IMPROVING THE EXPERIENCE REPLAY OF DEEP REINFORCEMENT LEARNING

ZHANG RUI

Department of Computer Science

City University of Hong Kong

CS6534 Guided Study

April 30, 2018

ABSTRACT

Experience replay is proposed by Lin in 1992 that is a core part of the deep reinforcement learning. It drives an agent learning more efficient and break the undesirable correlation among transitions via offering the earlier memories to the agent. Using this technique, we need to define the rules of storing and replaying. However, although there are many improved versions of ER focusing on the rule of replaying (e.g. Prioritized Experience Replay, Schaul et al, 2016), we still have a very limited understanding of the inner properties of experience replay. This is a reason of being lack of improvement for the mechanism of memory. Inspiringly, the problem recently was addressed by formulating a dynamical systems ODE model of Q-learning with standard ER (Liu, Zou, 2017). Meanwhile, the authors of the paper have proposed a simple adaptive experience replay (aER) algorithm which lets the memory making adjustment of size automatically based on the old absolute TD-error $|\delta_{old}|$. In this paper, I propose a new memory structure – Time-Tag memory which stores transitions based on the learning time step. Because of its data structure, it can efficiently extract the transitions (which is assign a time tag) at any learning time step. It is helpful for the adjustment step of aER algorithm. Moreover, I perform the idea of utilizing adaptive mechanism on prioritized experience replay (pER) which is called adaptive-prioritized experience replay (apER), including the key methods or techniques I used in this algorithm.

KEY WORD: [reinforcement learning; experience replay; prioritized experience replay; adaptive experience replay; adaptive-prioritized experience replay; Time-Tag memory structure; forgotten value; review]

1. Introduction

ONLINE Q LEARNING

Reinforcement learning is a sequential decision problem developed based on the Markov Decision Process. The nature of the problem is to find an optimal policy that can make a maximal benefit decision. In other words, the problem for agent is how to optimize their control of a given environment.

Two standard approaches to solving the control problem are policy iteration and value iteration working around the Bellman Equation. Online Q Learning (Eq.1), is an Q iteration algorithm developing based on the value iteration, using "max" trick to approximate the expected state value function $\mathbb{E}_{s_{t+1}\sim p(s_{t+1}|s_t,\pi(s_t))}[V(s_{t+1})]$ for solving model-free control problem (i.e. finding target policy π in the given unknown environment without transition model $p(s_{t+1}|s_t, a_t)$ and reward model).

$$\boldsymbol{Q}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)) \qquad (1)$$

In the practice, for some small-scale problems, we can use dynamic programming to find the optimal policy by building a Q value table (tabular look-up table[1]) for storing the $Q(s_t, a_t)$, in order to conveniently compute the $\max_{a'} Q(s_{t+1}, a')$ during the iteration.

DEEP Q LEARNING

However, the real problems are more complexity with more high-dimensional sensory inputs, such as the game GO has about 10170 states. Thus, the way of dealing large-scale problem is using a "black box" to approximate the value function shown in Figure 1.1

Figure 1.1 Estimate a value function of given current state ***s*** with a function approximation

---

Algorithm 1: online Q learning

Step1: take an action $\boldsymbol{a}$ based on behavior policy $\boldsymbol{\mu}$ and observe $(\boldsymbol{s_t}, \boldsymbol{a_t}, \boldsymbol{r_t}, \boldsymbol{s_{t+1}})$

Step2: TD target $\boldsymbol{y_t} = \boldsymbol{r_t} + \boldsymbol{\gamma max_{a'} Q(s_{t+1}, a'; W)}$ (2)

Step3: Update $\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{\alpha} \cdot (\boldsymbol{y_t} - \boldsymbol{Q(s_t, a_t; W)}) \cdot \boldsymbol{\nabla_W Q(s_t, a_t; W)}$ (3)

Repeat Step 1 - Step3 until convergence

---

[1] Tabular look-up table is a specific linear approximation: a state is treated as a feature. When individual stays at the specific state, the relevant feature is equal to 1, others are equal to 0 (i.e. 1-hot vector). The dimension of parameter $\boldsymbol{W}$ is equal to the number of states that each state $\boldsymbol{s_i}$ corresponds to a parameter $\boldsymbol{w_i}$.

The above process shows the simplest form of online Q learning that updates weights per step and discard incoming data immediately after the weights update. However, there are some obviously fatal issues: first, the input data s_t and s_{t+1} are sequential states between which have a strong correlation that break the i.i.d assumption of many popular gradient-based optimal algorithm; second, the input data only be used for once which the approximation model cannot sufficiently learn from. If the rare data is discarded immediately, the model will lose the chance of learning more surprising things from the rare data, and it is difficult to encounter again; the last issue is there is no gradient through TD target value (Eq.2) which will be changed as the weights are updated.

EXPERIENCE REPLAY

For solving the above issues, the experience replay is proposed (Lin, 1992), and becomes powerful core behind many advanced deep RL, like DQN (Mnih et al., 2015). An agent with the help offered by experience replay is able to learn from a stream of experiences and update its policy. The experience can be generated by the agent itself, or got from other sources. An experience is called a transition that is a tuple of (state, action, reward, next state), where the reward and next state are generated based on the given state and the given action. Because of the mechanism of replay, the transitions can be reuse and the correlations among transitions can be broken.

However, the limitation of experience replay is also obvious. The mechanism of standard ER's replay is randomly sampling the transitions, and discard them without considering any factors. That is the result of which the standard ER is a fixed-size memory. When the buffer is full, oldest memories will be discarded. Basically, it is similar to a FIFO buffer. Moreover, according to some research, they showed that the size of memory affects the learning dynamics and performance of the agent (Liu & Zou, 2017). The ER and pER both occur this phenomenon. It is worse that the pER being supposed to have better performance, will slow down the learning performance with a bad memory size. The phenomenon will be showed later in the preliminary experiment part of this paper.

CONTRIBUTIONS

In this paper, firstly I focus on the study of the performances between ER and pER, and the effect of the different memory size. The result shows that memory size has a non-monotonic effect on the performance of learning. Secondly, I develop a new memory structure Time-Tag that consist of two main part: hash map and sub-map. The hash map realizes fast operations on the transitions, like extracting *k* oldest transitions, updating the transitions' time tag, or searching the specific time node,

etc. The sub-map is used for storing the time node which include a set of transitions. I represent more detail in the methodology part of this paper. Based on the Time-Tag, I also propose two concepts: forgotten value and review, which support an idea of apER which is the adaptive version of pER.

2. Related Work

The experience replay technique has been widely utilized in many RL algorithms or experiments. Inspiringly, the results using experience replay are good in many cases, such as the DQN algorithm (Mnih et al., 2015) and double Q learning (Van Hasselt et al. 2016). Meanwhile, the improvement of ER technique has a good developing trend as well. For example, the prioritized experience replay (Schaul et al, 2016) that determine the priority of a transition by the magnitude of the temporal difference error (TD-error):

$$\delta = R + \gamma Q_{target}\big(S', argmax_{a'} Q(S', a') - Q(S, a)\big) \qquad (4)$$

where *S* and *a* are given state and action, the *S'* and *a'* is the next state and the action with maximal Q value according to the *S'* respectively. At the earliest, using TD-error as priority is proposed and reported to be useful, efficient, and more intuitive in many experiments finished by Moore & Atkeson (1993). Recently, this method is implemented again (Van Seijen & Sutton, 2013; Schaul et al, 2016). Except this method, there are many other measure ways as well, like weight experience, samples with rewards (Tessler et al, 2016), etc.

For figuring out the property of the ER, Liu and Zou propose to use ODE model. They derive an ODE model to simulate the learning process of Q learning and use a natural evaluation metric (i.e. the difference between the real parameters and the approximate parameters) to demonstrate the performance of learning. The experiment shows that the size of memory has a non-monotonic effect on learning rate. After that, they develop a simple adaptive experience replay which highly relies on the oldest transitions.

3. Analysis of ER's performance

In the prioritized Experience Replay, the mechanism of pER only consider how much information of a transition can be learned. However, it does not consider the properties of the ER. Thus, the issue is that whether the properties of the ER (e.g. memory size) affect the performance of pER. Is the pER always better than the simple ER? For solving the issue, we need to figure out the hidden mechanism of the ER. Hence, I have design a simple toy game called Treasure (figure 2.1).

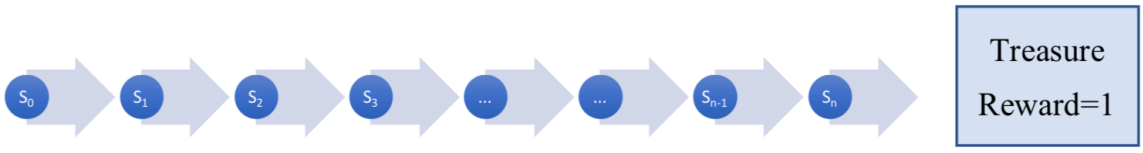

Figure 2.1 The toy game: Treasure

The rule of the game is simple. There are n states and 2 actions -- "right" and "wrong". When the agent takes a "wrong" action at any states, the agent will be back to the front step, and get 0 reward. Taking "right" actions will lead the agent to the treasure through a sequence of n states (the big grey arrows), and get the reward at the treasure. Then episode restart from S_0. Hence, if the agent wants to get the reward, there is only one optimal sequence that is hidden in a mass of "wrong" sequences. In other words, the experience of that unique successful sequence is rare.

For gaining convincing analysis, I totally have done about 20 trials with different memory size (i.e. the memory size is the unique variance in this experiment). For better comparing, I choose 4 representative results which demonstrate the main problems of ER and pER. I discuss the details and analysis of the results in the second part of this section (i.e. analysis of memory effect in ER)

MODEL SETUP

In this game, the space of state is one dimensional (i.e. the agent only considers one condition during the game). The reward function is

$$r(s) = sign(s); \quad (5)$$

$$sign(s) = \begin{cases} 0, if\ s\ is\ not\ Treasure \\ 1, otherwise \end{cases} \quad (6)$$

and the action is binary $a \in \{1, -1\}$, where **1** means the right action, **-1** means the wrong action. In a transition, the next state is generated according to the given s and a:

$$s' = s + a \quad (7)$$

Considering the large number of states setting in the game will cause a high resource requirement, so I use a simple linear function to approximate the Q value:

$$Q(s,a) = \phi(s,a)^T W \quad (8)$$

where $\phi(s,a)$ is a one-hot vector, and the corresponding one-hot matrix is $\Phi(S,A) = \begin{bmatrix} \phi(s_1,a_1) \\ \phi(s_2,a_2) \\ \cdots \\ \phi(s_n,a_n) \end{bmatrix}$.

At the beginning of the game, the hyperparameter W is randomly initialized and the one-hot matrix $\Phi(S,A)$ is set. As the agent exploring and training, the $Q(s,a)$ will approach to the $Q_{target}(s,a)$ and the TD-error Eq (4) will approach to 0 (in the ideal situation).

In this game, I used the mini-batch Q learning for training the agent.

ANALYSIS OF MEMORY EFFECT IN ERs

In this part, I have done a study to analyze the memory effect on standard ER and prioritized ER via comparing gradient descending and total learning steps of standard ER and prioritized ER. There are total 4 experiments with different size of memory. Except of the size of memory, other parameters are same.

Firstly, I chose the size of memory *N*=75 for the first experiment. Let's focus on the gradient graph. In the earlier stage (about 0 ~ 1500 steps), the different between ER and pER is significant that pER keeps a high gradient in this prior, but ER rarely performs high-level TD updates. In other words, the pER use its advance of priority to learn the experiences as much as possible. However, in the middle term (about 1500 ~ 2500), the pER is stuck. On the contrary, ER starts to learn and keep a high gradient periodically. This phenomenon of pER has been mentioned by Schaul (2016). Because of the priority mechanism, the initially high error transitions (which means the transitions with high TD-error at the beginning) can get a higher probability for being sampled. Thus, they get replayed frequently. However, when the transitions are sampled for learning too much times, the information of them is not surprising to the agent any more. Thus, the transitions become redundant. In my analysis, this is main reason of this phenomenon. Let's move to the graph (c) – total training step. From the graph, we can easily find the cross point of two lines at about 2000+ steps. This is exactly the prior of the phenomenon occurring.

Secondly, I changed the size of memory to 100. The performances of two ERs in the earlier stage is similar to the first experiment result. Surprisingly, the bad performance of pER in the first experiment seems to be solved via increment of memory size. However, the performance of standard ER gets worse. The efficient TD update is postponed to about 2500+ steps.

For the third experiment, I set the memory size equal to 150. The gradient of pER in the earlier stage is unexpectedly low. The efficient TD update starts from about 1600+ steps. The effect of it in the earlier stage cannot be ignored. Although pER still have a huge advantage comparing with ER, the overall performance of pER is back to the performance showing in the first experiment (graph(i) and graph(c)). Hence, blindly increasing the memory size is not a way optimize the performance of pER. From the graph(g), we can learn that the ER performs worse. In the intuition, as the memory size increasing, it becomes more difficult for the standard ER to sample useful transitions.

The final experiment, with memory size N=300, the graph(l) obviously shows that pER is suffering an epic fail which is worse than the performance of ER.

Except of the analysis above, there is a very important phenomenon which both ERs have occurred. Firstly, we can find that every gradient graph shows an increment at the tail part of the graph. Secondly, let's put our focus on all graphs of total training step. We can find that both curves show linear increments of steps which means the model is well trained already, since only the agent approaches the Treasure state, the episode will be plus 1. In other words, the agent can get to the Treasure state in an optimal, nearly fixed number of steps which make the curves perform linear.

Combine with two points above, it seems it is unreasonable that the error is increased after the optimal policy being found. However, this overshooting phenomenon is also mentioned by Liu & Zou on their experiments. Meanwhile, Schaul also said the similar topic in his paper that "the memory requirements for deep Q learning are dominated by the size of the replay memory, no longer by the size of the neural network".

According to the experiments' results showed below, analysis I made, and other experts' comments, it can be concluded that:

- The memory size of memory is a main factor affecting the reinforcement learning process.
- Non-monotonic effect on the learning rate of both ERs
- Too much or too little memory both slow down the rate of learning
- An improper memory size causes redundant problem in pER
- Both versions of ER exist overshooting problem.

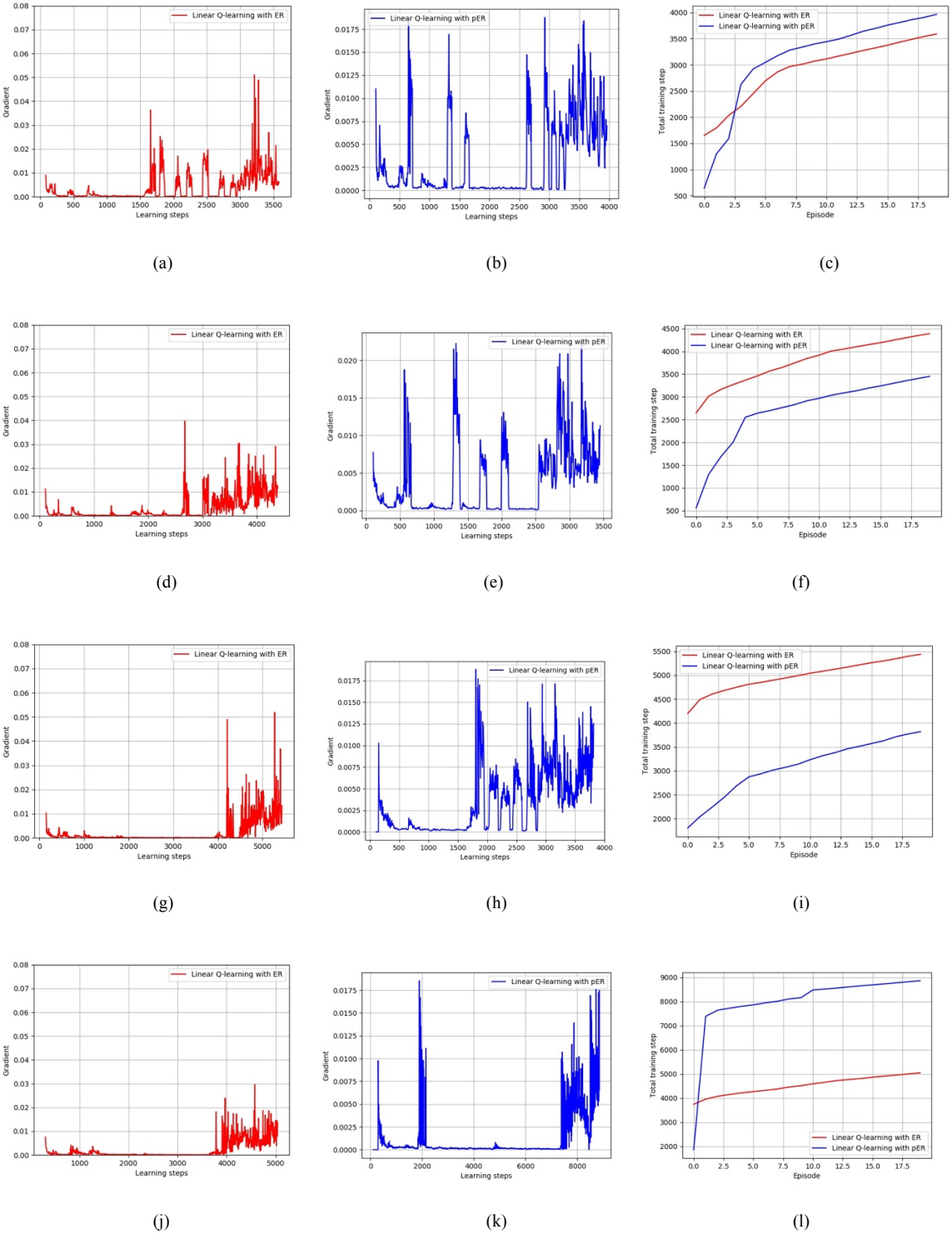

Figure 2.2 **The first row**: the experiment with memory size 75; **The second row**: the experiment with memory size 100; **The third row**: the experiment with memory size 150; **The fourth row:** the experiment with memory size 300

4. Dual memory system & aER

From the above results, we learn that the properties of ER have dominant effect on the performance of ER. Considering the properties of ER like memory size that is a new way for improving the performance of ER. In my study, I followed the new way and introduced an improved version of aER (algorithm 2).

A simple adaptive experience replay (aER), an algorithm proposed for this problem recently, controls the size of memory comparing the current TD-error of n-oldest transitions with the last one. Obviously, the algorithm's idea is quite intuitive and easy-to-understand, which treats TD-error as a criterion of the performance of learning and adjust the size of memory buffer based on the criterion. However, the trouble of implementation is that there is lack of memory structures storing the information of time step. In other words, the question is how to define a transition's age. For solving the problem, I introduced a dual memory system for improving the aER algorithm.

The dual memory system consists of two single memory structures: master memory and assistant memory. Both memory structures serve for different roles. The master memory is an adaptive memory which in charge of sampling the transitions for learning. The assistant memory is the Time-Tag memory I proposed that takes charge of the adjustment stage. It mainly helps the agent to control the size of the master tree according to the performance of learning.

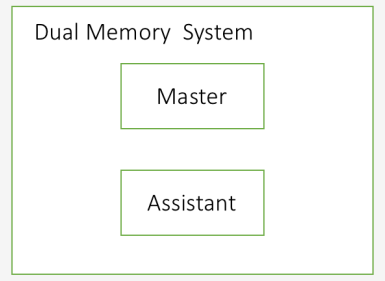

Figure 4.1 the structure of dual memory system

In the dual memory system, the master memory and Time-Tag memory both have transitions stored. The consistency of transitions needs to be ensured. Basically, the consistency is violated in the below 3 scenarios: storing, after learning and memory adjustment. In order to ensuring the consistency, the time tag is used thanks for the Time-Tag memory recording the time information. For clearly demonstration, I discuss the time tag and consistency in detail after I introduce the Time-Tag memory.

Algorithm 2: Improved aER with dual memory system

1:. Input: initial memory size N_0, mini-batch size m, discount factor γ, initial greedy ϵ, learning rate α, #checked oldest transitions n; memory adjustment internal k; replay period K; budget T; minimal memory size i;

2:. Initialize the weights W_0, the replay memory buffer with N_0, initialize $|\delta_{old}|$=0

3:. Interact with environment, store transitions

4:. For t=1 **to** T **do:**

5:. Interact with environment, store transitions into Time-Tag memory first

6:. Time-Tag memory returns a transition with time tag, and store the transition to master memory

7:. **If** mod (t, K) = 0 **then**

8:. Sample transitions from master memory buffer, learn from the samples, update time tag

9:. **If** mod (t, k) = 0 **and** memory **is FULL:**

10:. Select n oldest transitions along the time chain of the Time-Tag memory

11:. Calculate $|\delta_{old}|'$ of n oldest transitions

12:. **If** $|\delta_{old}|' > |\delta_{old}|$:

13:. Enlarge the size of memory with k: N = N + k;

14:. $|\delta_{old}| \coloneqq |\delta_{old}|'$

15:. **Else:**

16:. **If (**the current memory size – k) < i:

17:. **Continue;**

18:. Randomly select k oldest transitions from the set of n-oldest transitions

19:. Delete k oldest transitions from master memory and Time-Tag memory

20:. Shrink the memory N = N - k

21:. Re-select n oldest transitions along the new time chain after delection.

22:. $|\delta_{old}| \coloneqq$ recalculate the absolute of TD-error of n oldest transitions

23:. **End if**

24:. **End if**

25:. **End if**

26:. End for

5. Time-Tag Memory

In the aER algorithm and the apER algorithm I proposed, there are many operations for transitions which is highly based on the learning time step. For example, in the adjustment part of aER, we need to select *n* oldest transitions. The issue is how do we determine a transition that whether it is oldest. The structure of memory buffer utilized in standard ER is a sample FIFO stack. Besides, the sumTree structure used in priority experience replay only contain the information of priorities of transitions. Hence, the current memory structure is unfriendly for some algorithm which rely on the time information of transitions.

HASH DATA STRUCTURE

Hash table (Hash map) is a data structure that utilizes the mapping between keys and values for helping operations (e.g. searching, updating, extracting) reaching fast and efficient performance. Hashable objects that are considered equal return the same hash value. For off-policy reinforcement learning, firstly an agent explores the environment, get a feedback (i.e. reward and next state) from interaction with environment. Secondly, the agent stops playing, start to replay the transitions stored before and learn from those transitions. Thus, at a learning time step, the agent can collect a number of transitions and store them into memory. Obviously, the relation between transitions and learning time step is many-for-one. The idealized data structure for representing the relation between transitions and learning time step would be hash map which show more advance than other data structure, like stack, binary heap, queue, etc.

LINEAR MAP

In the process of design, I have considered the linear map first. Linear map is a simple hash table which can use tuples to realize the mapping between keys and values. The structure is showed below:

$$ITEMS = [(k_1, v_1), (k_2, v_2), (k_3, v_3), \dots]$$

Basically, we can simply map the relation between the learning time step t_i and the transition T_i as (t_i, T_i), and put the tuple into $ITEMS$. Since the time step is sequential, the tuples in $ITEMS$ must be ordered ascendingly. If we want to extract a specific transition from a set of transition corresponding with the specific time step, we need to do a binary search firstly for finding the key of the specific time step. This sub-operation's big O is O(logn). After that, we also need to search the transition set for finding the objective transition. Because the transition is not a number, the only thing we can do is search the transitions one by one. The big O of it is linear. Thus, the big O of the whole searching operation is linear.

Although linear map can store the (t_i, T_i) based on the time sequence, the big O is disappointing. It is not an issue if the training model only need a few learning steps for training. Unfortunately, in the practice, the deep reinforcement learning always need about ten thousand or even millions learning steps for training. It is impractical to use linear map.

HASH MAP: HASH THE LINEAR MAP TO BE SUB-MAP

For solving the limitation mentioned above, I introduce a hash function to segment the linear map to many pieces of sub-map. For keeping the sequence of the time step after hashing, I simply use the build-in hash function of Python 3.6, and use the below function for getting an index:

$$index = hash(t) \,\%\, \#\text{sub\_}map \qquad (9)$$

where $\#\text{sub\_}map$ is the number of segmented sub-map. The index can be used directly on hash map for extracting the corresponding sub map. The idea is simple, and be demonstrated by figure 5.1.

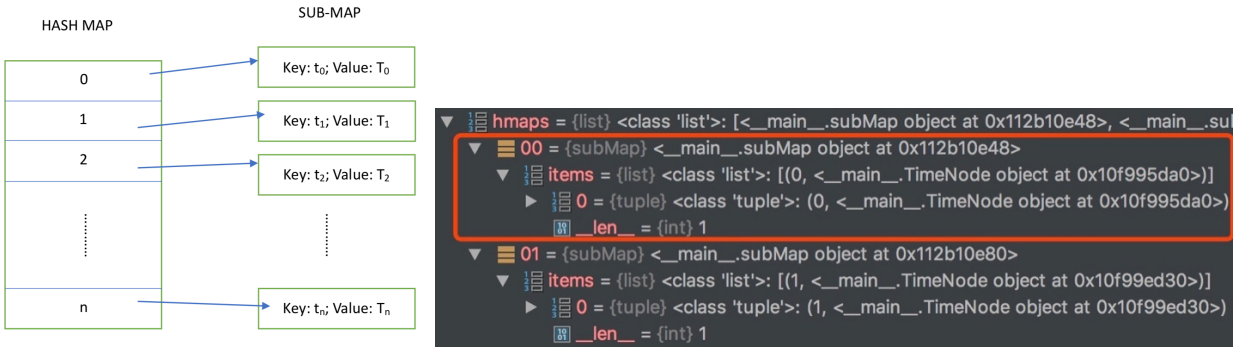

Figure 5.1 UP: The relation between hash map & sub-map; DOWN: The implementation of hash map &sub-map

TIME CHAIN

Searching or extracting transitions successively is an important requirement for aER algorithm. In the adjustment step of aER, we need to collect *n*-oldest transitions from the Time-Tag memory. However, as the training processing, the number of transitions corresponding to the oldest time step should be descend. Thus, there may be a case that the number of transitions of the oldest time's sub-map is less than *n*. In this case, we need to find the rest oldest transitions from the secondary oldest time's sub-map. It is even worse that we need to collect those transitions from several time's sub-map. Hence, it is inefficiency and it wastes the property of time, i.e. sequence.

Time chain is a chain table designed for highly utilizing the property of time. In the time chain, every node is a time step. Every time node stores a set of transitions collected by the agent at the relevant learning time step.

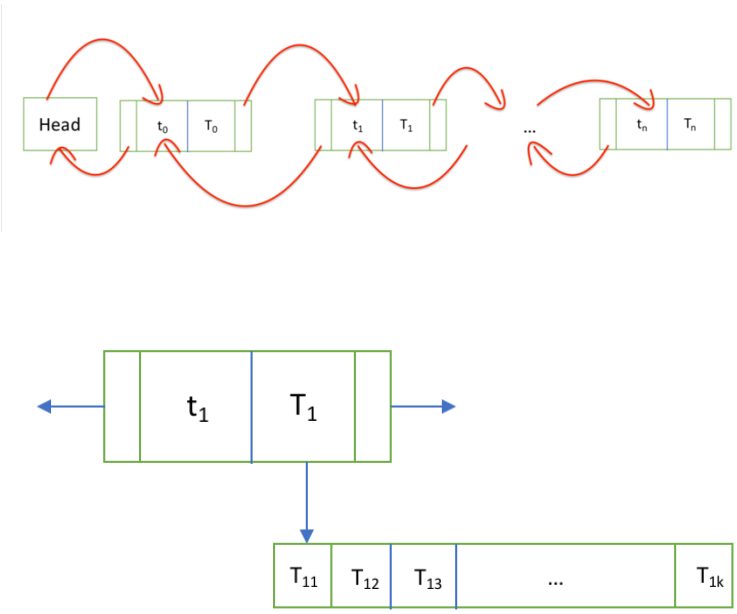

Figure 5.2 UP: The time chains; DOWN: A time node structure, says at time step t_l the agent has collected *k* transitions

In the design, every time node is a bidirectional node. A node consists of a time step and a set of transitions. In doing this, the sub-maps have been connected each other via their time nodes. The trick realizes searching on sequential sub-maps without recalculating the indies via Eq (9).

TIME TAG

The transition stored in the Time-Tag memory is a bit different from the normal transition consisting of four basic elements. When a new transition income, the transition will be assign a time tag at the tail of the transition before it being inserted to the transition set *i*. The time tag is:

$$time\ tag = [t_i, idx_i]$$

where t_i is a learning time step, idx_i is the index that points at the end of the transition set *i*. After assigning, the transition will extend to be (S, A, R, S', t, idx). A transition with a time tag is unique so that we can accurately update the specific transition. Thus, the time tag is not only designed for quickly extracting transitions from the transition stack, but also helping Time-Tag memory communicating (or keeping consistency) with other memories in the experience replay with dual memory system.

USING TIME TAG for CONSISTENCY

Basically, the transitions may reach inconsistency in three operations: storing, learning, adjustment. For these cases, I designed a process for each case. For storing case, the new transition should firstly be stored into the TT memory and the TT memory will return a marked transition with a time tag. At the end, the marked transition is stored into sumTree. The process is demonstrated as figure 5.3. For

the second case, because the samples have been used for learning, the time tags of them need to be updated. Thus, after learning we need to update their time tags on both memories. The main process is demonstrated as figure 5.4. For the last case, when the memory shrinks, the oldest transitions need to be removed from both memories at the same time. The process of adjustment is showed as figure 5.5.

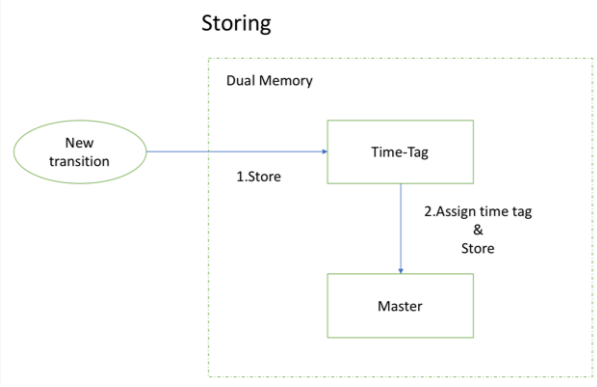

Figure 5.3 The process of storing

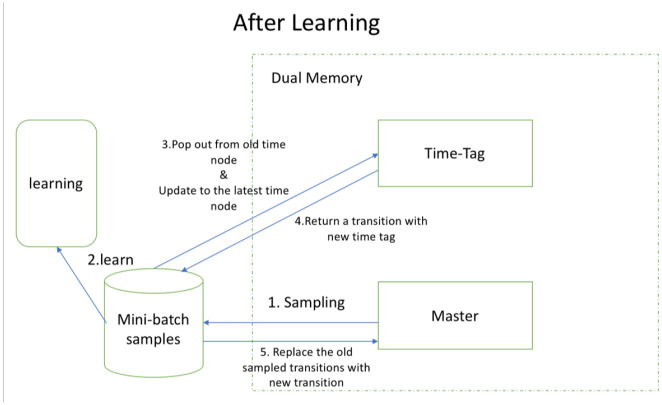

Figure 5.4 the process of updating time tag after learning

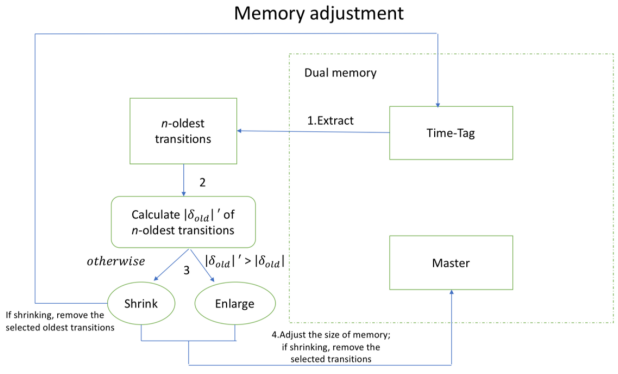

Figure 5.5 the process of memory adjustment

TIME-TAG MEMORY

Combining all parts above, the structure of Time–Tag memory (TT memory) is built as the figure 5.6. There are two main parts: hash map and sub-map. The part of hash maps consists of every index of every sub-map. It is the main entrance of accessing a time node or a specific transition. The part of sub-map is used to store time nodes. Every sub-map item only stores the relevant time node (i.e. every sub-map item only contains one time node) for protecting the memory from hash crashing and maintaining the time sequence. For solving the connection problem between two neighbor sub-maps, values in sub-maps are time nodes which contain sets of transitions, instead of saving stacks of transitions directly.

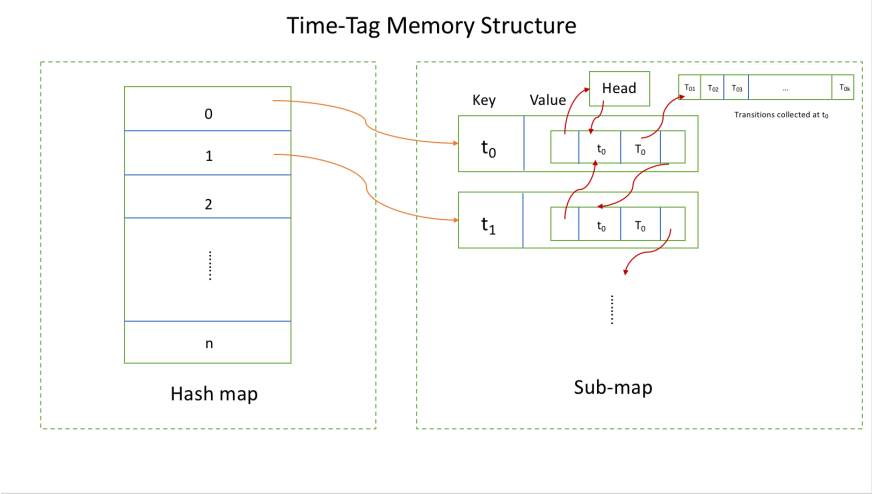

Figure 5.6 The structure of Time-Tag memory

By using this memory, the transitions are stored based on the learning time step. When the agent collects a new transition from interaction, the transition will be assigned a time tag first before being save. Secondly, the memory system will check whether a time node exists which corresponds to the current learning time step. If the time node does not exist, the system will create a new time node for the current learning time step and connect the new node to the tail of the time chain. After that, the transition is inserted to the transition stack of the node. The transition is inserted to the stack directly while the node exists. There are only two cases that cause the transitions to be removed from the transition of the time node. One of the cases is the removed transitions is sampled for training. After training, it will be assigned a new time tag and stored into the latest time node. In more intuitive way to explain, the sampled transition's old-time tag is refresh. Another case is that the removed transitions are very old and it is always ignored (i.e. not be sampled). Thus, they have a high probability of being removed during the shrinking of the memory.

IMPLEMENTATION

According to the structure representing above, the TT memory is assembled with 3 data structure: hash table (the hash map), linear table (the sub-map item), and node (time chains). So, we need to define 3 classes for each data structure. Except of these, we need to define a class of operations of the TT memory. In the implementation, there is a technical issue that we cannot directly remove transitions from the stack, since direct removal will cause chaos of the time tag. For this problem, I use a trick that assign a *None* value to removing transition. After that, I check whether the elements in stack are all *None*. If it is true, the time node becomes useless and to be delete, since the time node does not contain any transitions, and the reverse of time step is impossible.

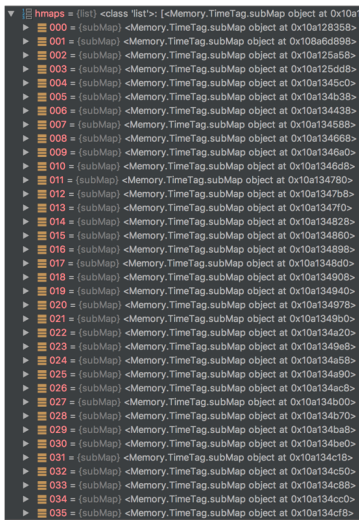

Figure 5.7 The implementation of the hash map

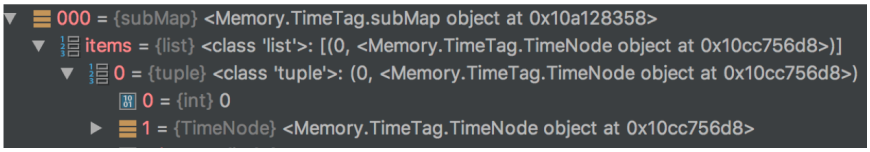

Figure 5.8 The sub-map of 0^{th} time step

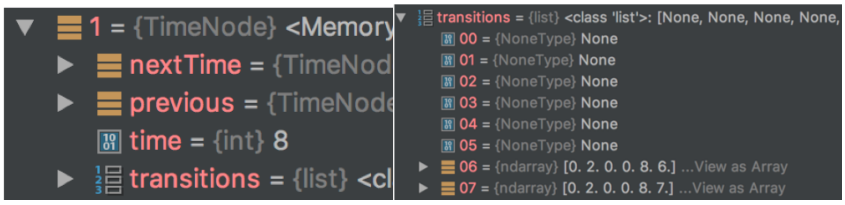

Figure 5.8 Left: The time node of 8^{th} time step; Right: the part of transition set of 8^{th} time step

ADVANCES

The TT memory lets an agent be able to efficiently utilize time information of the transitions. By using time information, we can calculate the *n*-oldest transitions' TD-error that is a criterion for deciding enlarging or shrinking the size of memory. Except of this, the forgotten value I mentioned later is also calculated via time information.

Thus, in some extent, TT memory can assist the use of time information during the learning. The time information should not be ignored. Focusing on utilizing time information of a transition may be another way to improve the experience replay.

LIMITATION

However, the limitation of TT memory is also obvious. During the agent exploring and training, the TT memory need to generate a sub-map and a time node for every learning time step. As I mention above, the number of required training steps is huge, which means the TT memory is not suitable for some complex models because of the large computing resources required.

USING DUAL MEMORY SYSTEM ON aER

After finishing the design of dual memory system, I have implemented it on my improved version of aER. Meanwhile, I ran the same mini-batch Q learning with my aER and dual memory system in the toy game. Then, I compared the results with the one of ER and pER which I have shown above (figure 2.2 the third row). Inspiring, the results demonstrate the aER makes a good progress on the performance of learning.

In my experiment, I set initial memory size M equal to 150, batch size m equal to 50, n-oldest transition n equal to 30. Every k=30 learning steps, memory calculates the n-oldest transition's TD-error and adapt the size by itself.

Firstly, let's focus on the performance comparison (figure 5.9). The left-hand plot is the performance of aER and the right-hand plot is the performance of ER and pER. Although the aER uses the same replay mechanism as the one of ER, the aER performs significantly better than ER. The total training step of aER is only about 3900, but the counterpart of ER is nearly 5500. Moreover, it is surprising that the learning performance of aER is similar to that of pER.

Secondly, I compare the aER with ER via observing the gradient plot. As I said above, the replay mechanism of aER also is random selection. But the aER starts to learn at about 2000th time step, approximately half the time of ER. Besides, the aER can keep learning until the end of the training.

One more thing, the tail of the plot of aER also demonstrates that the overshooting problem is solved since the trend of gradient keeps descending at the end stage.

Finally, we move to the figure 5.11 which is the change of memory size during the learning process. I have set the minimal memory size to 120 so that the minimal size in the plot is equal to the parameter I set. The maximal size is 300. Comparing this figure with the left-hand plot of figure 5.10, the trend of gradient and the trend of the memory size changed is similar. Intuitively, the memory size is dominated by the performance of learning at the specific time step. When gradient increases, memory enlarges itself for collecting more interesting information for learning. On the contrary, the memory shrinks itself and deletes some useless transitions for limiting some useless information being stored again. Thus, this characteristic can alleviate the redundant problem.

Overall, the improvement points caused by the aER with dual memory system are:

- The learning performance is better the standard ER
- Overshooting problem is solved
- Shrinking alleviates redundancy
- The memory buffer can be self-adaptive that not only improves the performance of learning, but also saves resources.

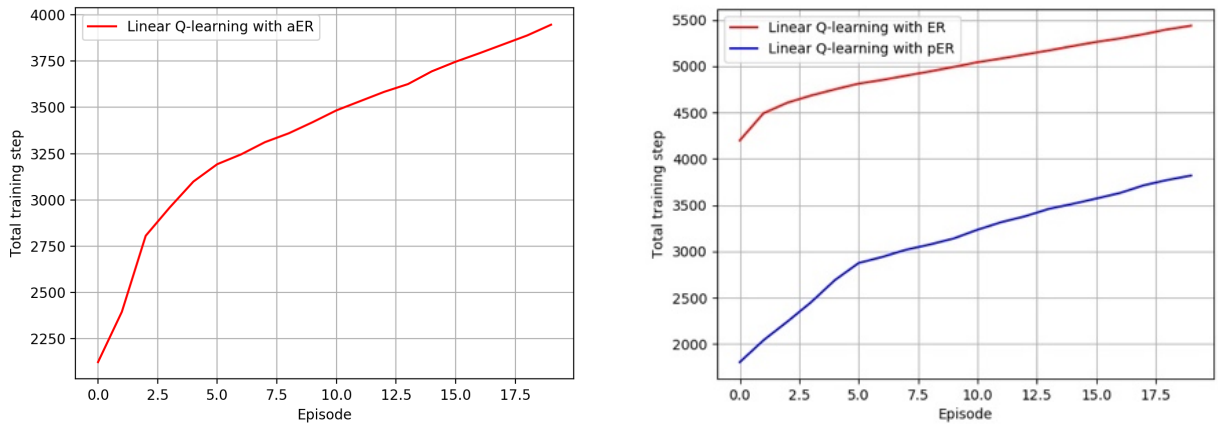

Figure 5.9 Left: The performance of aER with 150 initial memory sizes; Right: The performance of ER and pER with 150 fixed memory sizes

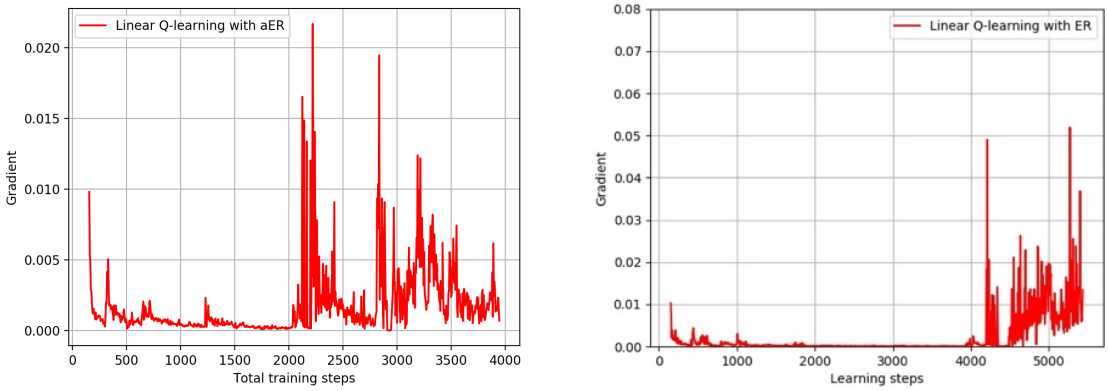

Figure 5.10 Left: The gradient of aER with 150 initial memory sizes; Right: The gradient of ER and pER with 150 fixed memory sizes

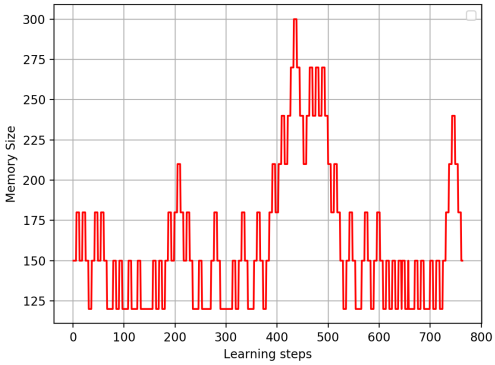

Figure 5.11 The change of memory size of aER

6. Adaptive – Prioritized Experience Replay

The adaptive experience replay (aER) has showed a better performance compared to having a fixed-size replay buffer. In this part, I make an effort on designing the algorithm which utilizes adaptive concept on the prioritized experience replay. I call this algorithm as adaptive prioritized experience replay (apER). I will discuss the main components of this algorithm below.

DUAL MEMORY FOR apER

The dual memory also is one of the main cores of the apER. The memory system consists of sumTree and Time-Tag I proposed above. Both memory structures serve for different roles. The master memory is sumTree which in charge of sampling the transitions for learning. Basically, its role is same as the one of pER. Meanwhile, when we talk about the effect of memory size, this effect is based on the size of sumTree. Hence, it is the master memory dominating the learning process. The Time-Tag memory

takes charge of the adjustment stage. It mainly assists the sumTree, like controlling the size of sumTree, improving the priority of the transitions with the long-term low priority via review.

THE FLEXIBLE SUMTREE

The normal version of sumTree using in the prioritized experience replay is simple and efficient. The structure of the normal sumTree consists of two main parts: tree and data frame. Both parts are separated and do not have any connection (e.g. node) between them. Firstly, the tree's structure is:

$$Tree: [\, N-1 \mid N \,]$$

where *N* is fixed capacity of the memory. The *N-1* stores all parent nodes and the Tree [0] is the root. The *N* part stores all leaves nodes which record priorities of the transitions which stored in the data frame. If the *N* is fixed, the index of mapping priorities to the relevant transitions is fixed too. Says the index of transition is ***i***, then the corresponding index of tree leaf node is ***i* + *N-1***. It is very efficient way for retrieving the corresponding priority and transition. The more intuitive way to show the structure of the sumTree is like the figure6.1.

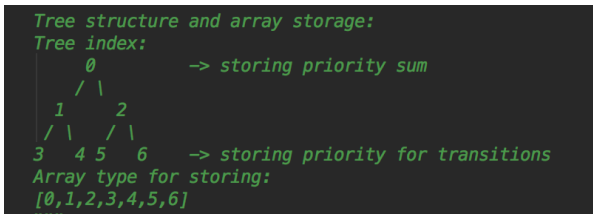

Figure 6.1 The structure of sumTree

However, the limitation of sumTree is not flexible. The connection between tree and data frame highly relies on the fixed capacity of the memory. In other words, this simple version can work well on the prioritized experience replay, but it is not suitable for the adaptive version of prioritized experience replay. This is the main problem for realizing the apER. Actually, it is difficult to finding a data structure that is flexible and is able to represent the priorities' information efficiently, as well as make a strong matching with the corresponding transitions. Unfortunately, my study about apER is stuck in this problem. However, I do not give up and tried to use the database and set a maximum capacity to simulate a memory buffer, which I called oraSumTree. The table of the database consists of 8 tuples: (ID, STATE_, ACTION_, REWARD_, STATE_NEXT, PRIORITY, TIME_STEP, IDX). The example of the table is showed as figure 6.2

ID	STATE_	ACTION_	REWARD_	STATE_NEXT	PRIORITY	TIME_STEP	IDX
1	1	2	0	0	0.154731	1	17
2	0	1	0	1	0.127837	2	22
3	1	1	0	2	1	0	152
4	2	1	0	3	0.113034	1	19
5	3	2	0	2	0.0643233	2	23
6	2	1	0	3	1	0	155
7	4	2	0	3	0.148712	2	24
8	3	2	0	2	0.125235	1	20
9	2	1	0	3	1	0	8
10	3	1	0	4	0.144668	1	21
11	4	1	0	5	0.0967332	2	25
12	5	2	0	4	0.101311	2	26
13	4	1	0	5	1	0	12
14	5	2	0	4	0.101311	2	27
15	4	1	0	5	0.126775	1	23
16	5	1	0	6	1	0	15
17	6	2	0	5	0.152127	2	28
18	5	2	0	4	0.122045	1	24
19	4	1	0	5	1	0	18
20	5	1	0	6	0.159909	1	25
21	6	2	0	5	0.152127	2	29
22	5	2	0	4	0.122045	1	26
23	4	2	0	3	0.148712	2	30
24	3	1	0	4	0.131242	2	31
25	4	1	0	5	0.126775	1	27
26	5	2	0	4	1	0	25
27	4	2	0	3	1	0	26

Figure 6.2 The table of oraSumTree

In this version of sumTree, the ID and (TIME_STEP, IDX) both are the key for finding the unique transitions stored in the database.

In the implementation, the result of using this version sometimes is quite weird. Sometimes, after enlarging the memory (actually enlarging the limitation of max capacity), there are some transitions are stored with 0 priorities which deeply effects the learning process. Because of the limitation of time, I do not have deeper research on it.

However, I still make an effort on finding the optimal data structure for improving the sumTree.

FORGETTING VALUE

The famous hypothesize Hermann Ebbinghaus forgetting curve demonstrates the forgetting rate as the time goes on. A typical graph of the curve purports to show that humans tend to master well their newly learned knowledge in a short period. The period may be a few days or a week. The speed of forgetting is different caused by many factors, such as the individual memory ability. Inspired by the hypothesize, I introduced a concept about forgetting value to control which experiences to erase from the memory, rather than dropping them via the FIFO way.

The forgetting factor (Eq.10) can well simulated the forgetting curve shown as Figure 6.3,

$$f^t = e^{-\frac{T}{S}} \quad (10)$$

where T is the time interval that is equal to subtract the initial time step t_0 from the current time step t, where the initial time step t_0 is equal to the specific time step of the transitions being stored in the memory, or the specific time step of the transitions after being sampled. The initial time step t_0 can be extract from the Time-Tag memory. When an experience is newly stored in the memory or has been

sampled, the initial time step t_0 of the experience will be reset to the current time step (as I mentioned above). The S represents the ability of the memory (Eq11).

$$S = \beta \cdot n \quad (11)$$

where n is the batch size of sampling, β is a weight for controlling the performance of the forgetting value influenced by the change of n.

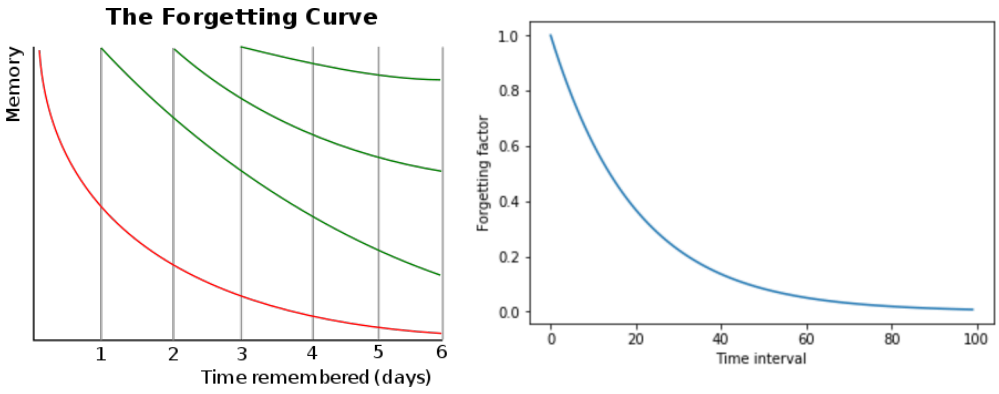

Figure 6.3. **Left:** the Hermann Ebbinghaus forgetting curve[2]; **Right:** the simulated forgetting curve of a single experience in the time interval from 0 to 100

Considering to a huge computing quantity, the forgetting value should be calculated every C step rather than every step. Meanwhile, there are two choices: partially calculating or completely calculating. I think the first choice may be better. It reduces the cost of computing and focuses on the low-priority experiences. The high-priority experiences can be frequently sampled so that its forgetting value will not descend much before the experiences have been exploited sufficiently. When these experiences have been learned sufficiently, their priority will be decreased and they will naturally enter the list of partially calculating. I need more experience later to demonstrate which is better.

Based on the considering above, it is intuitive that execute the calculating of forgetting values when $|\delta_{old}|' \leq |\delta_{old}|$, i.e. the memory size need to be shrink, for giving a criterion to decide which oldest transitions should be removed. However, there is an implementation problem which is the choice of threshold.

REVIEW

Review is the best way to confront the forgetting. Thus, I introduce a way to monotonically increase the priority in order to increase the probability of sampling (Figure 6.4).

[2] The graph is collected from Wikipedia: Forgetting curve https://en.wikipedia.org/wiki/Forgetting_curve

The review increment (Eq.12),

$$\mathrm{Rew}^{t} = \tau \cdot \frac{p^{t}}{(f^{t} + \mathrm{Rew}^{t-1})} \quad (12)$$

where Rew^{t-1} is the review increment at $(t-1)$ time step, p is the priority of the experiences, f^{t} is the forgetting factor at time step t and τ is the discount factor which control the increment in order to keep the stability and prevent the low prioritized experience overwhelming the high priority experience after reviewing.

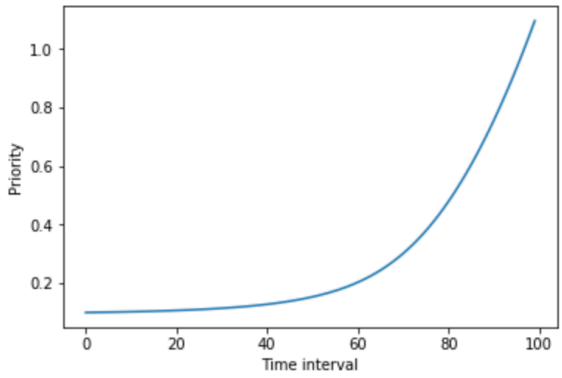

Figure 6.4 The increment of priority

The Figure 6.4 demonstrates that the low priority's increment state. At the beginning stage, the increment is slow since it intuitionally seems that the experience with low priority is not important at the current training stage. For efficiency and stability reason, the slow increment at the beginning stage not only gradually increase the agent's attention on the low priority, but also make agent keep learning the current high priority experience without effect caused by the review process. As the training goes on, the increase rate ascends sharply at the late stage. In this stage, the forgetting value is very small (it gets no chance to be sampled, the initial time step t_0 is not updated, the time interval is accumulated), thus the increase uprush can increase the probability of sampling in maximum.

APER ALGORITHM

Combining all ideas above, I propose an adaptive algorithm that using on prioritized experience replay.

Algorithm 3: Adaptive prioritized experience replay

Input: initial memory size N_0, mini-batch size m, discount factor γ, initial greedy ϵ, learning rate α, #checked oldest transitions n; memory adjustment internal k; replay period K; budget T
Initialize the weights W_0, the replay memory buffer with N_0, initialize $|\delta_{old}|$=0
Interact with environment, store transitions
For t=1 **to** T **do:**
    Interact with environment, store transitions into sumTree and Time-Tag
    If mod (t, K) = 0 **then**
        Sample transitions from sumTree memory, learn from the samples, update priorities
        If mod (t, k) = 0 **and** memory **is FULL:**
            Select n oldest transitions from Time-Tag
            Calculate $|\delta_{old}|'$ of n oldest transitions
            If $|\delta_{old}|' > |\delta_{old}|$:
                Enlarge the size of memory with k: N = N + k;
                if necessary, do a synchronization between sumTree and Time-Tag
                $|\delta_{old}| \coloneqq |\delta_{old}|'$
            Else:
                Calculate the forgetting values of n oldest transitions
                Delete k oldest transitions which are below the threshold;
                Shrink the memory N = N - k
                Synchronization of Time-Tag and sumTree
                Re-select n oldest transitions after synchronization
                $|\delta_{old}| \coloneqq$ recalculate the absolute of TD-error of n oldest transitions
            End if
        Review to improve the priorities
        End if
    End if
End for

---

7. Conclusion

In this study, firstly I do a research on the effect of different memory size. The results of those experiments are very helpful for us to recognize the effect of size of memory. Meanwhile, by reading the relevant papers, I have found some phenomenon occurring on my experiments which is similar that reported on the papers, and get a reasonable interpretation for it. Thus, according the analysis of experiments and interpretation from the papers, the bad memory size (too small or too large) will cause a disappointing learning performance. Besides, the fixed-size memory always has an overshooting problem because they cannot adjust the size of memory based on the learning process, which can be measure by TD-error. In my opinion, how to control or how to make memory self-adaptive is a topic which deserve to be researched in this field.

In the second stage of this study, I have designed a memory structure which stores transitions based on learning steps. It is more like an assistant for helping the master memory control their size or helping the transitions which has long-term low probability of being sampled, in order to keep the diversity of the transitions. The limitation of it is the issue of resources while the model is complex and need a bunch of steps for training.

In the third stage of the study, I proposed the algorithm adaptive – prioritized experience replay. The algorithm uses adaptive concept on the prioritized experience replay. It can self-control the size of memory based on the learning performance (e.g. TD-error). However, the implementation of the algorithm still has some technical problem, such as sumTree's flexible problem.

Overall, the study I did is mainly focus on improving the experience replay via control the size of memory.

8. Future Work

Although I have proposed the idea of apER algorithm in the third stage of this study, it is still on developing. There are some technical problems need to be solved. Meanwhile, since the limitation of the time, the performance of this algorithm does not have a comprehensive experiment for proving its effectiveness. Thus, in the future works, I will go on improving this algorithm, and make an effort on solving the technical problem, like developing a flexible sumTree, improving the Time-Tag memory on the resources problem, etc.

9. Code

- aER: https://github.com/royukira/MyAI/blob/master/Q_learning/aER.py
- Time-tag memory: https://github.com/royukira/MyAI/blob/master/Memory/TimeTag.py
- Toy game experience:
  - https://github.com/royukira/MyAI/blob/master/Q_learning/Q_Brain_Simply.py
  - https://github.com/royukira/MyAI/blob/master/Q_learning/QpER.py;
  - https://github.com/royukira/MyAI/blob/master/training_env/Simply_Teasure_Game.py

REFERENCES

[1] Lin, Long-Ji. Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine learning, 8(3-4):293–321, 1992. □

[2] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, Petersen, Stig, Beattie, Charles, Sadik, Amir, Antonoglou, Ioannis, King, Helen, Kumaran, Dharshan, Wierstra, Daan, Legg, Shane, and Hassabis, Demis. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015. □

[3] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver. Prioritized Experience Replay. ICLR 2016, arXiv:1511.05952v4, 2016. □

[4] Watkins, Christopher JCH and Dayan, Peter. Q-learning. Machine learning, 8(3-4):279–292, 1992.

[5] van Seijen, Harm and Sutton, Richard. Planning by prioritized sweeping with small backups. In Proceedings of The □30th International Conference on Machine Learning, pp. 361–369, 2013. □

[6] Moore, Andrew W and Atkeson, Christopher G. Prioritized sweeping: Reinforcement learning with less data and □less time. Machine Learning, 13(1):103–130, 1993. □

[7] Guo, Xiaoxiao, Singh, Satinder, Lee, Honglak, Lewis, Richard L, and Wang, Xiaoshi. Deep Learning for Real- □Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N.D., and Weinberger, K.Q. (eds.), Advances in Neural Information Processing Systems 27, pp. 3338–3346. Curran Associates, Inc., 2014. □

[8] Hinton, Geoffrey E. To recognize shapes, first learn to generate images. Progress in brain research, 165:535–547, 2007. □

[9] Ruishan Liu, James Zou. The Effects of Memory Replay in Reinforcement Learning. arXiv:1710.06574v1, 18 Oct 2017. □