

Basic Reproduction Method with Accurate Reproduction of Deep Q-network

Ver 1.0 17 June 2018

Shigeru Matsumura

© 2018 Elliott Trade Laboratory

Abstract

Although Deep Q-network (DQN) (Mnih et al., 2015) was published in February 2015, there are opinions in the reality as of 2018 that it is difficult to reproduce the Reinforcement learning algorithm (Rahtz, 2018).

In order to solve the difficulty that arises in reproducing Reinforcement learning application, I recommend applying Basic Reproduction Method (BRM) which I suggest. BRM shows the direction to which the reproducer should go by explaining the basic knowledge and techniques necessary for reproducing the application program with neural network(s).

In this article I explain BRM and show that DQN3.0 (an implementation of Mnih et al., 2015) (DeepMind, 2017a) was reproduced accurately as a result of applying BRM.

1. Introduction

In April 2017, I thought about applying Deep learning and Reinforcement learning to solve my own problems, so I started learning about that programming method. The best and fastest way to learn programming: By studying the source code of the programs with work well to solve the same problem, understanding its mechanism and learning how to write the code. In Reinforcement learning, DQN proposed together with its source code is best teaching material. And it became my first work to reproduce DQN3.0 created using Lua and Torch using Python and Tensorflow.

However, I spent about 8 months (about 944 hour; 118 work days(8h/d)) achieve to accurately reproduce DQN3.0: Ultimately, I achieved accurately reproduce DQN3.0 with exactly the same creation method as DQN3.0 in the last 3 months; but the first about 5 months were wasted by selected wrong reproduction method—but more generally fashion—as follows:

Wrong environment was selected. First I selected an environment that *Breakout-v0* of OpenAI Gym (Gym) (Brockman et al., 2016). Because I regarded that Gym is a standard Reinforcement learning environment with the environment collection and the algorithm evaluation platform. But it returns different frames than alewrap (DeepMind, 2014) used by DQN3.0 because it leads to difficult to implement the *Preprocessing* described in Mnih et al., 2015 P.6 and its probability distribution is different from DQN3.0 due to random frame-skipping. However, I did not perceive that Mnih et al., 2015 implemented the *Preprocessing* in alewrap, because alewrap was provided separately from DQN3.0 package and where these are implemented was not explained nowhere. **Wrong implement**

method of DQN loss-function was selected. The implementation of the DQN loss-function and optimizer in DQN3.0 is very strange compared to the implementation of the loss-function and optimizer of the general Deep learning application. There was no document describing such an implementation method anywhere. Although I do not know the reason why the method of implementing extremely important algorithms like DQN has not been explained even two years after the published, I have a hypothesis: DQN was too innovative, but we left behind Mnih et al., 2015 because there is no concrete explanation on how to implement innovative DQN loss-function in DQN3.0: Mnih et al., 2015 might considered that since DQN is a variant of Q-learning, many people can understand its semantics with knowledge of the basics of Reinforcement learning key-techniques when they see expression of DQN loss-function in paper and source code of DQN3.0, but unfortunately the fact was not as expected by Mnih et al., 2015—rather, many people may have been understand that DQN applying the neural networks containing some convolution layers, which is commonly used for image recognition problems with many successful cases, is a just image classifier which classifies actions with respect to an Atari game screen.

Wrong functions provided by Deep learning framework and library was selected. The RMSprop used in DQN3.0 is a special one created by Mnih et al., 2015 from scratch, but in their paper only “see Hinton et al., 2012” was explained (Mnih et al., 2015 P.6). For this reason, first I selected RMSprop with “not centered” provided by Tensorflow, but I understood that it is wrong as study progresses. On the other hand, DQN3.0 use `image.scale()` which resizing an image by interpolation “bilinear” as default (DeepMind, 2017a dqn/Scale.lua#L25), for resizing of Atari game frames. So I selected `cv2.resize()` provided by OpenCV2 with interpolation “`cv2.INTER_LINEAR`” which the same interpolation as “bilinear” of `image.scale()`'s default I thought, but I found that it result is different from `image.scale()`'s result. In addition, I was confused when comparing the performance of my DQN with DQN3.0. Because the performance after learning the Breakout in DQN3.0 was very different from result described in Mnih et al., 2015 P.11 Extended Data Table2. I considered that one of the reason of this is that it may be affected by `nn.SpatialConvolution` modified from the currently unavailable `nn.SpatialConvolutionCUDA`¹ (DeepMind, 2017a dqn/convnet.lua#L19).

Wrong evaluation method was selected. The other one of the reason of I confused in comparison of my DQN with DQN3.0 is: Since the evaluation method implemented in DQN3.0 can not accurately evaluate the test result of DQN including large variance, it was difficult to confirm whether my DQN was accurately reproduced.

It is difficult to build the running environment of some DQN

¹ `nn.SpatialConvolutionCUDA` was published by Chintalain, 2015, but I could not install it because I could not know how to install it.

implementation. In order to evaluate the implementation of various DQNs, I attempted to get the source code and build the running environment, but in some implementation, I could not complete it, in the face of some problems: First, too old libraries to use will not work together with the recently libraries. Second, since the source code is written corresponding to the API of the old-version library, it was necessary to get the old-version of the library or to modify the source code corresponding to the latest-version. Finally, the installation procedure is complicated.

However, fortunately, I had the knowledge and techniques to solve them from about 28 years of programmer experience. By using them to resolve the problems one by one, I eventually achieved accurate reproduction of DQN3.0. So I will summarize these knowledges and techniques as Basic Reproduction Method (BRM). BRM consists of knowledge and techniques used in migrating general application program obtained from my experience, and knowledge and techniques specific to reproduction of application with neural network(s). I think that BRM born from the accurate reproduction work of DQN3.0 is a fundamental method to work effectively even in the reproduction scenes of other various proposals.

In the rest this article, in *Section 2* explains how to recreate BRM, in *Section 3* and *Section 4* explains how to recreate DQN 3.0 using BRM, and shows DQN3.0 could be reproduced accurately as a result.

Please note that there are two subjects in this article: One is explanation of BRM; the other one is explanation of accurate reproduction method of DQN. So, if you want to read about the latter case only you can read *Section 3* and *Section 4*.

2. Basic Reproduction Method

BRM consists of four parts and iterates them until the same result as the original work is obtained, as shown in **Figure 1**: *Section 2.1* explains the importance of understanding the basic principles of programming languages—however, regardless of the field, all programmers must first understand this basic principle—. *Section 2.2* explains the importance of understanding semantics and how to do it. *Section 2.3* explains how to reproduce functions similar to those used in original work. Note that there is no ordering in *Section 2.2* and *Section 2.3*—both sequential section numbers are only for identification—. Finally, *Section 2.4* explains the importance and method of verifying that our work accurately reproduced the original work.

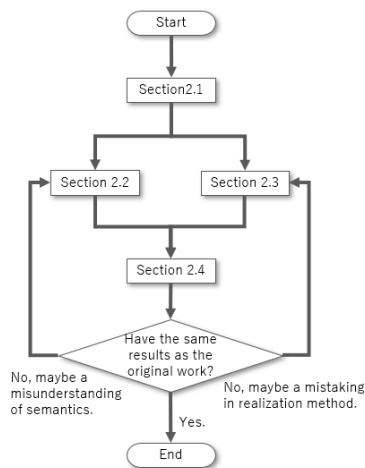


Figure 1: Flowchart of BRM.

2.1. Understand principle of programming language

The programming language has unique syntax so there is different how to write of source code but finally executed by same CPU using machine language compiled from source code. Therefore if it is possible to compile the same machine language from source code written using different programming language, the CPU will return the same computation result as them. It seems that this can be understood intuitively with very simple logic like $1 + 1 = 2$. However, even with complicated program with many functions like DQN, they are running on CPU under the same principle.

In the development of application program, improvement of computer performance in recent years has made it possible to write only specialized source code for our problem solving by using high-level scripting language such as Python, and it is no longer necessary to write source code that directly controls the hardware and executes more efficiently using low-level programming language such as assembly language and C language; at the same time, the high-level scripting language hidden “compile” from the programmers to convert source code to machine language. As a result, it may not be necessary to have the intuition that “the programming language is ultimately compiled into a machine language and executed by the CPU”, but this intuition, when confronted with difficulty in reproducing the proposal, it will ultimately make it possible to maintain motivation.

In addition, we must not forget that the CPU will be executed according to the source code we wrote, regardless of our wishes: Humans sometimes make mistakes, but the CPU executes human order accurately without any mistakes—regardless of whether the returns wished by humans—. If the result returned by the CPU is different from what we wished, there is a cause somewhere in the source code we wrote—debugging work will not be completed forever if we think “I am right”. It may be a painful work to find the mistakes inside of me, but I believe that many great achievements are the result of overcoming similar pain more—.

2.2. Understand the semantics of original work

The semantics is a specification of map that the relationship between input and output. And the semantics gives a guarantee of the relationship between inputs and outputs: Java as a programming language defined the specification as semantics called “Java Platform, Standard Edition” (Java SE) (ORACLE, 2018) and this guarantees the users that one Java source code will return the same result among multiple Java virtual machines created by different vendors.

So even simple expressions such as $1 + 1 = 2$ can not be reproduced accurately if implemented without understanding its semantics. For example, if we want to reproduce $1 + 1 = 2$ with semantics “the addition of number of fruits” using different programming language, we need to reproduce it with semantics “the addition of number of fruits” similarly rather than “the addition of number of apples”. Because, if we assumed “the addition of number of apple” to semantics of our $1 + 1 = 2$, one apple+one apple=two apples is OK, but one apple+one orange, may throw an illegal argument exception for one orange, or obtain unexpected results.

Therefore it is very important to understand the semantics of original work and apply it to our reproduced work. In order to understand the semantics, we firstly read the specifications and references. However, they may be thick documents consisting of many pages, reading it may be tedious and painful work, but there is no shortcut to get accurate understanding without going through that work.

2.3. Examine the reproduction method of realization method of semantics in original work

In the reproduction of $1 + 1 = 2$, if we use a subtract function rather than an addition function, we can not obtain result we expected. Similarly, if we use a addition function but it return 2.2 when given 1 and 1, we can not obtain result we expected. That is, considering what kind of function should be use is to examine which function has the same semantics as realization method in oritinal work. The way of know whether a function has the semantics we expected:

See API document

but unfortunately many API document described only simple explanation for function, that is, it did not explain actually implementation in detail. And, even if the same function, the argument name will be different if the library that provides it is different. In order to specify the arguments correctly, we need to know how it is used, but it is difficult to read it exactly from the API document. Therefore, after all it has to be examined in the way described below. **See the source code of function**

This way can accurately know how to implement it but may be more difficult: Libraries in high-level programming languages usually have two parts: Interface part for improving the convenience of users written in the same programming language; The Parts written in C/C++ for efficient program execution (more speed; reduce memory use). For this reason, you will need skills to read and understand both the programming language you use and C/C++.

In addition, in the case of Python, some source file is generated at the time of installation, so in the repository you clone from such Github you may not find the source file containing the source code of the function you want to examine. So you need to search both repository and package installation directory.

Finally, in large libraries such as Deep learning frameworks, the polymorphic structure for the abstraction of functions make it difficult to read and understand. In order to read and understand such programs, you need to learn program abstraction techniques—however, library source files are the best textbooks made by fully exploiting the abilities of the programming language, so reading and understanding them is useful for learning how to write the programming language you use—.

Comparing output between original function and select one

In this way, comparing of simple function like $1 + 1 = 2$ is very easy that just compare the scalar return value when the same input is given, but if return is data object(s) like a tensor with multiple elements, it become time-consuming work. In this case, as shown in **Figure 2**, it is one of the easiest ways to copy the output generated in context B to context A through a binary file, then compare both outputs. However, note the following: **Consider that the floating point value of programming language is an approximate value**: that is, they are not completely equal. In that case ignore smaller values (e.g. ignore less than 5 decimal places). **Consider the influence of cuDNN**: When the estimation by the neural network is executed on GPU, even if the random seed is fixed due to the influence of cuDNN, the output with respect to the same input is slightly different everytime. In this case, when estimated on the CPU, the output with respect to the same input are the same everytime. However, there are points to keep in mind when running on the CPU: Naturally it takes more time than

GPU, so the number of trials will be less. As a consequence, it is necessary not to mind short-term errors caused by ignorable differences in implementation of framework and library. Because this kind of error converges to the optimum average value by law of large numbers (Wikipedia, 2018a) when the number of trials

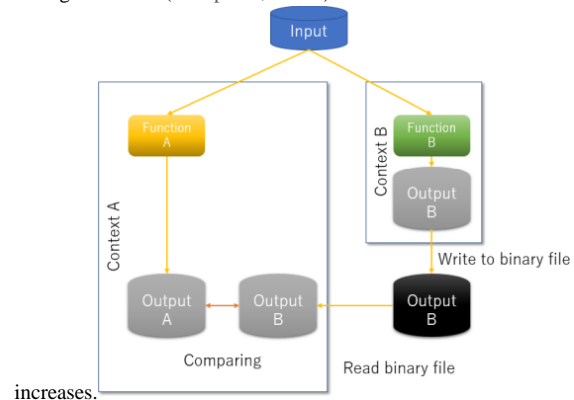


Figure 2: Example of comparison of outputs between function A and function B.

2.4. Verification of reproduced work

Finally we need to verify that the semantics we understand and the applications we reproduce are the same as the semantics and applications of the original work: If we do not pass this, we misunderstood the semantics of the original work, and we have made a different application from the original work; Although we can terminate the reproduction in this state, we should not go further (e.g. publish it as reproducing the original work; apply original work to different problems; further work on the original work the study), although you can finish the reproduction work in this state, proceeding to the next stage in that state may lead to undesirable situations (e.g. misunderstanding of original work diffuses due to published reproduced work; expected solutions can not be obtained; research on wrong direction). However, for various reasons that we do not want, there are cases where we have to terminate without accurate understanding and reproduction (e.g. Lack of time; Lack of cost). In that case, due to the problems caused by it, we need to return to the previous stage again—and if it has already been published, it is desirable to fix it to a more accurate reproduction work—. Therefore, when using work that reproduces the original work being published, it is necessary to verify it in the same way.

"understand" and "Can accurately reproduce" are two sides of the same coin: "We understand the original work" means "We can reproduce the original work accurately"; "We can reproduce the original work accurately" means "We understood the original work". However, the following cases exist: even if it no can be reproduced it can be understood; in contrast, even if you do not understand it can be reproduced (this is my actual case). Therefore, in order to obtain accurate understanding and reproduction, it is necessary to carefully search out the missing parts by verifying the reproduced work.

What the meaning of "Understanding"

The explanation of how to verification of reproduced work begins with explaining this. Because it is necessary to debug our understand and reproduce. Also, the same can be said for reproduction.

"Understanding" means that the two set of knowledge (the set of original knowledge; the set of knowledge that reproduced original

knowledge) is the same: Let's consider the context "Mr. Shigeru understands A as B ". Where A is a set of knowledge of original work; B is a set of knowledge on A understanding by Shigeru; and assume that each set contains knowledges, a, b, c and a relationship R^{ab} ; R^{ab} denote for knowledge a being related with knowledge b (e.g. a knowledge a is based on knowledge b ; that is, understanding of knowledge b is essential for understanding knowledge a).

In this case, if Shigeru accurately understands A , $A = B$ holds as shown in **Figure 3-(1)**, that is, individual knowledges and a relationship in A and B are completely equal. On the other hand, as shown in **Figure 3-(2)**, knowledge b lacks in B , that is, $A \neq B$, in this case, we can not say that Shigeru understood A . In addition, as shown in **Figure 3-(3)**, also we can not say that Shigeru understood A because there is no relationship R^{ab} , that is, $A \neq B$. In this case, there is a possibility that Shigeru can not have the same relationship R^{ab} due to misunderstanding of knowledge a and/or knowledge b .

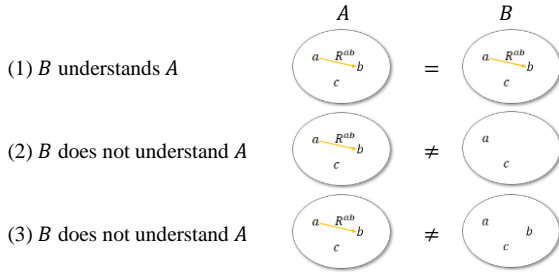


Figure 3: Visualization of understand A as B . Verification of the understanding of original semantics

Verification of the understanding of the original semantics is whether we can explain all the knowledge of the original semantics with reasonably and easily understandable using the knowledge we understood in our words—It is desirable not to use technical terms: If we do not understand, there is a part that can not be explained, or there is a part that is explained vaguely (**Figure 3-(2)**); also, if the original semantics has knowledge with relationships, we can not say we understood accurately when we can explain each knowledges without relationships (**Figure 3-(3)**).

However, the final verification of the correctness of the understanding of the original semantics obtained by the reproducer is possible only by the author of the original work, but that is not a realistic task. So, the reproducers need to work with good sense.

Verification of reproduced application program

The verification of the reproduced application program compares the output of the original application when the same input is given with the output of the reproduced application program (Training curve; Best score; etc.). If the reproduced application is accurate, its training curve approximates the training curve of the original application (However, as mentioned above, there is nothing to be perfectly matched). When not approximating, the following causes are considered: It may not be accurately reproduced; evaluation method may not be suitable.

In the former case: We will again compare the output of each function of the reproduced application with the output of the function of the original application again separately in order to find a function which can not be accurately reproduced; or search for the function of the original application not implemented in the reproduced application; we doubt the possibility of implementing it using the incorrect method due to the misunderstanding of

semantics.

In the later case: Try increasing the number of training steps, because, the actual convergence point may be further behind; if we are using the evaluation method described in the original paper, we will review the evaluation method: In this case, it is necessary to be able to fully explain the part where the evaluation method of the original paper is inappropriate; otherwise it may be worse for evaluation method which is convenient for us with inherently not suitable.

3. Reproduction method of Deep Q-network

This section explains reproduction method of DQN based on BRM.

3.1. Understand the semantics of Deep Q-network

During iteration based on the BRM flowchart, I noticed that the semantics of DQN has two large knowledges: One, DQN is an application created to actively solve Atari2600. Other one, DQN is a novel variant of One-step Q-learning. In addition, we need to understand their background as related knowledges. So, to explain the semantics of DQN, three parts are required as follows: First, in *Background of Deep Q-network*, I will explain the basics key technique of Reinforcement learning and the environment of Atari2600 which indispensable for understanding DQN. Second, *The semantics as an actively solver of Atari2600 domain of DQN* focuses on functions specialized in solving the Atari2600 domain of DQN. I think that this part is a part that may be replaced when applying DQN to other problems. Finally, *The semantics as a novel variant of One-step Q-learning of DQN* focuses on the feature as a variant of One-step Q-learning which is the root part of DQN.

Background of Deep Q-network

Here based on Sutton and Barto, 2018, I emphasize the fundamental understanding of key technique, which is indispensable for understanding DQN. In addition, I will clarify the prior knowledge necessary for selecting the environment to use in our DQN.

Finite Markov Decision Processes. Almost Reinforcement learning task can formulate the interaction between the environment and the agent as finite Markov Decision Process (MDP). The environment-agent interaction is represented the transitions that at each time step $t = 0, 1, 2, \dots, T$, the agent obtain the *state* s and *reward* r from the environment, then, the agent taken an *action* a by estimate with respect to a state s and given it to the environment, after that, the agent obtain s and r from the environment again, and these are continued until appear the *terminal state*. That is, these can be denoted as $s_0, r_0, a_0, s_1, r_1, a_1, \dots, s_T, r_T$ and $s \in \mathcal{S}, a \in \mathcal{A}(s), r \in \mathcal{R}$, where $\mathcal{A}(s)$ is set of possible actions in a state s . While the environment-agent interaction, the agent has learn that how to take actions aiming obtain maximum cumulative reward according to *policy* π . The cumulative rewards is denoted as follows:

$$R_t = \sum_{t'=t}^T \gamma^{t'-t} r_{t'}, \quad (1)$$

where γ is discount rate, $0 \leq \gamma \leq 1$. The policy π is probability that taken action a in s , that is

$$\sum_a \pi(a | s) = \sum_a p(a | s) = 1 \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s). \quad (2)$$

The environment has dynamics that conditional probability distribution of random variable next state s' and reward r depend only on taken action a in preceed state s as follows:

$$\sum_{s',r} p(s',r | s,a) = 1 \quad \text{for all } s \in \mathcal{S}, a \in \mathcal{A}(s), \quad (3)$$

where $p(s',r | s,a) \doteq \Pr\{s' = s_{t+1}, r = r_{t+1} | s = s_t, a = a_t\}$. Why does it depend only on the precedence (s,a) of the random variable (s',r) ? If the states has the Markov property, the state at the time step t , denote s_t , contains all the previous information s_0, s_1, \dots, s_{t-1} , that is, $p(s_t | s_0, s_1, \dots, s_{t-1})$. For example, in the role-playing game, the player's item-menu as a state contains unused items which player has acquired during the journey. On the other hand, the Hit Point (HP) as also a state, may has decreased as much as injured in the previous hardship battles. If the HP is full despite being injured in the previous battles, number of the healing item in the item-menu may be decreasing for restores the HP. Therefore, the player can be decide whether it is necessary to restore HP by looking at only the current HP, and if it is necessary to restore the HP, it can be decide whether it is possible to restore by the healing item by looking at only the current item-menu, that is, these current states determines the probability distribution of the next state. Therefore, the current state consists of all previous states so maybe all states in an episode are unique.

In order to achieve obtain maximizing cumulative rewards, the agent requires select a best action in a state. The meaning of best action in a state is an action which can arrive at the next state which has the next action with the maximum expected return. For that purpose, first of all, we needs estimate value of action: In MDP, it is formulated as the *Action-value function* as follows:

$$q_\pi(s,a) = \mathbb{E}[R_t | s_t = s, a_t = a] \quad (4)$$

$$= \mathbb{E}[r + \gamma R_{t+1}] \quad (5)$$

$$= \mathbb{E}[r + \gamma q_\pi(s',a') | s_{t+1} = s', a_{t+1} = a'] \quad (6)$$

$$= \sum_{s',r} p(s',r | s,a) \left[r + \gamma \sum_{a'} \pi(a' | s') q_\pi(s',a') \right], \quad (7)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. The Eq (7) is called *the Bellman equation* q_π , and $\gamma \sum_{a'} \pi(a' | s') q_\pi(s',a')$ is discounted the sum of expected returns for all action $a' \in \mathcal{A}(s')$. Expanded expression (8) of Eq (7) represents components of the action-value function:

$$q_\pi(s,a) = \begin{bmatrix} p(s^{(1)}, r^{(1)} | s,a) \left[r^{(1)} + \gamma \begin{bmatrix} \pi(a'_{[1]} | s^{(1)}) \cdot q_\pi(s^{(1)}, a'_{[1]}) \\ \pi(a'_{[2]} | s^{(1)}) \cdot q_\pi(s^{(1)}, a'_{[2]}) \\ \vdots \\ \pi(a'_{[|\mathcal{A}(s^{(1)})|]} | s^{(1)}) \cdot q_\pi(s^{(1)}, a'_{[|\mathcal{A}(s^{(1)})|]}) \end{bmatrix} \right] \\ p(s^{(2)}, r^{(2)} | s,a) \left[r^{(2)} + \gamma \begin{bmatrix} \pi(a'_{[1]} | s^{(2)}) \cdot q_\pi(s^{(2)}, a'_{[1]}) \\ \pi(a'_{[2]} | s^{(2)}) \cdot q_\pi(s^{(2)}, a'_{[2]}) \\ \vdots \\ \pi(a'_{[|\mathcal{A}(s^{(2)})|]} | s^{(2)}) \cdot q_\pi(s^{(2)}, a'_{[|\mathcal{A}(s^{(2)})|]}) \end{bmatrix} \right] \\ \vdots \\ p(s^{(n)}, r^{(n)} | s,a) \left[r^{(n)} + \gamma \begin{bmatrix} \pi(a'_{[1]} | s^{(n)}) \cdot q_\pi(s^{(n)}, a'_{[1]}) \\ \pi(a'_{[2]} | s^{(n)}) \cdot q_\pi(s^{(n)}, a'_{[2]}) \\ \vdots \\ \pi(a'_{[|\mathcal{A}(s^{(n)})|]} | s^{(n)}) \cdot q_\pi(s^{(n)}, a'_{[|\mathcal{A}(s^{(n)})|]}) \end{bmatrix} \right] \end{bmatrix} \quad (8)$$

where n is number of outcome of random variable (s',r) . These expressions denote that the action-value function is the immediate reward plus the sum of next action-values, for all next state and reward in current state and action, but these are weighted their

probability as environment dynamics.

Next, we needs find a best value of action from $q_\pi(s,a)$: As above, $q_\pi(s,a)$ contains all value of possible next actions in a next state. Among them, we takes a maximum next action value as a best next action value of a next state. However, if the environment has dynamics which returns multiple next states and rewards for an action the agent has taken, the best next action value is the sum of the next maximum action values, still weighted by environment dynamics $p(s',r | s,a)$ as follows:

$$q_*(s,a) = \max_{\pi} q_\pi(s,a) \quad (9)$$

$$= \sum_{s',r} p(s',r | s,a) \left[r + \gamma \max_{a'} q_*(s',a') \right], \quad (10)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. Where q_* is called *optimal action-value function* and also called *the Bellman optimal equation for q_** , and represents one optimal action value in a state. That is, expanded expression of Eq (10) is:

$$q_*(s,a) = \begin{bmatrix} p(s^{(1)}, r^{(1)} | s,a) \cdot r^{(1)} + \gamma \max_{a' \in \mathcal{A}(s^{(1)})} \left\{ \begin{bmatrix} q_\pi(s^{(1)}, a'_{[1]}) \\ q_\pi(s^{(1)}, a'_{[2]}) \\ \vdots \\ q_\pi(s^{(1)}, a'_{[|\mathcal{A}(s^{(1)})|]}) \end{bmatrix} \right\} \\ + \\ p(s^{(2)}, r^{(2)} | s,a) \cdot r^{(2)} + \gamma \max_{a' \in \mathcal{A}(s^{(2)})} \left\{ \begin{bmatrix} q_\pi(s^{(2)}, a'_{[1]}) \\ q_\pi(s^{(2)}, a'_{[2]}) \\ \vdots \\ q_\pi(s^{(2)}, a'_{[|\mathcal{A}(s^{(2)})|]}) \end{bmatrix} \right\} \\ + \\ \vdots \\ + \\ p(s^{(n)}, r^{(n)} | s,a) \cdot r^{(n)} + \gamma \max_{a' \in \mathcal{A}(s^{(n)})} \left\{ \begin{bmatrix} q_\pi(s^{(n)}, a'_{[1]}) \\ q_\pi(s^{(n)}, a'_{[2]}) \\ \vdots \\ q_\pi(s^{(n)}, a'_{[|\mathcal{A}(s^{(n)})|]}) \end{bmatrix} \right\} \end{bmatrix} \quad (11)$$

Now we could obtain action value for one of possible action a in state s . After that, we need to obtain value of action for all rest of possible actions in the same way. And finally, we take a index of maximum $q_*(s,a)$ from among them as a best action:

$$\text{The best action } a \text{ in } s = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} \begin{bmatrix} q_*(s, a_{[1]}), \\ q_*(s, a_{[2]}), \\ \vdots \\ q_*(s, a_{[|\mathcal{A}(s)|]}) \end{bmatrix} = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} v_\pi(s), \quad (12)$$

where $v_\pi(s)$ is *state-value function*. Surprisingly, we also could obtain value of state.

As above, the optimization of action-value function is computed for s and a for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ individually, that is,

the Bellman optimality equation is actually a system of equations, one for each state, so if there are n states, then there are n equations in n unknowns.

Sutton and Barto, 2018 P.50~51

This contrasts with the supervised learning to optimize only one function as a single model.

Q-learning. In real situations, for achieving our purpose, sometimes we must take an action can arrive at next desirable situation which has very low occurrence probability. For example, in the role-playing game, some time only a player who has a special item with very low occurrence probability can achieve their purpose. In this case, these algorithms influenced by environmental dynamics may not be able to solve the situation because they cannot obtain special items underestimated its value by scaling using environmental dynamics. In addition, in real situations, we may not be able to obtain fully transitions like game record of official match of board game such as Go, Chess and

Shogi. The off-policy Temporal-Difference Learning (TDL) algorithm called *Q-learning* can solve that situation because it estimates only the value of the next action in a next state due to the selected an action in a current state, therefore, not influenced by environment dynamics and not required fully transitions.

These computation are formulated as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (13)$$

where α is step size parameter called *learning rate*. At length, we cannot find environment dynamics in Eq (13), so the agent with Q-learning can be select an action with maximum future returns even if it occurrence probability is very lower.

Policy. Q-learning is Off-policy TDL, so it uses two policies, the *target policy* and the *behavior policy*: The target policy is more optimal or maybe deterministic. It used for optimal target of learning; On the other hand, the behavior policy is more stochastic: It used for exploration of transitions, while optimized itself toward to target policy. For the behavior policy, often use ϵ -greedy policy, because, it behavior is changed from the almost random selection to the almost deterministic selection, using a parameter epsilon stochastically step by step. It is because, in early stage, the agent should be more exploration for find to better transitions: If the agent use deterministic policy with unoptimal at this time, its exploration range became narrow by the agent selects only unoptimal fixed actions, so learning does not proceed; In contrast, if the agent use only stochastic policy, it is too difficult to arrive at the state where on deeply path in transitions of the more complex task; These are fatal problems for TDL which propagates value only from the next state.

In addition, it is also important to consider the tie-breaking for behavior policy. The tie-breaking selects one action from multiple maximum action values. If the agent does not use it, the behavior policy always select one action with smaller index even if it become unoptimal action when there are multiple maximum action values.

Environment of the challenging domain of classic Atari2600 games. While study of the challenging domain of classic Atari2600 games, in most cases one of the two directions is chosen: First one is, they use the atari env provided by Gym; The other one is, they use Arcade Learning Environment (ALE) (Bellemare et al., 2013) directly or extended by themselves.²In the former case, under the environment defined by OpenAI, we can compare our algorithm and its implementation with other algorithms to solve the same problem: Gym has concept that provides the benchmark collections which corresponds various problems, and to equitably evaluate various algorithm which aims to solve these problems (Brockman et al., 2016. P.1 Introduction, P.2 Design Decisions); In order to realize their concept, they implemented the AtariEnv named ‘*GameName-v0*’, (e.g. ‘*Breakout-v0*’) which is generally used has the frameskip with skip k frames, $k \sim \mathcal{U}(\{2, 3, 4\})$ and `repeat_action_probability`³ with 0.25. On the other hand, several AtariEnv corresponding to other uses are also available: For example, one of the AtariEnv named ‘*GameNameDeterministic-v4*’, (e.g. ‘*BreakoutDeterministic-v4*’) has deterministic frameskip with $k = 4$ or 3 (only *Space Invaders*) and `repeat_action_probability` with 0; other one of the AtariEnv named ‘*GameNameNoFrameskip-v4*’, (e.g. ‘*BreakoutNoFrameskip-v4*’) is get

every frames and `repeat_action_probability` with 0.

In the latter case, ALE provides low-level APIs that are also used by Gym’s AtariEnv through atari-py (OpenAI, 2017), so we can use a lighter environment than AtariEnv and also can extend it as we want.

The semantics as an actively solver of Atari2600 domain of DQN

DQN was developed to achieve that a single algorithm can be develop competencies for multiple problems (Mnih et al., 2015 P.1). For that purpose, Mnih et al., 2015 select Atari2600 and devised DQN to have the ability to play various games well. Among those ingenuity is preprocessing of information obtained from Atari 2600. Here I will explain the information preprocessing technique of Atari 2600 devised by Mnih et al., 2015.

Preprocess. Mnih et al., 2015 have found that in several Atari2600 games, important objects are lost due to blinking sprites or being drawn only in certain frames (DeepMind, 2014 alewrap/GameScreen.lua#L24-L30). And it can be demanding in terms of computation and memory requirements when use raw pixel images that has shape $210 \times 160 \times 3$. Because of this, Mnih et al., 2015 applies the six step preprocess to the raw pixel images for create better input data for learning and prediction. Let denote \tilde{t} is actual time-step in environment, and the environment returns a RGB frame at time-step \tilde{t} , denote $x_{\tilde{t}} = \{x_{\tilde{t}}^R, x_{\tilde{t}}^G, x_{\tilde{t}}^B\}$. (a)

Normalize: First, all frames are normalized as $\hat{x}_{\tilde{t}}$

$$\hat{x}_{\tilde{t}} = \frac{x_{\tilde{t}}}{255} \quad (14)$$

(b) Maximization: they take the maximum value between $\hat{x}_{\tilde{t}}$ and $\hat{x}_{\tilde{t}-1}$ as $\hat{x}_{\tilde{t}}$

$$\hat{x}_{\tilde{t}} = \max\{\hat{x}_{\tilde{t}-1}, \hat{x}_{\tilde{t}}\}. \quad (15)$$

(c) Frameskip: then, they uses simple frame skipping technique that obtain frames every 4 times

$$\begin{aligned} \hat{x}_{t-3}, \hat{x}_{t-2}, \hat{x}_{t-1}, \hat{x}_t &= \hat{x}_{t-12}, \hat{x}_{t-8}, \hat{x}_{t-4}, \hat{x}_t \\ &= \max\{x_{t-13}, x_{t-12}\}, \max\{x_{t-9}, x_{t-8}\} \\ &\quad, \max\{x_{t-5}, x_{t-4}\}, \max\{x_{t-1}, x_t\} \end{aligned} \quad (16)$$

where t is time-step seen from the agent. The rest in this article, all t is in that meaning. Note that, more precisely, when terminal state is occurred during frame skipping, break the frame skipping immediately; thus, the number of frame skip is not always 4.

(d) Grayscale conversion: then, they extract the Y channel (it is called luminance) from \hat{x}_t as \check{x}_t (it is generally called the grayscale conversion)

$$\check{x}_t = 0.299\hat{x}_t^R + 0.587\hat{x}_t^G + 0.114\hat{x}_t^B. \quad (17)$$

(e) Resization: then, they resize \check{x}_t to 84×84 with the bilinear interpolation as s_t

$$s_t = \text{resize}^{\text{Bilinear}}(\check{x}_t, 84, 84). \quad (18)$$

(f) Stack frames into ϕ : As better input data ϕ_t for learning and prediction, they stacks the four recentry s_t

$$\phi_t = \{s_{t-3}, s_{t-2}, s_{t-1}, s_t\}, \quad s_t \neq s^+, \quad (19)$$

² In some implementations RL-Glue (Tanner and Write, 2009) was used, but development may have been terminated in 2009.

³ `repeat_action_probability` is a probability that repeat a precede action even if a current action is not equal to a precede action.

where s^+ is terminal state, that is, the agent does not learn and predict from terminal state. And now, ϕ_t has shape that $4 \times 84 \times 84$ or $84 \times 84 \times 4$. In addition, if the terminal state is exist in s_{t-3} , s_{t-2} or s_{t-1} , they replaces all s in episode contain s^+ in Φ_t to null state, denote s^{null} , that means that state is filled with zero. For example, if ϕ_t contains the terminal state occurred at $t - 2$, denote

$$\phi_t = \{s_{t-3}, s_{t-2}^+, s_{t-1}, s_t\}, \quad (20)$$

in this case, ϕ_t contains states occurred in two different episodes: One is episode⁻¹ contained s_{t-3} and s_{t-2}^+ (that is, episode⁻¹ is terminated at $t - 2$); and other one is episode⁰ contained s_{t-1} and s_t (that is, episode⁰ is started at $t - 1$). Let add the episode number in square bracket subscript to (20) as follows:

$$\phi_t = \{s_{t-3}^{[-1]}, s_{t-2}^{+[-1]}, s_{t-1}^{[0]}, s_t^{[0]}\}. \quad (21)$$

Then, to ignore the states in episode⁻¹, by replace s_{t-3} and s_{t-2}^+ to s^{null}

$$\phi_t = \{s^{\text{null}[-1]}, s^{\text{null}[-1]}, s_{t-1}^{[0]}, s_t^{[0]}\}. \quad (22)$$

Thus, their agents will learn and predict with preprocessed states ϕ_t only during survival.

Finally, **(g) start each episode randomly**: Except for the first episode of training, each episode is randomly started skipping frames between 1 and 30 steps.

In DQN3.0, these codes written separate source code files: (a)~(c) and (g) are written in GameScreen.lua in alewrap (DeepMind, 2014) package; (d), (e) both written in scale.lua in DQN3.0 package; and (f) is written in TransitionTable.lua in DQN3.0 package.

Episode separation for training. In the game of Atari 2600 there is a ‘‘Lives counter’’ that indicates how many more times player can fail. For example, in the case of the Breakout, firstly, it given 5 lives for player. If player cannot hit a ball with a paddel, then 1 Lives will be loss, and when 5 Lives are lossed, then the game is over. In this usual case, as shown in **Figure 4-** (a), the agent also learns these actions selected toward to the loss of Lives during journey to end of the game, because these actions are also given the value by the backup of value. In order to avoid this, Mnih et al., 2015 separated the episode by replacing a normal state to a termination state at the loss of Lives. Thereby, as shown in **Figure 4-**(b), the agent does not learn these actions with no value which selected toward to the loss of Lives. However, the agent recognizes by the Lives count obtained by the function of xitari (ALE extended) (DeepMind, 2017b) instead of the Lives counter on the game frame. That is, more precisely, their agent is observed not only the pixels and the game score. This function was implemeted in alewrap.

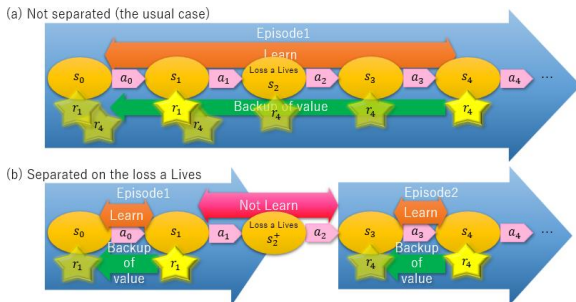


Figure 4: Comparison of learning range with or without episode separation. (a) In the case of the episode which not separated as default, the sequences reaching failure are learned by

backup of the future reward r_4 . (b) In the case of the episode separated on the failure and where s_2^+ is the terminal state which replaced from s_2 , the action a_1 is not learned because the reward r_4 occurred in the episode 2 is not backuped to s_2^+ as is not the future reward for s_2^+ .

The semantics as a novel variant of One-step Q-learning of DQN

DQN is an epoch-making algorithm that recognizes the game screen using a neural network that imitates the object recognition in the receptive field, and reactivates the trajectory of memorized experience using the replay memory that imitates the hippocampus, and then learning under One-step Q-learning. The distinction between target prolicy and behavior policy as an off-policy algorithm and random sampling of replay memory helped stabilize the algorithm. Also, the method of updating Q-fucntion has been greatly changed by using the parameters of the neural network in the space to accumulate the expected return. I will refer to those specifications here.

Transition Table as Replay memory. For the experience replay, Mnih et al., 2015 stores tuple of experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ obtained from transitions into a replay memory (called TransitionTable) $\mathcal{D}_t = \{e_{t-N}, e_{t-(N+1)}, \dots, e_t\}$, where $N \in \mathbb{R}^{[0, 1 \times 10^6]}$ is number of experiences. In fact, in DQN3.0, instead of it, used a tuple of experience, $e_t = (s_t, a_t, r_t, term_t)$, where $term_t$ is boolean represented whether state s_t is terminal or not. Note that, r_t is outcome of environment with s_{t+1} when given a_t estimated with respect to s_t , and is notation different from r_{t+1} denoted in Sutton and Barto, 2018. The replay memory is a large sliding window (Schaul et al., 2015) which represented a part of all transitions occurred during training, that is, stored recenry 1×10^6 experiences, and forget experiences earlier than $t - 1 \times 10^6$.

The TransitionTable cycles through the table and inserts a new experience, instead of use a queue. During first 50,000 steps of training, the agent is not learn but only stack experiences into TransitionTable. After that, the agent has learn from experiences sampled from TransitionTable. In learning iteration i th, the agent obtains a mini-batch \mathcal{E}_i from TransitionTable. TransitionTable will samples randomly the same number of experiences e' as mini batch size M

$$\mathcal{E}_i = \{e'_1, e'_2, \dots, e'_M\}, \quad (23)$$

where e' is a experience for learning,

$$e' = (\phi, a, r, \phi', term') \in \mathcal{D}_t,$$

$$(\phi, a, r, \phi', term') = (\phi_j, a_j, r_j, \phi_{j+1}, term_{j+1}),$$

$$j \text{ is sampled index of } \mathcal{D}_t, j \sim \mathcal{U}(2, |\mathcal{D}_t| - h),$$

$$h \text{ is number of frames in } \phi,$$

$$\phi_j = \{s_{j-3}, s_{j-2}, s_{j-1}, s_j\},$$

$$\phi_{j+1} = \{s_{j+1-3}, s_{j+1-2}, s_{j+1-1}, s_{j+1}\}.$$

As mentioned in *Preprocess (f)*, frame stacking would be applied to ϕ and ϕ' , but, only ϕ' allows including terminal state. Because, ϕ' is used for only estimate the target action value and the agent need to learn final reward in the episode.

Q-function update with neural network. As One-step Q-learning, DQN needs update each Q-functions to optimal. Recall equation of One-step Q-learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad \text{by (13)}$$

A simple Q-learning accumulates the state-action value in a scalar corresponding to a state-action pair, but this method is not practical for realistic problems: It is increased the number of scalars as the number of state-action pair increase. Mnih et al., 2015 tried to solve this problem by applying a deep neural network which familiar as an approach of the image classification

$$Q(s, a; \theta) \approx Q^*(s, a),$$

where θ is network parameter. Each Q-function shares θ , so there are as many Q-functions as the number of features.

In this case, the update formula of the Q-function was changed to the update formula for many elements corresponding to the features in the parameter, instead of a simple update expression between scalars. In addition, they need to accumulate state-action values of each actual state which has different history information as feature quantities corresponding a features within elements in the parameter. For this reason, Mnih et al., 2015 devised a novel update method: First, by Extremal property (Wikipedia, 2018b), to compute the average of gradient over all trials of each features from the squared TD error computed when each actual state is given. Second, to minimize the average of gradients using RMSprop of Hinton et al., 2012.

The Extremal property, $\mathbb{E}[X - c]^2$, $c \in \mathbb{R}$, is estimates a error between a target constant c and outcome of random variable X over all trials. Here the DQN loss function is defined in Mnih et al., 2015 as follows:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r} [\mathbb{E}_{s'} [y | s, a] - Q(s, a; \theta_i)]^2 \quad (24)$$

$$= \mathbb{E}_{s,a,r} [(y - Q(s, a; \theta_i))^2] + \mathbb{E}_{s,a,r} [\mathbb{V}_{s'} [y]] \quad (25)$$

$$= \mathbb{E}_{s,a,r,s'} [(y - Q(s, a; \theta_i))^2], \quad (26)$$

$$\text{with } y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-) \quad (27)$$

The optimal target y corresponds c of Extremal property, and $Q(s, a; \theta_i)$ corresponds random variable X of Extremal property. However, y contains a estimated value with respect to a next state, so y contains variance, therefore is not constant. I think that, Mnih et al., 2015 were considered can be regarded that y is constant by to extract and ignore a variance, $\mathbb{E}_{s,a,r} [\mathbb{V}_{s'} [y]]$, as shown Eq (25) (Mnih et al., 2015 P.7).

However, outcome of $y - Q(s, a; \theta_i)$ in a mini-batch at each iterations is a TD-error obtained from single trial.

$$\delta_i = \left\{ r^{(m)} + \gamma \max_{a' \in \mathcal{A}(s^{(m)})} Q(s^{(m)}, a'; \theta_i^-) - Q(s^{(m)}, a^{(m)}; \theta_i) \right\}_{m=1}^M \quad (28)$$

where δ_i is set of TD-error, and

$$(s^{(m)}, a^{(m)}, r^{(m)}, s'^{(m)}) = (\phi^{(m)}, a^{(m)}, r^{(m)}, \phi'^{(m)}) \in \mathcal{E}_i$$

The neural network does not has history of the outcome of all trials of each features but instead has a present value which is accumulated the part of the future return as gradient obtained in each trial so far. For this reason, I think, that Mnih et al., 2015 did consider to apply the exponential moving average of gradient, instead simple mean, for obtain average of gradient over all trials. Therefore, I will be able to write that a derivative function of DQN loss-function (Eq (24)) as follows:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} [(y - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i)] \quad (29)$$

$$= \aleph \nabla_{\theta_{i-1}, L_{i-1}(\theta_{i-1})} + (1 - \aleph) \sum_{m=1}^M \delta_i^{(m)} \nabla_{\theta_i} Q(s^{(m)}, a^{(m)}; \theta_i) \quad (30)$$

where \aleph is a gradient momentum described Mnih et al., 2015_P.10 Extended Data Table 1. Note that, Mnih et al., 2015 did not take the average over a mini-batch by mean squared error function such provided by the Deep learning framework, denote MSE^{SV}

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} [(y - Q(s, a; \theta_i))^2]$$

$$\neq \text{MSE}^{\text{SV}}(y, Q(s, a; \theta_i))$$

$$= \frac{1}{M} \sum_{m=1}^M \delta_i^{(m)^2},$$

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} [(y - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i)]$$

$$\neq \frac{d}{d\theta_i} \text{MSE}^{\text{SV}}(y, Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i)$$

$$= \frac{2}{M} \sum_{m=1}^M \delta_i^{(m)} \nabla_{\theta_i} Q(s^{(m)}, a^{(m)}; \theta_i).$$

Recall that each optimal action-value function exists as separate systems for all state-action pair. That is, in DQN, each models are individually learned with mini batch containing 32 data for learning 32 models corresponding to 32 state-action pairs. So, for DQN, we can not use MSE^{SV} which is designed for supervised learning, assuming that all mini batches has datas for learn one model. In addition, commonly, the derivative of mean squared error, is $\frac{2}{M} \sum_{m=1}^M (x^{(m)} - y^{(m)})$, so the derivative of Eq (29) should become $2 \mathbb{E}_{s,a,r,s'} [(y - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i)]$. However, since Eq (26) can be considered assuming One Half MSE (Ng, 2016), $\frac{1}{2M} \sum_{m=1}^M (x^{(m)} - y^{(m)})^2$, that is, its derivative is $\frac{1}{M} \sum_{m=1}^M (x^{(m)} - y^{(m)})$, so Eq (29) is not multiplied by 2.

Next, these are not describe in Mnih et al., 2015, but I think that they also devised a modified RMSprop of Hinton et al., 2012. I think that, Mnih et al., 2015 considered that to obtain the stability of the algorithm, it was necessary clip the TD error to be between -1 and 1 , and at the same time they thought that it is necessary to update the parameters using the gradient maintaining that range, because the gradient becomes positive values by the square of RMSprop of Hinton et al., 2012. For that reason, , Mnih et al., 2015 modified RMSprop of Hinton et al., 2012 to distribute the gradient around zero. Eq (29) was useful for that. Let $g_i = \nabla_{\theta_i} L_i(\theta_i)$ and $d\theta_i = \delta_i \nabla_{\theta_i} Q(s, a; \theta_i)$, the modified RMSprop of Hinton et al., 2012 can be write as follows:

$$g_i = \aleph g_{i-1} + (1 - \aleph) d\theta_i \quad \text{by (29)} \quad (31)$$

$$n_i = \aleph n_{i-1} + (1 - \aleph) d\theta_i^2 \quad (32)$$

$$\Delta_i = \beth \Delta_{i-1} + \lambda \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \gamma}} \quad (33)$$

$$\theta_{i+1} = \theta_i + \Delta_i \quad (34)$$

where λ is learning rate, γ min squared gradient, these are described in Mnih et al., 2015 P.10 Extended Data Table 1, and \beth is momentum, it is not described Mnih et al., 2015 but used in DQN3.0. Then, the equations very similar to RMSprop of Graves, 2013 appeared.

Finally, each equations corresponds source code of DQN3.0 as **Table 4**:

Policy. As One-step Q-learning, also DQN uses the behavior policy and the target policy: The behavior policy in DQN uses a network with parameter θ for estimate the action value. It also has tie-braking method; On the other hand, the target policy in DQN uses a target network with parameter θ^- for estimate the next action value. The parameter θ^- is copied from parameter θ at fixed intervals, and unchange it until next copy. In the meantime,

the parameter θ will be change by learning, that is, $\theta \neq \theta^-$. Therefore, it will be able to make two independed policies as the behavior policy and the target policy. This method was devised by Mnih et al., 2015 for the purpose of improving the stability of the algorithm.

3.2. Examine the reproduction method of realization method of semantics in DQN3.0

While large number of iterative updates, the One-step Q-learning accumulates the next value of state-action pair scaled by the step size parameter little by little to current value of state-action pair. So, even if there is a very small difference, by enlarging them during a large number of iteration updates, we get a very different result. For this reason, in order to improve the reproducibility of DQN3.0, it is necessary to consider that how to make it possible to obtain the same result as DQN3.0, but, it is not just using functions and parameters which named the same name and/or refer to the same paper. By try to reproduce DQN3.0 more accurately, we can be understood that having deeply examining the functions provided by the Deep learning frameworks and libraries that we are considering to apply to our application has comparable importance to the tuning of hyperparameters.

Note that, the version of Deep learning framework I used are shown **Table 1**. So, this article is based on these versions.

Table 1: The Deep learning frameworks used in this work. All framework test uses Tensorflow for visualization. And All DQN version correspond Python 3.5.

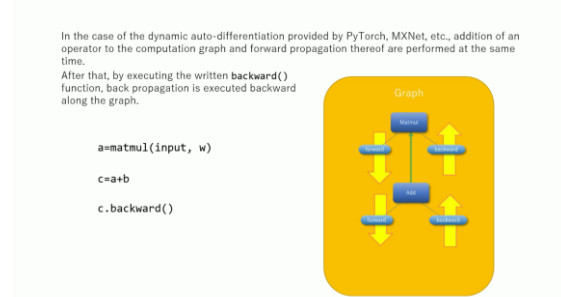
DQN version	Install libraries	CUDA ver	cuDNN ver
PyTorch(Legacy) and PyTorch(New)	PyTorch0.3.0.post4 Tensorflow1.4.1(CPU)	8.0	6.0
MXNet	MXNet 1.0.0 Tensorflow1.4.1(CPU)	8.0	5.1
Tnesorflow	Tensorflow-gpu1.4.1 Keras 2.1.2	8.0	6.0
CNTK	CNTK 2.3.1 Keras 2.1.2 Tensorflow1.4.1(CPU)	8.0	6.0

Realization of Q-function update in DQN

When we see the source of DQN3.0, we can find the implementation of the loss-function of DQN3.0 (DeepMind, 2017a dqn/NeuralQLearner.lua#L180-L277), which seems to be completely different from the common sense of modern Deep learning framework which has the automatic-differentiation. However, as mentioned in *Q-function update with neural network*, this unusual implementation is just a code that realized the update equation of Q-function (Eq.(28)~(34)) using neural network parameters devised by Mnih et al., 2015. Here, first I explain the common concept among the automatic differentiation functions of each Deep learning framework, understand that the same implementation is possible with the modern Deep learning framework, and then see how to implement in each Deep learning framework.

The concept of automatic-differenciatio. The Deep learning framework is commonly has one of the automatic differentiation which can be divided largely two kind of types: One of them, it requires definition and compilation of computation graphs before

execution. Such type is provided such as Tensorflow, CNTK, Theano, and called “static auto-differentiation” in this article; the other one is, it constructs the computation graphs and execute it on the fly. Such type is provided by such as PyTorch, MXNet and called “dynamic auto-differentiation” in this article. **Video 1** shows, these has common concept despite these differ in when the computation graph is constructed and how to execute it.



Video 1: A concept of the Automatic differentiation

When you watch **Video 1**, you may have a question that why does not use a loss function before the backpropagation and does not compute a scalar loss? Because the loss is just only indicates how good our network model, and a value of loss does not affect backpropagation. However, in most Deep learning frameworks emphasize writing as follows

```
# such as PyTorch, MXNet
loss.backward()
```

or

```
# Tensorflow
optimizer.minimize(loss).
```

This reason of that it is necessary to use a scalar loss computed by loss-function for backpropagation is only to make our aware of “to minimize loss” as a ceremony. In fact, the purpose of use a loss for backpropagation is just to obtain the backward function of the operation which was created a loss, as a function of first call in backpropagation, and give a scalar which has 1.0 (rather than a value of loss) as the default initial gradient. However, many Deep learning framework can be replace the default initial gradient to arbitrary initial gradient we want, and can be start backpropagation from arbitrary middle output we want, as follows:

```
# such as PyTorch, MXNet
middle_output.backward(init_gradient)
```

or

```
# Tensorflow
tf.gradient(middle_output, weights, init_gradient)4.
```

Therefore, we can create arbitrary derivative of loss-function like Eq (30).

Realization of DQN loss-function. For PyTorch and MXNet which has dynamic automatic-differentiation can write about the same as DQN3.0 as follows:

PyTorch(Legacy):

https://github.com/ElliottTradeLaboratory/DQN/blob/6ef872a83e1d618563339bd1ab1d21ab662cbd3b/dqn/pytorch_optimizer.py#L45-L129

PyTorch(New):

⁴ Note that, Keras provides `K.gradients()` but it cannot specify the initial gradient.

https://github.com/ElliottTradeLaboratory/DQN/blob/969e0e8eb54fdb18e9943f7b059aedf7fee00dc6/dqn/pytorch_optimizer.py#L232-L280

MXNet:

https://github.com/ElliottTradeLaboratory/DQN/blob/969e0e8eb54fdb18e9943f7b059aedf7fee00dc6/dqn/convnet_mxnet.py#L202-L246

For Tensorflow with Keras which has static automatic differentiation, it is necessary to write construction and execution of computation graph separately, but the description up to construction is about the same as PyTorch except use `tf.gradients()` instead of `backward()` as follows:

Tensorflow:

- (1) https://github.com/ElliottTradeLaboratory/DQN/blob/969e0e8eb54fdb18e9943f7b059aedf7fee00dc6/dqn/convnet_keras.py#L167-L250
- (2) https://github.com/ElliottTradeLaboratory/DQN/blob/969e0e8eb54fdb18e9943f7b059aedf7fee00dc6/dqn/tensorflow_extensions.py#L117

As Figure 7 shows, this reproduction method can be construct the computation graph having the same computation procedure as DQN3.0.

CNTK has static automatic differentiation, many computation procedure can be share with Tensorflow through Keras. But CNTK seems not to provide API which can be operate of the backpropagation procedures for Python users, so I applied the tricky way: Create a user function which provide a targets as initial gradient to backward function of bias-add operator of final layer as follows:

CNTK:

- (1) https://github.com/ElliottTradeLaboratory/DQN/blob/969e0e8eb54fdb18e9943f7b059aedf7fee00dc6/dqn/convnet_keras.py#L167-L250
- (2) https://github.com/ElliottTradeLaboratory/DQN/blob/969e0e8eb54fdb18e9943f7b059aedf7fee00dc6/dqn/cntk_extensions.py#L40-L59

Because CNTK does not provide the API like `tf.gradients()` for Python users.

Realization of RMSprop in DQN. Some Deep learning framework provided centered RMSprop like a RMSProp of Graves, 2013, but, these are implemented different equation from both RMSprop of Graves, 2013 and RMSprop of DQN3.0 (DeepMind, 2017a `dqn/NeuralQLearner.lua#L256-L277`) as shows

Table 5. So we need to create RMSprop the same as DQN3.0 from scratch in optimizer creation method explained in document of each Deep learning framework.

PyTorch(Legacy) and PyTorch(New):

https://github.com/ElliottTradeLaboratory/DQN/blob/02427dc1bd5eae8e79d7b7c715f8049cb1dca66/dqn/pytorch_optimizer.py#L135-L154

MXNet:

https://github.com/ElliottTradeLaboratory/DQN/blob/02427dc1bd5eae8e79d7b7c715f8049cb1dca66/dqn/mxnet_extensions.py#L83-L121

Tensorflow and CNTK with Keras:

https://github.com/ElliottTradeLaboratory/DQN/blob/02427dc1bd5eae8e79d7b7c715f8049cb1dca66/dqn/convnet_keras.py#L261-L317

Other realization method

Maximization in preprocessing. As mentioned in Section 0. Preprocess-(a), it is necessary to the frame maximization between a current frame and a preceding frame, but, the meaning of “preceding” is in actual timestep rather than in agent’s timestep, because agent’s timestep was through the frameskip. So we need to select a maximization method among two ways: One is, use ALE directly; other one is, use env named ‘*GamenamNoFrameskip-v4*’, (e.g. ‘*BreakoutNoFrameskip-v4*’) of Gym. And in either case, it is necessary to implement a frame maximization function and frameskip function. Note that, if we use the basic env of Gym named “*Gamenam-v0*”, or, implement a frame maximization function after a frameskip function, we obtains different maximized frames than DQN3.0, as shown Figure 5-(b).

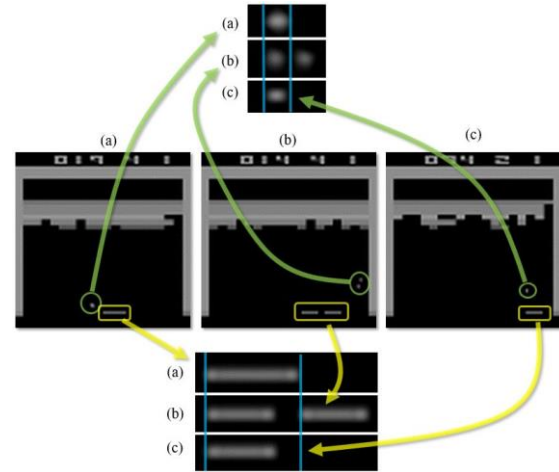


Figure 5: The comparison of how to implement the maximization. (a) implemented before frameskip; (b) implemented after frameskip; (c) not implemented; The top (a)~(c) are comparison of how the balls are displayed; The middle (a)~(c) are comparison of how inputs are displayed; The bottom (a)~(c) are comparison of how the paddles are displayed: (a) shows, the object is emphasized in the traveling direction when the object moved fast; On the other hand, (b) shows, the object is replicated when the object moved fast affected by frameskip.

Grayscale conversion in preprocessing. Although it is a small thing, the method of grayscale may be different depending on the image processing library to be used. In the first place, the grayscale conversion method varies depending on the application (Wikipedia, 2018c). Many image processing libraries applied a method that extracting the Y channel from the RGB image by the following formula

$$Y = 0.299R + 0.587G + 0.114B.$$

It is equal to the formula of the grayscale function as `rgb2y` provided the `image` package of Torch, and it is also equal to the formula of the grayscale function provided OpenCV (OpenCV), `tf.image` (Tensorflow1.4.1) and Pillow (Pillow). However, scikit-image does not has a such grayscale function, instead it has a grayscale function as `rgb2gray()` (scikit-image) with the formula as follow

$$Y = 0.2125R + 0.7154G + 0.0721B.$$

Resizing in preprocessing. In generally, each image processing library provides resize function with Bilinear interpolation (Bilinear) as default. In DQN3.0, it was also used `image.scale()` with ‘`bilinear`’ as a default interpolation. However, if we use different image processing library, we may obtain different output

of it resize function, even if it interpolation was named as “Bilinear”. As **Figure 6** shows, in the Space invaders which has frames with finer-designed objects, the object was lacked it shape or seems to another shape, if we use Bilinear of each image processing libraries on Python, and also shows, the resize function and the interpolation of library which is most close to `image.scale()` with 'bilinear' is `cv2.resize()` with `cv2.INTER_AREA`, and the next closest is `tf.image.resize_area()`. Not that, `tf.image` is very slow.

Trainable parameter initialization. In DQN3.0, the parameters of each layer of network are initialized by default initialization manner in `nn` package of Torch⁵ as follows

$$stdv = \frac{1}{\sqrt{fan_inWeight}} \quad (35)$$

$$\theta^{Weight} \sim \mathcal{U}(-stdv, stdv) \quad (36)$$

$$\theta^{Bias} \sim \mathcal{U}(-stdv, stdv). \quad (37)$$

Torch's layer class initialize both weight and bias using the same `stdv`. This is in contrast to each Python Deep learning framework initializing weight and bias separately. For this reason, in order to achieve that implement the same initializer as Torch, we need create a custom initializer shared a `stdv` between weight initializer and bias initializer. For example as follows:

<https://github.com/ElliottTradeLaboratory/DQN/blob/969e0e8eb54fdb18e9943f7b059aedf7fee00dc6/dqn/initializer.py#L55-L86>

I used class methods, but if you use `__call__` method may make it more sophisticated.

Rectifier layer. Mnih et al., 2015 created a Rectifier layer (DeepMind, 2017a `dqn/Rectifier.lua`) with their own forward/backward expressions: I considered this reason that, they may have thought that Torch's `ReLU` layer was slow. Because it compares elements and thresholds one by one during `for` loop.^{6,7} Or, by applied their own expressions, they may have aimed at stabilizing the algorithm and improving learning. In any case, except Tensorflow, when we want to reproduce their Rectifier layer, it is necessary to create custom operation(or user function etc.) and layer class in these creation method explained in document of Deep learning framework. On the other hand, Tensorflow officially only allows the creation of custom operations with C++ (Tensorflow.org, 2018). This reason is, probably because Tensorflow is low-level framework enough, they believe that in most cases it is possible to create what users want by combining existing operations. However, it is possible to create arbitrary operations in unofficial way, but In my case, I could not create a Rectifier layer in that way. For this reason, my DQN in the Tensorflow version usually uses the `tf.nn.relu` function provided by Tensorflow. Finally, for your information, we can override only the gradient function of the `tf.nn.relu` to the custom gradient function using `tf.Graph.gradient_override_map()`.

PyTorch(Legacy):

https://github.com/ElliottTradeLaboratory/DQN/blob/6ef872a83e1d61856339bd1ab1d21ab662cbd3b/dqn/pytorch_extensions.py#L105-L111

PyTorch(New):

https://github.com/ElliottTradeLaboratory/DQN/blob/6ef872a83e1d61856339bd1ab1d21ab662cbd3b/dqn/pytorch_extensions.py#L114-L137

MXNet:

https://github.com/ElliottTradeLaboratory/DQN/blob/969e0e8eb54fdb18e9943f7b059aedf7fee00dc6/dqn/mxnet_extensions.py#L48-L74

CNTK with Keras:

- (1) https://github.com/ElliottTradeLaboratory/DQN/blob/6ef872a83e1d61856339bd1ab1d21ab662cbd3b/dqn/cntk_extensions.py#L71-L111
- (2) https://github.com/ElliottTradeLaboratory/DQN/blob/6ef872a83e1d61856339bd1ab1d21ab662cbd3b/dqn/cntk_extensions.py#L118-L129

Tensorflow with Keras:

https://github.com/ElliottTradeLaboratory/DQN/blob/6ef872a83e1d61856339bd1ab1d21ab662cbd3b/dqn/tensorflow_extensions.py#L80-L103

3.3. Variation of reproduced work

Verification of the understanding of original semantics

Practice of “Verification of the understanding of original semantics” in BRM is to write this article. I think that I could write this article with a more accurate understanding of the semantics of DQN: The verification result of my DQN which will be described later is the same result as DQN3.0; each knowledge was able to explain the relevance to other knowledge and source code of DQN3.0. However, I have not confirmed the correctness of this article to Mnih et al., 2015, so there is no guarantee that I actually understand it accurately.

Verification of reproduced application program

Here I will explain in what methods I verified and the result.

The environment spec of experiments. DQN is required about 10GB memory per one process. I considered need the experiment environment that multiple process can be running in parallel, so I build self-made PC as **Table 2**. In addition, as shown in Table 1, finally, only MXNet 1.0.0 recommended the use of cuDNN 5.1, but I experimented with NVIDIA-Docker for the flexibility of the environment configuration.

Table 2: The environment spec of experiments.

Hardware	Spec.
CPU	Intel Core i7-7700K
Memory	64GB
GPU	GTX1080ti × 2
OS	Ubuntu 16.04

Basic performance evaluation. Here, I evaluate my DQNs with full-implementation, that is, each my DQN has the same implementation as DQN3.0. However, strictly speaking, Tensorflow ver. only uses Activation, `tf.nn.relu()`, which is different from DQN3.0.

Evaluation Method: I thought that it is necessary to consider for DQN evaluation method: The performance of learned parameters of DQN is greatly different in each evaluation step; The variance of each episode's score are very large in the one epoch, that is, by occasionally occurring the amazing maximum score as the outlier,

⁵ <https://github.com/torch/nn/blob/872682558c48ee661ebff693aa5a41fcdefa7873/SpatialConvolution.lua#L34-L55>

⁶ <https://github.com/torch/cunn/blob/master/lib/THCUNN/generic/Threshold.cu>

⁷ <https://github.com/torch/cutorch/blob/master/lib/THC/THCApply.cuh>

pull up the average score and thereby cause overestimation of the parameter; In addition, sometimes made the best average test score with parameter created in middle phase of training; On the other hand, the evaluation method in DQN3.0 is ambiguous: Since the evaluation in DQN3.0 is the average score per episode in the number of evaluation steps, that is, as learning progresses, the number of steps per episode increases, thereby the number of evaluated episodes per epoch is reduced, that is, the number of episodes per epoch is not constant, therefore, the evaluation of DQN3.0 compares the average episode score over the different number of episode at each epoch. In fact, as **Table 3** shows, actual measurement result of DQN3.0 using original evaluation method is very low than Mnih et al., 2015 P.11 Exented Data Table 2. For this other reason also I can consider that it replaced `nn.SpatialConvolution` instead of `nn.SpatialConvolutionCUDA`⁸, because `nn.SpatialConvolutionCUDA` was deprecated so is not available now. For both evaluation under constant condition and find more stable parameter, I apply the own evaluation method: First, getting average test scores over 100 consecutive trials from each parameters created between 60 epoch and 200 epoch for all my DQNs. Second, in each my DQN, select an average test score with the lowest standard deviation among Top 5 of average test scores as the Best average test score of a best parameter. Finally, compare the Best average test score between each DQNs. In addition, to make it possible to comparing the results of Mnih et al., 2015, I taken the average score over the first 30 episodes in the epoch which used a parameter becoming the best average test score. But, trial of training of each my DQN ver. are only once, so it is very simple and included variance.

Table 3: Comparison of best average test score of DQN3.0 with Minh et al., 2015 in Breakout using original evaluation method. the DQN version “Mnih et al., 2015” shows a Breakout results described in Mnih et al., 2015 P.11 Exented Data Table 2.

DQN version	Test 30 episodes		
	Average score	std	%Mnih et al
Mnih et al., 2015	401.20	±26.9	—
DQN3.0 Actual measurement	359.26	±89.8	89.55%

On the other hand, for comparison with other DQN implementations, training curves of `deep_q_rl` (Sprague, 2016, see Section 4) are also shown. However, it was difficult to obtain the Best average test score of `deep_q_rl` for me—perhaps it is necessary to modify some functions of the some classes and/or add some functions.

Result: As **Figure 8** and **Table 7** shows, in Breakout, as I expected, all my DQNs drew a similar training curve as DQN3.0, and the best average test score was nearly equal to DQN3.0. In Space invaders, on the other hand, the results were distinct between Group A and Group B: Group A with DQN3.0-level performance included PyTorch (New) and CNTK; Group B with low performance than DQN3.0 included PyTorch (Legacy), MXNet and Tensorflow, but these training curves are the same as DQN3.0 until about 120-160 epochs. In the case of `deep_q_rl`, in Breakout, the training curve was finally finished with close to DQN3.0-level but overall it was lower then DQN3.0; in Space invaders on the other hand, `deep_q_rl` had learned to a much lower level than DQN3.0. **Comparison of training curves by creating DQN with different reproduction method.** Here, I evaluate the result of simulating DQN where important implementation is lack or mis-implemented. Each function can switch enable or disable

with command-line argument, so it can be simulate DQN which only one function is lack or mis-implementation.

Evaluation Method: In this experiment, I evaluated only the training curve when changing each setting only. Only Episode separation only tried with Breakout, because the result of this experience shows noticeable difference also in Breakout. In other experiments tried with Space invaders which showed a more noticeable difference in the training curves.

Results: As **Figure 9**-(a),(b),(c) shows, my DQNs which the important functions are lack or mis-implementation are underperform DQN3.0 largely. In contrast, as **Figure 9**-(c) shows, the performance of `deep_q_rl` with `cv2.INTER_AREA` is make big improved and finally close to Group B of my DQN, but it is underperform DQN3.0 and my DQN until about 120 epochs. On the other hand, as **Figure 9**-(d) shows, there is no big difference by the Activation selection between Rectifier layer and ReLU function.

Training time. A comparison of the execution speed of the Deep learning frameworks is interested in many people. Although it deviates from the purpose of this article, comparison is more precisely possible in this work, so we will discuss it here.

Evaluation Method: As shown **Figure 10**, my DNQ is made possible to compare training time under equivalent condition among each my DQN vers. by applying the Strategy pattern (Erich et al., 1995). In experiments, the number of running process is one for each my DQNs, and GPU device is specified GPU#0 except CNTK. Because it was not possible to freely select a GPU device with CNTK using the `cntk.device.try_set_default_device()` function which was used to share the GPU selection method with Tensorflow.

Results: As **Figure 11** shows, PyTorch shown preeminently speed than other Deep learning frameworks. In contrast, no major difference was seen in other Deep learning frameworks, even Tensorflow-gpu 1.7 that more recent version in Deep learning frameworks I evaluated.

4. Related works

Many DQNs have been created and released so far, so I tried to evaluate them as much as possible. This section introduces two tasks from among them.

One of them, among the DQNs I evaluated, Sprague, 2016's DQN (called `deep_q_rl`) learned Atari 2600 games well. It was created using Python with Theano, and using ALE directly. In addition, it was created in the same creation method as DQN3.0 except loss-function, RMSprop and the bias initializer. For the loss-function, they applied Huber loss-function which they made from scratch as follows:

$$\begin{aligned}\delta_i &= y - Q(s, a; \theta_i) \\ \delta_i^{\text{quadratic}} &= \min(|\delta_i|, 1) \\ \delta_i^{\text{linear}} &= |\delta_i| - \delta_i^{\text{quadratic}} \\ L_i^{\text{Sprague}}(\theta_i) &= \sum \delta_i^{\text{linear}} \cdot 1 + \frac{\delta_i^{\text{quadratic}^2}}{2}\end{aligned}$$

Their Huber loss-function uses only the sum instead of the mean⁹,

⁸ `nn.SpatialConvolutionCUDA` was published by Chintalain, 2015, but I could not install it because I could not know how to install it.

⁹ but can select if we want.

over a mini-batch, then, obtain a scalar as a loss, denote $L_i^{\text{Sprague}}(\theta_i)$. After that, update the parameters using RMSprop they created from scratch like RMSprop of DQN3.0. Let $d\theta_i^{\text{Sprague}} = \nabla_{\theta_i} L_i^{\text{Sprague}}(\theta_i)$, their RMSprop can be denote as follows:

$$\begin{aligned} g_i &= \kappa g_{i-1} + (1 - \kappa) d\theta_i^{\text{Sprague}} \\ n_i &= \kappa n_{i-1} + (1 - \kappa) d\theta_i^{\text{Sprague}^2} \\ \theta_{i+1} &= \theta_i - \lambda \frac{d\theta_i^{\text{Sprague}}}{\sqrt{n_i - g_i^2 + \gamma}} \end{aligned}$$

This RMSprop is the same as DQN3.0, except, it does not use a momentum and subtraction is used for parameter update. The reason for subtracting for the parameter updating is that if we use automatic-differentiation for the any squared error loss-function, the sign of the gradient become opposite to DQN3.0. In order to reduce the instability of the algorithm resulting from these differences, Sprague, 2016 has filled the initial bias of each layer with 0.1.

Other one of them, Roderick et al., 2017 was published 2017 as scientific paper. Roderick et al., 2017 described the same aspect of DQN as this article, and they also release their source code (using Java with Caffe and Reinforcement learning library) at that time. However, I could not run it because it is too difficult to build the running environment for their source code for me. However, at least as far as saw their source code, it seems to be implemented in the same creation method as DQN3.0 except RMSprop, parameter initializer and a learning rate. Because, as described in Roderick et al., 2017, it is difficult to create RMSprop which the same as DQN3.0 in libraries their used, so their DQN has slightly low performance than DQN3.0, if their agent play the Breakout of Atari2600.

5. Discussion

5.1. About verification result

As shows in **Figure 8-(b)**, the results of my DQNs in Space invaders was separated into Group A and Group B. About this I can assume two reasons: One is the possibility that there is still a difference between my DQN and DQN 3.0; the other is that the errors due to differences in the implementation method of the framework may be delayed in convergence by enlarging by the detailed design of Space invaders. The verification of this assumption it is necessary to attempt further training: Training trials with different random seeds set; Training trials with more steps. However, as shown in **Figure 8-(a)**, in Breakout, since all my DQNs show the same training curve as DQM 3.0, it is assumed that the cause is the delay of the convergence point, I did not verify further.

5.2. About improving reproducibility

Although BRM was created by reverse engineering the actual DQN reproduction work, I think that there is room for BRM to incorporate opinions from people of various positions and to improve. In addition, BRM is a method of reducing the difficulty of reproduction that the reproducer feels by showing the reproduction procedure to the reproducer, however, I think that there is a method that can reduce the difficulty of reproduction by the author of the paper and the library provider; Building an environment that makes it easy to reproduce the outcome of papers

has the same meaning as providing a high quality textbook to beginners: I think that it is an one of important point that there is make it possible to expand the base of Artificial General Intelligence (AGI); So, with this article, I hope that the discussion on improving the reproducibility of the proposal will spread.

Acknowledgments

Thanks to Twitter people who support me secretly.

References

1. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep RL. *Nature*, 518(7540), 529. URL <https://www.nature.com/articles/nature14236>.
2. Rahtz, M. Lessons Learned Reproducing a Deep Reinforcement Learning Paper. (2018). Amid Fish. Web 16 June 2018. URL <http://amid.fish/reproducing-deep-rl>
3. DeepMind. Lua/Torch implementation of DQN (Nature, 2015). (2017a). URL <https://github.com/deepmind/dqn>
4. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. arXiv preprint arXiv:1606.01540. URL <http://arxiv.org/abs/1606.01540>
5. DeepMind. A lua wrapper for the Arcade Learning Environment/xitari, December 2014. URL <https://github.com/deepmind/alewrap>
6. Hinton, G., Srivastava, N., & Swersky, K. (2012). Lecture 6a overview of mini-batch gradient descent. URL https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
7. Chintala, Soumith. cuNNiCUDA:some depreceated, ugly and old modules. (2015). Web 29 November 2017. URL <https://github.com/soumith/cuNNiCUDA>
8. MXNet, rmspropalex_update. MXNet API. Web 11 June 2018. URL https://mxnet.incubator.apache.org/api/python/ndarray/ndarray.html#mxnet.ndarray.rmspropalex_update
9. ORACLE. Java SE at a Glance. (2018) Web 14 June 2018. URL <http://www.oracle.com/technetwork/java/javase/overview/index.html>
10. Wikipedia contributors. (2018a). Law of large numbers. In Wikipedia, The Free Encyclopedia. Retrieved 23:44, June 10, 2018, URL https://en.wikipedia.org/w/index.php?title=Law_of_large_numbers&oldid=839522786
11. Sutton, R. S., & Barto, A. G. (2018). RL: An introduction. Second edition, in progress * * * * Complete Draft * * * * . URL <http://incompleteideas.net/book/bookdraft2018jan1.pdf>
12. Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The Arcade Learning Environment: An evaluation platform for general agents. *J. Artif. Intell. Res. (JAIR)*, 47, 253-279. URL <https://arxiv.org/abs/1207.4708>
13. Tanner, B., and Write, A. RL-Glue: Language-Independent Software for Reinforcement-Learning Experiments. (2009). *Journal of Machine Learning Research*, 10(Sep):2133--2136, . (BibTex). URL <http://glue.rl-community.org/images/c/c2/RIglue-mloss-jmlr-2009-PREPRINT.pdf>

14. OpenAI. atari-py. (2017). URL <https://github.com/openai/atari-py>.
15. DeepMind. Xitari is a fork of the Arcade Learning Environment v0.4 (2017b). URL <https://github.com/deepmind/xitari>.
16. Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. arXiv preprint arXiv:1511.05952. URL <https://arxiv.org/abs/1511.05952>.
17. Wikipedia contributors. (2018b). Expected value 2.10 Extremal property. In Wikipedia, The Free Encyclopedia, 22 Mar 2018. Web 16 Apr 2018 URL https://en.wikipedia.org/w/index.php?title=Expected_value&oldid=837791052
18. Ng, A. Lecture 2.2 — Linear Regression With One Variable | CostFunction. (2016). Machine Learning — Andrew Ng, Stanford University. Web Dec 2017 URL <https://www.youtube.com/watch?v=yuH4iRcggMw>
19. Graves, A. (2013). Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850v5. URL <https://arxiv.org/abs/1308.0850>
20. Wikipedia contributors. (2018c). Grayscale. In Wikipedia, The Free Encyclopedia. Web 4 May 2018 URL <https://en.wikipedia.org/w/index.php?title=Grayscale&oldid=827764362>
21. OpenCV. Color conversions RGB↔GRAY. OpenCV modules. Web 20 Sep 2017. URL https://docs.opencv.org/3.3.1/de/d25/imgproc_color_conversions.html#color_convert_rgb_gray
22. Tensorflow1.4.1. rgb_to_grayscale. Web 11 December 2017. URL https://github.com/tensorflow/tensorflow/blob/438604fc885208ee05f9eef2d0f2c630e1360a83/tensorflow/python/ops/image_ops_impl.py#L1097-L1126
23. Pillow. Image.convert. Pillow Doc>Reference. Web 15 Sep 2017. URL <https://pillow.readthedocs.io/en/5.1.x/reference/Image.html?highlight=convert#PIL.Image.Image.convert>
24. scikit-image, rgb2gray, API Reference for skimage 0.14.0. Web 20 Aug 2017. URL <http://scikit-image.org/docs/0.14.x/api/skimage.color.html#rgb2gray>
25. Tensorflow.org (2018, April, 28). Adding a New Op In Extend. Web 5 May 2018 URL https://www.tensorflow.org/extend/adding_an_op
26. Sprague, N. (2016). Theano-based implementation of Deep Q-learning. URL https://github.com/spragunr/deep_q_rl
27. Roderick, M., MacGlashan, J., & Tellex, S. (2017). Implementing the Deep Q-Network. arXiv preprint arXiv:1711.07478. URL <https://arxiv.org/abs/1711.07478>
28. Erich, G., Richard, H., Ralph, J., John, V. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

Appendix

A. Extended Tables

Table 4: Correspondence of equations and source code.

Source code (DeepMind, 2017a dqn/NeuralQLearner.lua)
Eq (27): $y = r + \gamma \max_a Q(s', a; \theta_i^-)$
<pre> 194 term = term:clone():float():mul(-1):add(1) 195 196 197 local target_q_net 198 if self.target_q then 199 target_q_net = self.target_network 200 else 201 target_q_net = self.network 202 end 203 204 205 -- Compute max_a Q(s2, a). 206 q2_max = target_q_net:forward(s2):float():max(2) 207 208 209 -- Compute q2 = (1-terminal) * gamma * max_a Q(s2, a) 210 q2 = q2_max:clone():mul(self.discount):cmul(term) 211 212 213 delta = r:clone():float() 214 215 216 if self.rescale_r then 217 delta:div(self.r_max) 218 end 219 delta:add(q2) </pre>
Eq (28): $\delta_i = \left\{ r^{(m)} + \gamma \max_a Q(s'^{(m)}, a; \theta_i^-) - Q(s^{(m)}, a^{(m)}; \theta_i) \right\}_{m=1}^M$
<pre> 217 local q_all = self.network:forward(s):float() 218 q = torch.FloatTensor(q_all:size(1)) 219 for i=1,q_all:size(1) do 220 q[i] = q_all[i][a[i]] 221 end 222 delta:add(-1, q) </pre>
Error Clipping
<pre> 224 if self.clip_delta then 225 delta[delta:ge(self.clip_delta)] = self.clip_delta 226 delta[delta:le(-self.clip_delta)] = -self.clip_delta 227 end </pre>
Part of Eq (30): $\sum_{m=1}^M \delta_i^{(m)} \nabla_{\theta_i} Q(s^{(m)}, a^{(m)}; \theta_i)$
<pre> 229 local targets = torch.zeros(self.minibatch_size, self.n_actions):float() 230 for i=1,math.min(self.minibatch_size,a:size(1)) do 231 delta[delta:le(-self.clip_delta)] = -self.clip_delta 232 end 233 self.network:backward(s, targets) </pre>
Eq (30) or Eq (31): $g_i = \aleph g_{i-1} + (1 - \aleph) d\theta_i$
<pre> 266 self.g:mul(0.95):add(0.05, self.dw) </pre>
Eq (32): $n_i = \aleph n_{i-1} + (1 - \aleph) d\theta_i^2$
Eq (33): $\Delta_i = \beth \Delta_{i-1} + \lambda \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \gamma}}$
Eq (34): $\theta_{i+1} = \theta_i + \Delta_i$
<pre> 267 self.tmp:cmul(self.dw, self.dw) 268 self.g2:mul(0.95):add(0.05, self.tmp) 269 self.tmp:cmul(self.g, self.g) 270 self.tmp:mul(-1) 271 self.tmp:add(self.g2) 272 self.tmp:add(0.01) 273 self.tmp:sqrt() 274 275 -- accumulate update 276 self.deltas:mul(0):addcdiv(self.lr, self.dw, self.tmp) 277 self.w:add(self.deltas) </pre>

Table 5: The expression of Centered RMSprop provided by each Deep learning framework. Red indicates a different point between the RMSprop of DQN3.0 and the central RMSprop, which is provided by the Deep learning framework. CNTK and Keras does not provide the centered RMSprop.

Centered RMSprop of PyTorch(New)¹⁰:

$$g_i = \aleph g_{i-1} + (1 - \aleph) d\theta_i \quad (38)$$

$$n_i = \aleph n_{i-1} + (1 - \aleph) d\theta_i^2 \quad (39)$$

$$\Delta_i = \beth \Delta_{i-1} + \lambda \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \gamma}} \quad (40)$$

$$\theta_{i+1} = \theta_i - \Delta_i \quad (41)$$

Argument mapping		
PyTorch	Symbol	Mnih et al., 2015 Data Table 1 / DQN3.0
lr	λ	learning rate
alpha	\aleph	gradient momentum or squared gradient momentum
eps	γ	min squared gradient
weight_decay	—	self.wc (DQN3.0)
momentum	\beth	\emptyset (DQN3.0)

Centered RMSprop of MXNet^{11,12}:

$$g_i = \aleph g_{i-1} + (1 - \aleph) d\theta_i \quad (42)$$

$$n_i = \aleph n_{i-1} + (1 - \aleph) d\theta_i^2 \quad (43)$$

$$\Delta_i = \beth \Delta_{i-1} - \lambda \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \gamma}} \quad (44)$$

$$\theta_{i+1} = \theta_i + \Delta_i \quad (45)$$

Argument mapping		
MXNet	Symbol	Mnih et al., 2015 Data Table 1 / DQN3.0
lr	λ	learning rate
gamma1	\aleph	gradient momentum or squared gradient momentum
gamma2	\beth	\emptyset (DQN3.0)
epsilon	γ	min squared gradient

Centered RMSprop of Tensorflow¹³:

$$g_i = \aleph g_{i-1} + (1 - \aleph) d\theta_i \quad (46)$$

$$n_i = \aleph n_{i-1} + (1 - \aleph) d\theta_i^2 \quad (47)$$

$$\Delta_i = \beth \Delta_{i-1} + \lambda \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \gamma}} \quad (48)$$

$$\theta_{i+1} = \theta_i - \Delta_i \quad (49)$$

Argument mapping		
Tensorflow	Symbol	Mnih et al., 2015 Data Table 1 / DQN3.0
learning_rate	λ	learning rate
decay	\aleph	gradient momentum or squared gradient momentum
momentum	\beth	\emptyset (DQN3.0)
epsilon	γ	min squared gradient

¹⁰ https://pytorch.org/docs/stable/_modules/torch/optim/rmsprop.html#RMSprop

¹¹ <https://mxnet.incubator.apache.org/api/python/optimization/optimization.html#mxnet.optimizer.RMSProp>

¹² https://mxnet.incubator.apache.org/api/python/ndarray/ndarray.html#mxnet.ndarray.rmspropalex_update

¹³ In the case of Tensorflow, the overall formula of the centered RMSprop is written only in `pythonx.x/site-packages/tensorflow/python/training/gen_training_ops.py#L192-L195` (in the case of ver.1.4.1). This file is created during Tensorflow installation.

Table 6: The original-setting. The original-setting aims to obtain the same results as DQN3.0. “original-HP” means the same hyperparameter as Mnih et al., 2015 P.10 Extended Data Table 1. “Torch nn layers default” means the same initializer as DQN3.0 with Eq (34)~(36).

DQN Version	Hyperparameter	Resize	Activation	Weight & bias initializer
PyTorch(Legacy)	original-HP	cv2 AREA	Rectifier	Torch nn layers default
PyTorch(New)	original-HP	cv2 AREA	Rectifier	Torch nn layers default
MXNet	original-HP	cv2 AREA	Rectifier	Torch nn layers default
Tensorflow with Keras	original-HP	cv2 AREA	tf.nn.relu	Torch nn layers default
CNTK with Keras	original-HP	cv2 AREA	Rectifier	Torch nn layers default

Table 7 Comparison of best average test score of each DQNs in Breakout and Space invaders(with original-hyperparameter and DQN reproduction method) . “Mnih et al., 2015” of *DQN version* is the score described in Mnih et al., 2015 Extended Data Table 2; “*DQN3.0 Actual measurement*” of *DQN version* is the result of the Best average test score of DQN3.0; “% *DQN3.0 Actual*” is the ratio to the measured score of DQN3.0; “% Mnih et al., 2015” is the ratio to the score described in Mnih et al., 2015.

DQN version	(a) Breakout							(b) Space invaders						
	Test 100 episodes			Test 30 episodes				Test 100 episodes			Test 30 episodes			
	Average score	std	%DQN3.0 Actual	Average score	std	% Mnih et al., 2015	%DQN3.0 Actual	Average score	std	%DQN3.0 Actual	Average score	std	% Mnih et al., 2015	%DQN3.0 Actual
Mnih et al., 2015	—	—	—	401.20	±26.9	—	109.70%	—	—	—	1976.00	±893.00	—	99.81%
DQN3.0 Actual measurement	369.57	±69.0	—	365.73	±84.9	91.16%	—	2018.83	±655.1	—	1979.70	±737.15	100.19%	—
deep_q_rl	—	—	—	—	—	—	—	—	—	—	—	—	—	—
PyTorch(Legacy)	377.74	±72.1	102.21%	380.50	±80.50	92.97%	101.99%	1807.45	±673.4	89.53%	2020.50	±664.2	102.25%	102.06%
PyTorch(New)	361.11	±82.3	97.71%	377.23	±72.04	94.03%	103.14%	2111.05	±742.4	104.57%	2020.50	±685.9	102.25%	102.06%
MXNet	392.39	±61.2	106.18%	387.60	±95.71	96.61%	105.98%	1743.35	±739.1	86.35%	1789.00	±772.2	90.54%	90.37%
Tensorflow with Keras	385.03	±70.3	104.18%	388.57	±46.51	96.85%	106.24%	1868.45	±608.0	92.55%	1854.67	±598.6	93.86%	93.68%
CNTK with Keras	369.74	±68.2	100.05%	384.57	±33.70	95.85%	105.15%	2094.15	±723.1	103.73%	2214.17	±731.7	112.05%	111.84%

B. Extended Figures

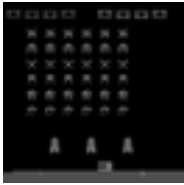

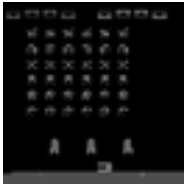
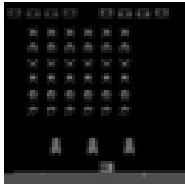
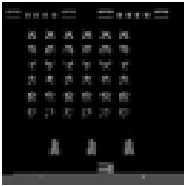
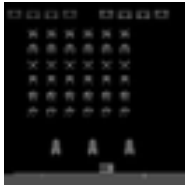
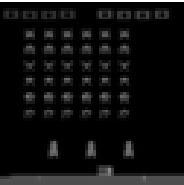

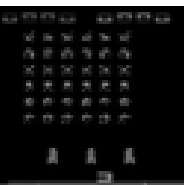

Library	Interpolation	
	Bilinear (MSE)	Area (MSE)
torch.image		
OpenCV	 (7.04E-04)	 (0.00E+00)
tf.image	 (1.13E-03)	 (1.10E-09)
Pillow	 (1.49E-04)	
scikit-image	 (7.04E-04)	

Figure 6: Comparison of resized frame by each image processor libraries. “MSE” is the mean squared error between a resized frame by `image.scale()` with ‘bilinear’ and a resized frame by each image processor libraries.

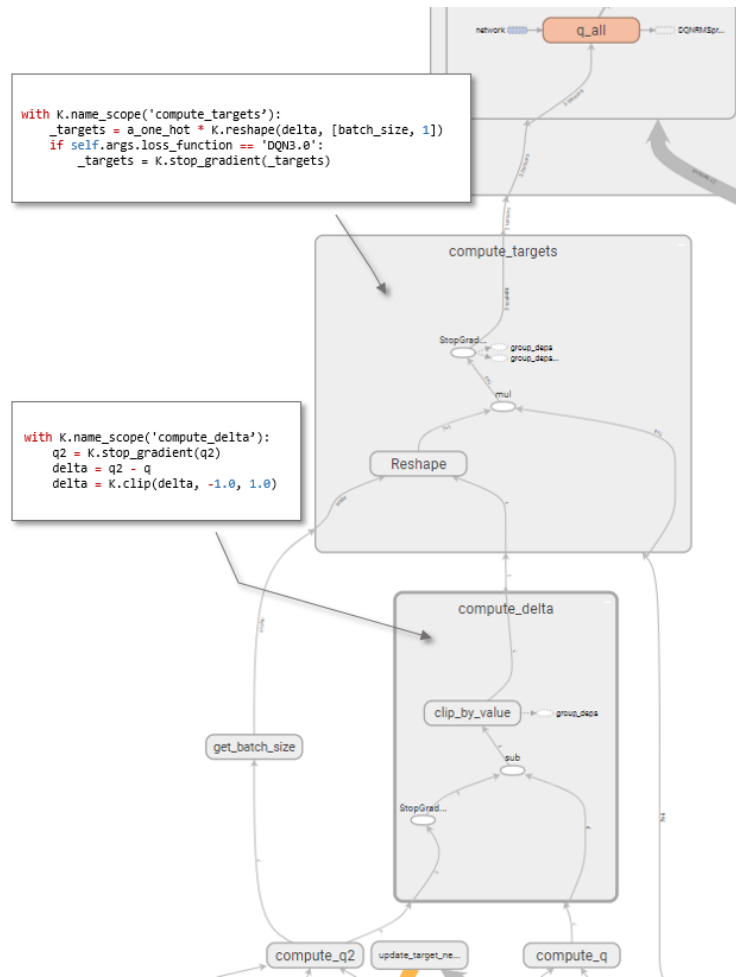


Figure 7: Part of the computation graph of DQN loss-function in Tensorflow.

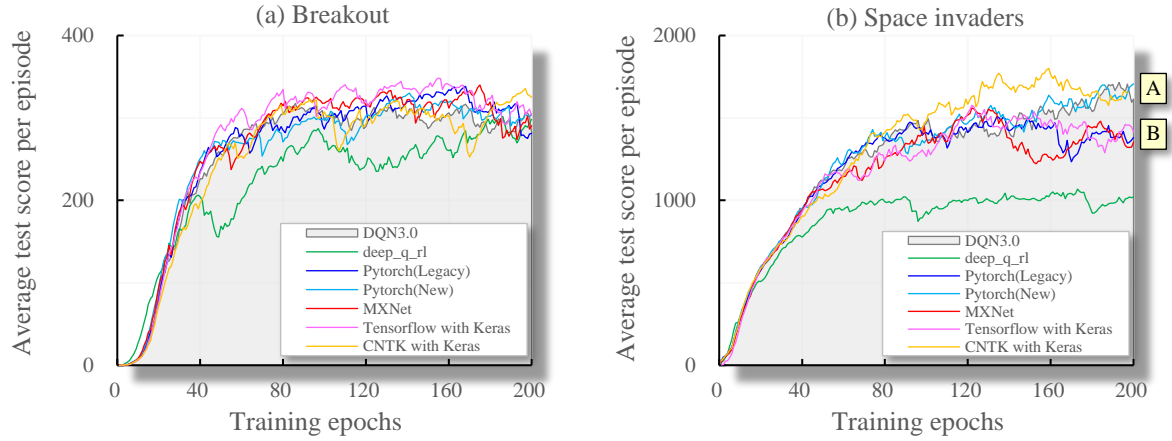


Figure 8 : Comparison of training curves in Breakout and Space invaders (with original-hyperparameter and DQN reproduction method). Training curves tracking each agent's average test score with the original-setting (Table 6). In the case of deep_q_rl, specifying `device=gpu` in `.theanorc` improved the training curve in Breakout. Both graphs applied smoothing 0.9. In (b) Space invaders, My DQNs converged to two results of Group A and Group B.

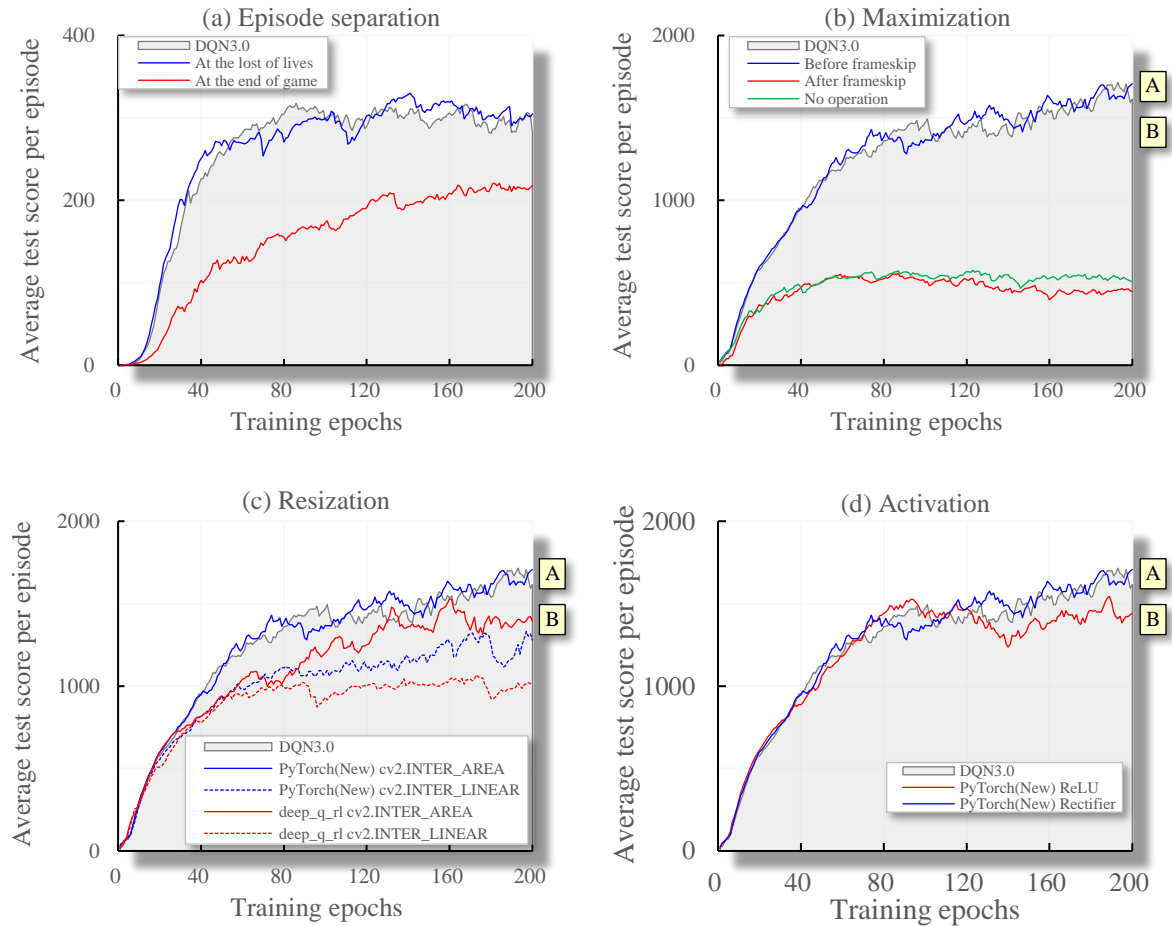


Figure 9: Comparison of training curves of each setting. (a) comparison of episode separation setting; (c) comparison of resization setting in Space invaders; (d) comparison of activation setting in Space invaders. (a)~(d) are used myDQN with PyTorch(New) with the original-setting and deep_q_rl. deep_q_rl with `cv2.INTER_AREA` in (c) was modified an argument "interpolation" of resize function of OpenCV in Sprague, 2016 ale_experiment.py#L186. These graphs applied smoothing 0.9.

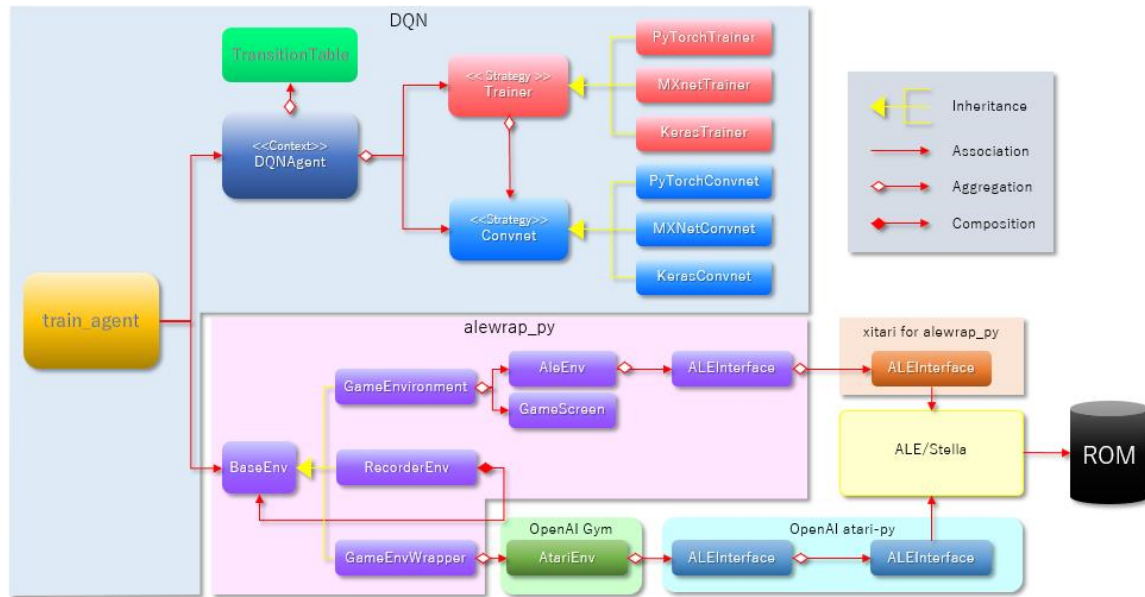


Figure 10: The architecture of my DQN. My DQN is an implement of the Strategy pattern (Erich et al., 1995): Client: train_agent, Context: DQNAgent, Strategy: Trainer and convet; Each ConcreteStrategy classes corresponds each Deep learning framework. alewrap_py reproduced alewrap in other package for improving reusability; In addition, comparison of the behavior of alewrap and OpenAI Gym's AtariEnv is made possible by having a common interface BaseEnv where GameEnvironment (reproduced GameEnvironment of alewrap) and GameEnvWrapper (calls AtariEnv of OpenAI Gym).

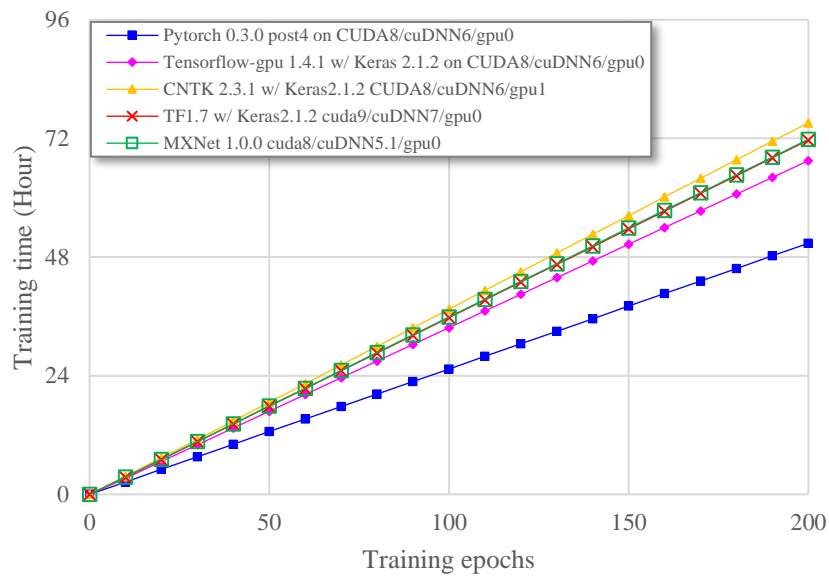


Figure 11: Comparison of training time of each Deep learning.