

The Creation Manner of Deep Q-network with DQN3.0-level Performance

Ver 1.0

Shigeru Matsumura

Elliott Trade Laboratory

elliott.trade.laboratory@gmail.com

Abstract

It is very simple way to create 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 manner as DQN3.0. But as far as I know, there is no anywhere to explanation of simplest way as above, and also its implementation is nowhere. So I studied the DQN creation manner in DQN3.0, and succeeded create DQNs with DQN3.0-level performance. This article is explained the creation manner 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) through the creation manner of DQN. After that, is proved that, we can get a DQN with DQN3.0-level performance, as a result. In addition, also explains a very important aspect which is indispensable for that purpose but, although not explained so far.

1. Introduction

DQN is an epoch making algorithm which applied a neural network and a replay memory on conventional reinforcement learning algorithm proposed Mnih et al., 2015 in Feb 2015. On the other hand, 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, accurate reproduction and explanation of exact creation method is not found on the Internet. As far as I know, the DQN implementation we can see has low performance than DQN3.0 released by Mnih et al., 2015 at that time.

So I studied Mnih et al., 2015 and its implementation DQN3.0 deeply. As a result, I understood that there are three important points necessary for creating DQN with DQN3.0 level-performance: First, we must understand that DQN is one of the reinforcement learning algorithm and DQN3.0 was created in the creation manner of reinforcement learning faithfully: DQN is not just a image classifier which estimate an action with respect to a game screen, and we do not forget that DQN loss-function is the expected value rather than the mean-squared error function such provided by deep learning frameworks. Second, we can train our network without “end-to-end” forward/backward propagation by automatic-differentiation: Even though it sounds like lack of common sense of deep learning framework, but, perhaps it is the common sense of deep reinforcement learning. Finally, we need to deeply examine the function of the framework and libraries which apply to our application: They may not yield the results we expect.

The motivation of this article is to breakthrough the mysterious aspect of DQN by explain the creation manner of DQN and how to implement it using Python, and to show that we can create DQN with DQN3.0-level performance. And, thereby, also will

explain the very important point mentioned above which no one has ever explained.

2. Background

This section explains the background necessary for understand the creation manner of DQN.

2.1. Key idea and algorithms of reinforcement learning

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

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

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

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

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

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

where $p(s', r | s, a) \doteq \Pr\{s' = s_{t+1}, r = r_{t+1} | s = s_t, a = a_t\}$. Why? If the transitions has the Markov property, the state at the time step t , s_t , contains all the previous information s_0, s_1, \dots, s_{t-1} , that is, $p(s_t | s_0, s_1, \dots, s_{t-1})$. For example, in the role-playing game, the player's item-menu as a state contains 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 recurring healing item in the item-menu may be decreasing for restores the HP. That is, the current state consists of all previous

states so maybe all states in an episode are unique¹.

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

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

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

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

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

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

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

where n is number of outcome of random variable (s', r) . These expressions denote that the action-value function is the sum of the immediate reward plus the discounted next action value weighted by the selection probability of next action and the occurrence probability of the next state and reward as the environment dynamics.

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

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

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

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

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

Now we could obtain action value for one of possible action a in state s . After that, we need to obtain a valu of action for 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:

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

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

As above, the optimization of action-value function is computed for s and a for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$ individually. Therefore, “*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. The off-policy time difference variance 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 without environment dynamics.

These calculations are formulated as equation (13):

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

where α is step size parameter called *learning rate*. So the agent 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,

¹Note that, this article is always explaining the states with Markov property, that is, we are always considering the transition as the sequence. In contrast, in *Example3.3 Recycling Robot* in Sutton and Barto, 2018 P.41 explains the

state-transition probability using the transition graph with the states without Markov property, thereby just looks like the robot can going back and forth between the same states in an episode.

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

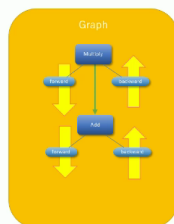
In addition, it is also important to consider the tie-breaking for behavior policy. The tie-breaking selects one action from multiple maximum action values. If the agent does not use it, the behavior policy always select one action with smaller index even if it become unoptimal action when there are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

2.2. Automatic differentiation of Deep learning framework

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 such as that possessed by PyTorch and MXNet, addition of operations to a graph and forward propagation thereof are executed at the same time. After that, by executing the written `backward()` function, back propagation is executed backward along the graph.

```
a=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. However the loss is just only indicates how good our network model, it does not affect backpropagation. In most deep learning frameworks emphasize writing as follows

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

or

```
# Tensorflow
```

```
optimizer.minimize(loss).
```

This reason why it is necessary to use a loss which is a scalar Tensor (or Variable) of outcome of a loss function is only to make our aware of "to minimize loss" as a ceremony. In fact, it purpose is to obtain the backward function of the operation which was created a loss, as a function of first call in differentiation of a computation graph, and give a scalar Tensor/Variable which has 1 (or a Tensor/Variable which has multiple elements and a shape the same as loss, to filled with 1) as the initial gradient rather than a value of loss, to the first backward function. This is based on the chain rule. However, many deep learning framework are provided other approach for differentiation of computation graph which can be starts of backpropagation from middle of graph by given initial gradient as follows:

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

or

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

But, such approach does not described in detail anywhere.

2.3. 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: One of them, they use the Arcade Learning Environment (ALE) (Bellemare et al., 2013) directly or extended by themselves; The other one is, they use the atari env provided by OpenAI Gym (Gym) (Brockman et al., 2016).

In the former case, we can get own environment that extended any additional functions to ALE for own algorithm: For example, in order to achieve their purpose, Mnih et al., 2015 devised xitari (DeepMind, 2017) and alewrap (DeepMind, 2014) which extended some functions described in エラー! 参照元が見つかりません。 and a function which can be obtain the Live counter described in *Episode separation for training*. to ALE.

In the latter case, we can comparison to other algorithms for the same problem: Gym has concept that provides the benchmark collections which corresponds various problems, and to equitably evaluate various algorithm which aims to solve these problems (Brockman et al., 2016. P.1 Introduction, P.2 Design Decisions); In order that, the atari env named '*Gamenam-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` with 0.25 based on their concept. On the other hand, several atari envs corresponding to other uses are also available: For example, one of the env named '*GamenamDeterministic-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 env named '*GamenamNoFrameskip-v4*', (e.g. '*BreakoutNoFrameskip-v4*') is get every frames and `repeat_action_probability` with 0.

3. The creation manner of DQN in DQN3.0

This section explains the creation manner of DQN considered from DQN3.0. It contains two aspects of ingenuity: For to solve Atari2600 domain; for as novel algorithm. By considering about these two aspects, I believe that good application to other tasks will be possible.

3.1. The ingenuity for solve Atari2600 domain in DQN3.0.

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 at time-step \tilde{t} , the environment returns a RGB frame denote $x_{\tilde{t}} = \{x_{\tilde{t}}^R, x_{\tilde{t}}^G, x_{\tilde{t}}^B\}$. (a) they take the maximum value between $x_{\tilde{t}}$ and $x_{\tilde{t}-1}$ as $\hat{x}_{\tilde{t}}$

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

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

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

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

$$\tilde{x}_t = 0.299\hat{x}_t^R + 0.587\hat{x}_t^G + 0.114\hat{x}_t^B. \quad (15)$$

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

$$s_t = \text{resize}^{\text{bilinear}}(\tilde{x}_t, 84, 84). \quad (16)$$

Finally, (e) As better input data ϕ_t for learning and prediction, they stacks the four recency s_t

$$\phi_t = \{s_{t-3}, s_{t-2}, s_{t-1}, s_t\}, \quad s_t \neq s^+, \quad (17)$$

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

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

in this case, ϕ_t contains states occurred in two different episodes: One is episode⁻¹ contained s_{t-3} and s_{t-2}^+ (that is, episode⁻¹ is terminated at $t-2$); and other one is episode⁰ contained s_{t-1} and s_t (that is, episode⁰ is started at $t-1$). Let add the episode number in square bracket subscript to (28) as follows:

$$\phi_t = \{s_{t-3}^{[-1]}, s_{t-2}^{+[-1]}, s_{t-1}^{[0]}, s_t^{[0]}\}. \quad (19)$$

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

$$\phi_t = \{s^{\text{null}}, s^{\text{null}}, s_{t-1}^{[0]}, s_t^{[0]}\}. \quad (20)$$

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

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

Episode separation for training. In the game of Atari 2600 there is a "Lives counter" that indicates how many more times player can fail. For example, in the case of the Breakout, firstly, it given 5 lives for player. If player cannot hit a ball with a paddel, then 1 Lives will be loss, and when 5 Lives are lost, then the game is over. In this usual case, as shown in **Figure 1**-(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. In order to avoid this, Mnih et al., 2015 took measures steps that, during training, when Lives was lost, replaced a normal state to a termination state. For this reason, as shown in **Figure 1**-(b), the agent does not learn these actions selected toward to the loss of Lives as these actions has no value. However, the agent recognizes the Lives counter 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 implemented in alewrap.

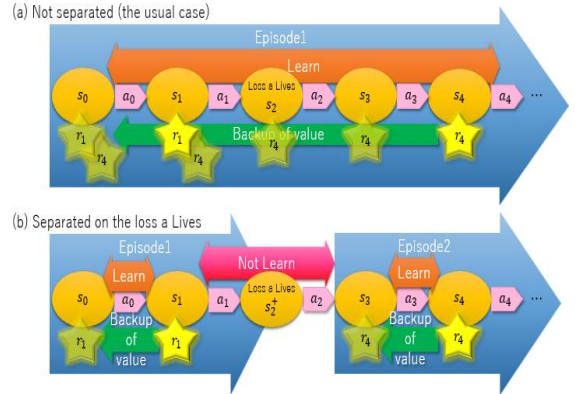


Figure 1: (a) In the case of the episode which not separated as default, the sequences reaching failure are learned by backup of the future reward r_4 . (b) In the case of the episode separated on the failure and where s_2^+ is the terminal state which replaced from s_2 , the action a_1 is not learned 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 ingenuity as novel algorithm

Transition Table as Replay memory. Mnih et al., 2015 stores tuple of experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ which is obtained from transitions into a replay memory $\mathcal{D}_t = \{e_{t-N}, e_{t-(N+1)}, \dots, e_t\}$, where $N \in \mathbb{R}^{[0, 1 \times 10^6]}$ is number of experiences, for the experience replay. 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 sliding window of all transitions occurred during training, that is, stored recently 1×10^6 experiences, and forget experiences earlier than $t - 1 \times 10^6$.

In DQN3.0, the replay memory is called TransitionTable and is circulate through the index for insert and freshen to the new

experiences, instead of use a queue. During first 50,000 steps of train, 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 E_i from TransitionTable. TransitionTable will samples randomly the same number of experiences e' as mini batch size M

$$\mathcal{E}_i = \{e'_1, e'_2, \dots, e'_M\}, \quad (21)$$

where e' is a experience for learning, and

$$e' = (\phi, a, r, \phi', term') \in \mathcal{D}_t \quad (22)$$

where $(\phi, a, r, \phi', term') = (\phi_{t'}, a_{t'}, r_{t'}, \phi_{t'+1}, term_{t'+1})$, t' is sampled time-step, $t' \sim \mathcal{U}(\mathcal{T}(\mathcal{D}_t))$, $\mathcal{T}(\mathcal{D}_t)$ is set of time-step included in \mathcal{D}_t .

As already mentioned, 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.

Policy update. DQN as One-step Q-learning also 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 independent policies as the behavior policy and the target policy.

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

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

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

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

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

In this case, the update formula of the Q-function was changed to the update formula for many elements corresponding to the generalized state 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 generalized state 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 generalized state from the squared TD error computed when each actual state is given. Second, minimizes the average of gradients using RMSprop of Hinton et al., 2012.

The extremal property, $\mathbb{E}[X - c]^2$, $c \in \mathbb{R}$, is estimates a error between a target constant c and outcome of random variable X

over all trials. Here the DQN loss function is defined in Mnih et al., 2015 as follows:

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

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

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

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

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, $\mathbb{E}_{s,a,r} [\mathbb{V}_{s'} [y]]$, 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.

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

where δ_i is set of TD-error, and

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

The neural network does not has history of the outcome of all trials of each 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 (25) as follows:

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

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

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 MSE^{SV}

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

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

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

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

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

Recall that each optimal behavior value function exists as a separate system for each state-action pair. That is, in DQN, each model is individually learned with one mini batch containing 32 data for learning 32 model corresponding to 32 state-action pair. So, for DQN, we can not use 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 $\frac{2}{M} \sum_{m=1}^M (x^{(m)} - y^{(m)})$, so the

derivative of equation (28) should become $2\mathbb{E}_{s,a,r,s'}[\nabla_{\theta_i} Q(s, a; \theta_i)]$. However, since equation (25) can be considered assuming One Half MSE (Ng, 2016), $\frac{1}{2M} \sum_{m=1}^M (x^{(m)} - y^{(m)})^2$, that is, its derivative is $\frac{1}{M} \sum_{m=1}^M (x^{(m)} - y^{(m)})$, so 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 et al., 2012. I think that, Mnih et al., 2015 considered that to obtain the stability of the algorithm, it was necessary clip the TD error to be between -1 and 1 , and at the same time they thought that it is necessary to update the parameters using the gradient maintaining that range, because the gradient becomes positive values by the square of RMSprop of Hinton et al., 2012. For that reason, Mnih et al., 2015 modified RMSprop of Hinton et al., 2012 to distribute the gradient around zero. 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 et al., 2012 can be write as follows:

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

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

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

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

where λ is learning rate, γ min squared gradient, these are described in Mnih et al., 2015 P.10 Extended Data Table 1, and λ 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)
(26): $y = r + \gamma \max_a Q(s', a'; \theta^-)$
<pre> 194 term = term:clone():Float():mul(-1):add(1) 195 196 197 local target_q_net 198 if self.target_q then 199 target_q_net = self.target_network 200 else 201 target_q_net = self.network 202 end 203 204 205 -- Compute max_a Q(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) </pre>
(27): $\delta_i = \left\{ r^{(m)} + \gamma \max_a Q(s^{(m)}, a'; \theta_i^-) - Q(s^{(m)}, a^{(m)}; \theta_i) \right\}_{m=1}^M$
<pre> 217 local q_all = self.network:forward(s):float() 218 q = torch.FloatTensor(q_all:size(1)) 219 for i=1,q_all:size(1) do 220 q[i] = q_all[i][a[i]] 221 end 222 delta:add(-1, q) </pre>
Error Clipping
<pre> 224 if self.clip_delta then 225 delta[delta:ge(self.clip_delta)] = self.clip_delta 226 delta[delta:le(-self.clip_delta)] = -self.clip_delta 227 end </pre>

Part of (29): $\sum_{m=1}^M \delta_i^{(m)} \nabla_{\theta_i} Q(s^{(m)}, a^{(m)}; \theta_i)$
<pre> 229 local targets = torch.zeros(self.minibatch_size, self.n_actions):float() 230 for i=1,math.min(self.minibatch_size,a:size(1)) do 231 delta[delta:le(-self.clip_delta)] = -self.clip_delta 232 end 233 self.network:backward(s, targets) </pre>
(29) or (30): $g_i = \kappa g_{i-1} + (1 - \kappa) d\theta_i$
<pre> 266 self.g:mul(0.95):add(0.05, self.dw) </pre>
(31): $n_i = \kappa n_{i-1} + (1 - \kappa) d\theta_i^2$
(32): $\Delta_i = \lambda \Delta_{i-1} + \lambda \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \gamma}}$
(33): $\theta_{i+1} = \theta_i + \Delta_i$
<pre> 267 self.tmp:cmul(self.dw, self.dw) 268 self.g2:mul(0.95):add(0.05, self.tmp) 269 self.tmp:cmul(self.g, self.g) 270 self.tmp:mul(-1) 271 self.tmp:add(self.g2) 272 self.tmp:add(0.01) 273 self.tmp:sqrt() 274 275 -- accumulate update 276 self.deltas:mul(0):addcdiv(self.lr, self.dw, self.tmp) 277 self.w:add(self.deltas) </pre>

Table 1: Correspondence of equations and source code.

4. Key points for improving the reproducibility of DQN3.0.

While large number of iterative updates, the One-step Q-learning accumulates the next value of state-action pair scaled by the step size parameter little by little to current value of state-action pair. So, even if 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**.

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

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

4.1.Preprocess

Maximization. As mentioned in *Preprocess*-(a), it is necessary to maximize 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. Also as mentioned in 2.3.Environment of the challenging domain of classic Atari2600 games, the basic env which named ‘*Gamename-v0*’ of Gym has frameskip set the arguments according to Brockman et al.,2016. So if we use it env, we obtains different maximized frames from DQN3.0 as shown **Figure 2**, because we can only take a maximum between frames which after the frameskip. So if we want to use Gym

instead of directly using ALE, it is necessary to use the env named 'GamenameNoFrameskip-v4', (e.g. 'BreakoutNoFrameskip-v4'), and implements maximization and then frameskip where after obtain frames from that env.

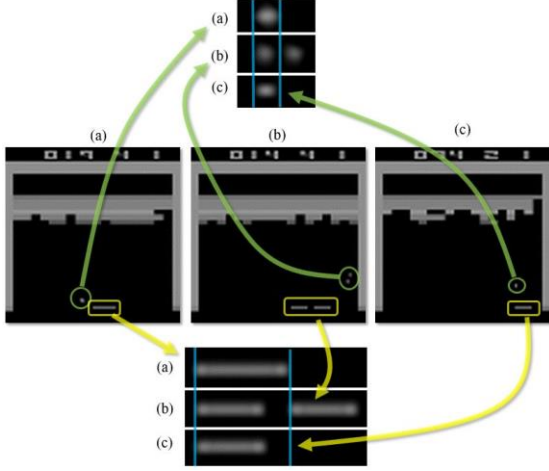


Figure 2: The comparison of how to implement the maximization. (a) implemented before frameskip; (b) implemented after frameskip; (c) not implemented; The middle (a)~(c) are comparison of how inputs are displayed; The top (a)~(c) are comparison of how the balls are displayed; The bottom (a)~(c) are comparison 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.

Grayscale. 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 adopt 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 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 provided Torch. 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 3** shows, in the Space invaders, which has objects with small design, the object was lacked it shape or seems to another shape, if we use Bilinear of each image

processing libraries on Python.

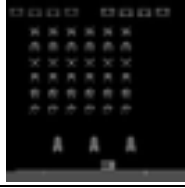
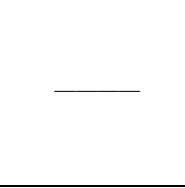
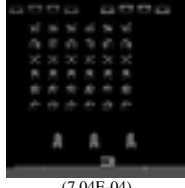
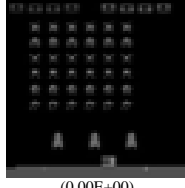
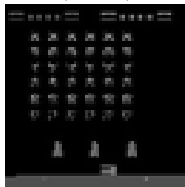
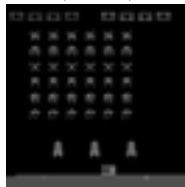
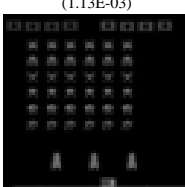
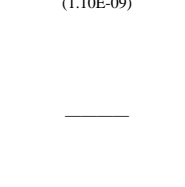
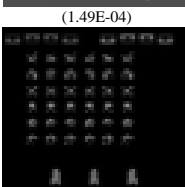
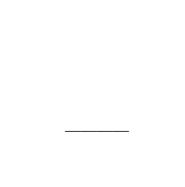
Library	Interpolation	
	Bilinear (MSE)	Area (MSE)
torch.image		
OpenCV		
tf.image		
Pillow		
scikit-image		

Table 3: 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. This table shows, the resize function and the interpolation of library which is most close to `image.scale()` with 'bilinear' of image is `cv2.resize()` with `cv2.INTER_AREA`. The next closest is `tf.image.resize_area()`. However, `tf.image` is very slow.

4.2.Initialization

In DQN3.0, the parameters of each layer of network are initialized by default initialization manner in `nn` package of Torch² as follows

$$stdv = \frac{1}{\sqrt{fan_in^{weight}}} \quad (34)$$

$$\theta^{weight} \sim \mathcal{U}(-stdv, stdv) \quad (35)$$

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

Torch's layer class initializes weight and bias together using the same stdv. This is in contrast to each Python deep learning framework initializing weight and bias separately. For this reason, we need create a custom initializer shared a stdv between weight initializer and bias initializer if we want to reproduce DQN3.0 completely. 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.

4.3. 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.^{3,4}; Or, by applied their own expressions, they may have aimed at stabilizing the algorithm and/or improving learning. In any case, in the case of except Tensorflow, when we want to reproduce their `Rectifier` layer, we need to create custom operation(or user function etc.) and layer class in these creation manner explained in document of deep learning framework. 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, they did not take into account that some users wants to implement arbitrary differential expressions. 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.

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

4.4. 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 is very different from the common sense of deep learning framework learned so far. At that time, we may think "Why did not you use the loss-function provided by Torch?". And finally, all other people chose to follow the common sense of the deep learning framework. As a result, their DQNs are different from DQN3.0. If we want to create a DQN with DQN3.0 level-performance, we need to breakthrough that common sense. But in other words, we need to follow the common sense of reinforcement learning rather than the common sense of supervised learning as common sense of deep learning framework.

In this case, as mentioned in 2.2. *Automatic differentiation of Deep learning framework*, 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 3** shows, this creation manner can be construct the computation graph having the same computation procedure as DQN3.0.

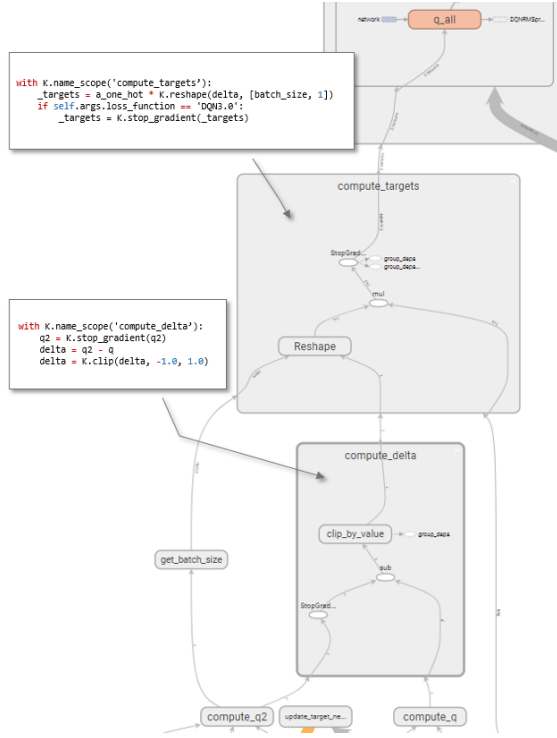


Figure 3: Part of the computation graph of DQN loss-function in Tensorflow.

For CNTK with Keras which also has static automatic differentiation, is the same as Tensorflow if we use Keras, but it is necessary to create a user function that provide the targets to backward() function of the add operation which is add the bias to input 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.

4.5.RMSprop

Some deep learning framework provided centered RMSprop like a RMSprop of Graves, 2013, but, these implemented different equation from both RMSprop of Graves, 2013 and RMSprop of DQN3.0 (DeepMind, 2017

`dqn/NeuralQLearner.lua#L256-L277`) as shows **Table 4**. So we need to create RMSprop the same as DQN3.0 from scratch in optimizer creation manner explained in document of each deep learning framework..

Centered RMSprop of PyTorch(New)⁵:

5

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

6

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

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

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

$$\Delta_i = \kappa \Delta_{i-1} + \lambda \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \gamma}} \quad (39)$$

$$\theta_{i+1} = \theta_i - \Delta_i \quad (40)$$

Centered RMSprop of PMXNet^{6,7}:

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

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

$$\Delta_i = \kappa \Delta_{i-1} - \lambda \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \gamma}} \quad (43)$$

$$\theta_{i+1} = \theta_i + \Delta_i \quad (44)$$

Centered RMSprop of Tensorflow⁸:

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

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

$$\Delta_i = \kappa \Delta_{i-1} + \lambda \frac{d\theta_i}{\sqrt{n_i - g_i^2 + \gamma}} \quad (47)$$

$$\theta_{i+1} = \theta_i - \Delta_i \quad (48)$$

Table 4: 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 manner as DQN3.0 except loss-function, RMSprop and the parameter initializer. For the loss-function, they use Huber loss-function which they made from scratch as follows:

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

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

7

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

8

<https://github.com/tensorflow/tensorflow/blob/8753e2ebde6c58b56675cc19ab7ff83072824a62/tensorflow/python/training/rmsprop.py#L30-L37>

they created from scratch like RMSprop of DQN3.0. Let $d\theta_i^{\text{Sprague}} = \nabla_{\theta_i} L_i^{\text{Sprague}}(\theta_i)$, that RMSprop can be denote as follows:

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

This RMSprop is the same as DQN3.0, except, it does not use 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, it is also a scientific publication, so it explanation is not sufficiently for accurately reproduce DQN3.0. In addition, 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 the source code, it seems to be implemented in the same creation manner 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 build self-made PC as **Table 5**. 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 5: The environment spec of experiments.

6.2. Final 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 parameters; In addition, sometimes made the best average test score with parameter created in middle phase of training; On the other hand, the

evaluation method in DQN3.0 is ambiguous: Since the evaluation in DQN3.0 is the average score per episode in the number of evaluation steps, that is, as learning progresses, the number of steps per episode increases, thereby the number of evaluated episodes per epoch is reduced, that is, the number of episodes per epoch is not constant, therefore, the evaluation of DQN3.0 compares the average episode score over the different number of episode at each epoch. For that reason, 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. Finally, compare the Best average test score between each DQNs. That is, this evaluation method is searching for DQN which can obtain a higher average score with more stably. 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 uncertain but they may be helpful.

On the other hand, for comparison with other DQN implementations, training curves of deep_q_rl which is an implementation by Sprague, 2016 are also shown. However, it was difficult to change the implementation of deep_q_rl for the above evaluation method, so we could not get the best average test score.

The performance of my DQNs with original-setting and the DQN creation manner. As **Figure 4** and **Table 7** shows that all my DQNs were able to have DQN3.0-level performance, if we create a DQN following the creation manner of DQN with its original-setting as shown **Table 6**.

On GPU, even if we were specified random seed, algorithm outputs the random results affected by cuDNN.

In Breakout, only PyTorch(New) slightly underperformed DQN3.0, otherwise slightly about the same result as DQN3.0.

In Space invaders, only Tensorflow without Rectifier layer underperform DQN3.0.

As mentioned in Results of Resize and Rectifier below, I supposes that the result was deteriorated by affected by resizing method and activation method in Space invaders, which has objects with small design.

The performance of my tuned DQNs in Breakout and Space invaders. Activation をと Bias イニシャライザを変更した結果, Breakout で PyTorch(New), Tensorflow and CNTK に誤差とも思えるレベルの若干の改善が見られた. しかし, 他のフレームワークでは逆に悪化した. as shown in **Table 8**. As **Figure 5-(a)** and **Table 9-(a)** shows, the performance of my tuned DQNs (PyTorch(New), Tensorflow and CNTK) in Breakout were slightly improved. Otherwise, tuned DQN were not improved, but instead deteriorated.

6.3. Comparison of training curves by creating DQN with different creation manner

Episode separation. As **Figure 6-(a)** shows, as expected, if we applied that the state terminates at the Lives is lost, we obtained good training curve, otherwise, we obtained training curve with

very slow growth.

Maximization. As Figure 6-(b) shows, as expected, if we applied the maximization before frameskip, we obtained good training curve affected by frames with objects emphasized in the traveling direction as Figure 2-(a) shows. On the other hand, if we applied that the maximization after frameskip, we obtained low training curve affected by frames with duplicated objects as Figure 2-(b) shows.

Resizing. As Figure 6-(c) shows, as expected, there are improved their training curves as big difference between `cv2.INTER_LINEAR` and `cv2.INTER_AREA` in both `myDQN` with Tensorflow and `deep_q_rl`. On the other hand, as Figure 6-(d) shows, in `myDQN` with Tensorflow, as expected, its training curve was improved, but, in `deep_q_rl`, there was no big difference in `deep_q_rl`. However, although I could not find out the cause of the difference in training curves between Breakout and Space invaders in `deep_q_rl`.

Rectifier layer.

6.4. Training time

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 7 shows, PyTorch shown preeminently speed than other deep learning frameworks. In contrast, other frameworks shown no big difference, even if in Tensorflow which most popular deep learning framework and its more recent version.

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

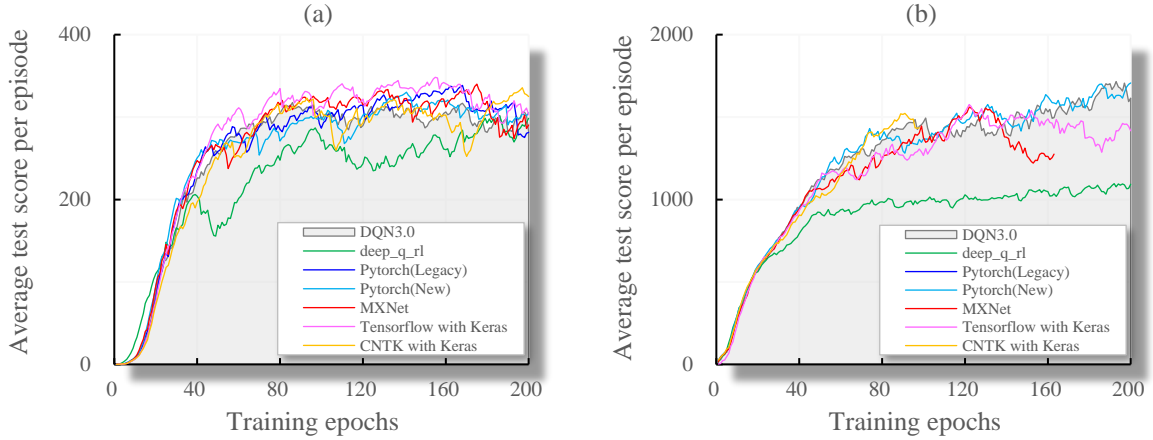


Figure 4: Comparison of training curves in Breakout (with original-hyperparameter and DQN creation manner). Training curves tracking each agent's average test score with the same hyperparameter (Mnih et al., 2015 P.10 Extended Data Table 1) and creation manner as DQN3.0 (only Tensorflow ver. uses ReLU function provided by Tensorflow). For `deep_q_rl`, specifying `device=gpu` in `.theanorc` improved the curve in Breakout. This graph applied smoothing 0.9.

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	±655.08	—	1979.70	±737.15	100.19%	—
deep_q_rl	—	—	—	—	—	—	—	—	—	—	—	—	—	—
PyTorch(Legacy)	377.74	±72.1	102.21%	380.50	±80.50	92.97%	101.99%	1870.10	±745.5	92.63%	1941.33	±734.4	98.25%	98.06%
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%	—	—	—	—	—	—	—
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%	—	—	—	—	—	—	—

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

score of DQN3.0; "%Mnih et al., 2015" is the ratio to the score described in Mnih et al., 2015.

DQN Version	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 8: The setting of my tuned DQNs. the own uniform initializer is

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

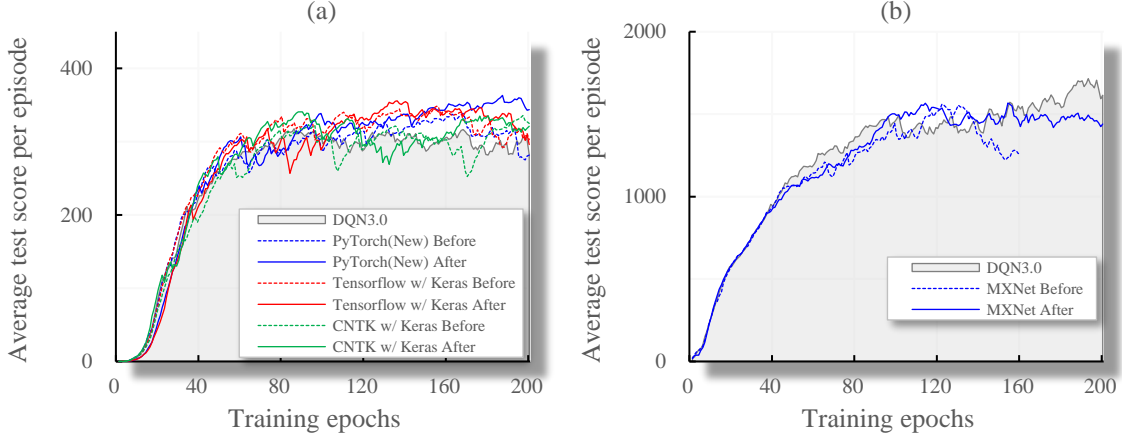
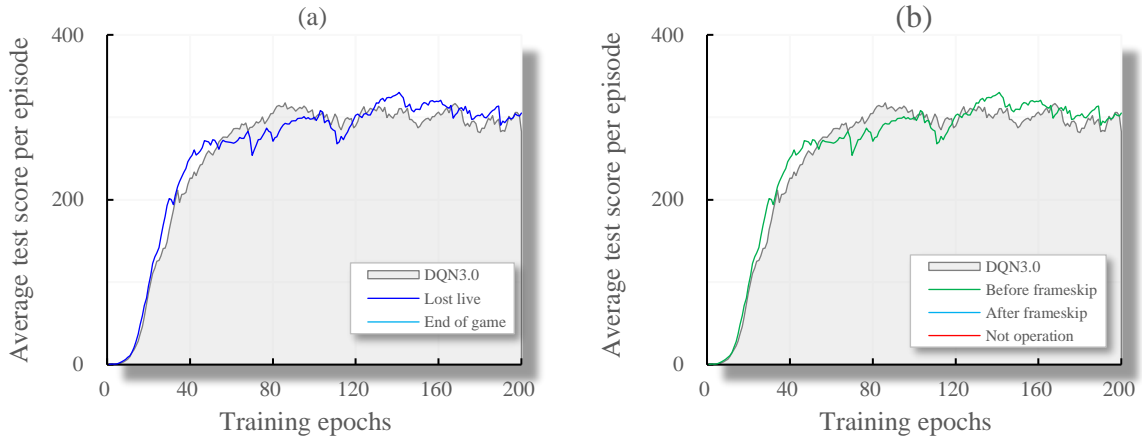


Figure 5: Comparison of training curves of my tuned DQNs in Breakout and Space invaders. (a) the training curves of my tuned DQNs in Breakout. (b) the training curves of my tuned DQNs in Space invaders. Both appeared smoothing 0.9.

DQN version	Game	Before/ After	Average score	std	%DQN3.0 Actual	Average score	std	% Mnih et al., 2015	%DQN3.0 Actual	% Before Avg-score
PyTorch(New)	Brakout	Before	361.11	±82.3	97.71%	377.23	±72.04	94.03%	103.14%	107.01%
		After	386.44	±64.1	104.57%	399.17	±46.7	99.49%	109.14%	
MXNet	Space invaders	Before								
		After	1848.80	±641.3	91.58%	2058.00	±613.3	104.15%	103.96%	
Tensorflow with Keras	Brakout	Before	385.03	±70.3	104.18%	388.57	±46.51	96.85%	106.24%	102.15%
		After	393.32	±45.3	106.43%	398.87	±38.8	99.42%	109.06%	
CNTK with Keras	Brakout	Before	369.74	±68.2	100.05%	384.57	±33.70	95.85%	105.15%	102.99%
		After	380.83	±61.1	102.97%	392.57	±47.5	97.85%	107.34%	

Table 9: Comparison of the original-hyperparameter DQN vs. tuned DQNs in Breakout



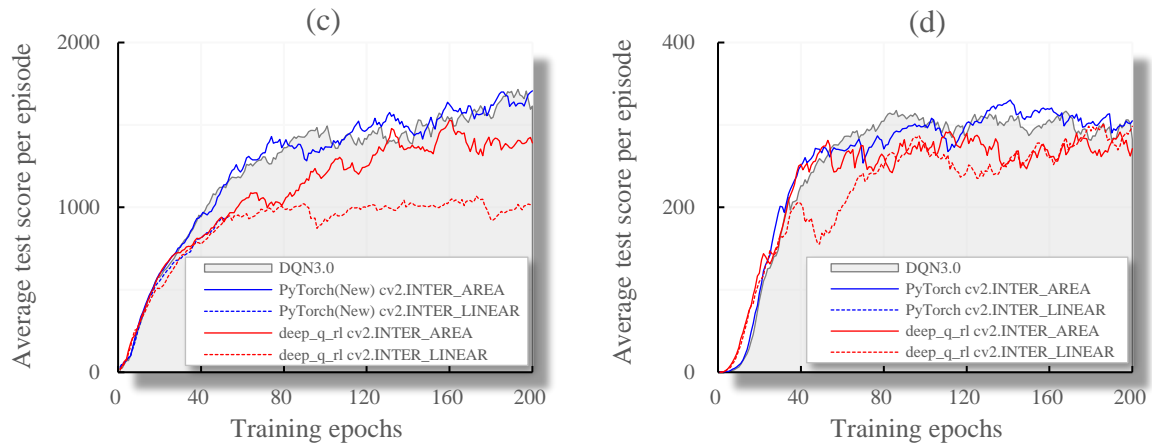


Figure 6: Comparison of each setting. (a) comparison of episode separation setting; (b) comparison of maximization setting; (c) comparison of resizing setting in Space Invaders; (d) comparison of resizing setting in Breakout: (a) and (b) are used myDQN with PyTorch(New) with original-HP and setting; (c) and (d) are used myDQN with Tensorflow and deep_q_rl. deep_q_rl with cv2.INTER_AREA was modified an argument “interpolation” of resize function of OpenCV in Sprague, 2016 ale_experiment.py#L186.

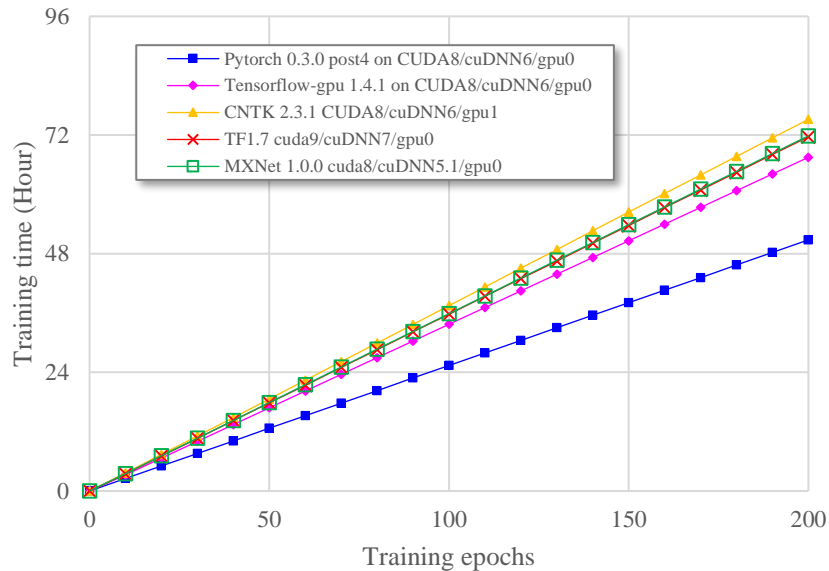


Figure 7: Comparison of training time of each deep learning.

7. Discussion

7.1. About results

In 6.2, with my PC, it can run 5 or 6 DQN processes in parallel, but doubles the processing time. If the same processing time as Figure 7 is required, up to 2 DQN processes can be executed. I have not been able to identify the cause, but I think that concurrent execution of many processes leads to lower throughput of PCIe. This is because even if the process is distributed to 2 GPUs, the speed reduction is not improved. For this reason, since it took a lot of time to test, the final performance evaluation uses only the result obtained by one training. However, more desirably, it is to select the best result from the multiple training results.

In 6.3, I tried only PyTorch(New) ver., because as Figure 7 shows, PyTorch(New) ver is most fastest in my DQNs. So note that, there are different ver may have different results.

7.2. About deep learning framework

Extendability. In order to more exactly reproduce DQN 3.0, extensibility of deep learning framework is necessary. Fact, in Radic as an example, they could not reproduce exactly the same as DQN3.0 due to extensibility of the reinforcement learning library they used which could not reproduce the DQN3.0's RMSprop. This suggests that the same is true when creating a more high-performance implementation of the novel algorithm. Because as can be seen in the 6.3-Rectifier layer example, implementing with well thought-out own expressions may be useful for performance improvement, because the implementation of different formulas which give the same results on the desk gives different results by approximation of floating point operations on the computer.

この作業のように、特定のアルゴリズムを完全に再現しようとした場合、使用するディープラーニングフレームワークや

ライブラリの拡張性が問題になってきます。事実、Redickらは使用したライブラリの拡張性の問題により、DQN3.0を完全に再現することができませんでした。他方で、拡張性の問題は、新しいアルゴリズムを構築する際に、フレームワークやライブラリにより提供されていない独自のアイデアでそれらを拡張したい場合にも存在すると考えます。今回使用したディープラーニングフレームワークの中では、TensorflowとCNTKの拡張性に問題が見られました。Tensorflowはカスタムオペレーションが簡単に作成できないため、Rectifierレイヤーが実装できませんでした。またCNTKは計算グラフの実行プロシージャを変更するAPIや各レイヤーの勾配を取得するAPIが提供されておらず、それらの実現のためにトリッキーな方法を取らなければなりません。これらの要因について正確な理由は不明ですが、どちらも静的自動微分機能を有するディープラーニングフレームワークであることから、それによる何らかの制限がある可能性があると思いました。他方でPyTorchとMXNetは、DQN3.0を再現する上での拡張性に問題は見られませんでした。

また、今回、主要なディープラーニングについて、同一ハードウェア上で同じトレーニングプラットフォーム上で同じアルゴリズムの実装での性能比較ができました。その結果、学習能面を比べた場合、Tensorflowのみが、恐らくRectifierレイヤーを再現できなかったことに起因して、スペースインベーダーでDQN3.0レベルの性能が得られなかったことを除いて、有意な違いは見られませんでした。また、速度面においては、PyTorchのみが他のフレームワークに比べて突出して高速であることは印象的でした。一般的にはAI研究のリーダー企業であるGoogleが提供するTensorflowが最も好まれるフレームワークである資料もありますが(Allen and Li, 2017)、この作業の結果ではTensorflowが特別に優秀であるという根拠は見られないだけでなく、むしろ他のフレームワークと比べ、柔軟性に欠ける面があることが露呈しました。この結果から、Tensorflowを強く選択する理由はTPUを使用することで生じるアドバンテージを期待すること以外ないと思われます。また、わずかな実装方法の違いで学習性能が変わってくるため、どのフレームワークであっても、実装方法によって最良の結果を得ることも、そうでないこともあり得ることがわかります。つまり、特定のフレームワークのみを使用するより、異なるフレームワークによる実装を同時に評価してみることに優位性があることがわかります。

We often stick to a specific deep learning framework, but the results of this work show that there is little advantage in doing so. Each framework can give the best results, but on the other hand it may not give the best results. So, there are advantages to trying as many frameworks as possible to get the best model. In that sense, the use of the high-level framework which provides the common API between low-level frameworks such as Keras and ONNX has great advantages. Because multiple frameworks can be evaluated with the code once written. On the other hand, this work shows that the individual functions provided by each framework and library are implementations that the author considered as good, so that may not be suitable for our purposes. Therefore, after carefully examining the implementation of the function, we can use it if it is available, otherwise we need to make it from scratch. For that reason, I think that deep learning frameworks are required for flexibility. The ability to flexibly add the custom functions in deep learning frameworks is essential to implement unknown

algorithms. As far as I evaluated, Keras was a flexible framework that allows us to seamlessly use the APIs of the low-level framework such as Tensorflow or CNTK to implement complex requirements while providing an easy-to-use API for building networks. But, because Tensorflow can not easily implement the custom operation we want, its advantages are somewhat diminished. On the other hand, ONNX which provides high-level API of PyTorch, MXNet and CNTK, could not be evaluated because it is not used in this work. However, if ONNX has convenience like Keras, its usefulness can be expected, because, PyTorch, MXNet, and CNTK can be extended with custom function.

8. Conclusion

We can create a DQN with DQN3.0-level performance in the creation manner of DQN.

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. DeepMind. A lua wrapper for the Arcade Learning Environment/xitari, December 2014. URL <https://github.com/deepmind/alewrap>
4. Roderick, M., MacGlashan, J., & Tellex, S. (2017). Implementing the Deep Q-Network. *arXiv preprint arXiv:1711.07478*. URL <https://arxiv.org/abs/1711.07478>
5. DeepMind. Xitari is a fork of the Arcade Learning Environment v0.4 (2017). URL <https://github.com/deepmind/xitari>.
6. Sutton, R. S., & Barto, A. G. (2018). RL: An introduction. Second edition, in progress * * * * Complete Draft * * * *. URL <http://incompleteideas.net/book/bookdraft2018jan1.pdf>
7. Sprague, N. (2016). Theano-based implementation of Deep Q-learning. URL https://github.com/spragunr/deep_q_rl
8. 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
9. 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>
10. 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
11. Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850v5* [cs.NE]. URL <https://arxiv.org/abs/1308.0850>
12. Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The Arcade Learning Environment: An evaluation platform for general agents. *J. Artif. Intell. Res. (JAIR)*, 47, 253-279. URL <https://arxiv.org/abs/1207.4708>
13. 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>

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. van Seijen, H., Fatemi, M., Romoff, J., Laroché, R., Barnes, T., & Tsang, J. (2017). Hybrid Reward Architecture for Reinforcement Learning. arXiv preprint arXiv:1706.04208.
17. Microsoft Research Montreal. (2017). Microsoft Research

Montreal@MSRMontreal, Twitter. URL

<https://twitter.com/MSRMontreal/status/87497770253815809>

18. 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>
19. 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. Application architecture

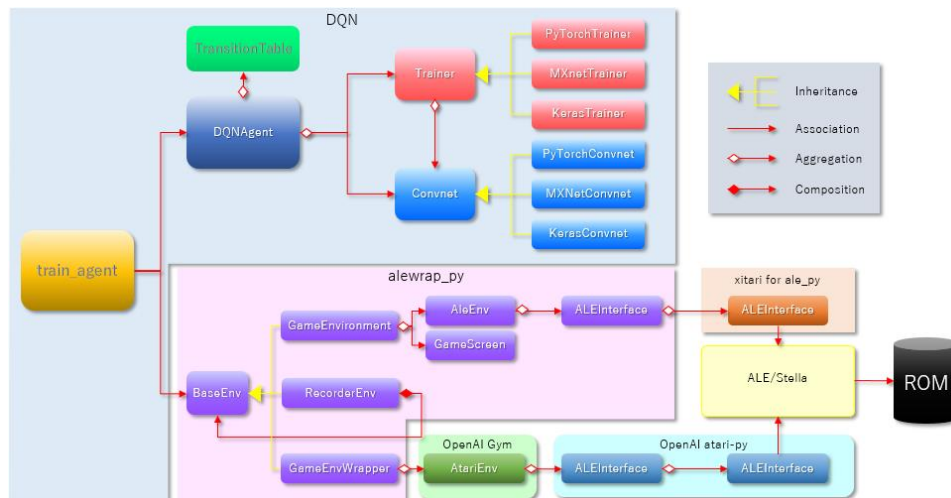


Figure 8: The architecture of my DQN. My DQN is implement of the Strategy pattern (Erich et al., 1995): Client: train_agent, Context: DQNAgent, Strategy: Trainer and convet; Each ConcreteStrategy classes corresponds each deep learning framework. alewrap_py reproduced alewrap in other package for improving 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).