Hardware-oriented deep reinforcement learning for edge computing

Yoshiharu Yamagishi¹, Tatsuya Kaneko¹, Megumi Akai-Kasaya², and Tetsuya Asai²

¹ Graduate School of Information Science and Technology, Hokkaido University, M-BLDG 2F, Kita 14, Nishi 9, Kita-ku, Sapporo, Hokkaido 060-0814, Japan
² Faculty of Information Science and Technology, Hokkaido University, M-BLDG 2F, Kita 14, Nishi 9, Kita-ku, Sapporo, Hokkaido 060-0814, Japan

¹) yamagishi.yoshiharu.63@ist.hokudai.ac.jp

Received January 8, 2021; Revised March 19, 2021; Published July 1, 2021

Abstract: A new deep reinforcement learning enhancement is proposed for edge computing. This work focuses on deep Q-networks (DQNs), which are used in deep reinforcement learning. Although DQNs are typically improved through a software-based approach, hardware-specific knowledge such as that on data paths and pipelines is used for improving a DQN. The DQN performance is improved and the number of resources are reduced through an efficient hardware design that considers the learning flow and parameter search. As the scale of the problem increases, the amount of reduction in the use of resources also increases. For example, when the size of the block catch game is $5 \times 10$, the memory requirement is reduced by approximately 50% compared to a previous DQN. The proposed hardware-oriented approach can be applied to any software technology. This study facilitates the development of novel technologies that can be realized through edge computing.

Key Words: deep reinforcement learning, deep Q-network, hardware, edge computing

1. Introduction

Reinforcement learning is a well-known recently developed machine learning method [1, 2] in which an agent learns how to interact with the environment to achieve a certain goal, and is expected to play an important role in various fields [3, 4]. The further development of this reinforcement learning is deep reinforcement learning (deep RL), which introduces deep neural networks (DNNs) into the reinforcement learning method [5]. Compared with reinforcement learning, deep RL can be applied to handle more complex problems, such as gaming and driving [6, 7]. In March 2016, deep RL was used to implement the AlphaGo gaming system, the performance of which was surprisingly good when it defeated one of the best Go players in the world [8]. Many researchers have recently focused on deep RL and considered its applicability in many different fields [9, 10].

Edge computing, which has also been gaining recognition in recent years, is a technology that completes information processing immediately without the need to connect to a powerful server computer through a network [11]. This has several advantages, including being unaffected by the network en-
environment in which it is used and reducing the load on the cloud, which significantly increases in size every year. Edge computing is also an excellent choice in terms of security because it prevents the leakage of confidential information. Although deep RL and edge computing are extremely useful technologies, they face many different problems, making it difficult to combine them. As one of the problems faced by deep RL, most related studies have been software-oriented and are often based on the assumption that large central processing units (CPUs) and graphics processing units (GPUs) are being used [12, 13]. CPUs and GPUs have the potential to solve a wide range of problems, but they also have a considerable amount of wasted motion, which results in substantial costs and power consumption. This is a problem that cannot be ignored in edge computing because there are only a few computational resources available for edge computing.

In this study, we attempted to realize a deep Q-network (DQN), which is a type of deep RL in edge computing, by applying various efficiency improvements and optimizations [14]. The key contribution is the methodology. Many researchers have tried to optimize the DQN in a software-oriented manner, whereas we attempted to optimize it through a hardware-oriented approach [15]. In addition, in the activation function of the deep learning part of deep reinforcement learning, we are dealing with a sigmoid function, which is nonlinear.

2. Necessity for thinking in a hardware-oriented way

Software is a behavioral description and is computed using the CPU, which reads the program from memory and processes it sequentially. However, hardware is a register transfer level description, and parallel processing can be applied by freely designing the data path. For example (Fig. 1), when conducting 100 operations, the software-oriented sequential processing requires an execution time of 100 cycles, whereas the hardware-oriented parallel processing can be performed in less than 100 cycles using multiple arithmetic logic units [16].

When considering how to optimize a theory, a software-oriented approach is often used. The approach mainly focuses on reducing the overall number of computations. By contrast, we considered the optimization in a hardware-oriented manner. In our approach, we considered parallelization and pipelining based on the constraints of available resources, performance, size, power, and other factors considering the “parts that should be computed simultaneously” and “parts that cannot be switched back and forth.” In addition, in hardware, because reducing the circuit size is directly related to the cost, we considered the bit limit of the numerical values and the data paths.

![Fig. 1. Difference between software-oriented and hardware-oriented thinking. Software-oriented processing is sequential, whereas hardware-oriented processing is parallel using multiple arithmetic logic units (ALUs); thus, the time required for the same calculation differs depending on the way of thinking.](image-url)
3. Hardware-oriented reinforcement learning

3.1 Q-Learning and the DQN from a hardware perspective

In this study, we used Q-learning and its advanced form, a DQN, in reinforcement learning [14, 17]. The flow of Q-learning is shown in Fig. 2(a), and the procedure consists of the following five stages.

1. Initialization: The starting state is set and random numbers are assigned to the values of the Q-table (a lookup table that stores the values of all possible actions in each state).

2. Action decisions: The agent refers to the Q-table using the current state and retrieves the value (Q-value) of each of the next possible actions. The agent compares the values and selects the action with the largest value. In addition, we used ε-greedy to ensure that the learning was not biased, and we initially set the probability of ignoring the output values (randomness) to a high value and then reduce the randomness as learning progresses.

3. State update: The state is updated based on the action selected in stage 2.

4. Reward confirmation: The immediate reward is checked in the new state.

5. Update Q-table: The value of the current action is updated using the immediate reward obtained in stage 4 and the maximum value of the next action is available. The equation for the value update is given by the following:

\[
Q_{s_t,a_t} \leftarrow Q_{s_t,a_t} + \eta (R_{t+1} + \gamma \max_a Q_{s_{t+1},a} - Q_{s_t,a_t}),
\]

where \(s\) is the state, \(a\) is the action, \(t\) is the number of steps, \(\eta\) is the learning rate, \(\gamma\) is the discount factor, \(R\) is the immediate reward, and \(Q\) is the value.

When stages 2-5 are repeated multiple times, the value of correct actions increases and the value of incorrect actions decreases, and the agent can apply correct actions. Figure 3 shows the hardware part. The color of the wiring indicates the procedure (STAGE 1, black; STAGE 2, red; STAGE 3, blue; and STAGE 4, green).

Fig. 2. Flow of solving a problem with reinforcement learning (top, Q-learning; bottom, DQN).

Fig. 3. Hardware-oriented design of the learning part. The color of the wiring indicates the procedure (STAGE 1, black; STAGE 2, red; STAGE 3, blue; and STAGE 4, green).
version of the learning part of Q-learning and a DQN. Most of the resources required for Q-learning are occupied by the Q-table, which is also shown in Fig. 3(a). As the total number of states increases, the memory increases exponentially. The size of this Q-table is

\[
\text{Memory}_{\text{Qtable}} = (N_a \times M + S_{\text{game}})S_{\text{all}},
\]

where \(N_a\) is the number of possible actions, \(M\) is the memory size, \(S_{\text{game}}\) is the game scale, and \(S_{\text{all}}\) indicates all possible states. The DQN is an attempt to solve this problem with a DNN. The DQN replaces the Q-table of Q-learning with a DNN and learns the DNN by updating the value in Q-learning. The updating of the DNNs requires data that can be used as a teacher for the results (output) of the given input and inference. Learning progresses by filling in the differences between the inference results and the teacher data.

For reinforcement learning, only the reward fits perfectly into the teacher data. However, because a reward is not available every time, the DQN generally treats the reward and temporal difference errors in Eq. (3), which is part of the value-update formula for Q-learning in Eq. (1), as the teacher data [3, 14].

\[
R_{t+1} + \gamma \max_a Q_{s_{t+1},a} - Q_{s_t,a_t}.
\]

The flow of the DQN is shown in Fig. 2(b). The DQN consists of the following eight stages.

1. Initialization: The starting state is set, and the DQN is set (the number of nodes in the layer are determined, and in each layer, random numbers are assigned to the weights and bias values of each node).

2. Action decisions: The agent inputs its current state into the DQN, makes inferences, and obtains the value (Q-value) of each of the next possible actions. The agent compares the values and selects the action with the largest value. In a DQN, we also use \(\epsilon\)-greedy.

3. State update: The state is updated based on the action selected in stage 2.

4. Reward confirmation: The immediate reward is checked in the new state.

5. Saving the parameters: Each parameter (state, action, reward, next state, and terminal) is saved to memory.

6. Select training data: A mini-batch number of experiences are selected and retrieved from the memory of stage 5.

7. Create teacher data

   (a) Infer the state before the action as the input of the DNN and output Q-value (previous).

   (b) Infer the state after the action (next state) as the input of the DNN and output the Q-value (after).

   (c) Create the teacher data in Eq. (3) using the part of the (previous) Q-value that was actually actioned. However, if the next state does not exist, \(\max_a Q_{s_{t+1},a}\) will be zero.

8. DNN backpropagation: DNN weights and biases are updated using a gradient descent.

Stages 1–4 of the DQN are almost the same as those of Q-learning, and stages 5–8 of the DQN correspond to stage 5 of Q-learning. The parameters obtained up to stage 4 were stored at stage 5. In stages 6–8, the parameters are randomly retrieved from the past experience stored in stage 5 and used for training the DNN. This is called an “experience replay,” and is described in a later section.

Figure 3(b) shows the hardware version of the learning part of the DQN. The DNN is used twice to obtain the Q-value (previous) from the input of the the state before the action and once to obtain the Q-value (after) as the target from the input state after the action. We can see that this part can be made more efficient by parallelizing it.
3.2 Object description

3.2.1 Rule description

In this study, we used the Block Catch game to test the accuracy. This is a simple game of catching blocks falling from above by moving the player. At the beginning of the game, the ball and player start at the initial location. In one step, the ball moves one square down, and the player can choose one of the following actions: stay in place, move one square to the left, or move one square to the right. When the ball reaches the bottom, the game is complete. In other words, if the game scale is $3 \times 5$, the game consists of two steps, and if the game scale is $5 \times 5$, the game consists of four steps. At the end of the game, if a player is able to catch the ball, the player is rewarded; if not, the player is penalized.

![Game scales](image)

(a) Game scale $3\times5$  
(b) Game scale $5\times5$

Fig. 4. Appearance of the Block Catch game. Blocks falling from above are caught by moving the player.

3.2.2 Compatibility with Q-learning

As a necessary condition for Q-learning, the problem is a Markov decision process, which consists of states, actions, transition probabilities, and rewards, each of which has the following characteristics.

- The environment has various states and is observable.
- In each state, the agent can arbitrarily select the available actions.
- When an agent performs an action, the environment probabilistically transitions the states, and rewards are accordingly obtained.
- The reward is an immediate reward or the expected value of the reward that is obtained when the state transitions.

Because the Block Catch game satisfies these conditions, we considered it to be a suitable object for this study.

3.2.3 Data form and retention

State  An array of type int, where the square in which the block or player exists is represented by 1, whereas other squares are represented by 0s.

Reward  Int type: 1 for a good reward, −1 for a penalty, and 0 otherwise.

Action  Int type: −1 for move left, 0 for remain in place, and 1 for move right.

Terminal  Bool type: true if it is the last step and is false otherwise.

3.3 Sizes of the object and resources required

In this section, we present the memory requirements against the game scale. For the game scale, the column is fixed (= 5), and only the row is varied.
3.3.1 Differences in resources required for Q-learning and DQN

Most of the resources required in Q-learning are occupied by the Q-table, which holds all values. The Q-table grows exponentially with the increasing game scale because the Q-table is derived as follows:

\[
\text{Memory}_Q = (N_a \times M + S_{\text{game}})S_{\text{all}} = (3 \times 32 + (5S_{\text{row}})) \times 2^{5S_{\text{row}}-2(S_{\text{row}}-1)} = (5S_{\text{row}} + 96) \times 2^{3S_{\text{row}}+2},
\]

where \(S_{\text{row}}\) is the row size. This shows that the memory requirement for Q-learning increases significantly as the game scale increases. This makes it unsuitable for edge computing, where the amount of resources is directly related to the cost.

The DQN replaces all Q-tables in Q-learning with a DNN. The resources needed for the DNN part (weights and bias values) are

\[
\text{DNN}_{\text{DQN}} = M(N_w + N_b) = 32((5S_{\text{row}}) \times (44 + 4S_{\text{row}}) + (44 + 4S_{\text{row}}) \times 3) + ((44 + 4S_{\text{row}}) + 3)) = 640S_{\text{row}}^2 + 7552S_{\text{row}} + 5728,
\]

where \(N_w\) is the number of weights, and \(N_b\) is the number of biases. In addition, the DQN often uses a method of storing past experiences in memory and reusing it (experience replay) to improve the efficiency of learning. As a result, the overall DQN memory requirements are

\[
\text{Memory}_{\text{DQN}} = \text{DNN}_{\text{DQN}} + (2S_{\text{game}} + A_{\text{all}} + R + T)N_{\text{history}} = \text{DNN}_{\text{DQN}} + (10S_{\text{row}} + 2 + 1 + 1) \times 1000S_{\text{row}} = 10640S_{\text{row}}^2 + 11552S_{\text{row}} + 5728,
\]

where \(A_{\text{all}}\) is the memory size to represent all actions, \(T\) is the memory size representing the terminal, and \(N_{\text{history}}\) is the number of histories. Therefore, it is suitable for edge computing.

However, in terms of accuracy and speed, the DQN is still inferior to Q-learning and should be improved. The next section focuses on our approach to improve it using the hardware-oriented approach.

3.3.2 Suitability of the game scale used in this study

The resource requirements of the DQN are almost the same as those of Q-learning when the game scale is \(3 \times 5\) \((S_{\text{row}} = 3)\). However, the resource requirements of Q-learning increase rapidly when the game scale exceeds \(3 \times 5\), and they are already much larger than those of the DQN.

Therefore, we first used \(3 \times 5\) to improve the accuracy of the DQN. Then, after some improvement in accuracy was expected, the game scale was changed to \(5 \times 5\) for verification.

3.4 Hardware-oriented approaches to improve the performance of the DQN

Compared to Q-learning, the DQN requires fewer resources but is less accurate and has a slower learning speed. Therefore, several software-oriented methods have been considered to improve the DQN performance. We revised the method in a hardware-oriented manner. We aimed to make the DQN more suitable for edge computing by attempting to reduce the required resources and avoid the use of the CPU and GPU as much as possible. We attempted to improve the efficiency and optimization of the “experience replay” and “fixed target Q-network.”

3.4.1 Hardware-oriented experience replay

“Experience replay” is a method of replaying the information of how the environment changes as a result of the agent’s previous behavior \([18]\). In the DQN, the primary goal is to train the DNN well. However, if we try to train DNN using only real-time experience, they will not train well. As one reason for this, real-time experiences are often similar; in other words, there is a large correlation
between experiences. To solve this problem, an experience replay is considered as an effective method. The training of a DNN using experience does not require the use of the past Q-value (DNN output) as it currently is. It uses the Q-value output from the past state input to the current DNN. In addition, similar to the flow of learning a DQN, there are five types of data that should be stored in memory, i.e., state, given as input for the DNN inference; action, used to identify the Q-value to be updated; reward, used as a part of the DNN teacher data; next state, used as an input for the DNN to create the teacher data; and terminal, used to determine the existence of the next state when STEP is the last.

In a software-oriented system, we are unaware of the location where memory is stored or how data are retrieved. However, in hardware implementation, it is extremely important to reduce the cost, power consumption, execution time, and scalability. By considering the data path, we can significantly reduce the memory required to store the experience. In the conventional method, the state, action, reward, next state, and terminal are stored. However, we realized that within the same game count (terminal = false when the game count = t), the next state when the game count = t is the same as the state when the game count = t + 1. We solved this problem by using multiple data paths and devising a way to access the memory, as shown in Fig. 6(a). Furthermore, it was found that by applying this technique, learning using not only one but two previous states can be achieved with the same amount of memory, as shown in Fig. 6(b).

![Fig. 5. Software-oriented storage of experiences. For each experience, the state, action, reward, next state, and terminal are stored.](image)

![Fig. 6. Hardware-oriented reduction of memory used. By comparing the figures in (a) and (b), we can see that the amount of memory remains the same even if the before state is saved in addition to the conventional method.](image)

### 3.4.2 Hardware-oriented fixed target Q-network

A "fixed target Q-network" is a method used to fix the DNN and generate max \( Q_{s_{t+1}, a_t} \), which is required to create teacher data for a certain period of time [19]. The main DNN changes each time it is used for training. If the same DNN is used to generate the teacher data, the teacher data will also fluctuate upon each use. This makes the learning unstable and the parameters diverge. To solve this problem, we prepare a target Q-network separately from the main DNN, as shown in Fig. 7. This network is not trained, and instead synchronizes by copying the weights and bias data of the main DNN at regular intervals. In this way, the teacher data are fixed to a certain extent, and the learning becomes stable.

In a DQN, the DNN is used twice, first to “obtain the Q-value with the state as input” and second to “obtain the Q-value as the target with the next state as input.”

In a conventional DQN, the same DNN is used; therefore, it cannot be parallelized. We found the target Q-network to be useful for not only stabilizing the learning but also for pipelining. Figure 8 indicates that we succeeded in four stages: STAGE 1, data selection from memory and periodic
target Q-network synchronization (parallel); STAGE 2, forwarding operation on both DNNs (parallel); STAGE 3, creating the training data; and STAGE 4, back propagation by conducting a hardware-oriented optimization. By contrast, the software-oriented approach typically requires six steps.

4. Evaluation

In this study, we monitored how much the agent can learn from an unlearned state to playing the Block Catch game thousands of times (referred to herein as the total game). In this section, we refer to “how much difference there is between Q-learning and the DQN,” “the results of focusing on and improving the way the DQN learns,” and “how each hyperparameter is involved in learning” using the experimental results. We compared the performance from two perspectives: learning progress and accuracy. These are shown in a graph according to the following rules.

Accuracy For the total game, the success rate of the last 100 games was considered as the accuracy of one trial. We conducted 100 trials under the same conditions and represented the results in a graph (horizontal axis, success rate; vertical axis, number of trials). From the graph, we can read the dispersion of the data and the accuracy under each condition.

Progress of learning Each trial was divided into 100 games (10 games only when comparing Q-learning and the DQN), and the number of successes for each game was calculated and represented in a graph (horizontal axis, epoch; vertical axis, number of successes). The graph shows the progress of learning within a single trial.

The meanings of the hyperparameters used in this study are described in the following. The hyperparameters, which were varied, are shown in the tables in each section.

Total game Total number of Block Catch games repeated in one trial.
**History store** Number of histories to maintain for replaying the experience.

**Epsilon** A parameter of the $\varepsilon$-greedy method, where $\varepsilon$ takes values of between 0–1. $\varepsilon = 0$: take action based only on the Q-value. $\varepsilon = 1$: ignore the Q-value and take a random action. The value of $\varepsilon$ varies within one trial, and the parentheses indicate the condition. GC indicates the number of games.

**RL discount factor** The degree to which the most recent or future rewards should be prioritized when creating the teacher data. Generally, a number of less than or equal to 1 is used. The discount factor is $\gamma$ in Eq. (1). In this study, we used 0.875.

**DNN learning rate** Decide by what extent the DNN weights and biases should be changed at a particular time. If the learning rate is too high, the values are likely to diverge. If it is too small, it is likely to fall into a local solution with few updates. Take a value of between zero and 1.

**DNN values** Weights and biases are the components of a DNN. Learning proceeds as these values are changed. The initial value is a random number, and the range of random numbers to be included is set.

**DNN layer** The DNN used was the game scale for the input layer, i.e., 64 for the middle layer and 3 for the output layer.

**DNN optimizer** As an optimizer, we used a minibatch stochastic gradient descent (minibatch-SGD), which is a method that randomly takes some of the data from the entire dataset used for training and trains them as mini-batches [20].

**Number of mini-batches** Batch training was conducted with 32 mini-batches.

**Target Q-network synchronization frequency** The target Q-network was synchronized with the main DNN once every 10 games.

### 4.1 Difference between Q-learning and the DQN

In this section, we compare the differences in performance between Q-learning and simple DQN without any innovations using the game scale $3 \times 5$.

Table I shows each hyperparameter, and for both Q-learning and DQN, we employed commonly used parameters.

Figure 9 shows the progression of learning for Q-learning and the simple DQN. Because Q-learning completed the learning quickly, we divided the epoch into 10 games and counted the number of successes for each. Q-learning completed its learning after approximately 200 trials. However, the DQN did not complete the learning even after 5000 trials. In addition, the DQN learned slowly as the number of learning cycles increased, but the dispersion was large.

| Table I. | Value of each hyperparameter of Q-learning and the DQN in Section 4.1. Commonly used parameters were employed for both Q-learning and the DQN. |
|----------|----------------------------------------------------------------------------------------------------------------------------------|
| Q-learning | Deep Q-network |
| Total game | 5000 | 5000 |
| History store | - | All |
| Epsilon | 0.8 (GC < 50) 0.3 (50 ≤ GC ≤ 100) 0 (100 < GC) | 0.8 (GC < 1000) 0.3 (1000 ≤ GC ≤ 4000) 0 (4000 < GC) |
| DNN learning rate | - | 0.1 |
| DNN initial weight | - | $-1$ − $+1$ |
| DNN initial bias | - | $-1$ − $+1$ |
Fig. 9. Graph showing the progress of Q-learning and simple DQN learning. Epochs were divided into 10 games each, and the number of successes for each was plotted. Compared to the DQN, Q-learning shows much faster convergence and less dispersion in learning.

Fig. 10. Histograms showing results of 100 trials of success probability after learning for Q-learning and the simple DQN, respectively. The success probability of one trial is the number of successes in the last 100 games of that trial. The accuracy of Q-learning is 100%, whereas the accuracy of the simple DQN is approximately 60%, indicating a large dispersion.

Figure 10 shows that the accuracy of the DQN from the 4901st to the 5000th times was approximately 60%, whereas that of Q-learning was 100%. Neither the convergence speed nor the accuracy of the simple DQN were as good as those of Q-learning; therefore, the DQN must be improved.

4.2 Result of improving the DQN
In this section, we focus on how DQNs learn to improve, based on the description provided in Section 3.4. For comparison, we used a game scale of $3 \times 5$.

| Total game | 5000 |
| History store | All |
| Epsilon | 0.8 ($GC < 2500$), 0.3 ($2500 \leq GC \leq 4900$), 0 ($4900 < GC$) |
| DNN learning rate | 0.1 |
| DNN initial weight | $-1 \rightarrow +1 \rightarrow -0.01 \rightarrow +0.01$ |
| DNN initial bias | $-1 \rightarrow +1 \rightarrow 0$ |

4.2.1 Fixed target Q-network
In Section 3.4.2, we indicated that a “fixed target Q-network” is effective for not only its original purpose of improving the accuracy but also for pipelining. Figure 11 shows the difference in accuracy before and after adapting the target Q-network. It can be observed that the accuracy is improved by approximately 10% by adapting the target Q-network.

4.2.2 Initial parameters of the DNN tailored to the DQN
In the DQN, the teacher data are the immediate rewards obtained when the state transitions and the
Q-value (after) after the action. The immediate reward always indicates the correct action. However, in the early stages of learning, the Q-value (after) is a random number; thus, the reliability of the correct action is low. To increase the influence of the immediate reward during the early stage of learning, we decided to reduce the initial values of the DNN parameters, i.e., the weight and bias. Figure 12 shows the difference in accuracy depending on the initial value of the DNN parameters. It can be seen that decreasing the initial value improves the accuracy and decreases the learning dispersion.

4.2.3 Learning of two previous states
In Section 3.4.1, we propose a hardware-oriented approach that involves devising a data path that combines the state and next state into a single version. By applying this approach, we found that we can implement the “learn two previous pasts” method with almost no increase in resources. Figure 13 shows the difference between the conventional method and our proposed approach. In addition, both the accuracy and dispersion of learning were improved by learning two previous pasts.

4.3 Number of total games
We varied the total number of games repeated in each trial to compare the accuracy and learning progress.

The hyperparameters are listed in Table III, where the total number of games varies from 3000.

| Total game (TG) | 500, 1000, 3000, 5000, 10000 |
|----------------|--------------------------------|
| History store | All                            |
| Epsilon       | 0.8 (GC < TG/2)                |
|               | 0.3 (TG/2 ≤ GC ≤ last 100)    |
|               | 0 (last 100)                   |
| DNN learning rate | 0.1                          |
| DNN initial weight | -0.01 + +0.01             |
| DNN initial bias | 0                             |

Table III. Value of each hyperparameter of the DQN from Section 4.3. The size of the total game was changed. The epsilon value of ε-greedy was 0.8 and decreased to 0.3 after half of the total game was complete and was set to 0 for the final 100 iterations.

Fig. 14. Graph showing the progression of each learning process when the total game varied. Each graph shows a sharp change in the percentage of correct answers at the time that the epsilon changes.
Fig. 15. Histogram showing the accuracy when the total game is set to 3000, 5000, and 10,000. The larger the total game, the higher the accuracy. When the total game is 10,000, the success rate reaches 100%.

5000, and 10,000. In the epsilon change of $\varepsilon$-greedy, the random rate was slightly reduced for half of the total game. Moreover, for the last 100 iterations, the random rate was set to zero. We also used a game scale of $3 \times 5$ to verify the results.

Figure 14 shows the progress of learning when the total game is varied. The probability that an action is randomly selected is 80% until halfway through the total game, and it becomes 30% from that moment until the last 100 iterations. In other words, even if learning is complete at that moment, the number of successes will inevitably stagnate. We call this phenomenon “hitting the ceiling” herein. In Fig. 14, we can see that when the total game is 1000 or later, a “hitting the ceiling” phenomenon was observed. In the next section, we verify the epsilon under all conditions with a total game of 500.

Figure 15 shows the accuracy for each of the total game changes of 3000, 5000, and 10,000. It can be seen that as the total game increases, the accuracy is improved and the learning dispersion is decreased.

4.4 Learning speed and accuracy

Based on the results described in Section 4.3, we vary the timing of the epsilon changes. We verified the relationship between the speed of learning and accuracy by reducing the value of epsilon at a time when learning was considered to be complete.

As shown in Fig. 16, when the reduction in the epsilon size is faster, the “hitting the ceiling” phenomenon described in Section 4.3 disappears. Although the learning converges faster, we can see that the oscillations still occur even if the learning continues after the epsilon reaches zero.

Figure 17 shows a decrease in accuracy when the total game is 5000 and 10,000, compared to the

| Table IV. Value of each hyperparameter of the DQN in Section 4.4. The total game was varied, but the change in epsilon was standardized to a total game of 500. |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| Total game      | 500, 1000, 3000, 5000, 10000 |
| History store   | All             |
| Epsilon         | 0.8 (GC < 250)  |
|                 | 0.3 (250 ≤ GC ≤ 400) |
|                 | 0 (400 < GC)    |
| DNN learning rate| 0.1             |
| DNN initial weight| −0.01 – +0.01  |
| DNN initial bias | 0               |

Fig. 16. Graph showing the progression of learning when the total game is varied. However, the number of epochs in which epsilon changes is unified, unlike in Section 4.3, where the epochs in which epsilon changes are different for each condition. The “hitting the ceiling” phenomenon is no longer present, but the graph continues to oscillate even when the total game is increased.
Fig. 17. Histogram showing the accuracy when the total game is set to 3000, 5000, and 10,000. All three graphs have similar levels of accuracy. Compared to Section 4.3, the accuracy improves when the total game is 3000. However, when the total game is 5000 or 10000, the accuracy decreases.

We can see that the “hitting the ceiling” phenomenon (searching with a large epsilon) is necessary for better learning. By contrast, when the total game is 3000, the accuracy improves. This indicates that it is better to reduce the value of epsilon as soon as possible if it is impossible to increase the number of training sessions.

4.5 Game scale

With an improved accuracy of the DQN, the success rate reaches almost 100% for a game scale of $3 \times 5$. In this section, we change the game scale from $3 \times 5$ to $5 \times 5$ and monitored the decrease in accuracy.

In Fig. 18, focusing on the graph for total games = 1000, 3000, and 5000, we can see that when the game scale is $5 \times 5$, the oscillation is larger than that when the game scale is $3 \times 5$.

Figure 19 shows that when the game scale is $5 \times 5$, the accuracy is lower than when the game scale was $3 \times 5$. Furthermore, Fig. 20 shows that the graph of the game scale of $5 \times 5$ is similar to the accuracy graph in Section 4.4. In other words, based on the discussion in Section 4.4, half of total

| Total game | 500, 1000, 3000, 5000, 10000 |
|------------|-------------------------------|
| History store | All                          |
| Epsilon | 0.8 (GC < TG/2), 0.3 (TG/2 ≤ GC ≤ last 100), 0 (last 100) |
| DNN learning rate | 0.1                         |
| DNN initial weight | −0.01 – +0.01               |
| DNN initial bias | 0                           |

Fig. 18. Graph showing the progress of learning when the game scale is changed from $3 \times 5$ to $5 \times 5$. The oscillation during the learning process is larger when the game scale is $5 \times 5$ than when the game scale is $3 \times 5$. 

Fig. 19. Graph showing the progress of learning when the game scale is changed from $3 \times 5$ to $5 \times 5$. The oscillation during the learning process is larger when the game scale is $5 \times 5$ than when the game scale is $3 \times 5$. 

Fig. 20. Graph showing the progress of learning when the game scale is changed from $3 \times 5$ to $5 \times 5$. The oscillation during the learning process is larger when the game scale is $5 \times 5$ than when the game scale is $3 \times 5$. 

Table V. Value of each hyperparameter of the DQN in Section 4.5. The game scale was varied with the same parameters (total game = 5000). In addition, as shown in Fig. 20, we monitored the case of total game = 10000 only when the game scale was $5 \times 5$. 

538
Fig. 19. Histogram showing the accuracy when the game scale is changed from $3 \times 5$ to $5 \times 5$. When the game scale was $3 \times 5$, the accuracy was approximately 100%. However, when the game scale was $5 \times 5$ under the same conditions, the accuracy decreased.

Fig. 20. Histogram showing the accuracy when the game scale is $5 \times 5$ and the total game is 10,000. There was not much change in accuracy compared to that in Fig. 19(b).

game = 5000 and 10,000 is insufficient for the number of searches with a large epsilon value. This indicates that we should increase the number of searches as the game scale increases.

4.6 Number of histories retained

From the results in Section 4.5, we found that we should increase the number of searches to solve complex problems. When we increased the number of searches, we also increased the history memory. For a hardware-oriented environment with edge computing, a large memory is not ideal. We attempted to solve this problem by checking if the entire history was necessary for learning. To verify this, in this section, we use a game scale of $5 \times 5$.

As shown in Table VI, we unified the total game to 5000 and varied the maximum size of the history memory. When the history exceeded the maximum value, the oldest ones were erased.

Figure 21 shows that greater the amount of history, the smaller the oscillations become during learning.

By contrast, Fig. 22 shows that the highest accuracy was obtained when the history was 3000. This indicates that it is better to have a moderate oscillation in the middle of the learning process.

| Total game (TG) | 5000 |
|-----------------|------|
| History store   | 1000, 3000, 5000 |
| Epsilon         | $0.8 \ (GC < 2500), 0.3 \ (2500 \leq GC \leq 4900), 0 \ (4900 < GC)$ |
| DNN learning rate | 0.1 |
| DNN initial weight | $-0.01 - +0.01$ |
| DNN initial bias | 0 |

Fig. 21. Graphs showing the progress of learning when the upper limit of the history memory is changed to 1000, 3000, and 5000. The more the past experience is retained, the smaller the oscillation becomes during the learning process.
4.7 Comparison of Q-learning, the simple DQN and the improved DQN

In Section 3.3 we showed that for Q-learning, the resource requirements grow exponentially as the size of the problem scale increases. In other words, even if the convergence speed and accuracy of Q-learning are good, it is unsuitable for edge computing, where the number of resources is directly related to the cost. Therefore, we turned our attention to a DQN. However, a simple DQN without any innovations could not replace Q-learning because the convergence speed and accuracy were too low. We aimed to achieve an accuracy similar to Q-learning by adding various improvements to the DQN. In this section, we evaluate how much our proposed DQN can improve the performance while reducing the required resources.

4.7.1 Comparison of hardware resources

As Fig. 23 shows, the hardware resources of Q-learning grow exponentially as the size of the problem scale increases. By contrast, that of the simple DQN increases polynomially. In our proposed improved DQN, because we used a target Q-network, the hardware resources of the deep neural network part are

\[
\text{DNN}_{\text{ImprovedDQN}} = 2 \times M(N_w + N_b)
\]

\[
= 64\left(5S_{\text{row}} \times (44 + 4S_{\text{row}}) + (44 + 4S_{\text{row}}) \times 3\right) + (44 + 4S_{\text{row}} + 3)
\]

\[
= 1280S_{\text{row}}^2 + 15104S_{\text{row}} + 11456.
\]

This requires twice as many hardware resources as the simple DQN. However, since we optimized the history memory and limited the history to be retained, the overall hardware resources of the improved DQN are

![Fig. 23. The graphs of Q-learning and the simple DQN are drawn from the equations in Section 3.3. Q-learning increases exponentially, whereas the simple DQN and improved DQN increase polynomially. The memory requirement of the improved DQN was less than that of the simple DQN when the row size is larger than 2.](image-url)
\[
\text{Memory}^{\text{ImprovedDQN}} = \text{DNN}^{\text{ImprovedDQN}} + (S_{\text{game}} + A_{\text{all}} + R + T + I)N_{\text{history}} \\
= \text{DNN}^{\text{ImprovedDQN}} + (5S_{\text{row}} + 2 + 1 + 1 + 1) \times (0.6 \times 1000S_{\text{row}}) \\
= 4280S_{\text{row}}^2 + 18104S_{\text{row}} + 11456,
\]

where \( I \) is the memory size to represent the initial state. As a result, the improved DQN requires fewer hardware resources than the simple DQN when the row size is two or more.

4.7.2 Comparison of convergence speed

In Section 4.1, we compare the difference in convergence speed between Q-learning and the simple DQN using a game scale of \(3 \times 5\). In this section, to make the difference easier to understand, we compare them using a \(5 \times 5\) game scale. As shown in Table VII, the timing of the epsilon change was unified. As Fig. 24 shows that the convergence speed of Q-learning is overwhelmingly fast. By contrast, the simple DQN does not converge, which indicates that it is unable to learn at the \(5 \times 5\) scale. The improved DQN is not as fast as Q-learning but is still able to learn on a \(5 \times 5\) scale, indicating that it is superior in performance to the simple DQN.

4.7.3 Comparison of accuracy

Based on Sections 4.3 and 4.4, we varied epsilon at the timings shown in Table VIII to stabilize the

| Total game | 5000 |
|---|---|
| History store | All (Simple DQN), 3000 (Improved DQN) |
| Epsilon | 0.8 (GC < 2500), 0.3 (2500 ≤ GC ≤ 4900), 0 (4900 < GC) |

Fig. 24. Graph showing the progress of Q-learning, the simple DQN, and the improved DQN. The improved DQN is not as good as Q-learning, but converges faster and learns better than the simple DQN.

Fig. 25. Histogram showing the accuracy of Q-learning, the simple DQN, and the improved DQN. The accuracy of Q-learning is 100%, but the simple DQN has a large scatter and does not learn well, whereas the improved DQN has some scatter but a high level of accuracy.
DQN learning. However, as described in Section 4.7.2, the simple DQN was unable to learn well. By contrast, the improved DQN succeeded in producing a high level of accuracy.

5. Discussion
In this study, we improved the DQN to make it suitable for edge computing using hardware-specific knowledge, such as data paths and pipelines.

This indicates that thinking in a hardware-oriented manner may allow for efficiency gains that are impossible with software orientation. This adds a new perspective to [4, 12, 13].

As a result, when the game scale was \(3 \times 5\), the accuracy was low before the improvement and was almost 100% after 5000 training sessions. When the game scale was expanded to \(5 \times 5\), we succeeded in preventing a rapid increase in the history memory by searching the necessary amount of history. However, the accuracy did not reach 100% even when the training frequency was increased to 10,000 iterations.

The method used in this study attempted to improve accuracy from two perspectives: the learning flow of the DQN and parameters such as the history. We also improved the hardware-oriented learning part and the memory.

However, when solving more complex problems, consideration of only these two perspectives is insufficient in terms of the accuracy and learning speed. Therefore, we believe that we require a new perspective for improving the DQN. Moreover, it is necessary to verify the entire hardware, including the data input and output. Based on these considerations, we plan to conduct research in two areas: examination of a new optimizer to improve the accuracy and learning speed, and an examination of the full hardware.

5.1 Optimizer
Here, the optimizer simply indicates a method of a back-propagation of DNNs. In this study, we used minibatch-SGD.

Other optimizers with a higher performance than minibatch-SGD, such as RMSProp and Adam, have been proposed [20]. However, they are not designed to be hardware-oriented, and many calculations, such as the root calculation and division, are time-consuming when converted into hardware. Because the optimizer is used in every learning process, these problems cannot be ignored. Therefore, one of our future goals is to optimize the RMSProp and other optimizers in a hardware-oriented manner.

5.2 Hardware design
In this study, we mainly optimized the learning part and memory for “experience replay” in a hardware-oriented manner because they are the most resource-intensive parts of a DQN. However, to implement a DQN in hardware, it is necessary to consider other parts as well. Figure 26 shows the overall structure of the designed hardware-oriented DQN applied in this study. Our future
goal is to discover new optimizations and efficiencies by considering the overall structure, including the data input/output and the search parts.

6. Conclusions
In this study, we focused on the DQN to solve the Q-learning problem in which the hardware resources required become exponentially larger as the problem scale increases. We improved the DQN in a hardware-oriented method by focusing on edge computing. As a result, we succeeded in improving the accuracy and even solved problems that could not be solved with the previous DQN. Furthermore, we succeeded in reducing the required hardware resources compared to the previous DQN. For example, in Section 4.7, it is shown that when the size of the Block Catch game is 5 × 10, the memory requirement is reduced by approximately 50% compared to the conventional DQN. In recent years, artificial intelligence technologies have been typically conceived and developed in a software-oriented manner, and hardware is often an afterthought. However, the concept of optimization is different for software researchers than for hardware researchers. Therefore, extensive knowledge and complex technology are required to convert a software-oriented product into a hardware product. Another problem is scalability: Even if the idea is ideal from a hardware-oriented perspective, it may be rejected or deleted from the software-oriented perspective. There is also a possibility that the idea will be biased toward the software-oriented approach.

In this study, we focused on the DQN, and the proposed hardware-oriented approach can be applied to all software technologies. As the novelty of our proposal, it considers the hardware perspective during the stage in which theory is first created. In this way, the latest technology will become more accessible and new ideas can be more likely conceptualized.

Acknowledgments
This study was supported by a JSPS Grant-in-Aid for JSPS Fellows and a Grant-in-Aid for Scientific Research on Innovative Areas [18H05288] from the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan.

References
[1] E. Nakai, *IT Enjinianotameno Kyokagakusyu Riron Nyumon [Introduction to Reinforcement Learning Theory for it Engineers]*, Gijutsu Hyoron-Sha Co., Tokyo, 2020.
[2] N. Brown and T. Sandholm, “Superhuman AI for heads-up no-limit poker: Libratus beats top professionals,” *American Association for the Advancement of Science*, vol. 359, no. 6374, pp. 418–424, December 2017. DOI: 10.1126/science.aao1733
[3] T. Morimura, *Machine Learning Professional Series Reinforcement Learning*, Kodansha Ltd., Tokyo, 2019.
[4] K. Arulkumaran, M.P. Deisenroth, M. Brundage, et al., “Deep reinforcement learning: A brief survey,” *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.
[5] J. Patterson and A. Gibson, *Deep Learning: A Practitioner’s Approach*, O’Reilly Media Inc., 2017.
[6] D. Silver, T. Hubert, and J. Schrittwieser, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” *arXiv preprint*, arXiv:1712.01815v1, 2017.
[7] V. Mnih, K. Kavukcuoglu, and D. Silver, “Playing Atari with deep reinforcement learning,” *arXiv preprint*, arXiv:1312.5602v1, 2013.
[8] D. Silver, A. Huang, C.J. Maddison, et al., “Mastering the game of Go with deep neural networks and tree search,” *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.
[9] D. Silver, J. Schrittwieser, K. Simonyan, et al., “Mastering the game of Go without human knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.
[10] J. Ikemoto and T. Ushio, “Application of deep reinforcement learning to networked control systems with uncertain network delays,” *NOLTA*, vol. 11, no. 4, pp. 480–500, 2020. DOI: 10.1587/nolta.11.480
[11] T. Kaneko, K. Orimo, I. Hida, et al., “A study on a low power optimization algorithm for an
edge-AI device,” *NOLTA*, vol. 10, no. 4, pp. 373–389, 2019. DOI: 10.1587/nolta.10.373

[12] A. Moore and C. Atkeson, “Prioritized sweeping: Reinforcement learning with less data and less real time,” *Machine Learning*, vol. 13, pp. 103–130, 1993.

[13] A. Farahmand and C. Szepesvári, “Model selection in reinforcement learning,” *Machine Learning*, vol. 85, no. 3, pp. 299–332, December 2011. DOI: 10.1007/s10994-011-5254-7

[14] V. Mnih, K. Kavukcuoglu, D. Silver, et al., “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, pp. 529–533, February 2015. DOI: 10.1038/nature14236

[15] D. Silver, R.S. Sutton, and M. Müller, “Sample-based learning and search with permanent and transient memories,” *The 25th International Conference on Machine Learning*, pp. 968–975, 2008. DOI: 10.1145/1390156.1390278

[16] K. Wakabayashi, “Use of high-level synthesis to generate hardware from software - Another alternative general-purpose program-executing mechanism to the CPU,” *IEICE*, vol. 6, no. 1, pp. 37–50, July 2012.

[17] M. Heger, “Consideration of risk in reinforcement learning,” *The 11th International Conference on Machine Learning*, pp. 105–111, July 1994. DOI: 10.1016/B978-1-55860-335-6.50021-0

[18] M. Riedmiller, “Neural fitted Q iteration - First experiences with a data efficient neural reinforcement learning method,” *In Machine Learning: ECML 2005*, pp. 317–328, 2005.

[19] Z. Wang, T. Schaul, M. Hassel, et al., “Dueling network architectures for deep reinforcement learning,” *The 33rd International Conference on Machine Learning*, pp. 1995–2003, 2016.

[20] D. Choi, C.J. Shallue, Z. Nado, et al., “On empirical comparisons of optimizers for deep learning,” *ICLR 2020 Conference Blind Submission*, June 2020.