# Improve Reproducibility
# Throught Accurate Reproduction of Deep Q-network

Ver 1.0  31 June 2018

Shigeru Matsumura

## Abstract

Low reproducibility of proposal is a serious problem that can lead to undesirable situations for the proposal:  underestimation; narrow the scope of applications and further study; false direction of further study.

Like with everyone can reproduce $1 + 1 = 2$ between arbitrary programming languages and libraries, in principal, the reproduction of an application program between different programming language and library is not difficult—but it is a time-consuming and heavy work—.  However, the reproduction of Deep Q-network (DQN) (Mnih et al., 2015) is difficult due to various obstacles.  As a result, in actual conditions, DQN may be underestimated.

This article address improvement of reproductivity of the proposal with suggest the solutions found by solving of problems occurred in the accurately reproduction of DQN3.0 (DeepMind, 2017).

## 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 spended 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 fashon—as follows.

### 1.1. Problems that occurred in reproduction of DQN3.0

#### 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 they 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 method of the DQN loss-function in DQN3.0 is not mentioned anywhere in the Deep learning primer or Deep learning framework document, but there was enough explanation to impress that DQN3.0's implementation method is wrong.  So I selected the method with more generally but thereby my DQN did not learning.  I don't know that the reason of why is not explained most famous algorithm in Reinforcement learning generally—as if they are pretending not to see the implementation method in DQN3.0—, but I have a hypothesis that 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 sees 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 including 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 the 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 at el., 2012" was explained.  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, 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].

### Wrong evaluation method was selected

One of the reason of I confused in comparison of my DQN with DQN3.0: Since the evaluation method implemented in DQN3.0 can not accurately evaluate the test result of DQN including high variance, it was difficult to confirm whether my DQN was accurately reproduced.

### I could not build the running environment of some DQN implementation

In general, each implementation created using author's favorite libraries. The library sometimes required specific version of low-level library such as CUDA and cuDNN. In evaluation of each implementation, it is necessary to build a running environment corresponding each implementation. In addition, the library sometimes need build from source code, but in some cases, complicated procedures were required, which could not be completed.

## 1.2. To alleviate the difficulty of reproduction

The above problems can be avoided by providing each stakeholder (the author of the paper, library provider, reproducer) with necessary information and knowing the method. I suggest the necessary actions for each stakeholder here.

### Author of paper should:

(1) Publish an executable source code reproducible to the experimental results indicated in the paper: The ability to reproduce your proposal easily means that its application range expands (e.g. further research based on it; apply it to different problems); It is difficult to execute in a environment with easy to prepare (e.g. it is premised that it runs on specific hardware such as a robot; a large scale parallel system is necessary) but as far as possible.

(2) Create Dockerfile or Container: There is a possibility that it may not be completely reproduced if you leave the execution environment construction to the reproducer: The installation procedure which the author thinks is easy may not be easy for the reproducer (Especially when the proposal need a library set that requires build from source code); Since the Docker container is separated from the environment of the reproducers's OS, reproducers can easy and accurate install without any concern that their favorite environment will be destroyed regardless of the installation set.

(3) An innovative proposal that is not in the conventional method describes a concrete explanation of its implementation method: The innovative implementation is not understood because there is no sample or explanation, and there is a high possibility of having another implementation applying the conventional method.

(4) Publish documents explaining application structure: An explanation of which module contains the function described in the paper can easily reproduce the proposal.

(5) Use stable values for the evaluation of the proposal: Values including large variance can not be precisely evaluated; Values with low variance make it easier to evaluate the application that reproduced the proposal—However, I think that it depends on the characteristics of the algorithm and environment how many times the trial will give the desired average value. I think that it is necessary to try the number of times until the result converges to a certain value at least once and obtain an appropriate number of evaluation trials—.

For example, Workd Models (Ha and Schmidhuber, 2018) shows a good example: Publish the executable source code and a web-page (Ha, 2018) explaining how to reproduce it, and accept the inquiries on the issue on Github.

The low reproducibility is a serious problem that may cause underestimation of the proposal and may narrow the scope of application of the proposal. I recommend that we do not assume that the reproducer can accurately reproduce the proposal.

### Library providers should:

(6) Explain the mathematical formulas used in the function (not only reference of paper) and explain the correspondence between the function argument and the term, in the API document: Reduce the labor of understanding the implementation by looking at the source code; And, it is not anyone who can do it; A good example in API document of MXnet (MXNet)

(7) (For Deep learning framework providers) Explain to make it positive that the Automatic-differentiation can create arbitrary differential expressions: The Automatic-differentiation is basically for the purpose of omitting the creation of the differential expression by the user, and it is necessary to affirm that the user makes arbitrary differential expression.

(8) Publish old versions: Allows to reproduce the implementation of algorithms proposed in the past (proposals may be evaluated in the future without being evaluated at that time)

### Reproducer should:

(9) Fully understand the basic knowledge necessary to understand the original work: We need to read many literature references and thick books, but there is no shortcut besides going through it.

(10) Check if the application we created is accurately reproduced: Focus on the cause of the parts that can not be reproduced, separating the part that can be reproduced and the part that can not be reproduced; For reasons why it can not be reproduced, it is necessary to isolate the problem in order to examine something. (Lack of basic knowledge, abuse of function of library to be used etc.)

(11) The decision of the library to use in our application is judged based on whether it suits the purpose of our application or not: Whether the library suits to the purpose of our application will become clear through deep examination of the properties of the library.

In this article, first in *Section.2* describe the basics reproduction method. And next in *Section.3* describe the reproduction method of DQN following the basics reproduction method and show the result that it can be accuracery reproduce DQN3.0 using Python. Finally, in *Section 6*, by mapping what I got from the reproduction

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

of DQN 3.0 to what I suggested here, I discuss the possibility that reproducing DQN becomes easier.

## 2. Basics reproduction method

In principle, an application program created using any programming language and library can accurate reproduce using different programming language and library which executable on the same CPU. For example, $1+1=2$ using Lua/Torch can reproduce using Python/Tensorflow everyone. But in order to this, it is necessary to satisfy some preconditions even in reproduction of only $1+1=2$.

### 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 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 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. 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 as per the source code we written, regardless of our wishes. Humans 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 not wrong". 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—.

### Grasp 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, that is, when the functions $A$ and $B$ have the same semantics, when the same input is given, these outputs become equal, even if both function's implementation are different. So even simple expressions such as $1+1=2$ can not be reproduced accurately if implemented without grasping 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 grasp the semantics of original work and apply it to our reproduced work.

### 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, very inconveniently, 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 part which aimed at speeding up written in C/C ++. For this reason, you will need reading comprehension 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 Github you may not find the source file containing the source code of the function you want to study. So you need to search both repository and package installation directory.

Finally, for large libraries such as Deep learning frameworks, the polymorphic structure for the abstraction of programs it has is difficult to read and understand. In order to read and understand such programs, you need to learn program abstraction techniques—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 tensor with multiple elements, it become time-consuming work. In this case, as shown in **Figure 1**, 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 number 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 the 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 anytime. In this case, when estimated on the CPU, the output with respect to the same input are the same anytime. 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, 2018) when the number of trials increases.

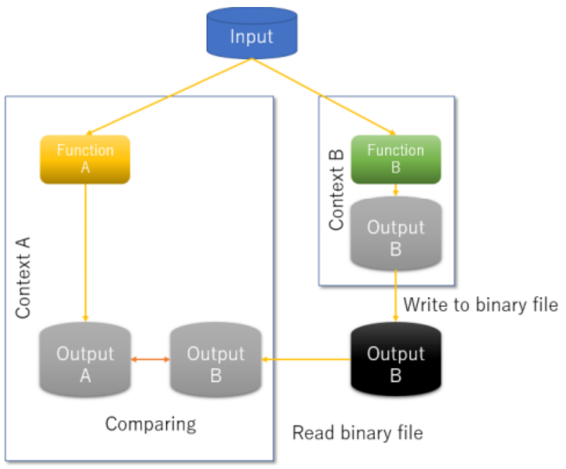


**Figure 1: Example of comparison of outputs between function A and function B.**

## 3. Reproduction method of Deep Q-network

In this section, based on the method described in the previous section, we examined the method of reproducing DQN in *3.1* and *3.2*, and in *5*.showed that DQN 3.0 could be accurately reproduced as a result. In addition, in *4*, I introduce examples of other implementations reproducing DQN.

### 3.1. Grasp the semantics of Deep Q-network

Three parts are required to explain the semantics of DQN: First, in *Background of Deep Q-network*, I will explain the basics key technique of reinforcement learning and the environment of Atari 2600 which indispensable for understanding DQN; next *The semantics as an aggressive 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 fomulate 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 $s0, r0, a0, s1, r1, a1\dots, s_T, r_T$ 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* $\pi$. The cumulative rewards is denoted as follows:

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

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

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

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:

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

where $p(s', r \mid s, a) \doteq \Pr\{s' = s_{t+1}, r = r_{t+1} \mid s = s_t, a = a_t\}$. 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 $s_0, s_1, \cdots, s_{t-1}$, that is, $p(s_t \mid s_0, s_1, \cdots, s_{t-1})$. 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:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}[R_t \mid s_t = s, a_t = a] && (4)\\ &= \mathbb{E}[r + \gamma R_{t+1}] && (5)\\ &= \mathbb{E}[r + \gamma q_\pi(s', a') \mid s_{t+1} = s', a_{t+1} = a'] && (6)\end{aligned}$$

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

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. The Eq (7) is called *the Bellman equation* $q_\pi$, and $\gamma \sum_{a'} \pi(a' \mid s')\, q_\pi(s', a')$ 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:

$$q_\pi(s,a) = \left[\begin{array}{c} p\left(s'^{(1)}, r^{(1)} \mid s, a\right) \left[ r^{(1)} + \gamma \left[\begin{array}{c} \pi\left(a'_{[1]} \mid s'^{(1)}\right) \cdot q_\pi\left(s'^{(1)}, a'^{(1)}_{[1]}\right) \\ + \\ \pi\left(a'_{[2]} \mid s'^{(1)}\right) \cdot q_\pi\left(s'^{(1)}, a'^{(1)}_{[2]}\right) \\ + \\ \vdots \\ + \\ \pi\left(a'_{[|\mathcal{A}(s'^{(m)})|]} \mid s'^{(1)}\right) \cdot q_\pi\left(s'^{(1)}, a'_{[|\mathcal{A}(s'^{(1)})|]}\right) \end{array} \middle| \, a' \in \mathcal{A}\left(s'^{(1)}\right) \right] \right] \\ + \\ p\left(s'^{(2)}, r^{(2)} \mid s, a\right) \left[ r^{(2)} + \gamma \left[\begin{array}{c} \pi\left(a'_{[1]} \mid s'^{(2)}\right) \cdot q_\pi\left(s'^{(2)}, a'^{(2)}_{[1]}\right) \\ + \\ \pi\left(a'_{[2]} \mid s'^{(2)}\right) \cdot q_\pi\left(s'^{(2)}, a'^{(2)}_{[2]}\right) \\ + \\ \vdots \\ + \\ \pi\left(a'_{[|\mathcal{A}(s'^{(m)})|]} \mid s'^{(2)}\right) \cdot q_\pi\left(s'^{(2)}, a'_{[|\mathcal{A}(s'^{2})|]}\right) \end{array} \middle| \, a' \in \mathcal{A}\left(s'^{(2)}\right) \right] \right] \\ + \\ \vdots \\ + \\ p\left(s'^{(n)}, r^{(n)} \mid s, a\right) \left[ r^{(n)} + \gamma \left[\begin{array}{c} \pi\left(a'_{[1]} \mid s'^{(n)}\right) \cdot q_\pi\left(s'^{(n)}, a'^{(n)}_{[1]}\right) \\ + \\ \pi\left(a'_{[2]} \mid s'^{(n)}\right) \cdot q_\pi\left(s'^{(n)}, a'^{(n)}_{[2]}\right) \\ + \\ \vdots \\ + \\ \pi\left(a'_{[|\mathcal{A}(s'^{(n)})|]} \mid s'^{(n)}\right) \cdot q_\pi\left(s'^{(n)}, a'^{(n)}_{[|\mathcal{A}(s'^{(n)})|]}\right) \end{array} \middle| \, a' \in \mathcal{A}\left(s'^{(n)}\right) \right] \right] \end{array}\right] \tag{8}$$

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 $p(s', r \mid s, a)$ as follows:

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

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

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:

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

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:

$$\text{The best action } a \text{ in } s = \operatorname*{argmax}_{a \in \mathcal{A}(s)} \left\{\begin{array}{c} q_*\left(s, a_{[1]}\right), \\ q_*\left(s, a_{[2]}\right), \\ \vdots \\ q_*\left(s, a_{[|\mathcal{A}(s)|]}\right) \end{array}\right\} = \operatorname*{argmax}_{a \in \mathcal{A}(s)} v_\pi(s), \tag{12}$$

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* $\boldsymbol{n}$ *states, then there are* $\boldsymbol{n}$ *equations in* $\boldsymbol{n}$ *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:

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

where $\alpha$ 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 $\varepsilon$-greedy policy, because, it behavior is changed from the almost random selection to the almost determistic 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 ALE directly or extended by themselves.In 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 Introdunction, 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`[2] with 0.25. On the other hand, several `AtariEnv` corresponding to other uses are also available: For example, one of the `AtariEnv` named '*Gamename*`Deterministic-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 '*Gamename*`NoFrameskip-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 aggressive 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 five 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 $x_{\tilde{t}} = \{x_{\tilde{t}}^{\mathrm{R}}, x_{\tilde{t}}^{\mathrm{G}}, x_{\tilde{t}}^{\mathrm{B}}\}$. **(a) Maximization:** they take the maximum value between $x_{\tilde{t}}$ and $x_{\tilde{t}-1}$ as $\dot{x}_{\tilde{t}}$

$$\dot{x}_{\tilde{t}} = \max\{x_{\tilde{t}-1}, x_{\tilde{t}}\}. \qquad (14)$$

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

$$\begin{aligned}\dot{x}_{t-3}, \dot{x}_{t-2}, \dot{x}_{t-1}, \dot{x}_t &= \dot{x}_{\tilde{t}-12}, \dot{x}_{\tilde{t}-8}, \dot{x}_{\tilde{t}-4}, \dot{x}_{\tilde{t}} \\ &= \max\{x_{\tilde{t}-13}, x_{\tilde{t}-12}\}, \max\{x_{\tilde{t}-9}, x_{\tilde{t}-8}\} \\ &\quad, \max\{x_{\tilde{t}-5}, x_{\tilde{t}-4}\}, \max\{x_{\tilde{t}-1}, x_{\tilde{t}}\} \qquad (15)\end{aligned}$$

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.

**(c) Grayscale conversion:** then, they extract the Y channel (it is called luminance) from $\dot{x}_t$ as $\ddot{x}_t$ (it is generally called the grayscale conversion)

$$\ddot{x}_t = 0.299\dot{x}_t^{\mathrm{R}} + 0.587\dot{x}_t^{\mathrm{G}} + 0.114\dot{x}_t^{\mathrm{B}}. \qquad (16)$$

**(d) Resization:** then, they resize $\ddot{x}_t$ to $84 \times 84$ with the bilinear interpolation as $s_t$

$$s_t = \mathrm{resize}^{\mathrm{Bilinear}}(\ddot{x}_t, 84{,}84). \qquad (17)$$

**(e) Stack frames into $\boldsymbol{\phi}$:** As better input data $\phi_t$ for learning and prediction, they stacks the four recentry $s_t$

$$\phi_t = \{s_{t-3}, s_{t-2}, s_{t-1}, s_t\}, \qquad s_t \neq s^+, \qquad (18)$$

where $s^+$ is terminal state, that is, the agent does not learn and predict from terminal state. And now, $\phi_t$ 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 $\Phi_t$ to null state, denote $s^{\mathrm{null}}$, that means that state is filled with zero. For example, if $\phi_t$ contains the terminal state occurred at $t-2$, denote

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

in this case, $\phi_t$ contains states occurred in two different episodes: One is $\mathrm{episode}^{-1}$ contained $s_{t-3}$ and $s_{t-2}^+$ (that is, $\mathrm{episode}^{-1}$ is terminated at $t-2$); and other one is $\mathrm{episode}^0$ contained $s_{t-1}$ and $s_t$ (that is, $\mathrm{episode}^0$ is started at $t-1$). Let add the episode number in square bracket subscript to (19) as follows:

$$\phi_t = \{s_{t-3}^{[-1]}, s_{t-2}^{+,[-1]}, s_{t-1}^{[0]}, s_t^{[0]}\}. \qquad (20)$$

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

$$\phi_t = \{s^{\mathrm{null},[-1]}, s^{\mathrm{null},[-1]}, s_{t-1}^{[0]}, s_t^{[0]}\}. \qquad (21)$$

Thus, their agents will learn and predict with preprocessed states $\phi_t$ only during survival.

Finally, **(f) 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), (b) and (f) are written in GameScreen.lua in alewap (DeepMind, 2014) package; (c) , (d) both witten in scale.lua in DQN3.0 package; and (e) is witten 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 2**- (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 2**-(b), the agent does not learn these actions with no value which selected toward to the loss of Lives. However, the agent recognizes the Lives count as the additional information obtained by the interface expansion of ALE instead of the Lives counter on the game frames. That is, more precisely, their agent is observed not only the pixels and the game score. This function was

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

implemeted in alewrap.

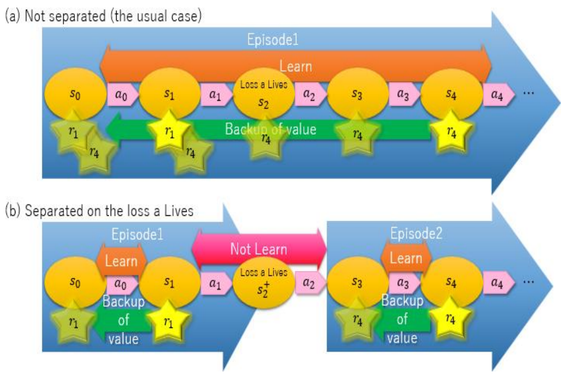

(a) Not separated (the usual case)
Episode1
Learn
$s_0$ $a_0$ $s_1$ $a_1$ Loss a Lives $s_2$ $a_2$ $s_3$ $a_3$ $s_4$ $a_4$ …
$r_1$ $r_4$ $r_1$ $r_4$ Backup of value $r_4$ $r_4$ $r_4$
(b) Separated on the loss a Lives
Episode1
Learn
Not Learn
Episode2
Learn
$s_0$ $a_0$ $s_1$ $a_1$ Loss a Lives $s_2^+$ $a_2$ $s_3$ $a_3$ $s_4$ $a_4$ …
Backup of value
$r_1$ $r_1$ $r_4$ $r_4$

**Figure 2: 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. We 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) $\mathcal{D}_t = \{e_{t-N}, e_{t-(N+1)}, \cdots, e_t\}$, where $N \in \mathbb{R}^{[0,1\times10^6]}$ 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 recentry $1\times10^6$ experiences, and forget experiences earlier than $t - 1\times10^6$.

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 $\boldsymbol{e}'$ as mini batch size $M$

$$\mathcal{E}_i = \{e'_1, e'_2, \cdots, e'_M\}, \tag{22}$$

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 0, *Preprocess (e)* would be applied to $\phi$ and $\phi'$, but, only $\phi'$ allows including terminal state. Because, $\phi'$ 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

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

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 $\theta$ is network parameter. Each Q-function shares $\theta$, 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 , 2018), 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 minimizes the average of gradients using RMSprop of Hinton at el., 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:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r}\left[\left(\mathbb{E}_{s'}[y \mid s, a] - Q(s, a; \theta_i)\right)^2\right] \tag{23}$$

$$= \mathbb{E}_{s,a,r}\left[\left(y - Q(s, a; \theta_i)\right)^2\right] + \mathbb{E}_{s,a,r}[\mathbb{V}_{s'}[y]] \tag{24}$$

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

$$\text{with } y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-) \tag{26}$$

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 (24) (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.

$$\delta_i = \left\{ r^{(m)} + \gamma \max_{a' \in \mathcal{A}(s'^{(m)})} Q\left(s'^{(m)}, a'; \theta_i^-\right) - Q\left(s^{(m)}, a^{(m)}; \theta_i\right) \right\}_{m=1}^{M} \tag{27}$$

where $\delta_i$ is set of TD-error, and

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

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 (23)) as follows:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(y - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right] \tag{28}$$

$$= \aleph \nabla_{\theta_{i-1}} L_{i-1}(\theta_{i-1}) + (1-\aleph) \sum_{m=1}^{M} \delta_i^{(m)} \nabla_{\theta_i} Q\left(s^{(m)}, a^{(m)}; \theta_i\right) \tag{29}$$

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 $\mathrm{MSE}^{\mathrm{SV}}$

$$\begin{aligned}
L_i(\theta_i) &= \mathbb{E}_{s,a,r,s'}\left[\left(y - Q(s,a;\theta_i)\right)^2\right] \\
&\neq \mathrm{MSE}^{\mathrm{SV}}\left(y, Q(s,a;\theta_i)\right) \\
&= \frac{1}{M}\sum_{m=1}^{M} \delta_i^{(m)^2}, \\
\nabla_{\theta_i} L_i(\theta_i) &= \mathbb{E}_{s,a,r,s'}\left[\left(y - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right] \\
&\neq \frac{d}{d\theta_i}\mathrm{MSE}^{\mathrm{SV}}\left(y, Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i) \\
&= \frac{2}{M}\sum_{m=1}^{M} \delta_i^{(m)} \nabla_{\theta_i} Q\left(s^{(m)}, a^{(m)}; \theta_i\right).
\end{aligned}$$

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 $\mathrm{MSE}^{\mathrm{SV}}$ 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 $\frac{2}{M}\sum_{m=1}^{M}\left(x^{(m)} - y^{(m)}\right)$, so the derivative of Eq (28) should become $2\mathbb{E}_{s,a,r,s'}\left[\left(y - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$. However, since Eq (25) can be considered assuming One Half MSE (Ng, 2016), $\frac{1}{2M}\sum_{m=1}^{M}\left(x^{(m)} - y^{(m)}\right)^2$, that is, it derivative is $\frac{1}{M}\sum_{m=1}^{M}\left(x^{(m)} - y^{(m)}\right)$, so Eq (28) 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 at el., 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 at el., 2012. For that reason, , Mnih et al., 2015 modified RMSprop of Hinton at el., 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 at el., 2012 can be write as follows:

$$g_i = \aleph g_{i-1} + (1-\aleph) d\theta_i \quad \text{by (29)} \tag{30}$$

$$n_i = \aleph n_{i-1} + (1-\aleph) {d\theta_i}^2 \tag{31}$$

$$\Delta_i = \beth \Delta_{i-1} + \gimel \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \daleth}} \tag{32}$$

$$\theta_{i+1} = \theta_i + \Delta_i \tag{33}$$

where $\gimel$ is learning rate, $\daleth$ 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 (DeepMind, 2017) 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 $\theta$ 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 $\theta^-$ for estimate the next action value. The parameter $\theta^-$ is copied from parameter $\theta$ at fixed intervals, and unchange it until next copy. In the meantime, the parameter $\theta$ 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. The reproduction method of the realization method of the semantics of Deep Q-network

While large number of iterative updates, the One-step Q-learning accumulates the next value of state-action piar scaled by the step size parameter little by little to current value of state-action piar. So, even if it is a very small difference, it will cause big differing results by being enlarged to a big difference among a large number of iterative updates. 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.

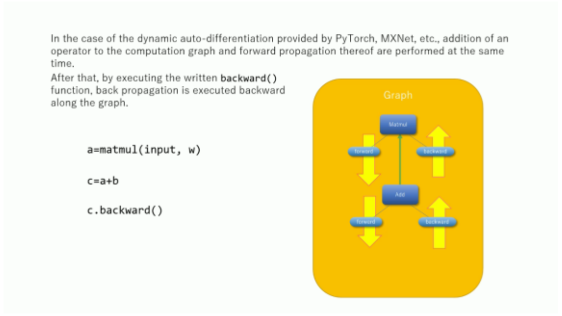
**Table 1**: The Deep learning frameworks used in this work. All framework test uses Tensorflow for visualization. And All DQN version correspond Python 3.5.

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

### 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, 2017 `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.(27)-(33)) 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 comcept of automatic-differenciation.*** 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.


In the case of the dynamic auto-differentiation provided by PyTorch, MXNet, etc., addition of an operator to the computation graph and forward propagation thereof are performed at the same time.
After that, by executing the written **backward()** function, back propagation is executed backward along the graph.

```
a=matmul(input, w)

c=a+b

c.backward()
```

**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 backword 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.backword(init_gradient)
```

or

```
# Tensorflow
tf.gradient(middle_output, weights, init_gradient)[3].
```

Therefore, we can create arbitrary derivative of loss-function like Eq (29).

***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):

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 4 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, 2017 `dqn/NeuralQLearner.lua#L256-L277`) as shows

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

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/02427dc1bd5eaeb8e79d7b7c715f8049cb1dca66/dqn/pytorch_optimizer.py#L135-L154

MXNet:

https://github.com/ElliottTradeLaboratory/DQN/blob/02427dc1bd5eaeb8e79d7b7c715f8049cb1dca66/dqn/mxnet_extensions.py#L83-L121

Tensorflow and CNTK with Keras:

https://github.com/ElliottTradeLaboratory/DQN/blob/02427dc1bd5eaeb8e79d7b7c715f8049cb1dca66/dqn/convnet_keras.py#L261-L317

## Other realization method

***Maximization in preprocessing.*** As mentioned in *Section0.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 '`*Gamename*NoFrameskip-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 "`*Gamename*-v0`", or, implement a frame maximization function after a frameskip function, we obtains different maximized frames than DQN3.0, as shown **Figure 3**-(b).

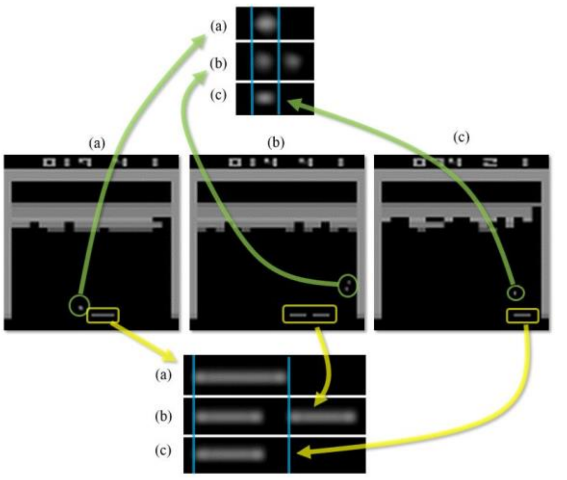


**Figure 3: The comparison of how to implement the maximization.** (a) implemented before frameskip; (b) implemented after frameskip; (c) not implemented; The top (a)~(c) are comparison of how the balls are displayed; The middle (a)~(c) are comparison of how inputs are displayed; The bottom (a)~(c) are 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, 2018, February 26). 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 **Table 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 Torch[4] as follows

$$stdv = \frac{1}{\sqrt{fan\_in^{Weight}}} \tag{34}$$

$$\theta^{Weight} \sim \mathcal{U}(-stdv, stdv) \tag{35}$$

$$\theta^{Bias} \sim \mathcal{U}(-stdv, stdv). \tag{36}$$

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 betwen 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, 2017 `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

[4] https://github.com/torch/nn/blob/872682558c48ee661ebff693aa5a41fcdefa7873/SpatialConvolution.lua#L34-L55

compares elements and thresholds one by one during `for` loop.[5,6]; 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/6ef872a83e1d618563339bd1ab1d21ab662cbd3b/dqn/pytorch_extensions.py#L105-L111

PyTorch(New):

https://github.com/ElliottTradeLaboratory/DQN/blob/6ef872a83e1d618563339bd1ab1d21ab662cbd3b/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/6ef872a83e1d618563339bd1ab1d21ab662cbd3b/dqn/cntk_extensions.py#L71-L111
(2) https://github.com/ElliottTradeLaboratory/DQN/blob/6ef872a83e1d618563339bd1ab1d21ab662cbd3b/dqn/cntk_extensions.py#L118-L129

Tensorflow with Keras:

https://github.com/ElliottTradeLaboratory/DQN/blob/6ef872a83e1d618563339bd1ab1d21ab662cbd3b/dqn/tensorflow_extensions.py#L80-L103

## 4. Related works

Many DQNs have been created and released so far, but I could not evaluate all of them. This section introduce two DQN implementations which well learn Atari2600 games among I evaluated.

First, 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:

$$\delta_i = y - Q(s, a; \theta_i)$$

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

Their Huber loss-function uses only the sum instead of the mean[7], 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:

$$\begin{aligned} g_i &= \aleph g_{i-1} + \left(1 - \aleph\right) d\theta_i^{\text{Sprague}} \\ n_i &= \aleph n_{i-1} + \left(1 - \aleph\right) d\theta_i^{\text{Sprague}^2} \\ \theta_{i+1} &= \theta_i - \gimel \frac{d\theta_i^{\text{Sprague}}}{\sqrt{n_i - g_i^2 + \daleth}} \end{aligned}$$

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.

On the other hand, 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. Experience and results

### 5.1. 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 builded 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 |

[5] https://github.com/torch/cunn/blob/master/lib/THCUNN/generic/Threshold.cu

[6] https://github.com/torch/cutorch/blob/master/lib/THC/THCApply.cuh

[7] but can select if we want.

| | |
|---|---|
| GPU | GTX1080ti × 2 |
| OS | Ubuntu 16.04 |

## 5.2. 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, 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 epcoch 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. Finaly, 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.

| | Test 30 episodes | | |
|---|---|---|---|
| DQN version | 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 of Sprague, 2016 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 5** and **Table 8** 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 perfomance 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.

## 5.3. 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 6**-(a),(b),(c) shows, my DQNs which the important functions are lack or mis-implementation are underperform DQN3.0 largely. In contrast, as **Figure 6**-(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 6**-(d) shows, there is no big difference by the Actication selection between Rectifier layer and ReLU function.

## 5.4. 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 7**, 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

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

Tensorflow.

### Results

As **Figure 8** 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 .

## 6. Discussion

In this section, I will discuss that how the solution to *Section 1.1*, got through *Section 3~5*, leads to the suggestion of *Section 1.2*.

### 6.1. Solution of "Wrong environment was selected"

This problem was finally solved by examining alewrap and Gym's AtariEnv source code, and comparison of output of my alewrap_py with alewrap (*Section 1.2-(10)*). By doing so, I understood that alewrap contains important implementations (Preprosessing, Frame Skip, Random Start), and that they are not included in Gym's AtariEnv. In this case, if the application structure of DQN and the specification of AtariEnv of Gym are released and it can be read, I think that there was no decision to use Gym. (*Section 1.2-(4)*)

In addition, this problem also includes a major cause that I chose Gym blindly by merely considering that "Gym is popular environment". This blindly choice can be avoided in an extremely basic way as menthoned *Section 1.2-(11)*. (*Section 1.2-(9)*&*(11)*)

### 6.2. Solution of "Wrong implement method of DQN loss-function was selected"

This problem was finally solved by deepening the understanding of the DQN loss-function by reading Sutton and Barto, 2018 and learning key technologies of reinforcement learning. Sutton and Barto, 2018 is a must-read for talking about reinforcement learning algorithms, but the problem is in my attitude that I was frightened by the number of pages of Sutton and Barto, 2018 and I tried to read a simpler document (as published on the Web) there were. For Extremal property, we had to study about expected value. Studying to obtain such basic knowledge can not reproduce the proposal if it does not pass through even if it is troublesome. (*Section 1.2-(9)*).

In the implementation of the DQN loss function, all the explanations of automatic differentiation that the deep learning framework exposes are explained and explained on the assumption that End-to-End forward/back propagation. This seems to deny the user to create an arbitrary differential equation. By adopting a custom operation that allows the creation of arbitrary differential equations, or backpropagation with arbitrary initial-gradient specified like DQN 3.0, we can create a loss function like DQN 3.0. Nevertheless, there are aspects that make it difficult. (*Section 1.2-(7)*)

In RMSprop reproduction, finally I checked all source code of RMSprop and understand that all the RMSprop formulas of the framework I used in this work are different from DQN3.0 and therefore in all versions of my DQN I decided to create the exactry same RMSprop as DQN3.0 from scratch. In the RMSprop API document of each Deep learning framework except MXNnet (MXNet), the mathematical expression is not written, so I had to examine the source code, but if the API document had a mathematical expression like MXNet written Then, I could have done other things at that time (*Section 1.2-(6)*&*(10)*).

In addition, since the implementation method of the innovative DQN loss function is not specifically shown, the possibility that the framework provider could not know the implementation method and could not provide examples or explanations. Algorithms underlying deep reinforcement learning such as DQN should be provided with appropriate examples and explanations by framework providers. (*Section 1.2-(3)*)

### 6.3. Solution of "Wrong functions provided by Deep learning framework and library was selected"

As mentiond in *Grayscale conversion in preprocessing* end *Resizing in preprocessing*, I found that some output of function in Python libraries are different from output of function used by DQN3.0. In the case of grayscale, tf.image need found and examined it source code, but other library explained their grayscale formula in API document, so these libraries except tf.image can speedy understood the specification of grayscale function. In case of resize, I finally compared the output of the resize function of each library with the output of `image.scale` in the manner mentioned in *Comparing output between original function and select one*. That result shown **Table 6**, I found `cv2.resize` with `cv2.INTER_AREA` which return the same resized-frames as `image.scale`. With `cv2.resize` with `cv2.INTER_AREA` is make big improved deep_q_rl's performance, that is, there are functions and/or their arguments with the same name is no guarantee of these returns the same result. Therefore, we need to do with focus on what expression is implemented in this function: reproducers needs to deeply examine it in a function their use; library providers needs to explain it in detail in API document. (*Section 1.2-(6)*&*(10)*).

### 6.4. Solution of "Wrong evaluation method was selected"

As mentioned in *Section 5.2*, the score in the tasks with large variance such as Atari domain should be more stable for the evaluation It: It is not only difficult to evaluate the equivalence of the reproduced work, but also lead to under/overestimation of the proposal. (*Section 1.2-(5)*)

### 6.5. Solution of "I could not build the running environment of some DQN implementation"

## 7. Conclusion

Like with everyone can reproduce $1 + 1 = 2$ between arbitrary programming languages and libraries, DQN can also be accurately reproduced by taking correct steps. In order to improve the reproducibility more, people related to AI application development need to select the desirable actions corresponding their own role.

## 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. DeepMind. Lua/Torch implementation of DQN (Nature, 2015), April 2017. URL https://github.com/deepmind/dqn
3. 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
4. DeepMind. A lua wrapper for the Arcade Learning Environment/xitari, December 2014. URL https://github.com/deepmind/alewrap
5. 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
6. Chintala, Soumith. cunnCUDA:some depreceated, ugly and old modules. (2015). Web 29 November 2017. URL https://github.com/soumith/cunnCUDA
7. Ha, D. (2018). World Models Experiments. Web 11 June 2018. URL http://blog.otoro.net/2018/06/09/world-models-experiments/
8. Ha, D., & Schmidhuber, J. (2018). World Models. arXiv preprint arXiv:1803.10122. URL https://arxiv.org/abs/1803.10122
9. MXNet, rmspropalex_update, MXNet API. Web 11 June 2018. URL https://mxnet.incubator.apache.org/api/python/ndarray/ndarray.html#mxnet.ndarray.rmspropalex_update
10. 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
11. OpenAI. atari-py. (2017). URL https://github.com/openai/atari-py.
12. Wikipedia contributors. (2018). 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
13. Sutton, R. S., & Barto, A. G. (2018). RL: An introduction. Second edition, in progress ＊＊＊＊Complete Draft＊＊＊＊. URL http://incompleteideas.net/book/bookdraft2018jan1.pdf
14. DeepMind. Xitari is a fork of the Arcade Learning Environment v0.4 (2017). URL https://github.com/deepmind/xitari.
15. Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. arXiv preprint arXiv:1511.05952. URL https://arxiv.org/abs/1511.05952.
16. Wikipedia contributors. (2018, April 23). 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
17. 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
18. Graves, A. (2013). Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850v5. URL https://arxiv.org/abs/1308.0850
19. Wikipedia contributors. (2018). Grayscale. In Wikipedia, The Free Encyclopedia. Web 4 May 2018 URL https://en.wikipedia.org/w/index.php?title=Grayscale&oldid=827764362
20. 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
21. 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
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. 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
24. Tensorflow.org (2018, April, 28). Adding a New Op In Extend. Web 5 May 2018 URL https://www.tensorflow.org/extend/adding_an_op
25. Sprague, N. (2016). Theano-based implementation of Deep Q-learning. URL https://github.com/spragunr/deep_q_rl
26. Roderick, M., MacGlashan, J., & Tellex, S. (2017). Implementing the Deep Q-Network. arXiv preprint arXiv:1711.07478. URL https://arxiv.org/abs/1711.07478
27. 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, 2017 dqn/NeuralQLearner.lua)

Eq (25): $y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$

```
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(s_2, 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)
```

Eq (27): $\delta_i = \left\{ r^{(m)} + \gamma \max_{a'} Q\left(s'^{(m)}, a'; \theta_i^-\right) - Q(s^{(m)}, a^{(m)}; \theta_i) \right\}_{m=1}^{M}$

```
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)
```

Error Clipping

```
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
```

Part of Eq (29): $\sum_{m=1}^{M} \delta_i^{(m)} \nabla_{\theta_i} Q\left(s^{(m)}, a^{(m)}; \theta_i\right)$

```
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
254 self.network:backward(s, targets)
```

Eq (29) or Eq (30): $g_i = \aleph g_{i-1} + (1-\aleph) d\theta_i$

```
266  self.g:mul(0.95):add(0.05, self.dw)
```

Eq (31): $n_i = \aleph n_{i-1} + (1-\aleph) d\theta_i{}^2$

Eq (32): $\Delta_i = \beth \Delta_{i-1} + \gimel \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \daleth}}$

Eq (33): $\theta_{i+1} = \theta_i + \Delta_i$

```
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)
```

**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)[9]:

$$g_i = \aleph g_{i-1} + (1 - \aleph) d\theta_i \tag{37}$$

$$n_i = \aleph n_{i-1} + (1 - \aleph) d\theta_i{}^2 \tag{38}$$

$$\Delta_i = \beth \Delta_{i-1} + \gimel \frac{d\theta_i}{\sqrt{n_i - g_i^2} + \daleth} \tag{39}$$

$$\theta_{i+1} = \theta_i - \Delta_i \tag{40}$$

Argument mapping

| PyTorch | Symbol | Mnih et al., 2015 Data Table 1 / DQN3.0 |
|---|---|---|
| `lr` | $\gimel$ | learning rate |
| `alpha` | $\aleph$ | gradient momentum or squared gradient momentum |
| `eps` | $\daleth$ | min squared gradient |
| `weight_decay` | — | `self.wc` (DQN3.0) |
| `momentum` | $\beth$ | `0` (DQN3.0) |

Centered RMSprop of MXNet[10,11]:

$$g_i = \aleph g_{i-1} + (1 - \aleph) d\theta_i \tag{41}$$

$$n_i = \aleph n_{i-1} + (1 - \aleph) d\theta_i{}^2 \tag{42}$$

$$\Delta_i = \beth \Delta_{i-1} - \gimel \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \daleth}} \tag{43}$$

$$\theta_{i+1} = \theta_i + \Delta_i \tag{44}$$

Argument mapping

| MXNet | Symbol | Mnih et al., 2015 Data Table 1 / DQN3.0 |
|---|---|---|
| `lr` | $\gimel$ | learning rate |
| `gamma1` | $\aleph$ | gradient momentum or squared gradient momentum |
| `gamma2` | $\beth$ | `0` (DQN3.0) |
| `epsilon` | $\daleth$ | min squared gradient |

Centered RMSprop of Tensorflow[12]:

$$g_i = \aleph g_{i-1} + (1 - \aleph) d\theta_i \tag{45}$$

$$n_i = \aleph n_{i-1} + (1 - \aleph) d\theta_i{}^2 \tag{46}$$

$$\Delta_i = \beth \Delta_{i-1} + \gimel \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \daleth}} \tag{47}$$

$$\theta_{i+1} = \theta_i - \Delta_i \tag{48}$$

Argument mapping

| Tensorflow | Symbol | Mnih et al., 2015 Data Table 1 / DQN3.0 |
|---|---|---|
| `learning_rate` | $\gimel$ | learning rate |
| `decay` | $\aleph$ | gradient momentum or squared gradient momentum |
| `momentum` | $\beth$ | `0` (DQN3.0) |
| `epsilon` | $\daleth$ | min squared gradient |

[9] https://pytorch.org/docs/stable/_modules/torch/optim/rmsprop.html#RMSprop

[10] https://mxnet.incubator.apache.org/api/python/optimization/optimization.html#mxnet.optimizer.RMSProp

[11] https://mxnet.incubator.apache.org/api/python/ndarray/ndarray.html#mxnet.ndarray.rmspropalex_update

[12] 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: 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.

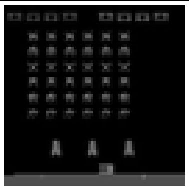
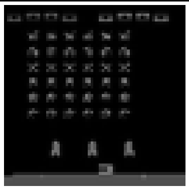
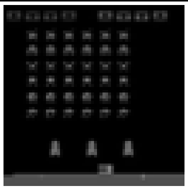
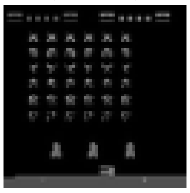
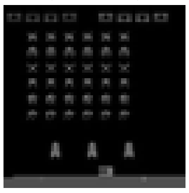
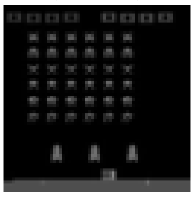
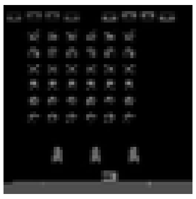
| Library | Interpolation<br>Bilinear<br>(MSE) | <br>Area<br>(MSE) |
|---|---|---|
| torch.image |  | ———— |
| OpenCV | <br>(7.04E-04) | <br>(0.00E+00) |
| tf.image | <br>(1.13E-03) | <br>(1.10E-09) |
| Pillow | <br>(1.49E-04) | ———— |
| scikit-image | <br>(7.04E-04) | ———— |

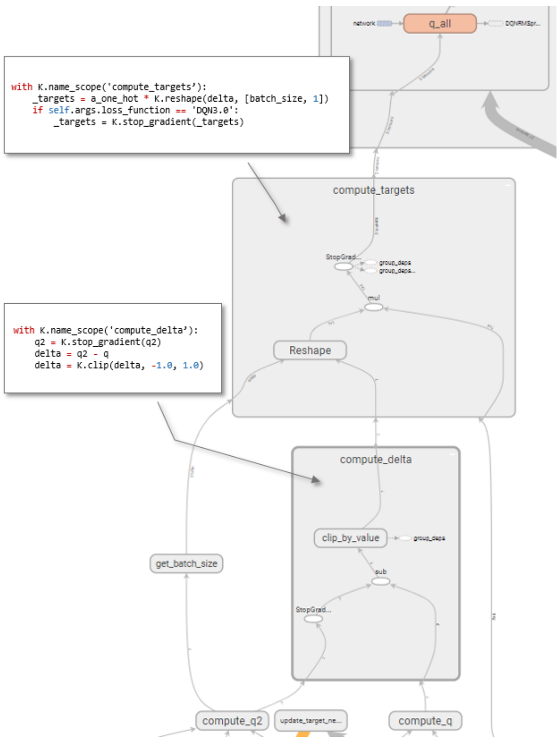
**Table 7: The original-setting.** The original-setting aims to obtain the same results as DQN3.0. "original-HP" means the same hyperparameter as Mnih et al., 2015 P.10 Extended Data Table 1. "`Torch nn layers default`" means the same initializer as DQN3.0 with Eq (34)~(36).

| 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 8 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 scpre 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.

| | | (a) Breakout | | | | | | | (b) Space invaders | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Test 100 episodes | | | Test 30 episodes | | | | Test 100 episodes | | | Test 30 episodes | | | |
| | DQN version | 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 | — | — | — | — | — | — | — | — | — | — | — | — | — | — |
| MyDQN | 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


```
with K.name_scope('compute_targets'):
    _targets = a_one_hot * K.reshape(delta, [batch_size, 1])
    if self.args.loss_function == 'DQN3.0':
        _targets = K.stop_gradient(_targets)
```

```
with K.name_scope('compute_delta'):
    q2 = K.stop_gradient(q2)
    delta = q2 - q
    delta = K.clip(delta, -1.0, 1.0)
```

**Figure 4: Part of the computation graph of DQN loss-function in Tensorflow.**

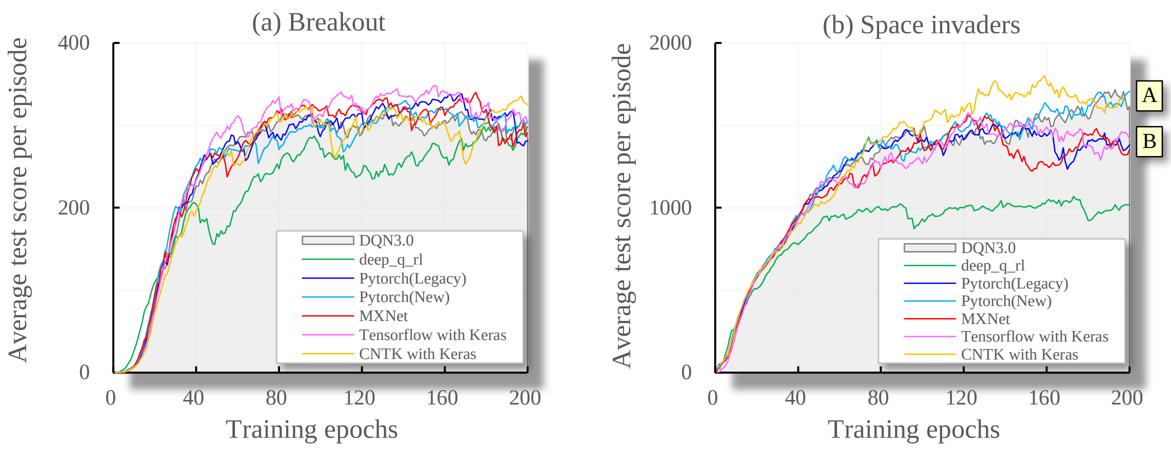


**Figure 5 : Comparison of training curves in Breakout and Space invaders (with original-hyperparameter and DQN reproduction method).** Training curves tracking each agent's average test score with the original-setting (Table 7). In the case of deep_q_rl, specifying `device=gpu` in `.theanorc` improved 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.

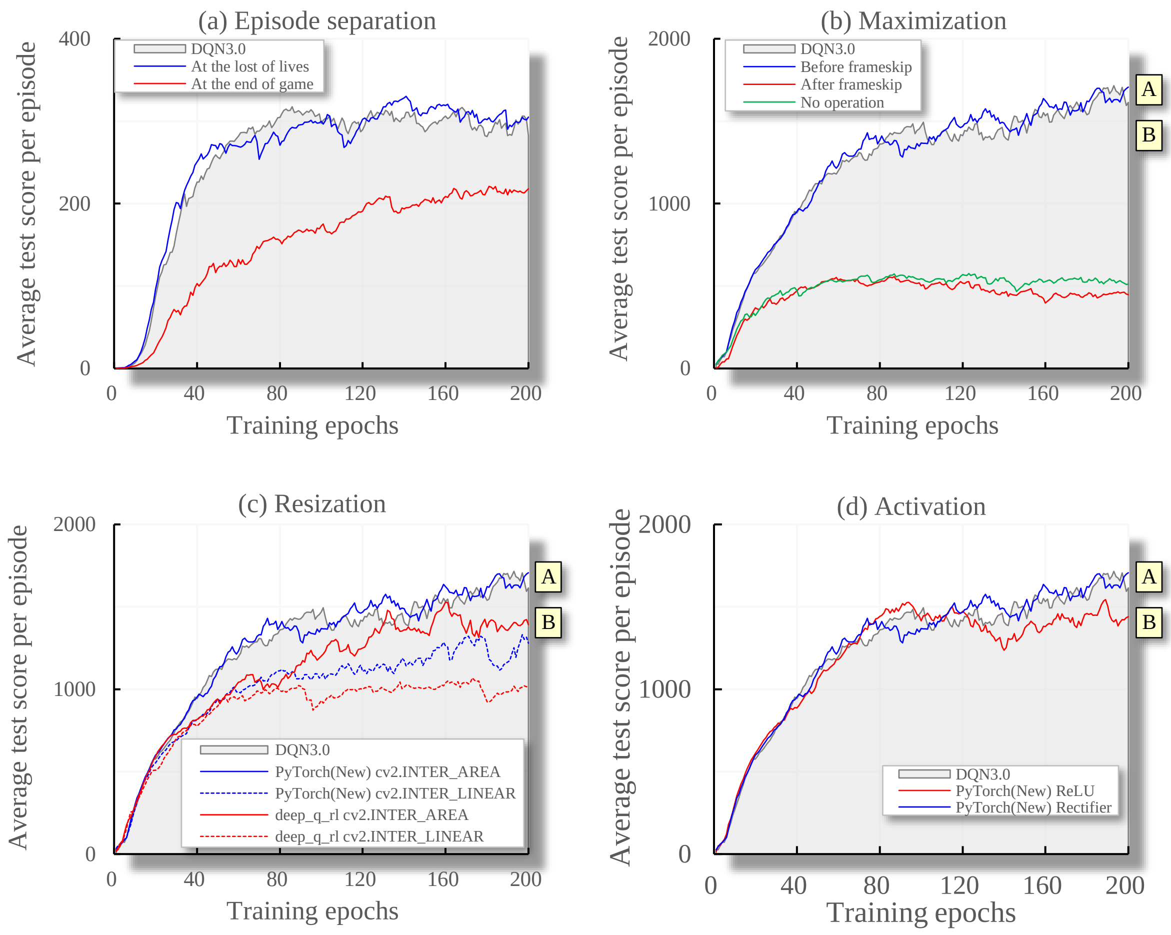


**Figure 6: Comparison of training curves of each setting.** (a) comparison of episode separation setting; (c) comparison of resization setting in Space invaders; (d) comarision of activation setting in Space invaders. (a)~(d) are used myDQN with PyTorch(New) with the original-setting and deep_q_rl. 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.

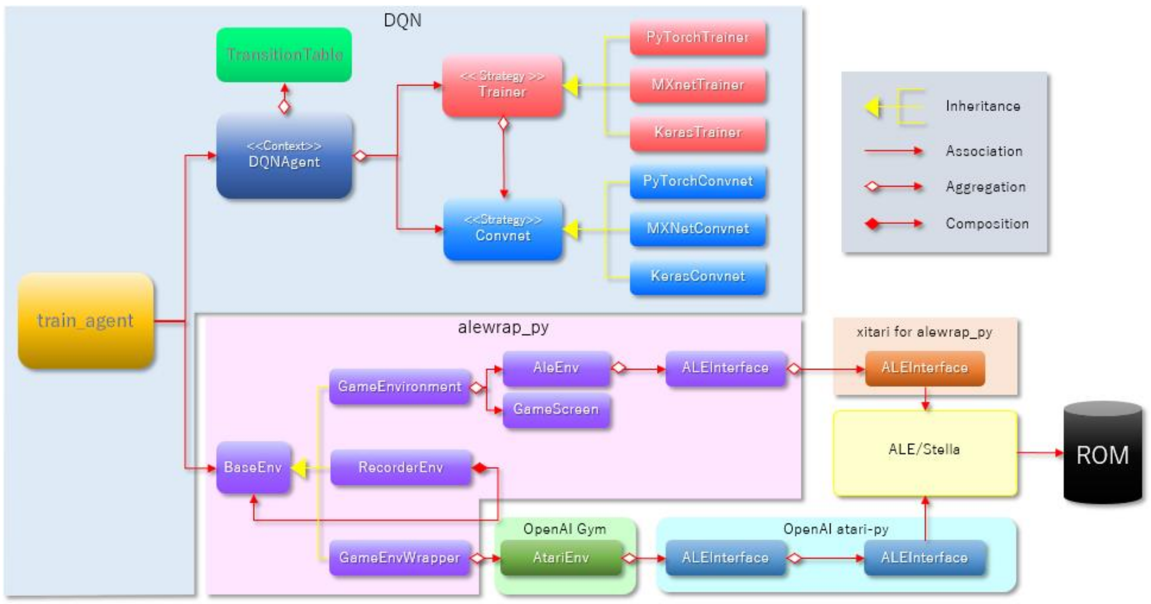


**Figure 7: The architecture of my DQN.** My DQN is an implement of the Strategy pattern (Erich et al., 1995): Client: `train_agent`, Context: `DQNAgent`, Strategy: `Trainer` and `Convet`; Each ConcreteStrategy classes corresponds each Deep learning framework. alewrap_py reproduced alewrap in other package 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).

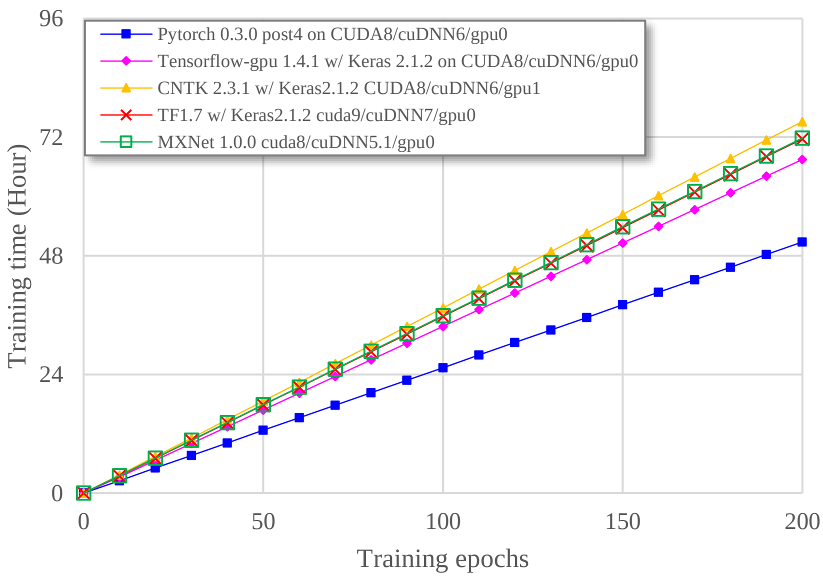


**Figure 8: Comparison of training time of each Deep learning.**