The Reproduction Method of Deep Q-network with DQN3.0-level Performance

Ver 1.0 31 May 2018

Shigeru Matsumura

Abstract

It is very easy way to reproduce a Deep Q-network (DQN) (Mnih et al., 2015) with the same performance as DQN3.0 (DeepMind, 2017), just create it in the same creation method as DQN3.0. But as far as I know, there is nowhere to explanation of easiest way as above, and also its implementation. So I studied the reproduction method of DQN in DQN3.0, and succeeded reproduce some version of DQNs with DQN3.0-level performance (**Figure 1**). This article will explained the reproduction method of DQN in detail and how to implement a DQN with DQN3.0-level performance using Python with major Deep learning frameworks such as PyTorch, MXNet, Tensorflow and CNTK (the last two uses Keras). After that, proves that reproducing DQN3.0 with other languages and Deep learning frameworks is the same as we can reproduce $1 + 1 = 2$ with any programming language and Deep learning framework.

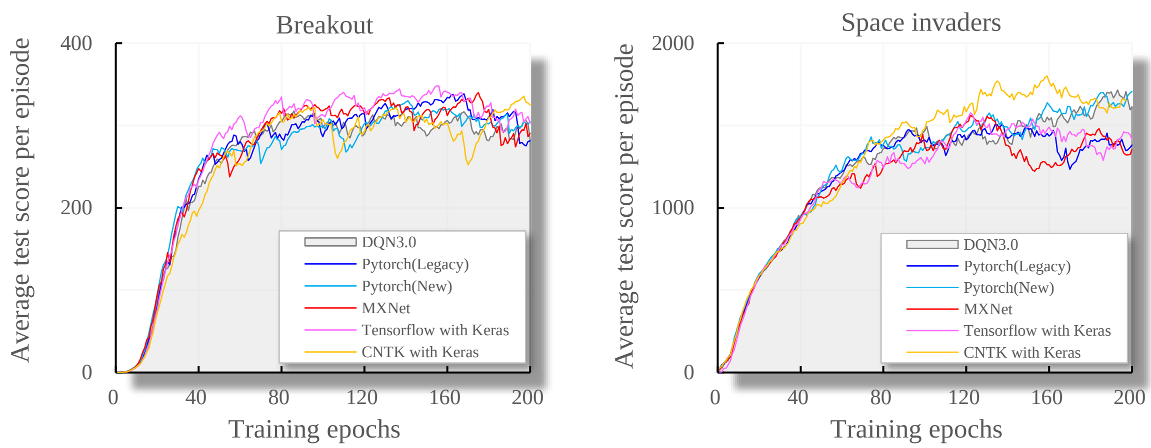


Figure 1: Comparison of training curves in Breakout and Space invaders.

1. Introduction

DQN is an epoch making algorithm which applying a neural network and a replay memory on conventional Reinforcement learning algorithm proposed Mnih et al., 2015 together with its implementation DQN3.0 (DeepMind, 2017) in February 2015. However, it has also a mysterious aspect that, despite the fact that DQN is the fundamental algorithm of Deep reinforcement learning which motivates the creation of many improved algorithms later, there are nowhere on the internet that the accuracy reproduced implementation and the explanation of the accuracy reproduction method. As far as I know, their DQN I evaluated has low performance than DQN3.0 or can not learn[1].

In contrast, as shown in **Figure 1**, I was able to create DQN with DQN3.0 level performance with four major Deep learning frameworks on Python. Because I knew the two important points necessary to reproduce a work created with other programming language and Deep learning framework:

First, in principle, even if the source code is written with different programming language and Deep learning framework, the same result can be obtained if it can be finally compiled into a similar machine language executable on our computer's CPU; so anyone should be able to reproduce it (eg. $1 + 1 = 2$) using any programming language and any Deep learning framework. That is true even in the case of repeoduction of DQN.

Second, on the other hand, in order to reproduce a work with different programming language and Deep learning framework, it is necessary to satisfy other two important points. One, **to inherit the semantics of source work**: For example, if we reproduce $1 + 1 = 2$ which has semantics "the addition of number of fruits", using different programming language and Deep learning framework, we need to assume "the addition of number of fruits" similarly rather than "the addition of number of apples" to semantics of our $1 + 1 = 2$, 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. Other one, **to accuracy reproduce the realization methods of the semantics**: For example, we can obtain unexpected result when we use subtract function rather than addition function for $1 + 1 = 2$, or use an addition function which returns 2.2 when 1 and 1 are given.

[1] Of course, the DQNs I evaluated are only a few of the many DQN implementations as a sample.

In the reproduction of DQN, the semantics of DQN is necessary to include two aspects depending on purpose of Mnih et al., 2015. One is, as an aggressive solver of Atari2600 domain: DQN was developed to achieve that a single model can be develop ability for multiple problems. In order to achieve their purpose, they chose the Arcade Learning Environment (ALE) (Bellemare et al., 2013) to learn Atari2600 games and extended it to explore solutions aggressively. The other one is, as a novel variant of One-step Q-learning: DQN is a One-step Q-learning extended with a neural network and a replay memory, even if DQN applied a neural network containing convolution layers like a neural network of model in image classification task. That is, DQN is not an image classifier which estimate an action with respect to a screen of Atari game. Likewise, even if the word "the mean-squared error" is written in Mnih et al., 2015 P.7, do not forget that the DQN loss-function is the expected value represented by the symbol $\mathbb{E}$ rather than the commonly mean squared error function like provided by Deep learning framework.

In addition, in order to achieve the realization of DQN semantics, we need to follow the realization methods of the semantics of DQN in DQN3.0 faithfully: Although it may seem to be out of the common sense of Deep learning seemingly, it is following the common sense of Reinforcement learning; and, for more accurate reproduction, we need to deeply examine the functions provided by Deep learning frameworks and libraries we use, because those functions do not necessarily return the results we expect, even if it has the same name and/or refer to the same paper.

The motivation of this article is to breakthrough the mysterious aspect of DQN together with to demonstrate an example of reproduction of proposed works, by these explanations.

2. Background

2.1. Key idea and algorithms of Reinforcement learning

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 $t = 0,1,2,\ldots,T$, 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 $s0, r0, a0, s1, r1, a1 \ldots, s_T, r_T$ and $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, $r \in \mathcal{R}$, where $\mathcal{A}(\mathrm{s})$ is set of possible actions in a state s. While the environment-agent interaction, the agent has learn that how to take actions aiming obtain maximum cumulative reward according to *policy* π. The cumulative rewards is denoted as follows:

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

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

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

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 equation (7) is called *the Bellman equation* q_π, 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 equation (7) represents components of the action-value function:

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

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 equation (10) is:

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

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:

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

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 n states, then there are n equations in n unknowns.***
> **Sutton and Barto, 2018 P.50~51**

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

Q-learning. In real situations, for achieving our purpose, sometimes we must take an action can arrive at next desirable situation which has very low occurrence probability. For example, in the role-playing game, some time only a player who has a special item with very low occurrence probability can achieve their purpose. In this case, these algorithms influenced by environmental dynamics may not be able to solve the situation because they cannot obtain special items underestimated its value by scaling using environmental dynamics. In addition, in real situations, we may not be able to obtain fully transitions like game record of official match of board game such as Go, Chess and Shogi. The off-policy Temporal-Difference Learning (TDL) algorithm called *Q-learning* can solve that situation because it estimates only the value of the next action in a next state due to the selected an action in a current state, therefore, not influenced by environment dynamics and not required fully transitions.

These computation are formulated as follows:

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

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

Policy. Q-learning is Off-policy TDL, so it uses two policies, the *target policy* and the *behavior policy*: The target policy is more optimal or maybe deterministic. It used for optimal target of learning; On the other hand, the behavior policy is more stochastic: It used for exploration of transitions, while optimized itself toward to target policy. For the behavior policy, often use ε-greedy policy, because, it behavior is changed from the almost random selection to the almost 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.

2.2. 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 OpenAI Gym (Gym) (Brockman et al.,2016); 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.

3. The semantics of Deep Q-network

3.1. The semantics as an aggressive solver of Atari2600 domain of DQN

DQN was developed to achieve that a single model can be develop ability for multiple problems. In order to achieve their purpose, as described below, Mnih et al., 2015 are aggressively taking various measures to solve the Atari 2600 games better.

Preprocess. Mnih et al., 2015 have found that in several Atari 2600 games, important objects are lost due to blinking sprites or being drawn only in certain frames. 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 $\dot{x}_{\tilde{t}}$

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

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

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

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)

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

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

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

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

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

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

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

in this case, ϕ_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:

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

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

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

Thus, their agents will learn and predict with preprocessed states ϕ_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 implemeted in alewrap.

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

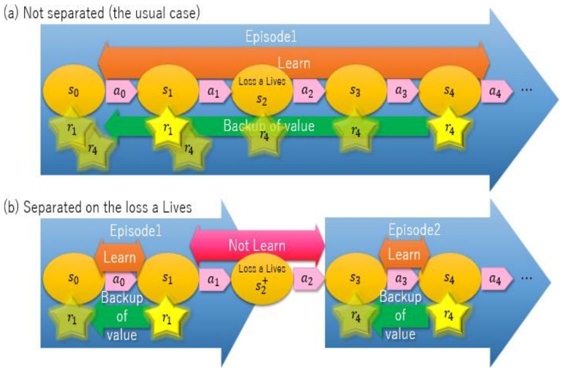


Figure 2: Comparison of learning range with or without episode separation. (a) In the case of the episode which not separated as default, the sequences reaching failure are learned by backup of the future reward r_4 . (b) In the case of the episode separated on the failure and where s_2^+ is the terminal 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^+.

3.2. 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. However, the some other DQNs seem that the use of a neural network with convolution layers with many success stories in the image recognition problem may have caused misunderstanding that DQN is just an image classifier. This section explains the semantics of replay memory in DQN3.0 and explains the ingenuity of Q-learning update when using the neural network seen in DQN3.0, thereby I emphasize that the semantics of DQN still inherit the semantics of Reinforcement learning even if it applying a neural network.

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 1×10^6 experiences, and forget experiences earlier than $t - 1 \times 10^6$.

The TransitionTable cycles through the table and inserts a new experience, instead of use a queue. During first 50,000 steps of training, the agent is not learn but only stack experiences into TransitionTable. After that, the agent has learn from experiences sampled from TransitionTable. In learning iteration ith, 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 *3.1*, *Preprocess (e)* would be applied to ϕ and ϕ', but, only ϕ' allows including terminal state. Because, ϕ' is used for only estimate the target action value and the agent need to learn final reward in the episode.

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

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

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

In this case, the update formula of the Q-function was changed to the update formula for many elements corresponding to the features in the parameter, instead of a simple update expression between scalars. In addition, they need to accumulate state-action values of each actual state which has different history information as feature quantities corresponding a features within elements in the parameter. For this reason, Mnih et al., 2015 devised a novel update method: First, by Extremal property (Wikipedia , 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, 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:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r}\left[\left(\mathbb{E}_{s'}[y \mid s, a] - Q(s, a; \theta_i)\right)^2\right] \tag{23}$$
$$= \mathbb{E}_{s,a,r}\left[\left(y - Q(s, a; \theta_i)\right)^2\right] + \mathbb{E}_{s,a,r}\left[\mathbb{V}_{s'}[y]\right] \tag{24}$$
$$= \mathbb{E}_{s,a,r,s'}\left[\left(y - Q(s, a; \theta_i)\right)^2\right], \tag{25}$$

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

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, hence y contains variance, thus 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, $\mathbb{E}_{s,a,r}\left[\mathbb{V}_{s'}[y]\right]$, as shown equation (24) (Mnih et al., 2015 P.7).

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

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

where δ_i is set of TD-error, and

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

The neural network does not has history of the outcome of all trials of each generalized state 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, equations (23) as follows:

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

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

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 $\mathrm{MSE^{SV}}$

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

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 $\mathrm{MSE^{SV}}$ which is designed for supervised learning, assuming that all mini batches aim to 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 equation (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 equation (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 equation(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. Equation (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:

$$g_i = \aleph g_{i-1} + (1-\aleph)d\theta_i \quad \text{by (28)} \quad (30)$$

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

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

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

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 follows:

Source code (DeepMind, 2017 dqn/NeuralQLearner.lua)

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

```lua
term = term:clone():float():mul(-1):add(1)


local target_q_net
if self.target_q then
    target_q_net = self.target_network
else
    target_q_net = self.network
end


-- Compute max_a Q(s_2, a).
q2_max = target_q_net:forward(s2):float():max(2)


-- Compute q2 = (1-terminal) * gamma * max_a Q(s2, a)
q2 = q2_max:clone():mul(self.discount):cmul(term)


delta = r:clone():float()


if self.rescale_r then
    delta:div(self.r_max)
end
delta:add(q2)
```

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

```lua
local q_all = self.network:forward(s):float()
q = torch.FloatTensor(q_all:size(1))
for i=1,q_all:size(1) do
    q[i] = q_all[i][a[i]]
end
delta:add(-1, q)
```

Error Clipping

```lua
if self.clip_delta then
    delta[delta:ge(self.clip_delta)] = self.clip_delta
    delta[delta:le(-self.clip_delta)] = -self.clip_delta
end
```

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

```lua
local targets = torch.zeros(self.minibatch_size, self.n_actions):float()
for i=1,math.min(self.minibatch_size,a:size(1)) do
    delta[delta:le(-self.clip_delta)] = -self.clip_delta
end
self.network:backward(s, targets)
```

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

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

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

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

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

```lua
self.tmp:cmul(self.dw, self.dw)
self.g2:mul(0.95):add(0.05, self.tmp)
self.tmp:cmul(self.g, self.g)
self.tmp:mul(-1)
self.tmp:add(self.g2)
self.tmp:add(0.01)
self.tmp:sqrt()

-- accumulate update
self.deltas:mul(0):addcdiv(self.lr, self.dw, self.tmp)
self.w:add(self.deltas)
```

Table 1: Correspondence of equations and source code.

Policy. As One-step Q-learning, also DQN uses the behavior policy and the target policy: The behavior policy in DQN uses a network with parameter θ for estimate the action value. It also has tie-braking method; On the other hand, the target policy in DQN uses a target network with parameter θ^- for estimate the next action value. The parameter θ^- is copied from parameter θ at fixed intervals, and unchange it until next copy. In the meantime, the parameter θ will be change by learning, that is, $\theta \neq \theta^-$. Therefore, it will be able to make two independed policies as the behavior policy and the target policy. This method was devised by Mnih et al., 2015 for the purpose of improving the stability of the algorithm.

4. The realization method of the semantics of DQN

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 framework and library 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 2**. So, this article is based on these versions.

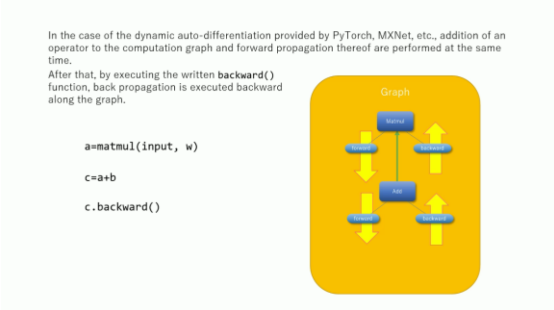
DQN version	Install libraries	CUDA ver	cuDNN ver
PyTorch(Legacy)	PyTorch0.3.0.post4	8.0	6.0
PyTorch(New)	Tensorflow1.4.1(CPU)		
MXNet	MXNet 1.0.0	8.0	5.1
	Tensorflow1.4.1(CPU)		
Tnesorflow	Tensorflow-gpu1.4.1	8.0	6.0
	Keras 2.1.2		
CNTK	CNTK 2.3.1	8.0	6.0
	Keras 2.1.2		
	Tensorflow1.4.1(CPU)		

Table 2: The Deep learning frameworks used in this work. All framework test uses Tensorflow for visualization.

4.1. Realization of DQN loss-function

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-L254`), which seems to be completely different from the common sense of modern Deep learning framework which has the automatic-differentiation. But it is right as it follows the common sense of Reinforcement learning faithfully. For understand the realization of the semantics of the Q-function update as DQN loss-function in modern Deep learning framework, first of all, we need to understand the mechanism of automatic-differentiation --it shows the DQN loss function of DQN3.0 is not illegal usage of backpropagation in modern Deep learning framework. After that, see its implementation.

The mechanism 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.



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 equation (29).

How to implement DQN loss-function. For PyTorch and MXNet which has dynamic automatic-differentiation can write

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

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 3** shows, this reproduction method can be construct the computation graph having the same computation procedure as DQN3.0.

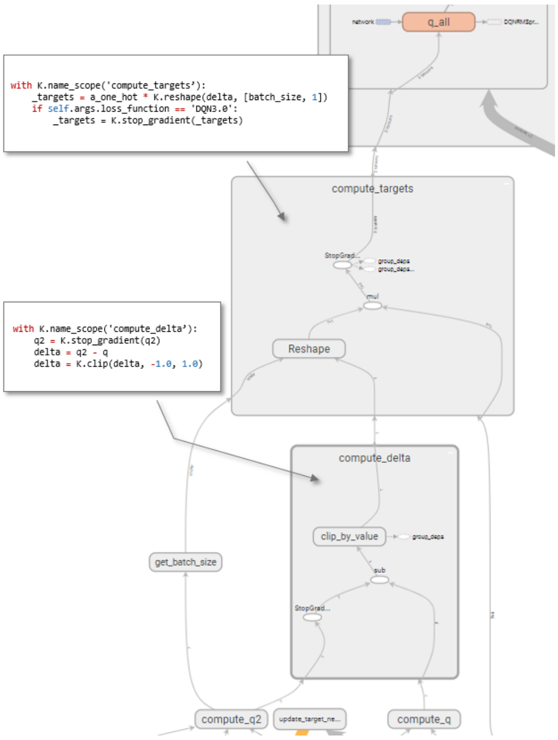


Figure 3: Part of the computation graph of DQN loss-function in Tensorflow.

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.

4.2. Other realization method

Maximization in preprocess. As mentioned in *Section3.1.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 4**-(b).

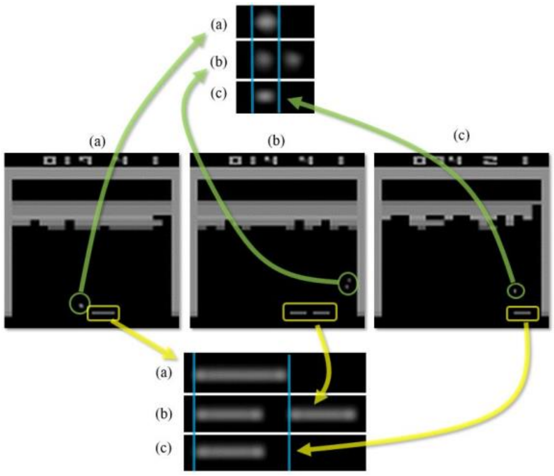


Figure 4: The comparison of how to implement the maximization. (a) implemented before frameskip; (b) implemented after frameskip; (c) not implemented; The middle (a)~(c) are comparison of how inputs are displayed; The top (a)~(c) are comparison of how the balls are displayed; The bottom (a)~(c) are comparison 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.

Grascaling in preprocess. 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, `tf.image` and Pillow. However, scikit-image does not has a such grayscale function, instead it has a grayscale function as `rgb2gray()` with the formula as follow

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

Resizing in preprocess. In generally, each image processing library provides resize function with Bilinear interpolation (Bilinear) as default. In DQN3.0, it was also used `image.scale()` with `'bilinear'` as a default interpolation. However, if we use different image processing library, we may obtain different output of it resize function, even if it interpolation was named as "Bilinear". As **Figure 5** 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.

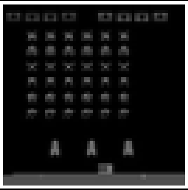
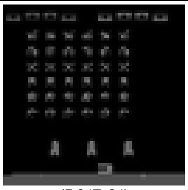
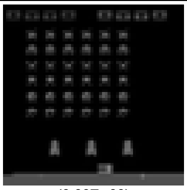
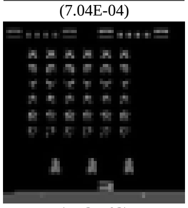
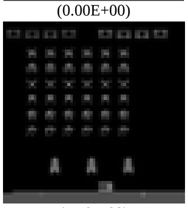
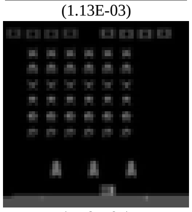
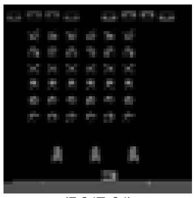
Library	Interpolation: Bilinear (MSE)	Area (MSE)
torch.image		———
OpenCV	 (7.04E-04)	 (0.00E+00)
tf.image	 (1.13E-03)	 (1.10E-09)
Pillow	 (1.49E-04)	———
scikit-image	 (7.04E-04)	———

Figure 5: 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.

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 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. But, on the other hand, however, it was not considered that some users wanted to create operations with different forward / back propagation expressions[7]. 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

[4] https://github.com/torch/nn/blob/872682558c48ee661ebff693aa5a41fcdefa7873/SpatialConvolution.lua#L34-L55
[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] Note that, this article is based on Tensorflow 1.4.1.

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

RMSprop. 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 **Table 3**. 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

Centered RMSprop of PyTorch(New)[8]:

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

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

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

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

Centered RMSprop of PMXNet[9,10]:

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

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

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

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

Centered RMSprop of Tensorflow[11]:

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

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

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

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

Table 3: 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.

5. 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:

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

Their Huber loss-function uses only the sum instead of the mean[12], 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

[8] https://pytorch.org/docs/stable/_modules/torch/optim/rmsprop.html#RMSprop
[9] https://mxnet.incubator.apache.org/api/python/optimization/optimization.html#mxnet.optimizer.RMSProp
[10] https://mxnet.incubator.apache.org/api/python/ndarray/ndarray.html#mxnet.ndarray.rmspropalex_update
[11] 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.

[12] but can select if we want

$d\theta_i^{\text{Sprague}} = \nabla_{\theta_i} L_i^{\text{Sprague}}(\theta_i)$, their RMSprop can be denote as follows:

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

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.

6. Experiments and results

6.1. 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 4**. In addition, as shown in **Table 2**, 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.

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

Table 4: The environment spec of experiments.

6.2. Basic performance evaluation.

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 evaluation, 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. 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.

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.

In addition, **Figure 6** and **Table 6** show that in Breakout, there is a difference between the measured value of DQN3.0 and the result described in Mnih et al., 2015 P.10 Extended Data Table 1. Probably, this seems to be due to the fact that `nn.SpatialConvolutionCUDA`, which was usable in those days, is now deprecated and can not be used. Therefore, the training curves of DQN3.0 and each score are actual measured values on my PC using `nn.SpatialConvolution` instead of `nn.SpatialConvolutionCUDA`.

The performance of my DQNs with original-setting and the DQN reproduction method. As **Figure 6** and **Table 6** 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. 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.

The performance of my tuned DQNs in Breakout and Space invaders. As a result of changing Activation and Bias Initializer as shown in **Table 7**, there was a case where the average test score and stability were slightly improved as shown in **Figure 7** and **Table 8**.

6.3. Comparison of training curves by creating DQN with different reproduction method

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.

Episode separation. As shown in **Figure 8**-(a), the training curves are clearly different by episode termination method between at the lost of lives and at the end of game. It shows our DQN can not obtain DQN3.0-level performance if the agent does not implement episode separation.

Maximization. As shown in **Figure 8**-(b), in Space invaders did not learn at all except for "Before frameskip". In Space invaders, the bullets blinks, so if the agent does not implement maximization before frameskip, the agent learns frames without bullets, therefore our DQN can not obtain DQN3.0-level performance.

Resization. As **Figure 8-(c)** shows that there is a clear difference between `cv2.INTER_LINEAR` and `cv2.INTER_AREA` in the training curve in Space invaders, even if we use `cv2.INTER_LINEAR` as the Bilinear interpolation, our DQN cannot obtain DQN3.0-level performance. In addition, by using `cv2.INTER_AREA`, the training curve of deep_q_rl was improved and thereby finally converged to Group B. However, even in this case, it was still underperform DQN3.0 overall.

Actovation. As **Figure 8-(d)** shows, in Space invaders, the training curve of PyTorch(New) with `torch.nn.functional.relu` function was converged to Group B. It shows a degrade from PyTorch (New) with the Rectifier layer, but on the other hand, it was converged to each framework in Group B. GroupB included PyTorch(Legacy) with Recrifier and Tensorflow with Keras with tf.nn.relu function, so I may not say that it is necessary to use Rectifier layer when DQN3.0-level performance is required in our DQN.

6.4. Training time

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

Results. As **Figure 9** 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 .

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 5: 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 equation (34)~(36).

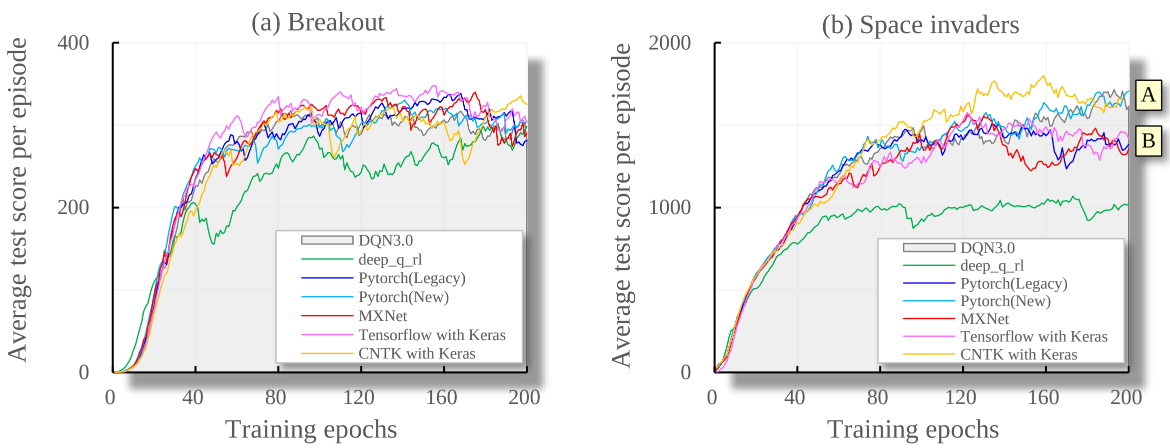


Figure 6: Comparison of training curves in Breakout and Space invaders (with original-hyperparameter and DQN reproduction method). Training curves tracking each agent's average test score with the original-setting (**Table 5**). In the case of deep_q_rl, specifying `device=gpu` in `.theanorc` improved 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.

DQN version	(a) Breakout							(b) Space invaders						
	Test 100 episodes			Test 30 episodes				Test 100 episodes			Test 30 episodes			
	Average score	std	%DQN3.0 Actual	Average score	std	% Mnih et al.,2015	%DQN3.0 Actual	Average score	std	%DQN3.0 Actual	Average score	std	% Mnih et al.,2015	%DQN3.0 Actual
Mnih et al.,2015	—	—	—	401.20	±26.9	—	109.70%	—	—	—	1976.00	±893.00	—	99.81%
DQN3.0 Actual measurement	369.57	±69.0	—	365.73	±84.9	91.16%	—	2018.83	±655.1	—	1979.70	±737.15	100.19%	—
deep_q_rl	—	—	—	—	—	—	—	—	—	—	—	—	—	—
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%
MyDQN 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%
MyDQN 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%
MyDQN 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%
MyDQN 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%

Table 6: 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.

DQN Version	Game	Resize	Activation	Bias initializer
PyTorch(New)	Breakout	`cv2 AREA`	`torch.nn.functional.relu`	`own uniform initializer`
MXNet	Space invaders	`tf AREA`	`mxnet.symbol.Activation relu`	`Torch nn layers default`
Tensorflow with Keras	Breakout	`cv2 AREA`	`tf.nn.relu`	`own uniform initializer`
CNTK with Keras	Breakout	`cv2 AREA`	`cntk.relu`	`own uniform initializer`

Table 7: The setting of my tuned DQNs. the *own uniform initializer* is

$$\theta^{Weight} \sim \mathcal{U}\left(-\frac{1}{\sqrt{fan\_in^{Weight}}}, \frac{1}{\sqrt{fan\_in^{Weight}}}\right) \text{ and } \theta^{Bias} \sim \mathcal{U}\left(-\frac{1}{\sqrt{fan\_in^{Bias}}}, \frac{1}{\sqrt{fan\_in^{Bias}}}\right).$$

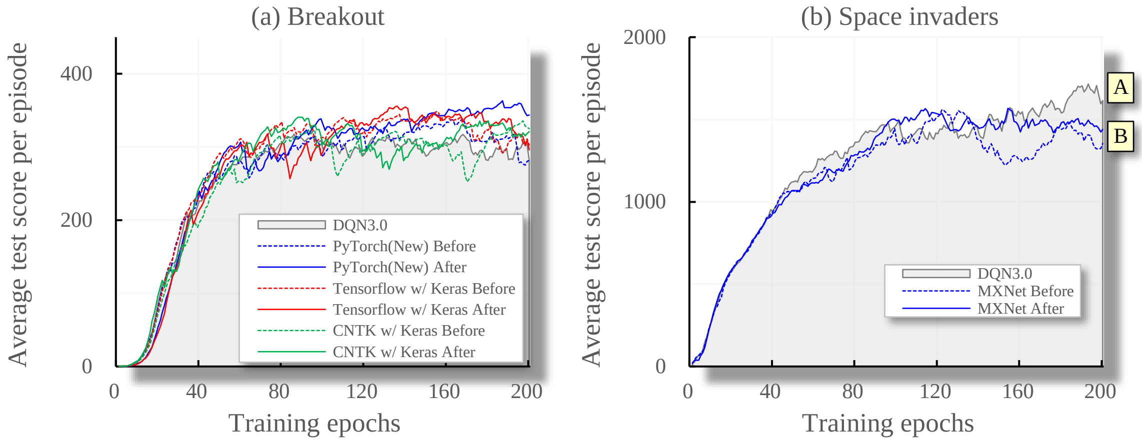


Figure 7: Comparison of training curves of my tuned DQNs in Breakout and Space invaders. (a) the training curves of my tuned DQNs in Breakout. (b) the training curves of my tuned DQNs in Space invaders. Both graphs appeared smoothing 0.9.

DQN version	Game	Before/ After	Average score	std	%Before Avg-score	%Before std
PyTorch(New)	Brakout	Before	361.11	±82.3	107.01%	77.88%
		After	386.44	±64.1		
MXNet	Space invaders	Before	1743.35	±739.1	106.04%	86.76%
		After	1848.80	±641.3		
Tensorflow with Keras	Brakout	Before	385.03	±70.3	102.15%	64.43%
		After	393.32	±45.3		
CNTK with Keras	Brakout	Before	369.74	±68.2	102.99%	89.53%
		After	380.83	±61.1		

Table 8: Comparison of best averace test score with the original-hyperparameter DQN vs. tuned DQNs in Breakout. The meaning of *Average score* and *std* are the best average test score and its standerd deviation in each setting.

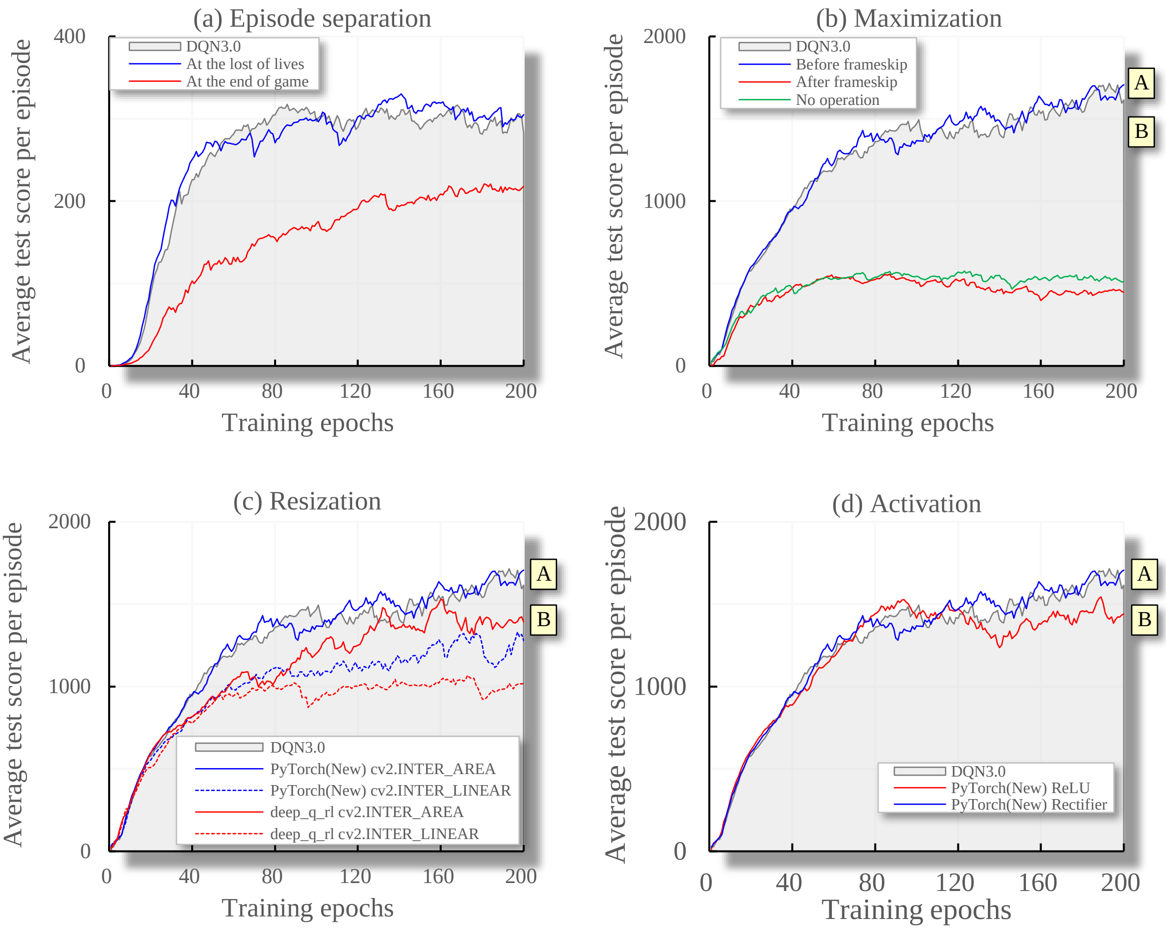


Figure 8: Comparison of training curves of each setting. (a) comparison of episode separation setting; (c) comparison of resization setting in Space invaders; (d) comarision of activation setting in Space invaders. (a)~(d) are used myDQN with PyTorch(New) with the original-setting and deep_q_rl. deep_q_rl 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.

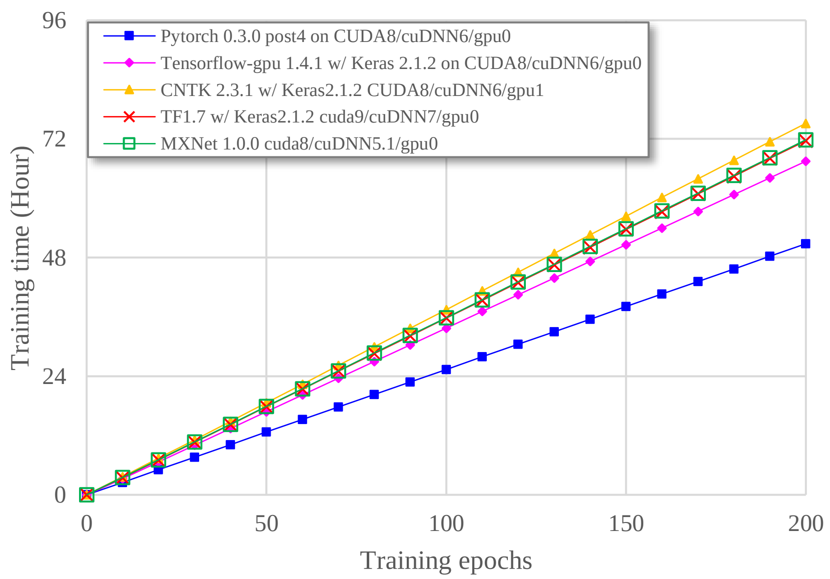


Figure 9: Comparison of training time of each Deep learning.

7. Discussion

7.1. About results

In Breakout, almost the same result as DQN3.0 was obtained in all versions of my DQN, but in Space invaders there was a difference in results after about 120 epochs of training, and finally the training curve was converged to either Group A or Group B. Although the cause is unknown, I think that slight errors each Deep learning frameworks has will be emphasized by frames composed of finer-designed objects of Space Invaders. On the other hand, deep_q_rl could improved by replacing interpolation to cv2.INTER_AREA which can be obtain exactly the same resized frames as DQN3.0, but still underperform DQN3.0, because deep_q_rl is still different the DQN loss function, RMSprop and Bias initializer from DQN3.0. I think that deep_q_rl can be obtain DQN3.0-level performance if replace them to the same realized method as DQN3.0 --but I did not try that. However, I can say that it is very important that following this reproduction method of DQN if DQN 3.0-level performance is required for our DQN.

Finally, as shown in *6.4*, in the case of on my PC, PyTorch's training time spends only about $\frac{2}{3}$ of other Deep learning frameworks. On the other hand, as shown in 6.2, PyTorch demonstrated performance of DQN3.0-level even in estimation of frames composed with finer-designed objects like Space invaders. Therefore, these results seem to show that the use of PyTorch is advantageous under a small development environment. However, PyTorch, which shares much with Torch, may have obtained results closer to DQN3.0. In addition, I could not identify the cause of the variation of the results of other Deep learning frameworks, and each Deep learning framework has its own adjustment method[13], but I have not tried all of them, moreover, we may be able to select more optimal implementation methods, so there is a possibility of improvement.

7.2. About Deep learning framework

In order to more accuracy reproduce DQN3.0, extensibility of Deep learning framework is necessary. Fact, in Roderick et al., 2017 as an example, they could not reproduce accuracy the same as DQN3.0 due to extensibility of the Reinforcement learning library they used which could not reproduce the DQN3.0's RMSprop. In contrast, there are flexible extensibility in each framework I evaluated --Although Tensorflow could not easy create a Python custom operation, I considered that the performance is not affected by whether use Rectifier layer.

In addition, this work shows that the function provided Deep learning framework or libraries we use may not return the same result, even if they have the same name and/or refer to the same paper. Moreover even if we write the same code, it may affect the implementation of Deep learning framework or library, resulting in errors. So it is desirable to try as many possible Deep learning frameworks and libraries as possible. In order to this purpose, it is advantageous to use Keras or ONNX that provides high-level API over each Deep learning frameworks.

Finally, this work shows a interestingly possibility: In developing an application which can be realized using only the basic functions of the Deep learning framework like the development of DQN in a small development environment, there is no advantage for preferentially selecting Tensorflow preferred by many people (Allen, 2017).

7.3. About Mnih et al., 2015

Finally, Mnih et al., 2015 did not use the grid search for hyperparameter, and the parameter initializer also uses Torch's default, but it is difficult to reach to the DQN3.0-level performance even using modern Deep learning frameworks. I can not hide my surprise in Mnih et al., 2015's outstanding achievement (even if it is a product of chance, their achievements remain unchanged).

8. Conclusion

These experimental results show that what each Deep learning framework provides is synonymous. In other word, they are representing $1 + 1 = 2$ in their own way. Therefore, in principle it is possible to accurately reproduce DQN 3.0 using any Deep learning framework. In practice, however, slight performance variance will occur depending on the implementation of Deep learning framework and the libraries. In addition due to limitations on the use of the Deep learning framework, misunderstanding of the semantics of DQN and misleading realization method, noticeable performance difference appears. Finally, the performance differences are more emphasized in the game which has frames with finer-designed objects.

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. 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
4. OpenAI. atari-py. (2017). URL https://github.com/openai/atari-py.
5. Sutton, R. S., & Barto, A. G. (2018). RL: An introduction. Second edition, in progress ＊＊＊＊Complete Draft＊＊＊＊. URL http://incompleteideas.net/book/bookdraft2018jan1.pdf
6. 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
7. DeepMind. Xitari is a fork of the Arcade Learning Environment v0.4 (2017). URL https://github.com/deepmind/xitari.
8. DeepMind. A lua wrapper for the Arcade Learning Environment/xitari, December 2014. URL https://github.com/deepmind/alewrap
9. Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015).

[13] Is MXNet faster than Tensorflow when using Keras-mxnet? (Krishnamurthy et al., 2018)

Prioritized experience replay. arXiv preprint arXiv:1511.05952. URL https://arxiv.org/abs/1511.05952.

10. 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
11. 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
12. Hinton, G., Srivastava, N., & Swersky, K. Lecture 6a overview of mini-batch gradient descent (2012). URL https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
13. Graves, A. (2013). Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850v5. URL https://arxiv.org/abs/1308.0850
14. Wikipedia contributors. (2018, February 26). Grayscale. In Wikipedia, The Free Encyclopedia. Web 4 May 2018 URL https://en.wikipedia.org/w/index.php?title=Grayscale&oldid=827764362
15. Tensorflow.org (2018, April, 28). Adding a New Op In Extend. Web 5 May 2018 URL https://www.tensorflow.org/extend/adding_an_op
16. Sprague, N. (2016). Theano-based implementation of Deep Q-learning. URL https://github.com/spragunr/deep_q_rl
17. Roderick, M., MacGlashan, J., & Tellex, S. (2017). Implementing the Deep Q-Network. arXiv preprint arXiv:1711.07478. URL https://arxiv.org/abs/1711.07478
18. Erich, G., Richard, H., Ralph, J., John, V. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.
19. Krishnamurthy, S., Chendke, K., Wei, L., Markham, A. Apache MXNet (incubating) adds support for Keras 2. (2018). AWS Machine Learning Blog. Web 30 May 2018. URL https://aws.amazon.com/jp/blogs/machine-learning/apache-mxnet-incubating-adds-support-for-keras-2/.
20. Allen, R., Li, M. Ranking Popular Deep learning Libraries for Data Science. (2017). KDnuggets. Web 20 May 2018 URL https://www.kdnuggets.com/2017/10/ranking-popular-deep-learning-libraries-data-science.html

Appendix

A. The architecture of my DQN

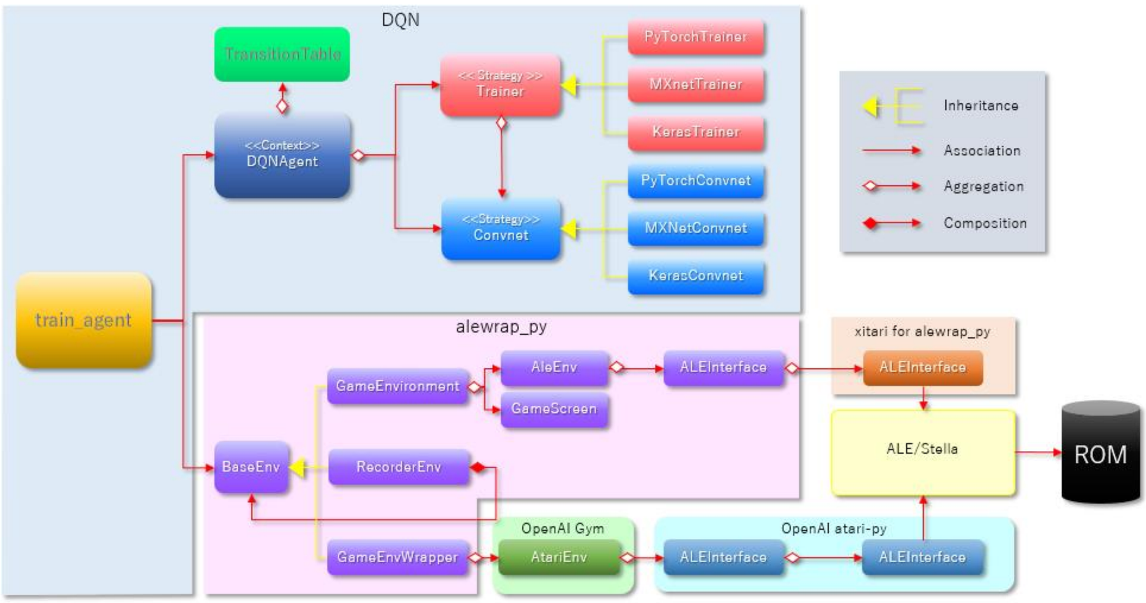


Figure 10: The architecture of my DQN. My DQN is an implement of the Strategy pattern (Erich et al., 1995): Client: `train_agent`, Context: `DQNAgent`, Strategy: `Trainer` and `Convet`; Each ConcreteStrategy classes corresponds each Deep learning framework. alewrap_py reproduced alewrap in other package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