (**1**) First convolution layer extracts features from observation s in MAC operations and threshold activation function. (**2**) Second and subsequent layers calculate the XNOR operations, the popcount operations, and the threshold activation functions. (**3**) Last layer's outputs are scaled by λ and used as action preference $P(s, a)$.

IV. Binarized P-Network

A. Network Architecture

BPN's network architecture is shown in Fig. 2. To alleviate the instability of reinforcement learning caused by a BCNN with low function approximation accuracy, our BPN adopts a robust value update scheme: Conservative Value Iteration. Thus, based on $P(s, a)$ represented by BCNN, action a is executed according to policy π in Eq. (9).

The network calculation to obtain $P(s, a)$ has three steps. (1) The first layer, which extracts features from an image to output $\boldsymbol{x}_1$, consists of MAC operators and a threshold activation function (Eq. (13)). MAC operators output $\boldsymbol{o}_1$, and the threshold activation function outputs $\boldsymbol{x}_1$. (2) In the second and subsequent layers, output $\boldsymbol{x}_2, \boldsymbol{x}_3, \ldots, , \boldsymbol{x}_{L-1}, \boldsymbol{o}_L$ is obtained by XNOR and popcount operations which output $\boldsymbol{o}_l$

Algorithm 1: Binarized P-Network

```
# Set parameters described in Table I
# Initialize network weights θ, θ⁻, replay memory D
Function DataCollect(θ, D):
    for e = 1, 2, ..., E do
        for t = 1, 2, ..., T do
            # Take action a_t with softmax policy
              Eq. (9) based on P(s_t, A; θ^b)
            # Receive observation s_{t+1}, reward r^{a_t}_{s_t s_{t+1}}
            # Push {(s_t, a_t, r^{a_t}_{s_t s_{t+1}}, s_{t+1})} to D
    # return D
Function PolicyUpdate(θ, D):
    # Set target network θ⁻ = θ
    for c = 1, 2, ..., C do
        # Set D' is index-shuffle local memory D
        for k = 1, 2, ..., round( |D|/B ) do
            # Sample the minibatch of transition
              D'[B×(k−1) : B×k]
            # Calculate loss on Eq. (15) and update θ
    # return θ
for i = 1, 2, ..., I do
    # D = DataCollect(θ, D)
    # θ = PolicyUpdate(θ, D)
```

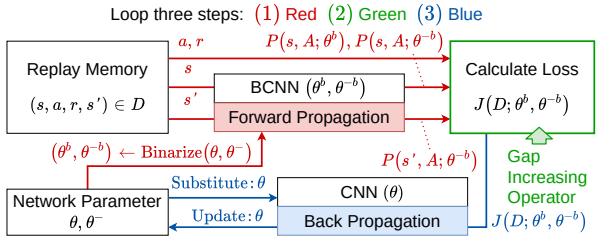


Fig. 3: Policy updating process of BPN: (1) Calculate action preferences, (2) Calculate loss function, (3) update network parameters.

and the threshold activation function which outputs $\boldsymbol{x}_l$. First half layers $\boldsymbol{o}_1, \boldsymbol{o}_2, \ldots, \boldsymbol{o}_c$ are obtained by the convolutional layers [8], and the second half layers $\boldsymbol{o}_{c+1}, \boldsymbol{o}_{c+1}, \ldots, \boldsymbol{o}_L$ are obtained by the FCLs (Eq. (12)). (3) In the last layer, FCL's output $\boldsymbol{o}_L$ is scaled by λ and output as action preference $P(s, a)$.

B. Network Details

To approximate action preference $P(s, a)$ with BCNN, which has low function approximation accuracy, we added the following three features to the network.

a) Unbinarized observation: To avoid reducing the features from observation s, BPN doesn't binarize the first layer's input s. Thus, only the weights are binarized in the input convolutional layer, and the input is kept as pixel values.

b) Batch normalization: BCNNs are prone to learning instability due to the binarization of weights and outputs in each layer. Thus, we added batch normalization to every BPN layer to stabilize the learning against dynamic changes in the target value. This dynamic changes of target values is unique to RL. Note that the calculation of batch normalization is slow on FPGAs because it has many floating-point computations. To speed up the calculation, the batch normalization and the activation function of Eq. (11b) are combined and converted into a threshold activation function:

$$x_{l,m} = \begin{cases} +1, & \text{if } o_{l,m} \geq \tau_{l,m}, \\ -1, & \text{otherwise}, \end{cases} \quad (13)$$

where $\boldsymbol{\tau}_l = [\tau_{l,1}, \ldots, \tau_{l,M}]^{\mathrm{T}} \in \mathbb{N}^M$ is the lth layer threshold [18].

c) Scaling network output: Eq. (11a) shows that the problem of using BCNN as a function approximator is that FCL's outputs are limited to range $[-N, N]$. BPN resolves this limitation by introducing scaling parameter λ to the last layer FCL's output $o_{L,m}$:

$$P(s, a_m) = \lambda \sum_{n=0}^{N} \text{Sign}(w_{L,m,n}) x_{L-1,n}. \quad (14)$$

Since it is difficult to set λ by hand, it is learned from data.

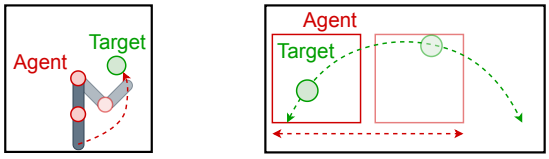
TABLE I: Learning parameters of BPN in two DOF manipulator reaching tasks.

Para.	Meaning	Value
α	GIO coefficient of CVI	0.95
β	Learning speed coefficient of CVI	1
γ	Discount factor of RL	0.99
C	Number of epochs	50
B	Minibatch size	32
I	Number of iterations	50
E	Number of episodes per iteration	10
T	Number of steps per episode	20
U	Number of iteration datasets in $\mathcal{D}$	3

C. Learning Process

The learning process consists of data collection and policy update steps. BPN uses the target network and the replay memory, as in the DQN method [19]. A target network technique uses two network parameters: P-network parameters $\boldsymbol{\theta}$ and target network parameters $\boldsymbol{\theta}^-$. $\boldsymbol{\theta}^-$ decides the actions during the data collection step. $\boldsymbol{\theta}$ is updated in the policy update step. $\boldsymbol{\theta}^-$ is updated to $\boldsymbol{\theta}$ at regular intervals to stabilize the learning and moderating the frequency of the network parameter updates. The details of the BPN learning process are shown below and summarized in Algorithm 1.

a) Data collection: First, target network parameters $\boldsymbol{\theta}^-$ are copied from P-network parameter $\boldsymbol{\theta}$. Then to calculate the action preferences, all the parameters in $\boldsymbol{\theta}^-$ are binarized to $\boldsymbol{\theta}^{-b}$ based on Eq. (10). In this paper, binarized network parameters $\boldsymbol{\theta}$ are denoted as $\boldsymbol{\theta}^b$. The training datasets are then sampled based on the current policy with $\boldsymbol{\theta}^{-b}$.



(a) Reaching (b) Tracking

Fig. 4: Simulation tasks: (a) reaching and (b) tracking.

In the data collection step, the target network first takes observation s as input and outputs action preference $P(s, \mathcal{A}; \boldsymbol{\theta}^{-b})$. Then, based on $P(s, \mathcal{A}; \boldsymbol{\theta}^{-b})$, the agent executes action a based on the softmax function in Eq. (9). The environment transitions and outputs next observation s' and reward r. (s, a, r, s') pairs are added to replay memory $\mathcal{D}$ as a training dataset.

b) Policy update: In the policy update step, the loss function is calculated based on dataset $\mathcal{D}$ and accumulated in the data collection step. Fig. 3 shows how to update P-network parameters $\boldsymbol{\theta}$ in three steps. (1) Sets of minibatches (s, a, r, s') are created from dataset $\mathcal{D}$. Action preferences $P(s, \mathcal{A}; \boldsymbol{\theta}^{b})$, $P(s', \mathcal{A}; \boldsymbol{\theta}^{b})$, $P(s, \mathcal{A}; \boldsymbol{\theta}^{-b})$ are calculated from s, s'. (2) Loss function $J(\mathcal{D}; \boldsymbol{\theta}^{b}, \boldsymbol{\theta}^{b-})$ derived from Eq. (7) is calculated as follow:

$$J(\mathcal{D}; \boldsymbol{\theta}^{b}, \boldsymbol{\theta}^{-b}) = \frac{1}{2}\left[r^{a}_{s,s'} + \gamma(m_{\beta}P)(s'; \boldsymbol{\theta}^{-b}) + \alpha\left(P(s, a; \boldsymbol{\theta}^{-b}) - (m_{\beta}P)(s; \boldsymbol{\theta}^{-b})\right) - P(s, a; \boldsymbol{\theta}^{b})\right]^{2}. \quad (15)$$

(3) Network parameters $\boldsymbol{\theta}$ are updated by back-propagation using a CNN composed of unbinarized network parameters $\boldsymbol{\theta}$, as described in Section III-C.

V. Simulation Experiment

In this section, we evaluated BPN's learning performance in a simulation study conducted with a Geforce RTX2080Ti GPU. As a comparison, we also evaluated the Binarized Q-Network (BQN) performance, which is a modified DQN with binarization for both the weights and outputs of every layer. Note that BQN is different from Binary Q-Network [20], which binarizes only the weights and cannot be implemented in FPGA. As shown in Eq. (14), the accuracy of the BPN output depends on the number of nodes N in the output layer. Thus, we verify that BPN can learn a policy robustly against a variation of function approximation accuracy due to the change in the number of nodes N.

A. Settings

1) Reaching Task: The target task is the 2DOF reaching task in Fig. 4(a). The agent rotates one joint at each step by a fixed angle. The target marker is fixed the entire time. The agent's learning goal is to match the hand coordinates with the target marker. The initial positions of the agent and the target are fixed. Let observation s be a gray-scale image of 84×84 pixels obtained from the entire simulation environment, such as Fig. 4(a). Agent's action a is selected from seven levels of target rotation angles: $[-90, -45, -30, 0, 30, 45, 90]$(degree). The number of selectable actions is $|\mathcal{A}| = 2 \times 7 = 14$. The number of pixels in the horizontal and vertical directions of the image obtained from observation s is defined as the XY coordinates. The robot's coordinates are $(x_{\text{agent}}, y_{\text{agent}})$, and the target's coordinates are $(x_{\text{target}}, y_{\text{target}})$. The reward is defined as $r = -\sqrt{(x_{\text{agent}} - x_{\text{target}})^2 + (y_{\text{agent}} - y_{\text{target}})^2}$. The network structure is consist of five layers, which are Conv(8,4,8), Conv(4,2,16), Conv(3,1,16), FC(N), FC($|\mathcal{A}|$). Conv() means convolutional layer, which parameters are kernels, strides, and channels, respectively. FC() means full-connected layer, which parameter is nodes. The training parameters are described in Table I.

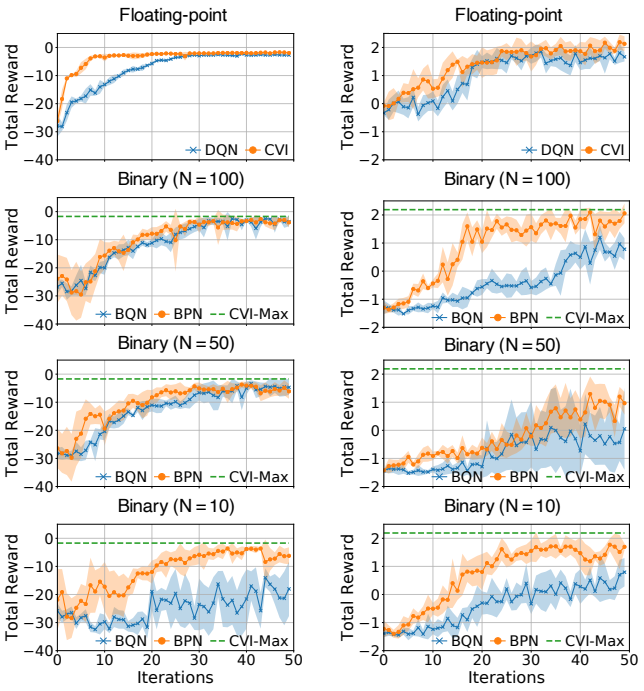
2) Tracking Task: Fig. 4(b) shows the experimental environment. The agent manipulates the red frame and learns that making the target always appears within it. The initial positions of the agent and the target are randomly assigned. The environment is represented by 120×180 pixels in height and width. The agent frame size is 84×84 pixels. To estimate the target's velocity, two consecutive frames are combined and used as observation $s \in \mathbb{R}^{6 \times 84 \times 84}$. The target is a circle with a 12-pixel radius and moves in an arc of a 60-pixel radius. Agent's action a moves the frame horizontally by the specified number of pixels in one step. Action a is selected from $[-8, -4, 2, 0, 2, 4, 8]$(pixel). Reward r is the distance between the center coordinates of the frame and the target. Reward calculation is identical as Section V-A.1. However, if the target moves out of the frame, we treat it as a tracking failure and the end of the episode. The network structure is identical as Section V-A.1. The difference between the training parameters and Table I is $T = 40$.

B. Results

The learning results of reaching task and tracking task are shown in Fig. 5(a), Fig. 5(b), respectively. In all simulation tasks, learning performance, such as, training stability, sample efficiency, maximum total reward, is decreased when the function approximation accuracy is reduced. Compared to BQN, the proposed method, BPN, mitigates the decrease of performance. Compared to the reaching task, BPN's learning performance in the tracking task remains high, although BQN's performance suffers. These results seem reasonable since the tracking task is more difficult than the reaching task because the initial positions of the agent and the target are randomly assigned. Fig. 6 shows the relationship between node number N and GIO coefficient α. We confirmed that the higher GIO coefficient α is, the more robustly the BPN can learn against a decrease in node number N. This result is consistent with the property of the GIO operator, where the higher the GIO coefficient α is, the more robust it is to function approximation errors.

VI. Real-Robot Experiment

This section shows the structure of a DRL system using FPGA and robots to learn control policies. Using the DRL system, we apply BPN and BQN to a real-robot object tracking task and verify the learning performance. We also



(a) Reaching (b) Tracking

Fig. 5: Learning curves of (a) reaching and (b) tracking tasks. CVI-Max indicates the maximum training reward using CVI with the floating-point NNs. Each curve plots mean and variance of total reward per iteration I over five experiments.

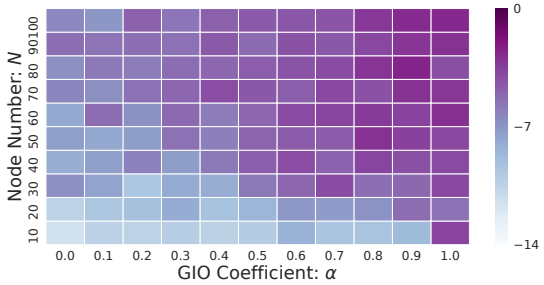


Fig. 6: Relationship between number of nodes N in output layer of a function approximation network and GIO coefficients α. Each value in heat map is maximum value in total reward's learning curve averaged over five experiments.

analyzed the calculation speed in the FPGA implementation to confirm that BPN is suitable for real-time control.

A. Learning System for DRL with FPGAs

The policy updates of BPN were conducted on the GPU server since the BPs of NNs in policy updates require many floating-point operations, and FPGAs do not have enough LBs to calculate them. Hence, the GPU server calculates the policy updates; the FPGA calculates only the policy executions.

Fig. 7 shows a learning system that consists of three steps: (1) FPGA and CPU control the robot to collect datasets for

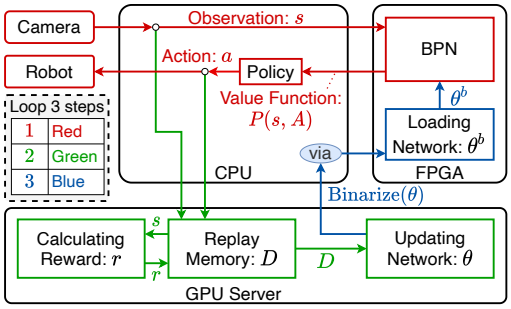


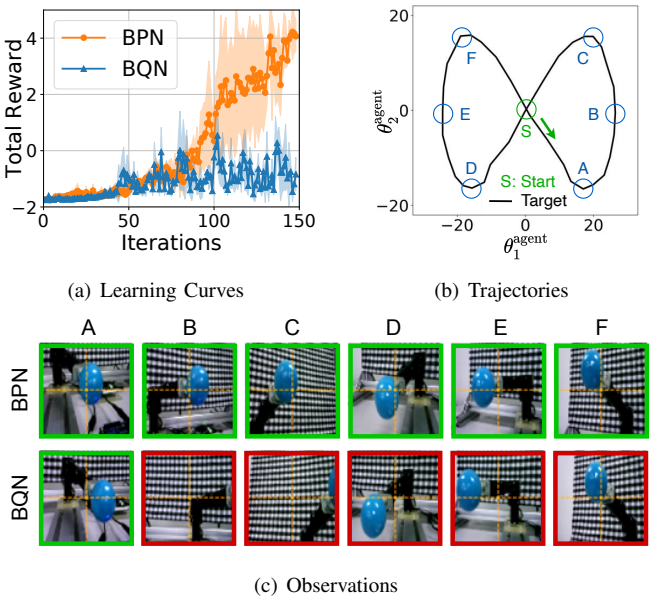
Fig. 7: Implemented learning system: Learning procedure consists of three steps. (**1**) FPGA and CPU control robot to sample data s, a into $\mathcal{D}$. (**2**) $\mathcal{D}$ is used by GPU server to update BPN network parameters $\boldsymbol{\theta}$. (**3**) FPGA updates BPN's network parameter $\boldsymbol{\theta}^b$.

learning. The CPU gets observation s from a camera. The FPGA calculates action preference $P(s, \mathcal{A})$ from s. Then the CPU determines action a based on the policy shown in Eq. (9). The robot executes a. (2) Based on the collected dataset in $\mathcal{D}$, network parameter $\boldsymbol{\theta}$ is updated based on Section IV-C. Reward r calculation is conducted on the GPU server instead of controlling the robot to maintain real-time control. (3) The GPU server binarizes and transfers the network parameter $\boldsymbol{\theta}$ to BRAM of the FPGA via the CPU. The FPGA calculates BPN using network parameter $\boldsymbol{\theta}^b$ loaded from BRAM. In this system, BRAM stores network parameters $\boldsymbol{\theta}^b$, which do not need to be compiled. The compiling time requires more than an hour. The DRL, which compiling network parameters $\boldsymbol{\theta}^b$ for updating $\boldsymbol{\theta}^b$ at each iteration I, has an extremely long learning time due to such a compilation time.

B. Learning Control Policies

1) Settings: The target task is the real-robot object tracking task shown in Fig. 1. The tracking target, a blue marker, moves in a figure-8 pattern. The agent learns to keep the object in the camera frame. Separate robots, consisting of two servo motors (Dynamixel XM430-W350-T), control the agent and the target. The agent's motors are controlled by position-control and wait for converging them to the objective angle before taking the following control. The agent's initial position is fixed, and the target's initial position is randomly assigned within the range where the target is included in the camera frame.

The observation is an RGB image of 84×84 pixels, as shown in Fig. 1(Upper Left). As in Section V-A.2, two consecutive frames are used as observation s. The motor rotation labels are $n = [-4, -2, 0, 2, 4]$(degree), and action a is defined as all the combinations of $(0, n), (n, 0), (-n, n), (n, -n)$ for two motor rotation angles $(\phi_1^{\text{agent}}, \phi_2^{\text{agent}})$. The number of actions is $|\mathcal{A}| = 17$. The trajectory of rotation angle $(\phi_1^{\text{target}}, \phi_2^{\text{target}})$ of the two motors manipulating the target, with angular velocity ω and time step t, is $\phi_1^{\text{target}} = 25 \sin \omega t$ (degree), and $\phi_2^{\text{target}} =$



(a) Learning Curves (b) Trajectories

(c) Observations

Fig. 8: Learning results of real-robot tracking task: (**a**) Learning curves plot mean and variance of total reward per iteration I over five experiments. Entire run time of Algorithm 1 is approximately 12 hours per experiment in $I = 150$. (**b**) Target trajectories of two motors of agent, θ_1^{agent} and θ_2^{agent}, as they complete figure-8 pattern. They are agent's motor trajectories when the target is at the center of the camera frame. (**c**) Observations obtained from learned policy. A to F mean observed timing shown in (**b**). Green and red frames indicate tracking success and failure.

$15 \sin 2\omega t$ (degree). The definitions of reward and episode are identical as in Section V-A.2. The network structure is same as in simulation tasks except for $N = 100$. The learning parameters are different from those of Table I: $\alpha = 0.95$, $\beta = 3$, $I = 150$, $T = 80$.

2) Results: The learning results are shown in Fig. 8. Fig. 8(a) shows that the BQN did not learn progressively, although the BPN did.

Fig. 8(b) shows the target trajectory of the agent for each motor θ_1^{agent} and θ_2^{agent}. Fig. 8(c) is an observation of the learned policy when the target is in the agent's target trajectory A to F in Fig. 8(b). The BPN can track the target to fit in the camera frame using raw images as input in a natural background environment. BPN can also track the target in real-time without being delayed by the target. However, BQN is out of the frame from point B.

The tracking time comparison between BPN and BQN is shown in Table II. BPN can track a target until the task end, which is six times longer than BQN. From Fig. 8(a), the BQN does not learn a suitable policy. Thus, the BQN achieved tracking only for 1.9 seconds up to around the agent's target trajectory A in Fig. 8(b).

C. Calculation Speed of FPGA

We verified that BPN can be implemented in edge FPGAs calculated in real-time by implementing BPN and DQN networks to FPGA. BPN's network (implemented in BCNN) is mainly calculated in logical operations. DQN's network (implemented in CNN) is mainly calculated in floating-point operations. In other words, we confirmed that BPN, implemented in logical operations for FPGA, has better hardware performance because its calculation speed is faster than DQN, which is implemented in floating-point operations. We implemented the networks on an FPGA evaluation board (Avnet Ultra96-V2) and verified the inference time. The network structures are same as Section VI-B.1. Table III, which shows the results of implementing the networks, indicates that DQN cannot be applied to tasks that require fast calculation and that BPN can be applied to real-time control policies.

We verify that the calculation of BPN on edge FPGAs is faster than that on other computers. The BPN has the same structure as Section VI-B.1. The experimental results of the calculation latency are shown in Table IV. The latency is the highest when using a server. The latency is the lowest when using the edge FPGA and is reduced to less than 20% of that of others.

TABLE II: Duration time of successful tracking of Real-Robot Experiment: Control period of agent is 145 ms consisted of 141 ms for sampling two consecutive camera images and controlling robot's motor positions, and 4 ms inference time for NN in the FPGA. The maximum step number is $T = 80$ and corresponds to 11.6 s (145 ms $\times$ 80 steps). Each time is the average of ten trials.

Tracking-Time	**RANDOM**	**BQN**	**BPN**
Second (Percent)	0.9 (8)	1.9 (16)	11.6 (100)

TABLE III: Network inference time: Function approximation networks of BPN and DQN are implemented in FPGA evaluation board (Avnet Ultra96-V2) and within it's resource capacity. Network inference time is evaluated by Xilinx Vivado-HLS.

Calculation-Time per Inference	**DQN**	**BPN**
ms / inference	1003	4

VII. Discussions

Section VI shows how to learn an object tracking task by DRL with a real robot and FPGA. An extension of this work might apply autonomous edge-robot control to exploit FPGAs' power-saving nature. To build a DRL system for such a purpose, we need an environment where FPGA agents can communicate with a server that updates the control policies.

Section VI-A suggests that learning by an autonomous robot requires a communication environment between the FPGA and the server. A bottleneck in implementing a learning algorithm on FPGAs is implementing a large-scale, error

TABLE IV: Latency for getting actions from observations: We measure the time from receiving an image from a camera sensor to a control input to the edge robot. The calculation time is measured in the server as Intel Core i9-9900KS, and the edge CPU and FPGA as ARM Cortex-A53 and Xilinx ZU3EG A484 on Avnet Ultra96-V2, respectively. All calculators get actions by calculating NNs (**Step 2**). The server has additional calculation steps, which are sending an image from the edge robot to the server (**Step 1**) and sending a control input from the server to the edge robot (**Step 3**).

Steps	**Server**	**Edge-CPU**	**Edge-FPGA**
1. Send Images	34 ms	-	-
2. Calculate NNs	2 ms	21 ms	4 ms
3. Send Controls	2 ms	-	-
Total-Time	38 ms	21 ms	4 ms

back-propagation (BP) algorithm, which might be addressed with [21]. In addition, extending the BP implementation method for servers [22, 23] may give some tips for implementing the BP fast in edge FPGAs.

The BPN shown in Section IV is a learning method that assumes a discrete action space. However, continuous actions are often required in robot control tasks. The extension of the proposed method to continuous action space remains our future work. To this end, we could adopt the actor-critic architecture [24]; however, we need to be concerned about how to represent the actor and critic accurately with BCNNs, which have low accuracy in function approximation.

VIII. Conclusion

We proposed a Binarized P-Network as a DRL algorithm suitable for FPGAs. We also implemented the BPN for an object tracking task with a real robot using image inputs and confirmed its effectiveness.

References

[1] P. Gohl, D. Honegger, S. Omari, M. Achtelik, M. Pollefeys, and R. Siegwart, "Omnidirectional visual obstacle detection using embedded FPGA," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015, pp. 3938–3943.

[2] L. Banjanovic-Mehmedovic, A. Mujkic, N. Babic, and J. Secic, "Hexapod robot navigation using FPGA based controller," in *International Conference "New Technologies, Development and Applications"*, 2019, pp. 42–51.

[3] X. Shi, L. Cao, D. Wang, L. Liu, G. You, S. Liu, and C. Wang, "HERO: Accelerating autonomous robotic tasks with FPGA," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2018, pp. 7766–7772.

[4] M. Quigley, K. Mohta, S. S. Shivakumar, M. Watterson, Y. Mulgaonkar, M. Arguedas, K. Sun, S. Liu, B. Pfrommer, V. Kumar, *et al.*, "The open vision computer: An integrated sensing and compute system for mobile robots," in *2019 International Conference on Robotics and Automation*, 2019, pp. 1834–1840.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of Go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[6] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen, "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection," *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 421–436, 2018.

[7] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," *Advances in neural information processing systems*, vol. 29, pp. 4107–4115, 2016.

[8] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[9] D. Honegger, T. Sattler, and M. Pollefeys, "Embedded real-time multi-baseline stereo," in *2017 IEEE International Conference on Robotics and Automation*, 2017, pp. 5245–5250.

[10] N. K. Manjunath, A. Shiri, M. Hosseini, B. Prakash, N. R. Waytowich, and T. Mohsenin, "An energy efficient edgeai autoencoder accelerator for reinforcement learning," *IEEE Open Journal of Circuits and Systems*, vol. 2, pp. 182–195, 2021.

[11] T. Kozuno, E. Uchibe, and K. Doya, "Theoretical analysis of efficiency and robustness of softmax and gap-increasing operators in reinforcement learning," in *The 22nd International Conference on Artificial Intelligence and Statistics*, 2019, pp. 2995–3003.

[12] J. Su, J. Liu, D. B. Thomas, and P. Y. Cheung, "Neural network based reinforcement learning acceleration on FPGA platforms," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, pp. 68–73, 2017.

[13] H. Watanabe, M. Tsukada, and H. Matsutani, "An FPGA-Based on-device reinforcement learning approach using online sequential learning," *arXiv preprint arXiv:2005.04646*, 2020.

[14] S. Shao, J. Tsai, M. Mysior, W. Luk, T. Chau, A. Warren, and B. Jeppesen, "Towards hardware accelerated reinforcement learning for application-specific robotic control," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors*, 2018, pp. 1–8.

[15] Y. Li, H. Li, Z. Li, H. Fang, A. K. Sanyal, Y. Wang, and Q. Qiu, "Fast and accurate trajectory tracking for unmanned aerial vehicles based on deep reinforcement learning," in *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2019, pp. 1–9.

[16] M. G. Azar, V. Gómez, and H. J. Kappen, "Dynamic policy programming," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 3207–3245, 2012.

[17] Y. Tsurumine, Y. Cui, E. Uchibe, and T. Matsubara, "Deep reinforcement learning with smooth policy update: Application to robotic cloth manipulation," *Robotics and Autonomous Systems*, vol. 112, pp. 72–83, 2019.

[18] H. Yonekawa and H. Nakahara, "On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an FPGA," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops*, 2017, pp. 98–105.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[20] Y. Li, Y. Fang, and Z. Akhtar, "Accelerating deep reinforcement learning model for game strategy," *Neurocomputing*, vol. 408, pp. 157–168, 2020.

[21] A. Nøkland, "Direct feedback alignment provides learning in deep neural networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2016, pp. 1045–1053.

[22] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "FA3C: FPGA-accelerated deep reinforcement learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 499–513.

[23] A. Asseman, N. Antoine, and A. S. Ozcan, "Accelerating deep neuroevolution on distributed FPGAs for reinforcement learning problems," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 17, no. 2, pp. 1–17, 2021.

[24] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International Conference on Machine Learning*, 2018, pp. 1861–1870.