A Universal Generalization for Temporal-Difference Learning Using Haar Basis Functions

Susumu Katayama KATAYAMA@FE.DIS.TITECH.AC.JP
Hajime Kimura GEN@FE.DIS.TITECH.AC.JP
Shigenobu Kobayashi KOBAYASI@DIS.TITECH.AC.JP
Department of Computational Intelligence and Systems Science, Tokyo Institute of Technology, 4259 Nagatsuta-cho, Midori-ku, Yokohama 226-8502, Japan

Abstract

We propose an algorithm efficiently implementing TD(λ) using (the infinite tree of) Haar basis functions. The algorithm can maintain and update the information of the infinite tree of coefficients in its finitely compressed form by taking advantage of the fact that the information obtained from finite training data is finite. Our algorithm computes the whole updating at each time step in time linear in the precision (measured by the number of bits) of each observation. The system of Haar basis functions includes both broad features, which have strong generalization and averaging ability, and narrow features, which have high precision approximation ability. Especially, since it can approximate arbitrary continuous functions on $[0, 1)$ in the limit, TD(λ) for Haar basis functions obtains the best solutions for all problems to obtain value functions on $[0, 1)$, apart from the possibility it may be slower to converge than other methods tuned with labor. The universality in this sense is precious because the main application of TD(λ) is reinforcement learning, where the environment is unknown. Although the only concern of our method is that the space complexity increases linearly in the progress in time steps, experimental results show that it yields no problem provided that it adopts an appropriate forgetting strategy.

1. Introduction

Temporal-difference (TD) learning (Sutton, 1988) is an adaptive prediction method based on dynamic programming. By using TD, we can empirically obtain the value for each state (or the expectation of the discounted sum of the rewards obtained after the state) from time sequences of states and rewards, without knowledge about the state transition and the reward distribution after each state. Many popular reinforcement learning algorithms including Q-learning (Watkins & Dayan, 1992), Sarsa (Sutton, 1996) and actor-critic (Barto, Sutton, & Anderson, 1983) are based on TD.

In cases of applying TD to problems with large state spaces, we have to generalize over the state space in order to make the time and space complexities per time step and the convergence rate practical. We can probably say most generalization methods are based on function approximation. As a special form of TD combined with function approximation, a linear TD(λ) that uses a linear function approximator is proved to converge with probability one under some conditions (Tsitsiklis & Van Roy, 1997).

On the other hand, the complexity of a naively implemented function approximator increases linearly with the number of the basis functions. For this reason it is impossible to implement function approximator using an infinite number of basis functions naively.

When we restrict the number of the basis functions, however, we cannot estimate the generalization error bound in advance if we do not know how the adopted function model fits the problem. Therefore, if the precision of approximation needed to solve the problem is unknown, it is difficult to select the basis functions adequately. Though we can select a function approximator from some sets of approximators by some information criterion, on-line implementation of this approach is difficult and it needs appropriate selection of the sets of functions in order to avoid the increase in the number of basis functions.

In this paper we propose an algorithm implementing the updating rule of linear TD(λ) using Haar basis functions (e.g., Chui, 1992). While conventional function approximators like neural nets (e.g., Lin & Mitchell, 1992) use finite subset of infinite basis functions, our approach uses the complete infinite of basis

functions. Although this may seem impossible because one must maintain information about an infinite set of coefficients, our algorithm can maintain and update it in its finitely compressed form because the information obtained from a finite training set is finite. Seen from the outside, our algorithm behaves as if it holds infinite number of coefficients.

The system of Haar basis functions includes both broad features, which have strong generalization and averaging ability, and narrow features, which have high precision approximation ability. Especially, the Haar expansion of each continuous function f on $[0,1)$ uniformly converges to f. Therefore, apart from the convergence rate our algorithm can universally be applied independently of the shape of the target functions. This means we no longer need to investigate appropriate basis functions by hand. This feature is precious because one objective of reinforcement learning algorithms is to reduce the burdens of developers.

Our algorithm implements the updating rule of TD(λ) using Haar basis functions in

- time linear in the precision of the observation, and
- space linear in the number of states visited until the time step.

The algorithm presented in this paper exploits a method that implements tabular TD(λ) updating in time logarithmic in the number of states (Katayama & Kobayashi, 1999).

One approach related to our own is memory-based function approximation (Santamaría, Sutton, & Ram, 1998). We can regard this as a combination of locally weighted learning (Atkeson, Moore, & Schaal, 1997) and TD(λ) — it computes TD(λ) assuming that there are radial basis functions (RBF) at all states visited until the time step.[1]

Santamaría et al.'s method has two common features with our algorithm:

- assuming that the memory space is supplied inexhaustibly, the approximation converges to the target function; and
- space complexity is linear in the number of states that has been visited.

However, its time complexity increases linearly in the number of states that has been visited. In order to cope with this problem, Santamaría et al. localize each RBF by assuming the function values relevant to the points far away from the center of the RBF to be zero. The increase in the number of bases causes not only high precision of the approximated value function but also a trade-off between the time complexity and the generalization power of broad features.

Methods for generalization by adaptive resolution (Chapman & Kaelbling, 1991; McCallum, 1995) also have some common features with our algorithm. Instead of using a hierarchical system of basis functions, they discriminate between and generalize over the states by state resolution and unification based on some strategies. The convergence of learning methods combined with the strategies is not assured. On the other hand, we can apply the knowledge about TD(λ) to our algorithm because it just computes linear TD(λ) using specific basis functions. For this reason, assuming a computer model with inexhaustibly supplied memory, the function approximated by our algorithm is theoretically assured to converge to the target function.

2. Linear TD(λ) using Haar Basis Functions

This section describes the specification of our algorithm. Subsection 2.1 describes the definition and the features of linear TD(λ). Subsection 2.2 describes the definition and the features of Haar basis functions including the features in case we apply our algorithm. Subsection 2.3 works out the detail of the specification of linear TD(λ) using Haar basis functions.

2.1 Linear TD(λ)

TD(λ) is a learning method for prediction of future reward in unknown Markov processes. It is the most popular learning method applied to value estimation in reinforcement learning, or learning to control by trial and error in unknown environments.

TD(λ) approximates the value function, or the expectation of discounted sum of rewards $V^*(y) = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{i+1} \middle| y_1 = y\right]$ where y_t and r_t are the state and the reward at time t respectively and $\gamma \in (0,1)$ is discount factor. Linear TD(λ) uses a linear function approximator $V(y) = \sum_{i=0}^{\infty} k(i)\psi_i(y)$ where ψ_i are basis functions. It obtains parameters k_i asymptotically as

$$k_t(i) \quad = \quad k_{t-1}(i) + \alpha_t \delta_t D_t(i) \tag{1}$$

where TD error δ_t is defined as $\delta_t = r_t + \gamma V_{t-1}(y_t) - V_{t-1}(y_{t-1})$. *The eligibility trace* D is a mechanism for assigning the more TD errors for the more recent states. *The accumulating eligibility trace* is defined as

$$D_t(i) \quad = \quad \psi_i(y_t) + \lambda\gamma D_{t-1}(i) \tag{2}$$

If $\forall y.\psi_i(y) \in \{0,1\}$, we can define *replacing eligibility trace* (Singh & Sutton, 1996) as

$$D_t(i) = \begin{cases} 1, & \text{if } \psi_i(y_t) = 1; \\ \lambda\gamma D_{t-1}(i), & \text{o.w.} \end{cases} \tag{3}$$

Accumulating traces are more conventional, while replacing traces can be quicker to converge.

[1] To be exact, it computes Sarsa(λ), which is an extension of TD(λ) to control.

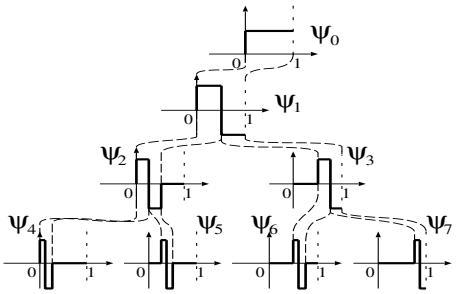

Figure 1. Haar basis functions. The system of Haar basis functions includes both broad features, which have strong generalization and averaging ability, and narrow features, which have high precision approximation ability.

Under some conditions on-line linear TD(λ) is proved to make V_t converge to near V^*, reducing the on-line error, i.e., the generalization error $\|V_\infty - V^*\|$ measured by the on-line distribution. The asymptotic generalization error has an upper bound linear in the minimal error that can be accomplished by the linear function model (Tsitsiklis et al., 1997). Therefore, a function approximator that can express all the continuous functions should converge to the target function at almost every state within the support of the on-line distribution function.

Tabular TD(λ) is a special case of linear TD(λ) learning that has $\psi_i(y) = \delta_i^y$ where δ is Kronecker's δ. It updates $V(y)$ $(= k(y))$ and D(y) directly. In Section 3 we first explain an efficient implementation of tabular TD(λ) (Katayama et al., 1999) before discussing the implementation of the target linear TD(λ).

2.2 Haar Basis Functions

We adopt the system of Haar basis functions (Figure 1), which has the following features:

precision It can express all the continuous functions on $[0, 1)$. (Our algorithm can implement TD(λ) for the infinite tree of Haar basis functions by compressing the information of the infinite tree of coefficients.)

generalization When the function value for some state improves, the values for all the states change toward the same direction. Near states can change greatly.

complexity Our algorithmic device saves the time complexity to the extent of the order of the precision of state observation and the space complexity the order of the number of states visited.

Figure 2 depicts an example of function approximation using Haar basis functions. It depicts the generalization ability at the beginning of learning and the averaging ability at the end of learning.

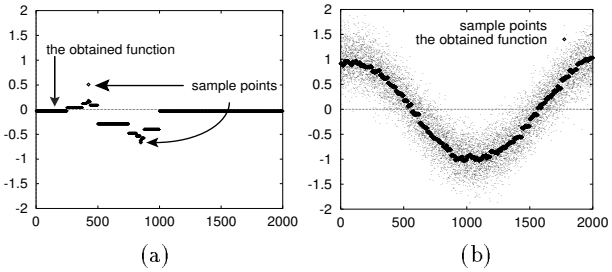

Figure 2. Example of function approximation using Haar basis functions. (a) The beginning of learning ($t = 2$). The sample points from the training set draw in the approximated function and generalization takes place. (b) The end of learning ($t = 10000$). The approximated function converges, averaging the sample points.

2.3 Application of TD(λ) to Haar Basis Functions

When updating $k_t(i)$, we should be careful to keep $V_t(y)$ finite for each y because it is an infinite series.

Our approach to this problem here is just discounting the learning rates α_t with the depth, namely,

$$k_t(i) = k_{t-1}(i) + \alpha_t(i)\delta_t D_t(i) \quad (4)$$

$$\alpha_t(i) = \alpha_t(1-\beta)\beta^{\lceil \log_2(i+1) \rceil} \quad (5)$$

where $0 < \alpha_t \leq 1$ and $0 < \beta < 1$. Discounting them with constant β makes the infinite series $\sum_{i=0}^{\infty} \alpha_t(i)$ (and hence the update of V_t at each t) finite.

Large β makes the learner sensitive to undulation and noise, while small β improves the averaging ability. We call β bias/variance parameter. Note, however, that even if β is inappropriately selected it only affects the convergence rate and the approximation converges to the true value function as long as α is scheduled appropriately.

Let $b_i = \sqrt{(1-\beta)\beta^{\lceil \log_2(i+1) \rceil}}$. By adopting $\tilde{\psi}_i(y) = b_i\psi_i(y)$, $\tilde{D}_t(i) = b_i D_t(i)$, and $\tilde{k}_t(i) = k_t(i)/b_i$ instead of $\psi_i(y)$, $D_t(i)$, and $k_t(i)$ respectively, we can see (2)/(3), (4), and (5) form a conventional linear TD(λ) with unique learning rate.

We initialize the coefficients to zero for each $i > 0$.

3. Our Efficient Implementation

This section describes our efficient algorithm for the learning specified in Section 2.

Subsection 3.1 prepares for Subsection 3.2 by explaining a logarithmic-time computation algorithm for tabular TD(λ) (Katayama et al., 1999). Subsection 3.2 describes how it can be applied to TD(λ) using Haar basis functions.

Here we only give their intuitive explanation. They are mathematically derived in Katayama (2000).

3.1 Logarithmic-time Computation Algorithm for Tabular TD(λ) updating

Tabular TD(λ) updating can be implemented in time logarithmic in the number of states (Katayama et al., 1999). In this subsection we explain the algorithm. Our resulting algorithm for Haar basis functions is based on this algorithm.

In this subsection first two subsubsections are preliminaries describing the essential ideas of the algorithm. The third subsubsection is its formalization.

3.1.1 Lazy Updating

The definition of TD(λ) requests that as long as $\lambda > 0$ $V(y)$ for all states y must be updated on every time step. The naive implementation of the overall updating costs at least $|Y|$ time where Y is the set of states.

Actually, however, the instant values for all states are not needed at every time step. We say an algorithm consuming only

- $O(\log |Y|)$ time for updating the data the program maintains and
- $O(\log |Y|)$ time for accessing each $V(y)$

is a logarithmic-time computation algorithm for every step. Now we call it *lazy updating* to update least at every time step and not to update $V(y)$ itself until y is visited (Wiering & Schmidhuber, 1998).[2]

3.1.2 The Data Structure

Assume a tree that has the same number of leaf nodes (or leaves) as the number of $|Y|$, and assign each state to some leaf exclusively.[3] By regarding each state y in the same light as its corresponding leaf L we can define $D_t(L)$ and $V_t(L)(= k_t(L))$. The algorithm lets each node preserve some information, and it updates the tree (by updating the information) in time linear in the depth of the tree (Figure 3). By adopting a minimal tree the depth is $O(\log |Y|)$. In this way we can update the tree in $O(\log |Y|)$ time and $O(|Y|)$ space.

We can regard the sequence describing "which child node to select" at each node in the same light as its corresponding state, because the route from the root node to each leaf is unique.

For time step t we call each node in the route to the current state y_t at time t *visited node at time* t and other nodes *non-visited nodes at time* t. Because the information at each leaf is updated when it is visited, the value stored at each leaf L at time t is not $V_t(L)$

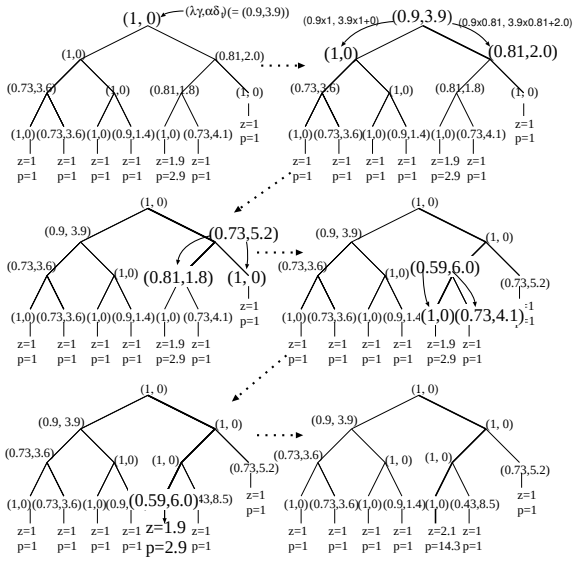

Figure 3. Computation flow in the algorithm for tabular TD(λ). Each tuple (d, s) means that the recency gap is d and the put-off update is s. When a state is observed, the algorithm collects the information of its value decentralized to each selected nodes, updates relevant nodes, and obtains its new value as new p. The values of the other states can be obtained through putting $(1, 0)$ into the tree after updating and following down the route to the states without updating the tree.

but $V_{\tau_{t+1}(L)}(L)$, that is, the value at the last visit to L, where $\tau_t(N)$ denotes *the last time to visit the node* N *before time* t. We let $p_t(L)$ denote $V_{\tau_{t+1}(L)}(L)$. In the same manner, $z_t(L)$ denotes the eligibility trace of L's corresponding state at the last visit to L.

The number of the visited nodes is equal to the depth of the tree. In addition, each visited node has a non-visited child node whose descendants are all non-visited. Therefore a minimal binary tree has the same number of "non-visited subtrees" (consisting of non-visited nodes) as its depth. It follows that if we put some pieces of information at the root nodes of the non-visited subtrees about what operations to apply to all the nodes in the non-visited subtrees, and can update the pieces of information in constant time per piece, we can update the whole binary tree in time linear in its depth (or in $O(\log |Y|)$ time) (Figure 3).

The addend to p's in each non-visited subtree, or the discounted sum of the addends while it is non-visited, is one of the pieces of information to be put at the root node of the non-visited subtree. Since it is cleared to 0 when the subtree is visited, the sum at time t is $\alpha \sum_{i=\tau_t+1}^{t} (\lambda\gamma)^{i-1-\tau_t} \delta_i$. We call this sum *the put-off update* and let $\tilde{s}_t$ denote it.

$\tilde{s}_t$ is a summation until t. In actual computations the information put at the node is not updated until its parent node will be updated. Therefore at node N we

[2] Wiering et al. (1998) categorizes this idea into lazy learning.

[3] Just like other divide-and-conquer algorithms, this algorithm enforces the tree structure to the set of states — the tree need not reflect the actual state structure.

do not have the instant $\tilde{s}_t$ but its difference from that of the parent node. We let s_t denote the difference.

The put-off update of a node needs to be added to the p's of all of its descendants after being multiplied by their eligibility traces. Therefore, in order to be able to compute each value, we have to be able to compute the eligibility trace of each leaf.

As well as the usage of p_t and $\tilde{s}_t$ for computing V_t, we use z_t and *recency gap* $\tilde{d}$ defined as $(\lambda\gamma)^{t-\tau_t}$ to compute D_t. Furthermore, we let d_t denote its difference (or rate) from that of its parent node.

3.1.3 Logarithmic-time Computation Algorithm

Now we give the rigorous definitions of the variables. We can implement the algorithm by a program that always preserves the instant values of d, s, z, and p.

$$d_0(N) = s_0(N) = 0 \tag{6}$$

$$d_t(N) = \begin{cases} 1, & \text{if } S_t(N); \\ \tilde{d}_t(g(N))d_{t-1}(N), & \\ \qquad \text{if } S_t(g(N)) \text{ and not } S_t(N); & \\ d_{t-1}(N), \quad \text{unless } S_t(g(N)) & \end{cases} \tag{7}$$

$$s_t(N) = \begin{cases} 0, & \text{if } S_t(N); \\ \tilde{s}_t(g(N))d_{t-1}(N) + s_{t-1}(N), & \\ \qquad \text{if } S_t(g(N)) \text{ and not } S_t(N) & \\ s_{t-1}(N), \quad \text{unless } S_t(g(N)) & \end{cases} \tag{8}$$

$$\tilde{d}_t(N) = \begin{cases} \lambda\gamma, \text{ if } N = \perp; \\ \tilde{d}_t(g(N))d_{t-1}(N), \quad \text{o.w.} \end{cases} \tag{9}$$

$$\tilde{s}_t(N) = \begin{cases} \alpha\delta_t, \quad \text{if } N = \perp; \\ \tilde{s}_t(g(N))d_{t-1}(N) + s_{t-1}(N), \\ \qquad \text{o.w.} \end{cases} \tag{10}$$

$$p_t(L) = \begin{cases} p_{t-1}(L) + z_{t-1}(L)\tilde{s}_t(L), & \\ \qquad \text{if } S_t(L); & \\ p_{t-1}(L), \quad \text{o.w.} & \end{cases} \tag{11}$$

$$z_t(L) = \begin{cases} \tilde{d}_t(L)z_{t-1}(L) + 1, & \text{if } S_t(L); \\ z_{t-1}(L), & \text{o.w.} \end{cases} \tag{12}$$

$$V_t(L) = p_{t-1}(L) + z_{t-1}(L)\tilde{s}_t(L) \tag{13}$$

In those equations, $\perp$ denotes the root node, $g(N)$ denotes the parent node of N, and $S_t(N)$ means that N is a visited node at time t. For above equations we can prove that $V_t(L)$'s yield the exact values for the relevant states (Katayama et al., 1999). We can obtain the same result for replacing eligibility traces by fixing all the z's to 1.

3.2 Extension to the Hierarchical Basis Functions

Since Haar basis functions form an infinite tree, and so do the coefficients $k(i)|_{i=0,1,\ldots}$, its naive implementation is impossible. However, it can be compressed into a small tree, because never visited nodes have no information, and because visited nodes are updated in a common way. Now we derive a possible and efficient implementation by repeated algorithm transformation within the same specification (Table 1).

Table 1. The derivation process of the working algorithm.

Algorithm 0 the naive implementation
⇓ apply the logarithmic-time algorithm.
Algorithm 1 using infinite list of finite trees
⇓ unify each tree.
Algorithm 2 using one infinite binary tree
⇓ compress the infinite tree into finite one.
Algorithm 3 using one finite binary tree

3.2.1 Algorithm 0

Algorithm 0 is the naive implementation of the specification described in Section 2.

3.2.2 Algorithm 0 → Algorithm 1

For one state observation, each depth of the tree of Haar basis functions has only one visited basis whose support includes the state. Therefore, Algorithm 0 corresponds to conducting separate tabular TD(λ) for each depth.[4] By applying the logarithmic-time algorithm to each depth, we obtain Algorithm 1 preserving the same specification as Algorithm 0 (Figure 4, left).

3.2.3 Algorithm 1 → Algorithm 2

By overlapping each tree in Algorithm 1, we find that the path of visited bases coincides at each time step. Thus the values of (d, s) at the same position always coincide throughout the trees, and therefore we can collect the trees into one tree by sharing the values of (d, s) (Figure 4, right). As a result we obtain Algorithm 2 using the infinite tree containing (d, s, z, p) at each node.

At each node (z, p) is updated with an updating rule similar to that of the logarithmic-time algorithm.

3.2.4 Algorithm 2 → Algorithm 3

The problem here is that the tree is an infinite tree. However, by following the updating process one can find the way to compress the tree into finite tree.

The initial tree T_0 at time $t = 0$ is depicted by Figure 5 (left). In this figure, each $(0, 0, 0, 0)$ describes the (d, s, z, p). The subtree under the root node has a recursive structure and if we let it v we find that v is the tuple of $(0, 0, 0, 0)$ and two v's.

At time $t = 1$ we obtain an observation which is, say, $y_1 = 0.1101\ldots$ in binary base (Figure 5, center). Only (d, s, z, p) of visited nodes and their direct children are

[4] To be exact, $\psi_i(y)$ can be -1 depending on the observation y and basis ψ_i.

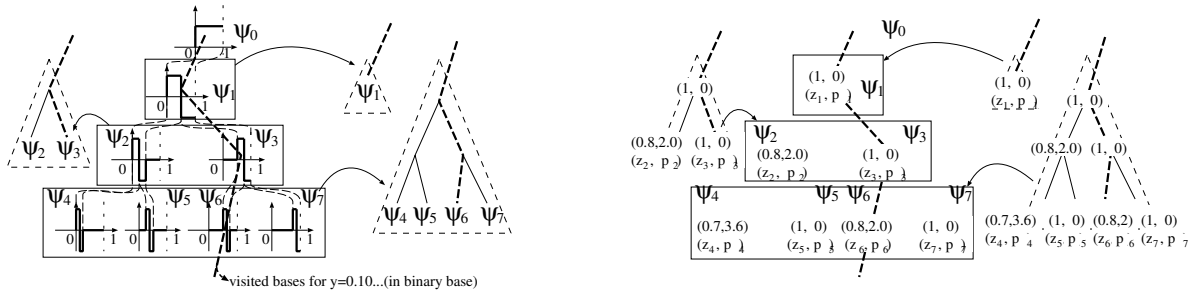

Figure 4. The derivation of Algorithm 1 from Algorithm 0 (left) and that of Algorithm 2 from Algorithm 1 (right).

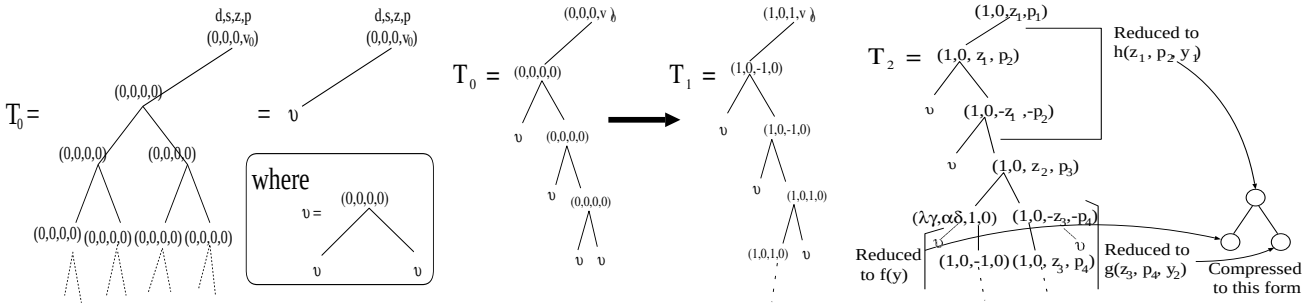

Figure 5. Data tree at time $t = 0$ (left), derivation of T_1 (center), and T_2 and its compression (right).

updated. However, from (7) and (8) the direct children remain v. In addition, (d, s, z, p) of visited nodes becomes $(1, 0, 1, 0)$. T_1 has no more information than y_1 (information about "which child nodes to go down"), and can be written as $f(y_1)$.

Assume that we obtain $y_2 = 0.1111\ldots$ in binary base at time $t = 2$. Figure 5 (right) depicts T_2. In this tree d and s of each *only child node* (or regular child node whose parent node has a single regular child node, where *regular* node means node that is not v) are 1 and 0 respectively. In addition the consecutive only children (or only descendants) share the same $e = z/h_b$ and $q = p/h_b$, where $h_b \in \{1, -1\}$ represents which child node is regular. By taking advantage of this fact we can compress the consecutive only children to obtain a binary tree with two leaves.

The same thing is true for T_t's for all t's in general. The compressed form of T_t is a binary tree with at most t leaves, which has at most $2t - 1$ nodes.

4. Experimental Evaluation

In this section we experimentally evaluate our algorithm. We demonstrate on two tasks described in Sutton (1996): puddle world and acrobot, comparing with CMAC: the conventional generalization method adopted in Sutton.

The puddle world is a goal seeking task with an obstacle. The learner observes its position and decides which way to go. The cost (or negative reward) per each time step is 0 at the goal, 1 at the plain, and 1 – 41 in the puddle, depending on the depth.

The acrobot is a gymnast-like two-link robot. Observing the positions and the velocities of the links, it has to swing up its tip only by giving torque at the waist (the first joint from the tip) without any torque at the hand (the second joint) as quickly as possible. The specifications of both tasks are the same as those in Sutton.

CMAC (e.g., Sutton, 1996) is a combination of several tilings. The approximated value is the sum of the values all the tilings return. We used the same tilings, the same exploration strategy, and the same parameters for both tasks as those in Sutton, except that we used $\alpha = 0.05$ for all the puddle world agents in order to clarify the asymptotical difference, and that we used the exploration rate $\epsilon = 0.1$ instead of $\epsilon = 0$ throughout the experiments in order to assure them to converge not to suboptimal solutions but to the optimal solution. We also tried some other CMACs for the puddle world.

4.1 The Detailed Description of the Learner

As long as possible, Haar agents use the same parameters as those for CMAC agents. In addition, both agents obtain the action-value function by assigning each action a function approximator on states and online learning by Sarsa(λ) (Sutton, 1996).

The following subsubsections describe some issues of the Haar agents.

4.1.1 Application to Multidimensional Observations

In both tasks, the observations are multidimensional. In order to apply our algorithm, they have to be arranged into some one-dimensional real values. To take an instance of the two-dimensional cases, we can obtain one-dimensional values by mixing each binary digit from two coordinates alternately. For example, observation $(0.375, 0.666)$ which is in binary base $(0.0110000, 0.1010101\ldots)$ yields $0.4229\ldots$ which is in binary base $0.01101100010001\ldots$.[5] In this way the feature that "the nearer the positions of two states are, the more bases they share" is preserved.

4.1.2 Replacing Eligibility Traces

When applied to puddle world, our algorithm with accumulating traces is slow to converge. In addition, Sutton (1996) only shows the result using replacing traces for puddle world. For the reasons mentioned above, we present the results using replacing traces.

It is difficult to settle the definition of replacing traces for basis functions which may return other values than 0 or 1. We defined replacing traces for ternary features:

$$D_t(i) = \begin{cases} \psi_i(y_t), & \text{if } \psi_i(y_t) \neq 0; \\ \lambda\gamma D_{t-1}(i), & \text{otherwise.} \end{cases} \quad (14)$$

4.1.3 Forgetting Strategy

Because our algorithm increases the data size by one leaf and one node per one time step, it may exhaust the memory. When this happens, it has to forget, or abandon a leaf and a node which seem to be useless.

The forgetting strategy we adopted for the presentation forgets the node that has the least influence on the value function.[6] We measured the influence by $|p|(1-\beta)\beta^{\lceil \log_2(i+1) \rceil}$,[7] i.e., the L^∞ norm of the difference between the value function before forgetting and that after forgetting. We implemented the forgetting by replacing a node and its two child leaves with a leaf.

Since the integral of a basis that is not at the root node is 0, forgetting just flattens a local relevant part of the value function. This fact reduces the influence of forgetting on the value function.

Our forgetting is not for generalization but only for space efficiency. Therefore, as long as there is free memory resource, we should exploit it. In the experiment, our algorithm started forgetting at $t = 100000$.

[5] By adopting this approach, the precision of the observation given to the learner increases by 2 bits as the precision per one dimension increases by 1 bit. In our experiment shown later we give 0.5 and 0.9 as β, whose substantial values must be 0.25 and 0.81 ($= \beta^2$). In general, β should be large for high-dimensional observations.

[6] Of course we can define different strategies.

[7] We used p instead of the actual coefficient because the computation of all the coefficients costs linear time in the number of states that have ever been visited.

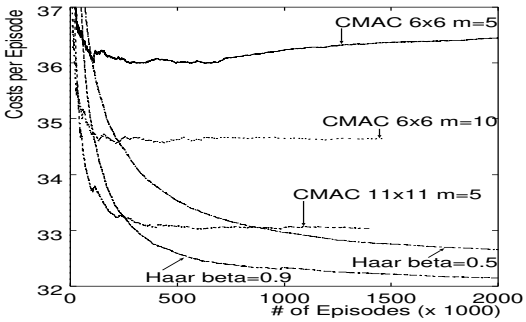

Figure 6. Results on the puddle world. m of each CMAC means it uses m tilings. As is not shown here, Haar $\beta = 0.5$ agent reached 32.1 costs per episode at 10,000,000 episodes.

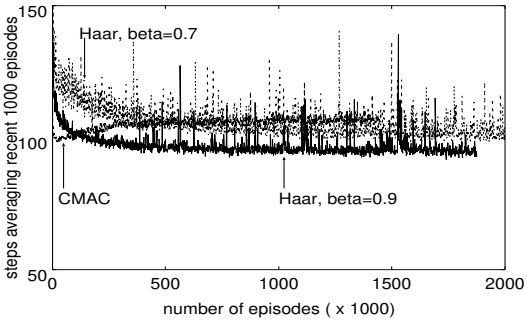

Figure 7. Results on the acrobot.

4.2 Experimental Results

We conducted experiments under the same random number sequence. Figure 6 shows the results on the puddle world task. It is a plot of "the costs so far" per "the number of episodes so far" on the y axis against "the number of episodes so far" on the x axis where episode means the interval from the start to the goal. Figure 7 shows the results on the acrobot task. It is a plot of "steps averaging 1000 consecutive episodes" on the y axis against "the number of episodes so far" on the x axis.

The CMAC agents are quick to converge to suboptimal solutions, while the agents with Haar basis functions always converge to the best solution steadily. We can say our algorithm with Haar basis functions are robust with respect to its parameter. Though CMAC could do better with fine grained features, its long-term experiment is difficult because it takes too long time.

5. Conclusion

We proposed an algorithm implementing TD(λ) updating for the infinite tree of Haar basis functions. They can approximate any value function on $[0, 1)$ without any generalization error as time $t \to \infty$.

Our algorithm computes TD(λ) updating in time linear in the precision of observations, which is small enough at least for the precisions under 100 bits. In fact, for the experiments shown in Section 4 our algorithm for Haar basis functions computes faster than the naive implementation of CMAC, and even faster than the fast implementation of CMAC based on Katayama et al. (1999). The main reason of this difference is that the computational time of CMAC agents is linear in the number of tilings, and the CMAC acrobot agents use 48 tilings.

On the other hand, at each time step it costs space linear in the number of the states visited until the time. For this reason, when we use actual computers instead of some theoretical computer model, it falls short of the heap space if the learning continues for a long time. In this case the learner has to forget unnecessary knowledge, which destroys the assurance that the approximation precisely converges to the target function. However, the experimental results suggest that, if we adopt the forgetting strategy based on the dependency of the value function upon the knowledge, it yields the best solution.

As for the examples presented, our algorithm is slower to converge than that using appropriately selected CMAC. However, the former converges to the best solution for any parameter β, while the latter converges to worse solutions. Considering that CMAC requires time and labor for appropriate selection of the size and the number of tiles, our method can be more useful.

One possible way to accelerate the convergence may be to decrease the learning rate α of each basis at each time it is visited. This device can easily be included in our algorithm, and we are working on it.

Our presented algorithm limits the states within $[0, 1)$. One possibility to apply it to the real state space may be regarding the floating point expression as a fragmental number in binary base. However, we have no experience on this approach.

Our algorithm generalizes only over states, and it cannot apply straightforwardly to the generalization over actions. Again, we are also working on this idea.

The most serious disadvantage of our algorithm may be the difficulty in its implementation. In order to avoid bugs we mathematically derived the algorithm and implemented it in a functional language. We are going to translate the program into a faster and more popular language and present it to the public.

References

Atkeson, G. C., Moore, A. W., & Schaal, S. (1997). Locally weighted learning, *Artificial Intelligence Review*, 11, 11–73.

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems, *IEEE Transactions on Systems, Man, and Cybernetics*, 13, 835–846.

Chapman, D., & Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons, *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* (pp. 726–731).

Chui, C. K. (1992). *An introduction to wavelets.* Boston: Academic Press.

Katayama, S., & Kobayashi, S. (1999) Logarithmic-time updating algorithm for TD(λ) learning. *Journal of Japanese Society for Artificial Intelligence*, 14, 119–130. (in Japanese).

Katayama, S. (2000). *Satisficing, efficient implementation, and generalization for on-line reinforcement learning.* Ph.D. thesis, Department of Computational Intelligence and Systems Science, Interdisciplinary Graduate School of Science and Engineering, Tokyo Institute of Technology. (in Japanese).

McCallum, A. K. (1995). *Reinforcement learning with selective perception and hidden state.* Ph.D. thesis, Department of Computer Science, The College Arts and Sciences, University of Rochester.

Lin, L. & Mitchell, T. M. (1992). *Memory approaches to reinforcement learning in nonmarkovian domains* , (Technical Report CS-92-138). School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Santamaría, J. C., Sutton, R. S., & Ram, A. (1999) Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior 6*, 163–218.

Singh, S. P. & Sutton, R.S.(1996). Reinforcement learning with replacing eligibility traces. *Machine Learning, 22*, 123-158.

Sutton, R. S. (1988) Learning to predict by the methods of temporal differences. *Machine Learning, 3*, 9-44.

Sutton, R. S. (1996) Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems 8*, (pp. 1038-1044).

Tsitsiklis, J. N. & Van Roy, B. (1997) An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42, 674–690.

Watkins, C. J. C. H., & Dayan, P. (1992) Q-Learning. *Machine Learning, 8*, 179-292.

Wiering, M., & Schmidhuber, J. (1998) Fast online Q(λ), *Machine Learning, 33*, 105–115.