BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **11**, No 3

Sofia • 2011

Time Hopping Technique for Faster Reinforcement Learning in Simulations

Petar Kormushev[1], *Kohei Nomoto*[2], *Fangyan Dong*[3], *Kaoru Hirota*[4]

[1] *Department of Advanced Robotics, Istituto Italiano di Tecnologia, via Morego 30, 16163 Genova, Italy*
[2] *Graduate School of Science and Engineering, Yamagata University, Yamagata, Japan*
[3, 4] *Department of Computational Intelligence and Systems Science, Tokyo Institute of Technology, 226-8502, Japan*

Abstract: *A technique called Time Hopping is proposed for speeding up reinforcement learning algorithms. It is applicable to continuous optimization problems running in computer simulations. Making shortcuts in time by hopping between distant states combined with off-policy reinforcement learning allows the technique to maintain higher learning rate. Experiments on a simulated biped crawling robot confirm that Time Hopping can accelerate the learning process more than seven times.*

Keywords: *Reinforcement learning, biped robot, discrete time systems, optimization methods, computer simulation.*

1. Introduction

Reinforcement Learning (RL) algorithms [16] address the problem of learning to select optimal actions when limited feedback (usually in the form of a scalar reinforcement function) from the environment is available. General RL algorithms like Q-Learning [19], SARSA and TD(λ) [15] have been proved to converge to the globally optimal solution (under certain assumptions) [1, 19]. They are very flexible, because they do not require a model of the environment, and have been shown to be effective in solving a variety of RL tasks. This flexibility, however, comes at a certain cost: these RL algorithms require extremely long training to cope with large state space problems [17]. Even for a relatively simple control task such as the cart-pole balancing problem on a limited-length track, they require tens of thousands of steps [3].

Many different approaches have been proposed for speeding up the RL process. One possible technique is to use function approximation [8], in order to reduce the effect of the "curse of dimensionality". Unfortunately, using function approximation creates instability problems when used with off-policy learning [18]. For instance, Q-Learning [19], one of the most popular RL algorithms, is known to diverge when used with linear function approximation, even for very simple environments. This divergent behavior is generated, at least in part, by the off-policy nature of the learning algorithm. The key problem is that the policy used to generate behavior and the target policy visit states with different frequencies [8].

Significant speed-up can be achieved when a demonstration of the goal task is available [12], as in Apprenticeship Learning [24]. Although there is a risk of running dangerous exploration policies in real world [22], successful implementation of apprenticeship learning for aerobatic helicopter flight exists [23].

Another possible technique for speeding up RL is to use some form of hierarchical decomposition of the problem [4]. A prominent example is the "MAXQ Value Function Decomposition" [2]. Hybrid methods using both apprenticeship learning and hierarchical decomposition have been successfully applied to quadruped locomotion [13, 14]. Unfortunately, decomposition of the target task is not always possible, and sometimes it may impose additional burden on the users of the RL algorithm.

A state-of-the-art RL algorithm for efficient state space exploration is E3 [6]. It uses active exploration policy to visit states whose transition dynamics are still inaccurately modeled. Because of this, running E3 directly in the real world might lead to a dangerous exploration behavior.

Instead of using value-iteration-based RL algorithms, some researchers have focused on significantly different algorithms, namely, policy search RL algorithms [7]. Examples include the Natural Actor-Critic architecture [20], as well as the Policy Gradient RL algorithm, which has been applied successfully to robot control [11]. An alternative way to represent states and actions also exists, known as Relational Reinforcement Learning [21], which generalizes RL by relationally representing states and actions.

This paper explores a completely different approach for speeding up RL: more efficient use of computer simulations. Simulations have been commonly used instead of executing RL algorithms in the real world. This approach has two main advantages: speed and safety. Depending on its complexity, a simulation can run many times faster than a real-world experiment. Also, the time needed to set up and maintain a simulation experiment is far less compared to a real-world experiment. The second advantage, safety, is also very important, especially if the RL agent is a very expensive equipment (e.g., a fragile robot), or a dangerous one (e.g., a chemical plant).

Whether or not the full potential of computer simulations has been utilized for RL, however, is a different question. A recently proposed technique, called "Time Manipulation" [25], suggests that using backward time manipulations inside a simulation can significantly speed up the learning process and improve the state space exploration. Applied to failure-avoidance RL problems, such as the cart-pole balancing problem, Time Manipulation has been shown to increase the speed of convergence by 260% [26].

In the same line of research, this paper extends the paradigm of time manipulations and proposes a RL technique, called "Time Hopping", which can be successfully applied to continuous optimization problems. Unlike the original Time Manipulation technique, which can only perform backward time manipulation, the Time Hopping proposed can make arbitrary "hops" between states and traverse rapidly throughout the entire state space. Time Hopping extends the applicability of time manipulations to include not only failure-avoidance problems, but also continuous optimization problems, by creating new mechanisms to trigger the time manipulation events, to make prediction about the possible future rewards, and to select promising time hopping targets.

The next Section 2 introduces the concept of Time Hopping and explains how it can be applied to RL in general. Section 3 proposes a concrete implementation of Time Hopping technique for Q-learning algorithm. Section 4 presents the results from experimental evaluation of Time Hopping on a particular continuous-optimization problem: a biped crawling robot.

2. The concept of Time Hopping

A. The problem with decreasing learning rate

Reinforcement learning works very similar to the natural trial-and-error learning that we, humans, use in real world. Let us consider the following example: a person is trying to learn how to ski. Usually the first day he falls many times, learning a lot of crucial motor skills and advancing significantly in the task. The second day, he learns how to keep better his balance on the ski during slow motion. Gradually, he increases the speed and the difficulty of the terrain. At some point he reaches an adequate level of skiing skill. As time goes by, the learning rate slows down and eventually the person needs to train very long time in order to advance his skill just a little bit more. A typical learning curve is depicted on Fig. 1.

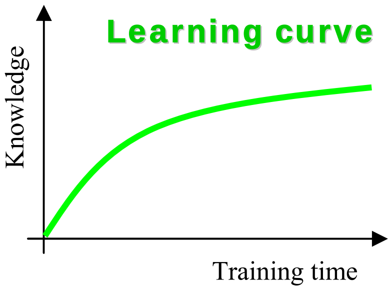

Fig. 1. A typical learning curve, showing the cumulative acquired knowledge (learned skill) with respect to the time spent for training. The learning rate is gradually decreasing, because the probability of experiencing unknown situations is becoming smaller

Exactly the same phenomenon is observed when a computer program uses reinforcement learning to acquire some new skill. It is important to understand why this is happening.

The explanation of this phenomenon is very simple: new skills can only be learned from new situations. As the skier becomes more experienced, he faces fewer new, unfamiliar situations/conditions of the environment. In other words, the probability of an unknown situation becomes so small, that the person has to ski for a very long time in order to find himself in a situation that allows him to learn something new.

Finding a way to prevent this phenomenon and keep the learning rate high throughout the entire training is a worthy objective. In real world probably very little can be done towards this objective. For a computer simulation, however, we propose one potential solution called "Time Hopping".

B. The concept of Time Hopping

Learning how to ski is essentially a continuous optimization problem. Let us consider a more formal definition of the same RL problem, given by Markov Decision Process (MDP) on Fig. 2. Each state transition has a probability associated with it. State 1 represents situations of the environment that are very common and quickly learned. The frequency with which state 1 is visited is the highest of all. As the state number increases, the probability of being in the corresponding state becomes lower. State 4 represents the rarest situations and therefore the most unlikely to be learned.

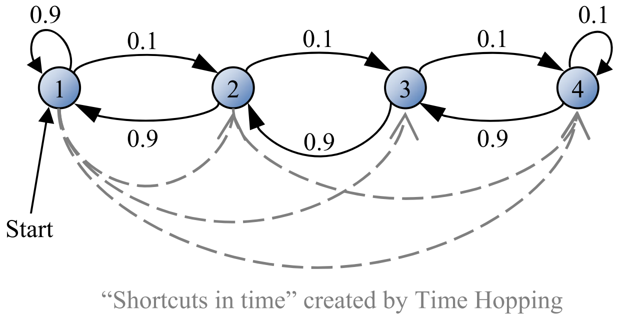

Fig. 2. An example of a MDP with uneven state probability distribution. Time Hopping can create "shortcuts in time" (shown with dashed lines) between otherwise distant states, i.e., states connected by a very low-probability path. This allows even the lowest-probability state 4 to be learned easily

The fundamental idea of Time Hoping is to provide "shortcuts in time" to such low-probability states, making them easier to learn, while at the same time avoiding unnecessary repetition of already well-explored states. This can be done by externally manipulating the computer simulation in a way which is completely transparent for the RL algorithm, as demonstrated in Section 3.

Time Hopping creates "shortcuts in time" by making direct hops between very distant states inside the MDP. Depending on how it is used, Time Hopping can potentially change the state probability distribution to, for example, an almost uniform distribution. In this way all the states can be visited (and therefore, learned) almost equally well. Fig. 3 shows what would be the effect of Time Hopping when applied to the same MDP from Fig. 2. In general, it is possible to do a complete probability redistribution of the MDP using Time Hopping, as shown in [27], but this falls out of the scope of this paper.

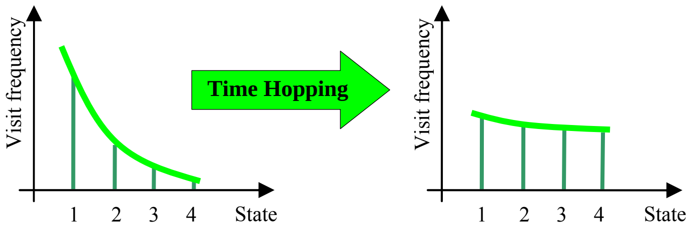

Fig. 3. Time Hopping can potentially change the state probability distribution to an almost uniform distribution. The four states shown correspond to the same four states from Fig. 2

The question we try to answer here is whether it is possible to efficiently implement Time Hopping for a continuous optimization RL problem. The following section defines what is required to be able to do this.

C. Components of Time Hopping

In order to accurately define Time Hopping in the context of RL, two things have to be specified: what components are necessary, and how they should interact with each other.

For failure-avoidance problems Time Manipulation technique [25] has proven to be very efficient. For continuous optimization problems, however, it can not be directly applied for various reasons. By analyzing these reasons we define the components required for successfully implementing Time Hopping for such problems.

Time Manipulation works by externally manipulating (modifying) the time within the simulation, in order to increase the speed of RL convergence. For failure-avoidance RL problems, such as the cart-pole balancing problem, the failure event provides a convenient trigger for a backward time manipulation. For continuous optimization problems, however, there are no such failure events and therefore the original Time Manipulation technique cannot be applied. A new trigger is needed for Time Hopping (Component #1 – *Hopping trigger*).

When the Time Hopping trigger is activated, a target state and time have to be selected, considering many relevant properties of the states, such as probability, number of times visited, level of exploration, connectivity to other states (number of state transitions), etc. In other words, a target selection strategy is needed (Component #2 – *Target selection*).

After a target state and time have been selected, hopping can be performed. It includes setting the RL agent and the simulation environment to the proper state, while preserving at the same time all the acquired knowledge by the agent (Component #3 – *Hopping*).

The flowchart on Fig. 4 shows how these three components of Time Hopping are connected and interact with RL algorithm. Now we can provide an accurate definition for the technique proposed.

D. Definition of Time Hopping

Definition: Time Hopping is an algorithmic technique which allows maintaining higher learning rate in a simulation environment by hopping to appropriately selected states.

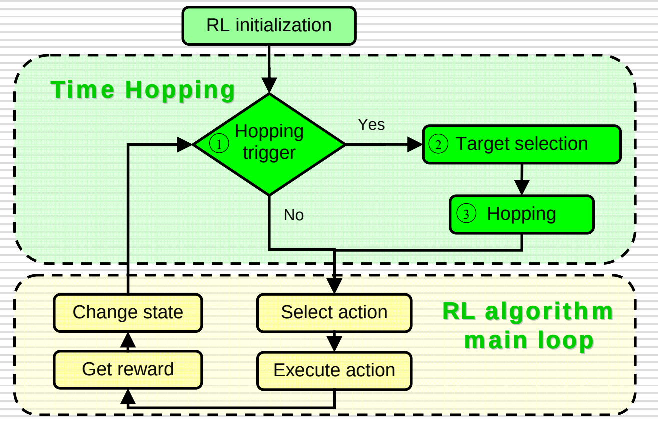

Fig. 4. Time Hopping technique applied to a conventional RL algorithm. The lower group (marked by a dashed line) contains the conventional RL algorithm main loop, into which the Time Hopping components (the upper group) are integrated

Appropriately selected states could be, for example, rarely experienced states, or states which are more promising for exploration. As a result of this state selection strategy, Time Hopping can potentially change the state probability distribution to, for example, an almost uniform distribution, or increase the visit probability of more promising states. This ability makes Time Hopping a tool for re-shaping the state probability distribution as desired.

When applied to a conventional RL algorithm, Time Hopping consists of three components:

1) Hopping trigger – decides when the hopping starts;
2) Target selection – decides where does it hop to;
3) Hopping – performs the actual hopping.

The components are connected according to Fig. 4 and integrated into a RL algorithm, allowing it to maintain higher learning rate throughout the entire training. Any proper implementation of Time Hopping for RL must provide concrete implementations of these three components. In Section 3 we propose one such possible implementation.

E. Convergence of Time Hopping

One very important question about Time Hopping is whether a RL algorithm using Time Hopping can converge or not. Conventional RL algorithms like TD(λ) and

SARSA [15] provably converge to the globally optimal solution [1, 19], but they both require gradual convergence of the exploration policy to the optimal policy. Such algorithms are known as on-policy RL algorithms. The value function that they learn is dependent on the policy that is being followed during the training. In the case of Time Hopping, however, it deliberately tries to avoid convergence of the policy in order to maintain high learning rate and minimize exploration redundancy. Therefore, if Time Hopping is used with such an on-policy RL algorithm, it is obviously not going to converge.

To ensure that Time Hopping converges, an *off-policy* RL algorithm must be used. One example of such algorithm is Q-Learning. The fundamental difference between Q-Learning and the previous two algorithms is that the learned policy is independent on the policy followed during learning. This makes Q-Learning much more suitable for Time Hopping.

Very often a good policy for the task we are attempting to learn is not known. Using an on-policy algorithm with an inappropriate training policy might cause the system not to learn the optimal policy. Using an off-policy algorithm, such as Q-Learning, frees us from worrying about the quality of the policy that the system follows during the training. In fact, Q-Learning even works when random training policies are used.

As a result, the convergence of Time Hopping is guaranteed by using an off-policy RL algorithm, regardless of the target selection policy, provided that it preserves the ergodicity of the underlying MDP (in order to guarantee sufficient exploration).

3. Implementation of Time Hopping technique

One possible implementation of Time Hopping is suggested in this section, by proposing concrete implementations for each of its three components (as defined in Section 2. D). Table 1 lists the proposed implementation for each component.

Table 1. Proposed implementation of each Time Hopping component

No of component	Component name	Proposed implementation
1	Hopping trigger	“Gamma pruning”
2	Target selection	“Lasso target selection”
3	Hopping	“Basic Hopping”

In the implementation proposed, Q-Learning is used as a representative off-policy reinforcement learning algorithm, in order to guarantee the convergence (as explained in Section 2. E).

A. Gamma pruning

What we would like to call “Gamma pruning” is an implementation of the Hopping trigger component of Time Hopping. By definition, a Hopping trigger component decides when to interrupt the current sequential exploration of RL algorithm and initiate a Time Hopping step. In the proposed Gamma pruning, this is done by predicting unpromising branches of exploration and triggering a Time Hopping step to avoid them. We call it *pruning*, because the idea is similar to Alpha-beta *pruning*, used in Mini-max algorithm. And we call it *Gamma* pruning, because the pruning is based on the γ discount factor.

The basic idea of Gamma pruning can be illustrated with the example on Fig. 5. Let us assume that the current best policy follows the transition ⟨state 1 → state T⟩, and the RL algorithm decides to explore a new transition ⟨state 1 → state 2⟩ to see whether it can achieve a bigger reward than the current best. After a few exploratory transitions, we can try to predict the "best-case scenario" (i.e., the biggest possible cumulative reward) if we continue this exploratory path. If even the best-case prognosis for the future path is not good enough to override the current best policy at state 1, then this exploratory path is unpromising and we can perform pruning here (i.e., activate the Hopping trigger to perform a Time Hopping step, leaving this unpromising exploratory path). In the particular example this is done after reaching state 3.

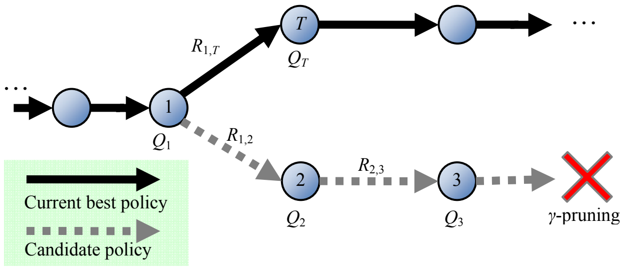

Fig. 5. An example of how Gamma-pruning works. After branching away from the current best policy (at state 1), the algorithm makes two state transitions, receives rewards $R_{1,2}$ and $R_{2,3}$, compares them with the calculated threshold and decides that it is unpromising to continue, so it performs pruning

The Q-value of (making) a state transition ⟨state i → state j⟩ is defined as follows:

$$Q_{i,j} = R_{i,j} + \gamma \max_s \left\{ Q_{j,s} \right\}, \tag{1}$$

where $R_{i,j}$ is the reward received from this state transition. For convenience, let us denote:

$$Q_i = \max_s \left\{ Q_{i,s} \right\}. \tag{2}$$

In the example shown in Fig. 5, state 1 and state T are part of the current best policy. In order for the candidate state 2 to become a part of a new best policy, the Q-value of the transition ⟨state 1 → state 2⟩ must become bigger than the current biggest Q-value: the Q-value of the transition ⟨state 1 → state T⟩. Therefore, state 2 can override the current best policy if and only if: $Q_{1,2} > Q_{1,T}$, which can be rewritten using (1) and (2) as

(3) $$R_{1,2} + \gamma Q_2 > R_{1,T} + \gamma Q_T.$$

The RHS of inequality (3) is the minimum value that has to be surpassed, in order to change the best policy at state 1. The necessary *Q*-value of state 2 must satisfy:

(4) $$Q_2 > \frac{R_{1,T} - R_{1,2}}{\gamma} + Q_T.$$

We can do this safely because $0 < \gamma < 1$. The RHS of inequality (4) shows the minimum *Q*-value of state 2 that has to be surpassed, in order to change the best policy from ⟨state 1 → state T⟩ to ⟨state 1 → state 2⟩. Let us call this value a *threshold* value and define it in the following way: a threshold $T_{s,t}$ is the minimum *Q*-value for state *t* that has to be surpassed, in order to change the current best policy at state *s* and make it pass through state *t*. For convenience, we assume that $T_{s,t} = Q_s$ when $s = t$.

For the particular example in Fig. 5

(5) $$T_{1,1} = Q_1 = \max_s \left\{ Q_{1,s} \right\} = R_{1,T} + \gamma Q_T.$$

Using the RHS of inequality (4) for state 2, the following equation for the threshold $T_{1,2}$ can be derived:

(6) $$T_{1,2} = \frac{R_{1,T} + \gamma Q_T - R_{1,2}}{\gamma} = \frac{T_{1,1} - R_{1,2}}{\gamma}.$$

In the same way the threshold after the next state transition ⟨state 2 → state 3⟩ can be calculated as

(7) $$T_{1,3} = \frac{T_{1,2} - R_{2,3}}{\gamma}.$$

This recursive formula can be generalized for the threshold after state transition ⟨state *n* – 1 → state *n*⟩ as

(8) $$T_{1,n} = \frac{T_{1,n-1} - R_{n-1,n}}{\gamma}.$$

Now that we have an efficient way to calculate the trigger thresholds, we need a way of *predicting future rewards* for the current state. One possible solution is to assume the best-case scenario: that all future rewards will be equal to the maximum possible reward $R_{\max}$. This prediction is actually plausible, since there might be a loop of states in which each transition has the highest possible reward $R_{\max}$, as illustrated on Fig. 6.

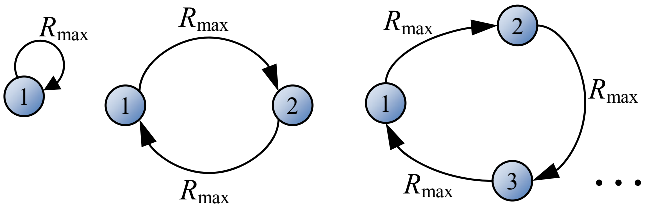

Fig. 6. The maximum-reward cycles with length 1, 2, 3 and etc., used for predicting the best-case future cumulative reward starting from a given state. The prediction assumes that it is possible to reach such a cycle in one step and calculates the maximum possible cumulative reward, which is a constant value regardless of the cycle length

There are infinitely many such loops possible but all of them have the same *Q*-value of the states, which can be calculated as

$$R_{\max} + \gamma\left(R_{\max} + \gamma\left(\ldots\right)\right) = R_{\max}\left(1 + \gamma + \gamma^2 + \ldots\right) = \frac{R_{\max}}{1-\gamma}. \tag{9}$$

According to (8), after every exploratory transition the threshold value increases (because $0 < \gamma < 1$). The value of the best-case prediction, however, remains fixed, as determined by (9). Therefore, at some point the threshold value will surpass the value of the best-case prediction. This means that at that point even the best-case scenario is not sufficient to change the current best policy. This is exactly the right moment to do Gamma-pruning and activate the Hopping trigger.

B. Lasso target selection

What we would like to call "Lasso target selection" is an implementation of component #2 (Target selection) of Time Hopping. The objective of this implementation is to construct a "lasso", which is a sequence of state transitions following the current best policy, starting at the initial state and ending when a cycle is detected. Fig. 7 illustrates why the name "lasso" is appropriate for such a construction.

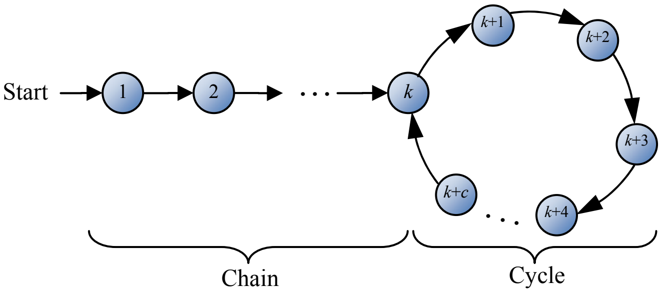

Fig. 7. A lasso in the state space, starting with a chain of states from the initial state and ending with a cycle of states

The procedure for constructing the lasso follows these steps:

Step 1. Start from the initial state and add it to the lasso.

Step 2. Select the action which has yielded the maximum *Q*-value for the current state and follow the corresponding state transition. If there is more than one action equal to the maximum, then select one randomly among them.

Step 3. If the new state is not a member of the lasso, then add it to the lasso and go to Step (2).

Step 4. Stop.

This procedure is guaranteed to finish, because there is only a finite number of states. It is important to note that the element of randomness in Step 2 is necessary for balanced exploration, especially during the early stages of the learning process, when the states have many equally-valued actions.

The so constructed lasso represents the current best policy starting from the initial state. Moreover, the yet unknown globally optimal policy is guaranteed to have a common subsequence with the lasso, starting from the initial state and following the lasso up to a certain state. In any case, at least the first (initial) state is always shared between both of them. Therefore, if we limit the targets for Time Hopping only to the states belonging to the current lasso, there is no risk of missing the globally optimal solution.

The Lasso target selection is doing exactly this: it re-calculates the current lasso every time when a change occurs in the current best policy, and selects a Hopping target among the states in the lasso. This forces the RL algorithm to better explore the states in proximity to the current best policy, which are in fact the most probable states to be part of the globally optimal policy.

Additionally, the Lasso target selection performs certain “load-balancing” of the states on the lasso by giving preference to those which are less explored. This is done by keeping the number of times each state was visited so far.

C. Basic Hopping

The implementation of the Hopping component is rather straightforward. After the Target selection has selected a specific state, it is necessary to set the state of the RL agent and the state of the environment to the corresponding target state. This is easy to do in a computer simulation, and the only concern is to have enough memory to store the representations of all states.

While doing this, care should be taken to preserve all the acquired knowledge so far by the RL algorithm. Additionally, all the threshold values for the Gamma pruning component have to be reset to reflect the new active state. After the Hopping is performed, the RL algorithm takes back control and continues executing its main loop. If the implementation of Time Hopping is proper, it is completely transparent for the RL algorithm and does not require any modification to it.

D. Eligibility propagation

Eligibility traces are one of the basic mechanisms for temporal credit assignment in reinforcement learning [16]. An eligibility trace is a temporary record of the occurrence of an event, such as visiting of a state or taking of an action. When a learning update occurs, the eligibility trace is used to assign a credit or blame for the received reward to the most appropriate states or actions.

Eligibility traces are usually easy to implement for conventional RL methods. However, in the case of Time Hopping, due to its non-sequential nature, it is not trivial to do so. Since arbitrary hops between states are allowed, it is impossible to directly apply the conventional (linear) eligibility traces. Instead, a different mechanism must be used, such as *Eligibility Propagation* [28].

Eligibility Propagation provides for Time Hopping similar abilities to what eligibility traces provide for conventional RL, except that it uses a state transitions graph to propagate values from one state to all of its temporal predecessors. The constructed oriented graph represents the state transitions with their associated actions and rewards and uses this data to propagate the learning updates. Because of the way Time Hopping works, this graph might be disconnected, consisting of many separate connected components. Using the transitions graph to obtain all predecessor states of an updated state allows the propagation to flow logically backwards in time.

4. Application of Time Hopping to a biped crawling robot

In order to evaluate the efficiency of the proposed Time Hopping technique in a continuous optimization problem, experiments on a biped crawling robot are conducted. The goal of the learning process is to find a crawling motion with the maximum speed. The reward function for this task is defined as the horizontal displacement of the robot after every action.

A. Description of the crawling robot

The crawling robot has two limbs, each one with two segments, for a total of four Degrees Of Freedom (DOF). Every DOF is independent from the rest and has three possible actions at each time step: to move clockwise, to move anti-clockwise, or to stand still. Fig. 8 shows a typical crawling sequence of the robot as visualized in the simulation environment constructed for this task.

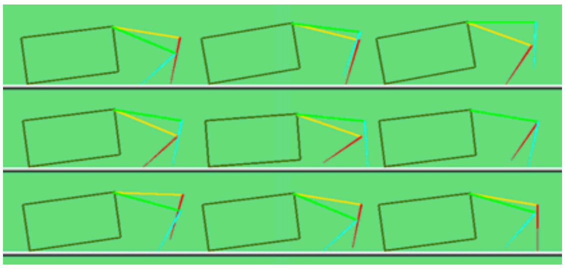

Fig. 8. A crawling robot with two limbs, each with 2 segments for a total of 4 DOF. Nine different states of the crawling robot are shown in a normal crawling sequence

When all possible actions of each DOF of the robot are combined, assuming that they can all move at the same time independently, it produces an action space with size $3^4 - 1 = 80$ (we exclude the possibility that all DOF are standing still). Using appropriate discretization for the joint angles (9 for the upper limbs and 13 for the lower limbs), the state space becomes divided into $(9\times13)^2 = 13\ 689$ states. For better analysis of the crawling motion, each limb has been colored differently and only the "skeleton" of the robot is displayed.

B. Description of the experimental method

The conducted experiments are divided in two groups: experiments using a conventional RL algorithm (conventional Q-Learning, as described in [26]) and experiments using the same algorithm modified with the Time Hopping technique. The experiments from both groups are conducted in exactly the same way, using exactly the same RL parameters (incl. discount factor γ, learning rate α, and the action selection method parameters). First, the conventional RL algorithm is used with a set of fixed algorithm parameters. After that, the Time Hopping technique is activated and the same set of parameters is used in exactly the same simulation environment starting from the same initial state. The robot training continues up to a fixed number of steps (45 000), and the achieved crawling speed is recorded at fixed checkpoints during the training. This process is repeated at least 10 times and the results are averaged, in order to ensure statistical significance.

C. Evaluation of Time Hopping

The conventional RL algorithm and the Time Hopping version of it are compared based on the best solution found (i.e., the fastest crawling sequence achieved) for the same fixed number of training steps. The comparison results are shown in Fig. 9. The achieved speed (as the amount of training steps increases) is displayed as percentage of the globally optimal solution.

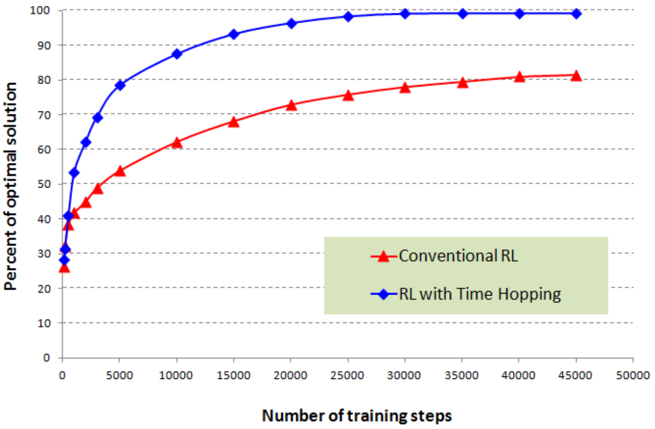

Fig. 9. Comparison of conventional RL and Time Hopping, based on the best solution achieved relative to the number of training steps. The fitness of the solutions is measured as a percentage from the optimal solution, i.e., the fastest possible crawling speed of the robot. The best solution achieved at each checkpoint is found following the current best policy at that point

Time Hopping achieves significant speed-up of the learning process. For example, it learns an 80%-optimal crawl in only 5 000 steps, while the conventional RL algorithm needs 35 000 steps to learn the same, i.e., in this case Time Hopping is seven times faster. The speed-up becomes even higher as the number of training steps increases. For example, Time Hopping reaches 90%-optimal solution with less than 15000 steps, while the conventional RL needs more than 50 000 steps to do the same.

The main reason for these results is that the conventional RL algorithm spends many steps exploring broadly (and quite randomly) the state space, regardless of whether such exploration is promising or not. Time Hopping, on the other hand, detects as early as possible unpromising branches using the proposed Gamma-pruning and avoids unnecessary exploration.

In addition, the Lasso target selection focuses the exploration on the most probable candidates for the best policy and thus provides a more purposeful exploration than the conventional RL. The cumulative effect of Gamma-pruning and Lasso target selection is shown in Fig. 10. It shows the sorted sequence of maximum Q-values of all explored states for the same number of steps (30 000) by the conventional RL algorithm and Time Hopping. The bigger steepness of the Time Hopping curve means bigger Q-values achieved with fewer explored states, i.e., more efficient exploration.

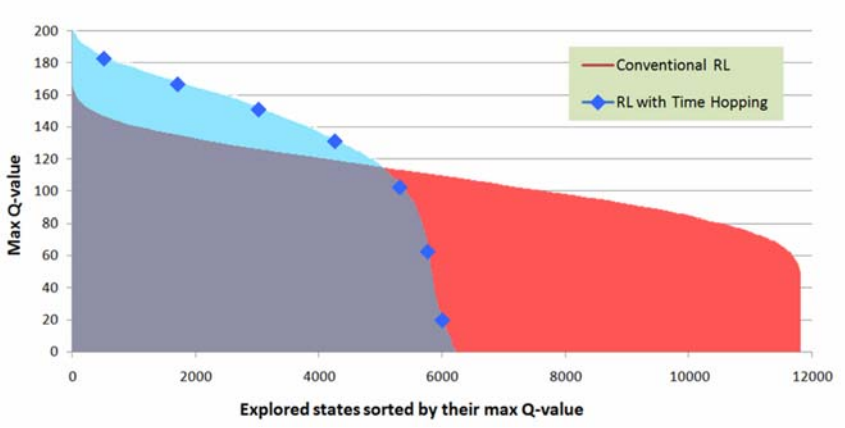

Fig. 10. Comparison of conventional RL and Time Hopping using the sorted sequence of maximum Q-values of all explored states after 30 000 steps of training. The steepness of the curves is used as an indication of the exploration efficiency. A steeper curve means better ability to find high-valued policies with fewer explored states

D. Evaluation of Gamma pruning

In order to evaluate the effectiveness of the proposed implementation for the Hopping trigger component (Gamma-pruning), a comparison with a different trigger is necessary.

For this purpose, a trigger called “Fixed trigger” was created. It is probably the simplest possible implementation of a Hopping trigger, because it is based on a single fixed parameter: number of steps N_s. After every N_s consecutive steps of the

RL algorithm, the Fixed trigger initiates one hopping step. The value of N_s is constant throughout the training, which means that the total number of hopping trigger activations is proportional to the number of training steps.

The smaller the value of N_s, the more trigger activations will be performed. There is a certain threshold, however, which cannot be exceeded: if the chosen value of N_s is smaller than the length of the optimal solution, the Fixed trigger would prevent the RL algorithm from reaching it.

Fig. 11 compares the cumulative number of hopping trigger activations for the two triggers: Gamma-pruning trigger and the best possible Fixed trigger. It clearly indicates that Gamma-pruning makes increasingly more trigger activations as the number of time steps increases. After a certain threshold (around 10 000 steps) the effectiveness of Gamma-pruning increases significantly, due to the fact that the maximum Q-values of the states have increased enough to enable early pruning of the exploratory sequences. This means that unpromising state space areas are pruned effectively, and explains why the proposed Time Hopping implementation performs better than the conventional RL.

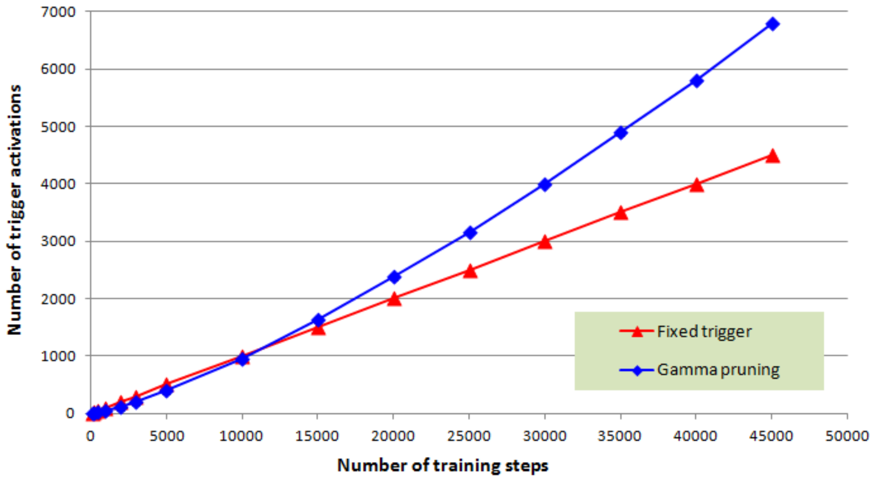

Fig. 11. Comparing a Gamma-pruning trigger with a Fixed trigger for Time Hopping, based on the cumulative number of trigger activations relative to the number of training steps. The Fixed trigger uses $N_s = 9$, which means that 1 out of every 10 consecutive transitions is a hopping step. This is the best that the Fixed trigger can do, because any value below 9 would cause failure to reach the optimal solution (which has a length of at least 9 steps)

E. Evaluation of Lasso target selection

In this experiment the effectiveness of the proposed implementation for the Hopping target selection component (Lasso target selection) is compared to a different target selection method. For comparison, a “Random target selection” method was created, which selects a target state randomly among all the currently known states. Fig. 12 compares the maximum Q-value achieved by both target selection methods as the number of training steps increases.

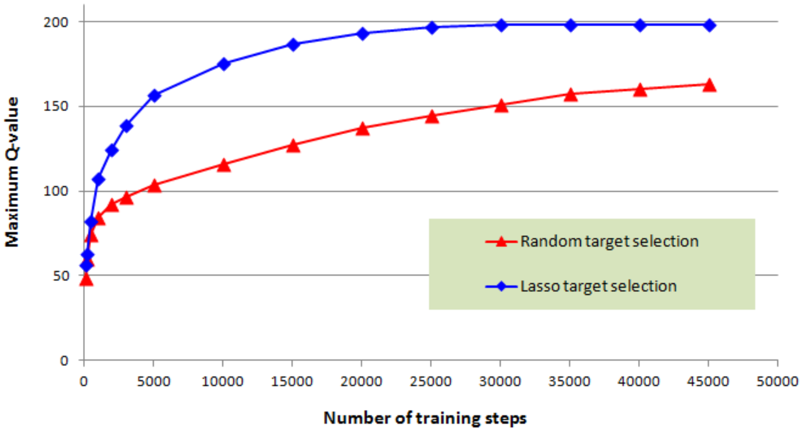

Fig. 12. Comparing Lasso target selection with Random target selection for Time Hopping. The maximum *Q*-value achieved relative to the number of training steps is used as an indicator of the ability of each selection method to discover early high-valued policies

The Lasso target selection performs significantly better because it manages to focus the exploration efforts on the most promising parts of the state space, while the Random target selection distributes the exploration uniformly throughout the state space. It is worth noting that for a different RL task in which uniform exploration is the goal, the Random target selection method might be a good candidate. For the robot crawling task, however, the Lasso target selection is more appropriate.

5. Conclusion

The general concept of Time Hopping is proposed and a concrete implementation for continuous optimization RL problems is developed.

The conducted experiments on a biped crawling robot show significant increase in the speed of learning when Time Hopping technique is used. This is due to the ability of Time Hopping to make direct hops between otherwise distant states, thus changing the state probability distribution favorably. The proposed Gamma-pruning trigger and Lasso target selection additionally boost the learning performance by predicting and avoiding unpromising branches of exploration, and also by selecting appropriate hopping targets.

An important advantage of the proposed implementation of Time Hopping is that no parameter tuning or manual adjustments are necessary during learning, which makes the technique easy to use.

The clear separation of the three well-defined components makes it straightforward to experiment with alternative component implementations.

Another strong point of Time Hopping is its complete transparency for the RL algorithm, which means that no modifications to RL algorithm are necessary, other than inserting Time Hopping as a part of the main RL loop. This offers future perspectives on combining Time Hopping with other approaches for speeding up the learning process.

The generality of the Time Hopping concept and the complete transparency of its implementation make another application feasible: Time Hopping as a tool for re-shaping the state probability distribution as desired.

Finally, an important drawback of the proposed technique is that it can only be used in a simulation, not directly applied in real world. Yet, it can be used as a part of the off-line computation of a real-world system.

Acknowledgments: This work was supported in part by the Japanese Ministry of Education, Culture, Sports, Science and Technology (MEXT).

6. References

1. Dayan, P., T. J. Sejnowski. TD(λ) Converges with Probability 1. – Mach. Learn., Vol. **14**, 1994, No 3, 295-301.
2. Dietterich, T. G. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. – Journal Artif. Intell. Res., Vol. **13**, 2000, 227-303.
3. Geva, S., J. Sitte. A Cart-Pole Experiment Benchmark for Trainable Controllers. – IEEE Control Systems Magazine, Vol. **13**, 1993, 40-51.
4. Barto, A., S. Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. – Discrete Event Dynamic Systems, Vol. **13**, 2003, 341-379.
5. Humphrys, M. Action Selection Methods Using Reinforcement Learning. Ph.D. Thesis, University of Cambridge, June 1997.
6. Kearns, M., S. Singh. Near-Optimal Reinforcement Learning in Polynomial Time. Machine Learning, 2002.
7. Peshkin, L. Reinforcement Learning by Policy Search. PhD Thesis, MIT, November 2001.
8. Precup, D., R. S. Sutton, S. Dasgupta. Off-Policy Temporal-Difference Learning with Function Approximation. – In: Proc. of the Eighteenth Conference on Machine Learning (ICML 2001), M. Kaufmann, Ed., 2001, 417-424.
9. Price, B., C. Boutilier. Accelerating Reinforcement Learning through Implicit Imitation. – Journal of Artificial Intelligence Research, Vol. **19**, 2003, 569-629.
10. Kaelbling, L. P., L. M. Littman, A. W. Moore. Reinforcement Learning: A Survey. – Journal Artif. Intell. Res., Vol. **4**, 1996, 237-285.
11. Kohl, N., P. Stone. Policy Gradient Reinforcement Learning for Fast Quadrupedal Locomotion. – In: Proc. of the IEEE International Conference on Robotics and Automation (ICRA 2004), New Orleans, LA, May 2004, 2619-2624.
12. Coates, A., P. Abbeel, A. Ng. Learning for Control from Multiple Demonstrations. – ICML, Vol. **25**, 2008.
13. Kolter, J., P. Abbeel, A. Ng. Hierarchical Apprenticeship Learning, with Application to Quadruped Locomotion. – Neural Information Processing Systems, Vol. **20**, 2007.
14. Kolter, J., M. Rodgers, A. Ng. A Control Architecture for Quadruped Locomotion Over Rough Terrain. – IEEE International Conference on Robotics and Automation, 2008.
15. Sutton, R. S. Learning to Predict by the Methods of Temporal Difference. – Mach. Learn., Vol. **3**, 1988, 9-44.
16. Sutton, R. S., A. G. Barto. Reinforcement Learning: An Introduction. Cambridge, MA, MIT Press, 1998.
17. Thrun, S. B. Efficient Exploration in Reinforcement Learning. Technical Report CMU-CS-92-102, Carnegie Mellon University, Pittsburgh, PA 15213, 1992.
18. Thrun, S., A. Schwartz. Issues in Using Function Approximation for Reinforcement Learning. – In: Proc. of the Fourth Connectionist Models Summer School, 1993.
19. Watkins, C. J. C. H., P. Dayan. Q-Learning. – Mach. Learn., Vol. **8**, 1992, 279-292.
20. Peters, J., S. Schaal. Natural Actor-Critic. – Neurocomputing, Vol. **71**, 2008, Issues 7-9, 1180-1190.

21. Tadepalli, P., R. Givan, K. Driessens. Relational Reinforcement Learning: An Overview. – In: ICML-2004 Workshop on Relational Reinforcement Learning, 2004.

22. Abbeel, P., A. Ng. Exploration and Apprenticeship Learning in Reinforcement Learning. – ICML, 2005.

23. Abbeel, P., A. Coates, M. Quigley, A. Ng. An Application of Reinforcement Learning to Aerobatic Helicopter Flight. – NIPS, Vol. **19**, 2007.

24. Ng, A. Reinforcement Learning and Apprenticeship Learning for Robotic Control. – In: Lecture Notes in Computer Science, Vol. **4264**, 2006, 29-31.

25. Kormushev, P., K. Nomoto, F. Dong, K. Hirota. Time Manipulation Technique for Speeding up Reinforcement Learning in Simulations. – International Journal of Cybernetics and Information Technologies, Vol. **8**, 2008, No 1, 12-24.

26. Kormushev, P. Time Hopping Technique for Reinforcement Learning and its Application to Robot Control. PhD Thesis, Department of Computational Intelligence and Systems Science, Tokyo Institute of Technology, September 2009.

27. Kormushev, P., F. Dong, K. Hirota. Probability Redistribution Using Time Hopping for Reinforcement Learning. – In: 10-th International Symposium on Advanced Intelligent Systems ISIS-2009, 2009.

28. Kormushev, P., K. Nomoto, F. Dong, K. Hirota. Eligibility Propagation to Speed up Time Hopping for Reinforcement Learning. – Journal of Advanced Computational Intelligence and Intelligent Informatics, Vol. **13**, 2009, No 6.