[Trans. of the Society of Instrument and Control Engineers
Vol.E-3, No.1, 73/81 (2004)]

An Actor-Critic Algorithm using a Binary Tree Action Selector

Reinforcement Learning to Cope with Enormous Actions

Hajime KIMURA* and Shigenobu KOBAYASHI**

In real world applications, learning algorithms often have to handle several dozens of actions, which have some distance metrics. Epsilon-greedy or Boltzmann distribution exploration strategies, which have been applied for Q-learning or SARSA, are very popular, simple and effective in the problems that have a few actions, however, the efficiency would decrease when the number of actions is increased. We propose a policy function representation that consists of a stochastic binary decision tree, and we apply it to an actor-critic algorithm for the problems that have enormous similar actions. Simulation results show the increase of the actions does not affect learning curves of the proposed method at all.

Key Words: reinforcement learning, actor-critic, action selector, binary tree, exploration strategy

1. Introduction

Reinforcement learning (RL) is a promising approach for robots to improve the behavior themselves or make up for lack of knowledge about the tasks [4]. Learning control in real world applications requires dealing with both continuous state and several dozens or more of actions. There are many works about the generalization techniques to approximate value functions over state space; CMAC [8] is one of linear architectures, and the others are neural-networks, Fuzzy methods [3], instance-based methods [2], etc. To deal with dozens of actions, not only the techniques of action value approximation but also exploration strategies are essential. Q-learning or SARSA algorithms with ϵ-greedy or softmax action selection rules are very popular, simple and effective in the problems which have a few actions. However, in these flat selection methods, the efficiency would decrease when the number of actions is increased.

You know, for example in our daily life, when we operate a handle, we will choose 'turn right' or 'turn left' at first, and there after we will select the amount of the operation, 'small', 'medium' or 'large'. Thus, hierarchical decision making is more effective than flat action selection when there exisis many similar actions. In this example, we group 'small turn left', 'medium turn left' and 'large turn left' into a higher hierarchical action 'turn left' unconsciously. In many real-world applications, the actions can be grouped and ranked in the same way. In this paper, a reinforcement lerarning approach that uses hierarchical action selection scheme for the problems that have enormous similar actions is proposed. It adopts a policy function representation that consists of a stochastic binary decision tree, and updates the policy parameters by an actor-critic algorithm [1) 5) 8)]. The knowledge of the order of the action groups is easily embedded in the tree structure of the decision making by designers in advance. Simulation results show the increase of the actions does not affect learning curves of the proposed method at all when the action space is assumed to have some good distance metric.

* Graduate School of Eng. Kyushu University
** Interdisciplinary Graduate School of Sci. and Eng. Tokyo Institute of Technology

2. Problem Formulation

2.1 Markov Decision Processes

Let $\mathcal{S}$ denote state space, $\mathcal{A}$ be action space, $\mathcal{R}$ be a set of real number. At each discrete time t, the agent observes state $s_t \in \mathcal{S}$, selects action $a_t \in \mathcal{A}$, and then receives an instantaneous reward $r_t \in \mathcal{R}$ resulting from state transition in the environment. In general, the reward and the next state may be random, but their probability distributions are assumed to depend only on s_t and a_t in Markov decision processes (MDPs), in which many reinforcement learning algorithms are studied. In MDPs, the next state s_{t+1} is chosen according to the transition probability $T(s_t, a, s_{t+1})$, and the reward r_t is given randomly according to the expectation $r(s_t, a)$. The agent does not know $T(s_t, a, s_{t+1})$ and $r(s_t, a)$ ahead of time. The objective of reinforcement learning is to construct a policy that maximizes the agent's performance. A natural performance measure for infinite horizon tasks is the cumulative discounted reward:

$$V_t = \sum_{k=0}^{\infty} \gamma^k \, r_{t+k} \ , \qquad (1)$$

where the discount factor, $0 \leq \gamma \leq 1$ specifies the importance of future rewards, and V_t is the value at time t. In MDPs, the value can be defined as:

$$V^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t r_t \middle| s_0 = s, \pi\right] , \qquad (2)$$

where $E\{\cdot\}$ denotes the expectation. The objective in MDPs is to find an optimal policy that maximizes the value of each state s defined by Equation 2.

2.2 MDPs with Enormous Similar Actions

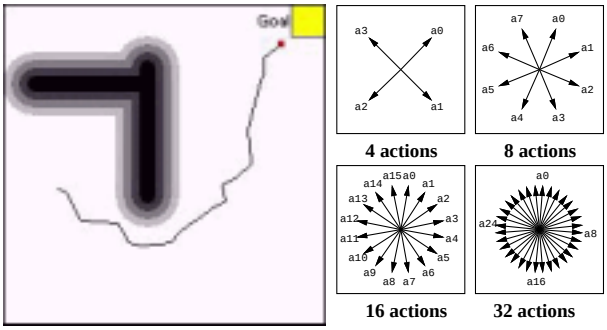


Fig. 1 A modified puddle-world problem. There are 4, 8, 16, 32, or 64 actions, which moves approximately 0.05 in these directions. The state space is continuous and two-dimensional, which is bounded by $[0, 1]$ for each dimension.

Many real world applications would have a feature that neighboring actions will almost result in similar state transitions. **Fig. 1** illustrates the example using a puddle-world task [7] but it is slightly modified from the original formulation. The state space is continuous and two-dimensional, which is bounded by $[0, 1]$ for each dimension. When the agent selects an action, the agent moves approximately 0.05 in that direction unless the movement would cause the agent to leave the limits of the space. A random Gaussian noise with standard deviation 0.01 was added to the motion along both dimensions. Obviously, the more actions increase (especially in 32 or 64 actions), the more similar state transitions occur. The details of the puddle world are as below. The rewards on this task were -1 for each time step plus additional penalties if either or both of the two oval puddles were entered. These penalties were -400 times the distance into the puddle (distance to the nearest edge). The puddles were 0.1 in radius and were located at center points $(0.1, 0.75)$ to $(0.45, 0.75)$ and $(0.45, 0.4)$ to $(0.45, 0.8)$. The initial state of each episode was selected randomly uniformly from the non-goal states.

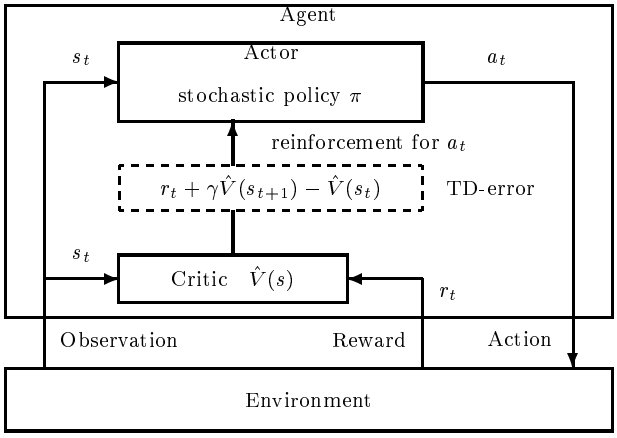


Fig. 2 A standard actor-critic architecture.

3. Actor-Critic Method for binary-tree action selector

Almost traditional value-based methods such as Q-learning or SARSA are dealing with the problems shown in Section 2.2 by generalization techniques over the action space. But such value-based methods work poor in non-Markovian environments, and it needs enormous number of iteration for estimating the state-action value. On the other hand, actor-critic algorithms are robust against non-Markovian environments, and its policy improvement is much faster. However, the actor-critic methods does not have any scheme for dealing with many similar actions shown in Section 2.2 making use of such problem's property. We propose a new action selection scheme for the actor-critic methods.

3.1 Actor-Critic Algorithm

As shown in **Fig. 2**, the actor-critic algorithm is composed of two modules; One is *actor* that selects action, and the other is *critic* that evaluates states. The actor implements a *stochastic policy* that maps from a representation of a state to a probability distribution over actions. The critic attempts to estimate the evaluation function $\hat{V}(s)$ for the current policy. After each action selection, the critic evaluates the new state to determine whether things have gone better or worse than expected. The evaluation is known as *temporal difference (TD)*, and it is used to evaluate the action just selected as an effective reinforcement. When the TD is positive, then the executed action would be good, therefore the probability of choosing that action is increased. Contrary, when the TD is negative, then the action would be bad, and the probability of that action is decreased.

Fig. 3 specifies an actor-critic algorithm we used [5]. It is noteworthy that both the actor and the critic adopt el-

(1) Observe state s_t, choose action a_t with probability $\pi(a_t, \theta, s_t)$ in the actor, and perform it.

(2) Observe immediate reward r_t, resulting state s_{t+1}, and calculate the TD-error according to

$$(\text{TD-error}) = r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t) \ , \qquad (3)$$

where $0 \leq \gamma \leq 1$ is the discount factor, $\hat{V}(s)$ is an estimated value function by the critic.

(3) Update the estimating value function $\hat{V}(s)$ in the critic according to the TD(λ) method as follows:

$$\begin{aligned} e_v(t) &= \frac{\partial}{\partial w}\hat{V}(s_t) \ , \\ \overline{e_v}(t) &\leftarrow e_v(t) + \overline{e_v}(t) \ , \\ \Delta w(t) &= (\text{TD-error})\,\overline{e_v}(t) \ , \\ w &\leftarrow w + \alpha_v \, \Delta w(t) \ , \end{aligned}$$

where e_v denotes the eligibility of the parameter w in the function approximator $\hat{V}(s)$, $\overline{e_v}$ is its trace, and α_v is a learning rate.

(4) Update the actor's stochastic policy according to

$$\begin{aligned} e_\pi(t) &= \frac{\partial}{\partial \theta} \ln\Big(\pi(a_t, \theta, s_t)\Big) \ , \\ \overline{e_\pi}(t) &\leftarrow e_\pi(t) + \overline{e_\pi}(t) \ , \\ \Delta\theta(t) &= (\text{TD-error})\,\overline{e_\pi}(t) \ , \\ \theta &\leftarrow \theta + \alpha_\pi \, \Delta\theta(t) \ , \end{aligned}$$

where e_π is the eligibility of the policy parameter θ, $\overline{e_\pi}$ is its trace, and α_π is a learning rate.

(5) Discount the eligibility traces as follows:

$$\begin{aligned} \overline{e_v}(t+1) &\leftarrow \gamma \lambda_v \, \overline{e_v}(t) \ , \\ \overline{e_\pi}(t+1) &\leftarrow \gamma \lambda_\pi \, \overline{e_\pi}(t) \ , \end{aligned}$$

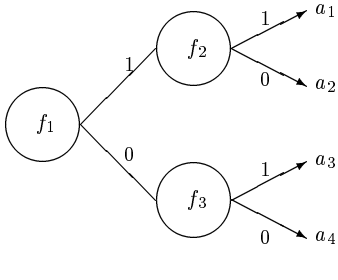
where λ_v and λ_π ($0 \leq \lambda_v, \lambda_\pi \leq 1$) are discount factors in the critic and the actor respectively.

(6) Let $t \leftarrow t+1$, and go to step 1.

Fig. 3 An actor-critic algorithm making use of eligibility traces in both the actor and the critic.

igibility traces; obviously TD(λ) in the critic refers to use it, and the actor uses eligibility traces on policy parameters. The calculation scheme of the TD(λ) in the critic is described in Step 3 and 5 in Fig. 3. The parameter λ_v specifies the eligibility trace of the TD($\lambda = \lambda_v$). In the actor, the policy is explicitly represented by its own function approximator, and updated according to the gradient of value function with respect to the policy parameters [5) 9)]. Let $\pi(a, \theta, s)$ denote probability of selecting action a under the policy π in the state s. That is, the policy is represented by a parametric function approximator using the internal parameter vector θ. The agent can improve the policy π by modifying the parameter θ. For example, θ corresponds to synaptic weights where the action selecting probability is represented by neural networks, or θ means weight of rules in classifier systems. The form of the function $\pi(a, \theta, s)$ is specified for the binary-tree action selector in this paper. The actor's learning rule is shown in Step 4 and 5 in Fig. 3. The eligibility e_π is the same variable defined in REINFORCE algorithms [12)]. The parameter λ_π specifies the actor's eligibility trace, but its features are somewhat different from TD(λ)'s. When λ_π is close to 0, the policy would be updated according to the gradient of the estimated value function $\hat{V}$, and when λ_π is close to 1, the policy would be updated by the gradient of the actual return, defined by Equation 1 [5)].

3.2 Binary-Tree Action Selector



(1) Start at the root of the tree, choose the branch 1 with probability f_1 (or the branch 0 with probability $1 - f_1$).

(2) At the selected branch node, continue the same process until a leaf is encountered. If the node f_2 is selected, choose 1 with probability f_2 (or 0 with probability $1 - f_2$). If the selected node is f_3, then choose 1 with probability f_3, etc. And take the corresponding branch.

(3) At the resulting leaf node, execute the corresponding action a.

(4) Calculate eligibilities for all nodes by

$$\frac{\partial}{\partial f_j} \ln \pi(a, \theta, s) = \begin{cases} \frac{y_j - f_j}{f_j(1-f_j)} & \text{if } f_j \text{ is active} \\ 0 & \text{otherwise,} \end{cases}$$

where $y_j \in \{0, 1\}$ is the outcome of the node f_j.

Fig. 4 A binary-tree action selection scheme for four actions, and its policy representation. f_i denotes a probability distribution function.

Here we propose a new technique for policy gradient methods to cope with enormous similar actions. **Fig. 4** illustrates the binary-tree action selector for four actions. The stochastic policy is represented by a binary decision tree. Leaves of the decision tree are primitive actions, and the other nodes branch to two ways following the corresponding probability f_i. Formally, the policy function is given by

$$\pi(a, \theta, s) = \begin{cases} f_1 \, f_2 & a = a_1 \\ f_1 \, (1 - f_2) & a = a_2 \\ (1 - f_1) \, f_3 & a = a_3 \\ (1 - f_1) \, (1 - f_3) & a = a_4 \end{cases} \qquad (4)$$

Then the eligibility for f_1 is given by

$$\frac{\partial}{\partial f_1}\ln\pi(a,\theta,s) = \begin{cases} 1/f_1 & a=a_1 \\ 1/f_1 & a=a_2 \\ -1/(1-f_1) & a=a_3 \\ -1/(1-f_1) & a=a_4 \end{cases} = \frac{y_1-f_1}{f_1(1-f_1)}\ , \quad (5)$$

where $y_1 \in \{0,1\}$ denotes the outcome of the unit f_1. Similarly, the eligibility for f_2 (or f_3) is given by

$$\frac{\partial}{\partial f_2}\ln\pi(a,\theta,s) = \begin{cases} 1/f_2 & a=a_1 \\ -1/(1-f_2) & a=a_2 \\ 0 & a=a_3 \\ 0 & a=a_4 \end{cases} = \begin{cases} \frac{y_2-f_2}{f_2(1-f_2)} & \text{if } f_2 \text{ is active} \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

This result says that the calculation of the eligibility at each node is the same and can be done by only local information. The eligibility on the executed path from the root of the tree to a leaf is simply given by $(y_j - f_j)x_i$, otherwise 0. This property remains in different size of the trees. The required memory size (the number of nodes) is proportional to the number of actions minus 1 (see **Fig. 5**).

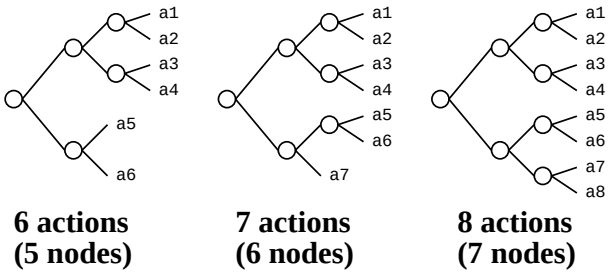


Fig. 5 The binary trees for 6, 7 and 8 actions.

3.3 Actor-Critic with Binary-Tree Action selector

This section specifies the scheme of the actor-critic algorithm shown in Section 3.1 using the binary-tree action selector shown in Section 3.2. A feature vector $(x_1, x_2, \cdots, x_i, \cdots, x_n)$ is given to the agent as the state input s. Especially, $(x_1(t), x_2(t), \cdots, x_i(t), \cdots, x_n(t))$ denotes the feature vector where the time step is t. The features must be constructed from the states in many different ways so that the feature vectors are linearly independent between the essential states. In the critic, the state value function is given by a linear function using weight parameters w_i and the features x_i as:

$$\hat{V}(s) = \sum_{i=1}^{n} w_i\, x_i\ . \quad (7)$$

In this case, the gradient-descent TD(λ) algorithm has a sound convergence property [10]. The eligibility e_{vi} of the parameter w_i is given by $(\partial\hat{V}(s))/(\partial w_i) = x_i$.

Using the feature vector, the branching probability f_j in each node is given by

$$f_j = \frac{1}{1+\exp(-\sum_{i=1}^{n}\theta_{ji}x_i)}\ , \quad (8)$$

where θ_{ji} denotes policy parameters. The f_j is monotone increasing function, and $0 < f_j < 1$. From Equation 5, 6 and 8, $\partial f_j/\partial\theta_{ji} = x_i\, f_j(1-f_j)$, then the eligibility $e_{\pi ji}$ of θ_{ji} is given by

$$e_{\pi ji} = \frac{\partial f_j}{\partial\theta_{ji}}\frac{\partial}{\partial f_j}\ln\pi(a,\theta,s) = \begin{cases} (y_j-f_j)x_i & \text{if } f_j \text{ is active} \\ 0 & \text{otherwise.} \end{cases} \quad (9)$$

where $y_j \in \{0,1\}$ denotes the selected branch (0 or 1) in the node f_j.

The specified scheme of the proposed algorithm for the number of actions is m and the number of features is n is given as:

(1) Initialize: Arrange parameters w_i $(i = 1\cdots n)$ in the critic, corresponding eligibility traces $\overline{e_{vi}}$, policy parameters θ_{ji} $(j = 1\cdots m-1)$ in the actor, and corresponding eligibility traces $\overline{e_{\pi ji}}$.

(2) Generate a feature vector $(x_1(t), x_2(t)\cdots x_n(t))$ from the state input s_t.

(3) Action selection in the actor by the binary-tree:

(a) Start at the root node $j = 1$.

(b) In the node f_j, calculate as follows:

$$f_j = \frac{1}{1+\exp(-\sum_{i=1}^{n}\theta_{ji}x_i(t))}$$

Choose the branch 1 following the probability f_j, otherwise choose the branch 0.

(c) Repeat the same calculation in the successor node until reaching the leaf node.

(d) Execute action a in the resulting leaf node.

(e) Calculate the eligibility in the all node j as

$$e_{\pi ji}(t) = \begin{cases} (y_j-f_j)x_i(t) & \text{if } f_j \text{ is active} \\ 0 & \text{otherwise,} \end{cases}$$

where $y_j \in \{0,1\}$ is the outcome of the unit f_j.

(4) Calculate TD-error in the resulting state s_{t+1} and the reward r_t as

$$\text{(TD-error)} = r_t + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t) = r_t + \gamma\left(\sum_{i=1}^{n} w_i\, x_i(t+1)\right) - \sum_{i=1}^{n} w_i\, x_i(t)$$

(5) Update $\hat{V}(s)$ in the critic:

$$\begin{aligned} e_{vi}(t) &= x_i(t) \ , \\ \overline{e_{vi}}(t) &\leftarrow e_{vi}(t) + \overline{e_{vi}}(t) \ , \\ \Delta w_i(t) &= (\text{TD-error})\, \overline{e_{vi}}(t) \ , \\ w_i &\leftarrow w_i + \alpha_v \, \Delta w_i(t) \ , \end{aligned}$$

(6) Update policy parameters in the actor:

$$\begin{aligned} \overline{e_{\pi ji}}(t) &\leftarrow e_{\pi_{ji}}(t) + \overline{e_{\pi ji}}(t) \ , \\ \Delta\theta_{ji}(t) &= (\text{TD-error})\, \overline{e_{\pi ji}}(t) \ , \\ \theta_{ji} &\leftarrow \theta_{ji} + \alpha_\pi \, \Delta\theta_{ji}(t) \ , \end{aligned}$$

(7) Discount the eligibility traces:

$$\begin{aligned} \overline{e_{vi}}(t+1) &\leftarrow \gamma \, \lambda_v \, \overline{e_{vi}}(t) \ , \\ \overline{e_{\pi ji}}(t+1) &\leftarrow \gamma \, \lambda_\pi \, \overline{e_{\pi ji}}(t) \ , \end{aligned}$$

(8) Increment the time step as $t \leftarrow t+1$ and go to step 2.

3.4 Features

When the number of actions are m, the action selection is executed by $\log_2 m$ alternatives. The calculation for the choosing action or finding eligibilities is executed only along the selected path in the tree structure, and also it is very simple using only local information in each node. Such a local calculation property is quite suited for large scale problems. The proposed method is strictly a subclass of the actor-critic algorithm [5], therefore it can avoid influence of the non-Markovian effect than Q-learning.

3.5 Conventional Methods and Related Works

In conventional actor-critic methods, Boltzmann action selection scheme [8] is frequently used. The same as Section 3.3, a n-dimensional feature vector is given to the agent. When the number of actions is m, policy parameters θ_{ji} $(j = 1 \cdots m,\ i = 1 \cdots n)$ and the corresponding eligibility traces $\overline{e_{\pi ji}}$ are arranged. The agent selects an action a_k with the following probability:

$$\Pr(a_k|s) = \frac{\exp\left(\sum_{i=1}^{n} x_i \theta_{ki}\right)}{\sum_{j=1}^{m} \exp\left(\sum_{i=1}^{n} x_i \theta_{ji}\right)} \ . \qquad (10)$$

The eligibility for the policy parameters are calculated in the following [6]:

$$e_{\pi_{ji}}(t) = \begin{cases} -\Pr(a_j|s)\, x_i & \text{if } a_j \neq a_k, \\ (1 - \Pr(a_j|s))\, x_i & \text{if } a_j = a_k. \end{cases}$$

The flat-action-selecting actor-critic is achieved by replacing the above with the process of the actor in Section 3.3.

The Q-learning and the SARSA(λ) approximate state-action values as $Q(s,a) = \sum_{i=1}^{n} x_i w_{ai}$, and use ϵ-greedy policy, that is, one selects random action with probability ϵ, or with probability $1-\epsilon$, one selects an action that has largest Q value. Let s_t be a state at time t, $x_i(t)$ be its feature, a_t be the selected action at t, s_{t+1} and $x_i(t+1)$ be the resulting state and its feature, a_{t+1} denote the selected action at $t+1$, then the Q-learning rule is

$$\begin{aligned} \Delta w_{a_t i} &= x_i(t) \left(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right) \\ w_{a_t i} &\leftarrow w_{a_t i} + \alpha \Delta w_{a_t i} \ , \end{aligned}$$

for all $i \in \{1, 2, \cdots n\}$, and α denotes a learning rate factor. The learning rule of SARSA(λ) is given by

$$\begin{aligned} (\text{TD-error}) &= r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \\ \overline{e_{a_t i}}(t) &= x_i(t) + \overline{e_{a_t i}}(t) \\ \Delta w_{ai} &= \overline{e_{ai}}(t)\, (\text{TD-error}) \qquad \text{for all } a \\ w_{ai} &\leftarrow w_{ai} + \alpha \Delta w_{ai} \qquad \text{for all } a \\ \overline{e_{ai}}(t+1) &= \gamma \, \lambda \, \overline{e_{ai}}(t) \qquad \text{for all } a, \end{aligned}$$

for all $i \in \{1, 2, \cdots n\}$. $\overline{e_{ai}}$ denotes eligibility traces. The Q-learning requires $n \times m$ variables for w_i to approximate $Q(s,a)$. The SARSA(λ) needs $2 \times n \times m$ variables for w_i and its trace $\overline{e_{ai}}$. It is noteworthy that the SARSA(λ) costs the memory exactly the same size as our method's.

4. Simulation Experiment

The purpose of this experiment is to investigate the influence of increasing the number of actions on the algorithms. We tested our algorithm on the puddle world problem shown in Fig. 1, and compared it with Q-learning and SARSA(λ) algorithms. We used a tile coding method [8] to construct a feature vector from 2-dimensional state input as shown in **Fig. 6**. That feature vector includes $1^2 + 2^2 + 3^2 + \cdots + 7^2 = 140$ features $(x_1, x_2, \cdots x_{140})$ from broad 1×1 to narrow 7×7 gridding tiles. Each tile corresponds with the component of the feature vector. When a state input is given to $m \times m$ gridding tiles, then the corresponding component takes $m^2/140$, and the others are 0.

Throughout the experiments, in our actor-critic algorithm, the discount factor $\gamma = 1$, the learning rate $\alpha_v = 0.1$, $\alpha_\pi = 0.01$, and $\lambda_v = \lambda_\pi = 0.9$. In the Q-learning, the learning rate $\alpha = 0.5$ and $\epsilon = 0.1$. In the SARSA(λ), the learning rate $\alpha = 0.1$, $\epsilon = 0.1$ and $\lambda = 0.9$ **Figs. 7**, **8**, **11**, **12** are the results showing influence of the number of actions on the binary-tree actor-critic, flat actor-critic, Q-learning and SARSA(λ) respectively. In each graph, 'Cost' denotes the total cost from the start position to the goal area, and 'Trial' denotes the number of trials, where one trial begins start position and ends in the goal. Although the numbers of actions were different, yet the actor-critic using the binary-tree policy performed very similar learning curves without any parameter tuning nor scheduling. On the other hand, the

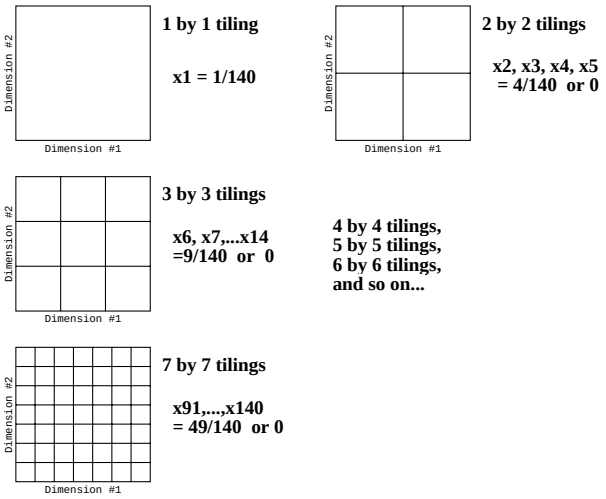


Fig. 6 A tile coding method to generate a feature vector, which contains all features from broad to narrow over a continuous two-dimensional state space. Any state is in exactly one tile of each tiling. Each tile is associated with one element of the feature vector $(x_1, x_2, \cdots x_{140})$. The number 140 is come from $1^2 + 2^2 + 3^2 + \cdots + 7^2$. When a state input is in some certain tiles, the corresponding tiles are activated and their elements are set to some positive value (e.g., 1×1 tiling is 1/140, etc.). The other elements are set to zero.

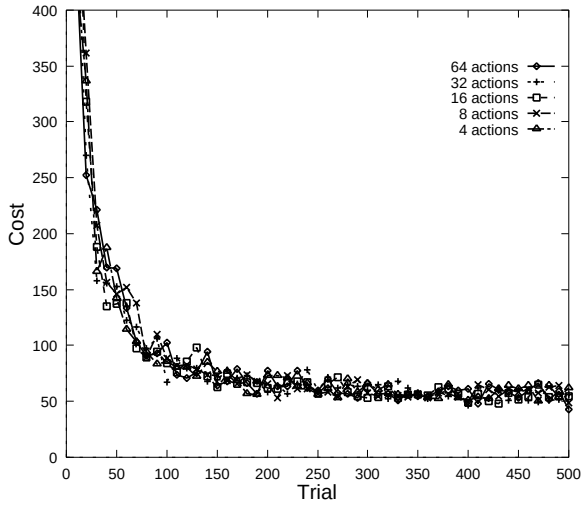


Fig. 7 Performance on the actor-critic method using the binary-tree action selector, averaged over 100 trials. $\alpha_v = 0.1$, $\alpha_\pi = 0.01$, $\lambda_v = 0.9$, $\lambda_\pi = 0.9$.

learning curves of the ϵ-greedy strategy were strongly affected with the numbers of actions on the Q-learning and the SARSA. **Fig. 10** shows the performance of the actor-critic using the binary-tree action selector in the puddle world with *randomly labeled actions* as shown in **Fig. 9**. It is noteworthy that the proposed method can work at least the same as the flat actor-critic in the puddle world with randomly labeled actions.

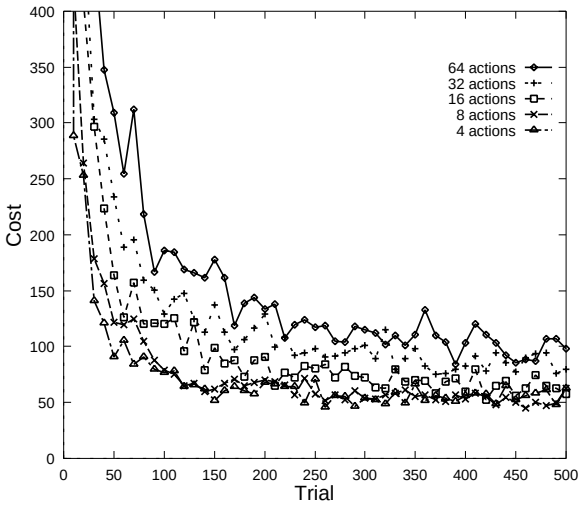


Fig. 8 Performance on the actor-critic method using the flat action selector, averaged over 100 trials. $\alpha_v = 0.1$, $\alpha_\pi = 0.01$, $\lambda_v = 0.9$, $\lambda_\pi = 0.9$.

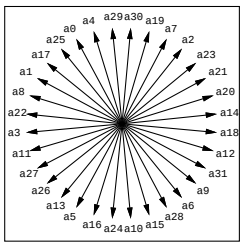


Fig. 9 An example which are randomly labeled actions in the puddle world.

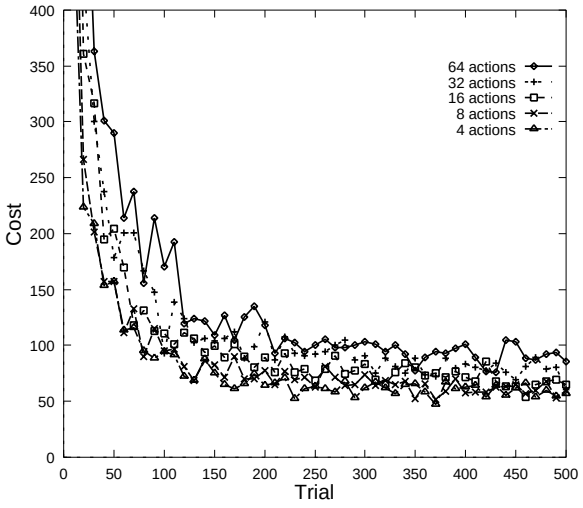


Fig. 10 Performance on the actor-critic method using the binary-tree action selector in the puddle world with randomly labeled actions, averaged over 100 trials. $\alpha_v = 0.1$, $\alpha_\pi = 0.01$, $\lambda_v = 0.9$, $\lambda_\pi = 0.9$.

5. Applying to Real Robots

We applied the algorithms to a learning task in real two robots that have 2 d.o.f. as shown in **Fig. 13**. The objec-

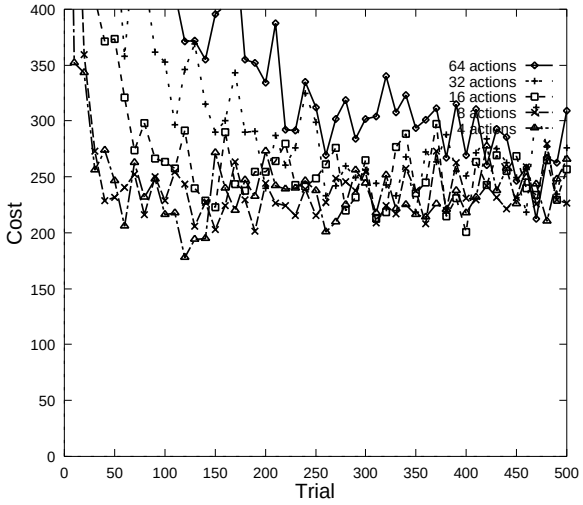


Fig. 11 Performance on Q-learning using ϵ-greedy policy averaged over 100 trials. $\epsilon = 0.1$, learning rate $= 0.5$.

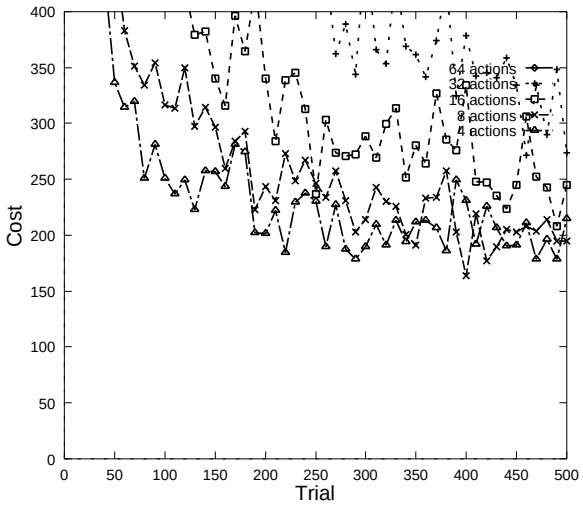


Fig. 12 Performance on SARSA(λ) using ϵ-greedy policy averaged over 100 trials. $\epsilon = 0.1$, learning rate $= 0.1$, $\lambda = 0.9$. Note that this algorithm costs the memory exactly the same size as our method's.

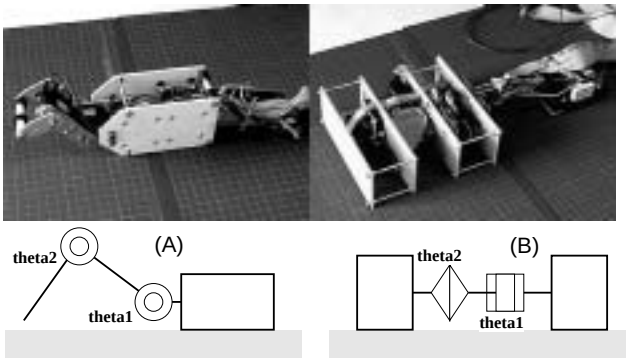


Fig. 13 Real two robots and their models. The Robot-A has a two-link arm. The Robot-B consists two boxes with an actuated joint which bends and twists the body.

tive of the learning is to find effective behavior to move forward. The learning agent interacts with the robot (i.e., the environment) as below:

(1) The agent observes the angles of the joints θ_1, θ_2 as a state input.

(2) The agent selects a destination of angular positions of the joints a_1, a_2 as an action output.

(3) The robot moves each joint to the destination.

(4) After about 0.2 sec, distance of the movement is measured and given to the agent as immediate reward.

(5) Go to step 1.

The angles of the joints θ_1, θ_2 as the state input and a_1, a_2 as the action output are arranged to take discrete value from 0 to 7. The reward signal also takes discrete value from -128 to 127. When the body does not move, the reward is 0.

Each joint is drived by a hobby-use servo motor that reacts to angular position commands. The angular position command is used as the current angle of the joint. Therefore, when the agent selects destination angles that is largely different from the latest angles (the difference is over 3), hidden state problem is occured quite often because the slow response of the motor cause errors between the true angle of the joints and the state input signal.

Since the agent selects action from the eight choices in each joint, the number of actions are $8 \times 8 = 64$ in the space of the two joints. In the flat action selection method, the agent deals with all the 64 actions equally. In the binary action selection method, the agent selects upper or downward groups of the angles with respect to the joint 1 in the root node (the first layer), and next the agent selects upper or downward groups of the angles with respect to the joint 2 in the node of the second layer. In the 3rd layer, the agent selects again upper or downward groups of the angles with respect to the joint 1 where the element of the groupes are composed of the group that is selected in the root node. In the 4th layer, the agent selects again upper or downward groups of the angles with respect to the joint 2 where the element of the groupes are composed of the group that is selected in the second layer. Thus, the decisions for joint 1 and 2 are assigned alternately to the layers in the binary-tree. The better implementation may exist, however, this is one of the simplest way to make a tree structure for decision making in two-dimensional action space. The feature vector for the state input is generated by using the tile-coding shown in Fig. 6, since the state space is the same two dimensional as the puddle world.

In order to measure the distance of the movement, a

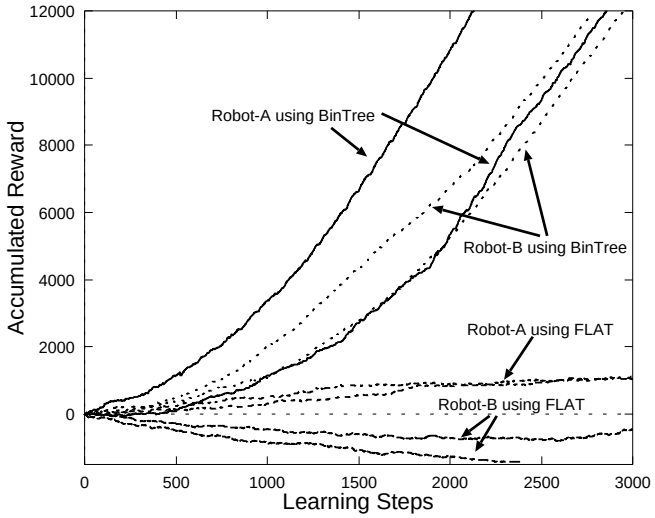


Fig. 14 Performances of actor-critic algorithms on the real robots.

wheel that has a diameter of 3cm is attached to the body, and turning full circle of the wheel generates 200 pulses. The reward signal is generated by the turning direction of the wheel as the positive or negative sign of the reward and by the number of pulses as the magnitude of the reward. Thus, 1,000 pulses are equivalent to about 50 cm. The experiment is executed in 3,000 steps, that takes about 8.5 minutes. This learning problem does not have any particular goal states such as in the puddle world, therefore we set the discount rate $\gamma = 0.9$. For practical use, the learning rate is set to somewhat large $\alpha_\pi = 0.1$. The other parameters are the same as in the puddle world, $\alpha_v = 0.1$, $\lambda_v = 0.9$ and $\lambda_\pi = 0.9$.

Fig. 14 shows learning curves of accumulated reward, that is, the distance of the movement from the initial position. Although the mechanics of the robot A is quite different from the robot B's, the proposed algorithm can learn smoothly in both robots. It is noteworthy that although the number of all the state-action pairs are $64 \times 64 = 4092$ since state space is $8 \times 8 = 64$ and actions are $8 \times 8 = 64$, the binary-tree approach obtains appropriate behavior with less than the half of 4092 steps. On the other hand, although the learning algorithms and parameters are the same as the binary tree approaches, the flat action selection approaches cannot learn at all. The difference of the performance between the binary tree and flat approaches is outstanding than the performance in the puddle world.

6. Discussion

Although the learning problem is very simple in the puddle world, the performance of the flat action-selection approach is strongly affected with the number of similar actions. On the other hand, the binary tree approach is not affected with the number of actions when the actions are arranged according to similarity. It is noteworthy that the performance of the random tree approach is the same or the better than the flat approaches in the case that the actions are randomly labeled. The reason of this result is considered that random groups that are not arranged according to similarity with using the hierarchical decision making is still effective in the problems that have many similar actions. The binary tree approaches will be weak when an optimum action exists in an action group where most of the actions have poor action-values. The reason is that the nodes in higher layers would not choose such an action group that the average of action value is poor even if an optimum action exists in that group. However, such an ill-natured problem is rare in real tasks. Replacing the flat action selection with the binary tree action selection achieves not only the effective learning but also to omit parameter tuning that is coming from the number of actions.

In the real robot problems, the difference of the performance between the binary tree and flat approaches is outstanding. One reason of this result would be that the action space is two-dimensional while it is one-dimensional in the puddle world. The analysis of the relation between the number of the action dimension and the learning performance is a future work. In this paper, we demonstrate that the proposed approach works effectively in the environments where the actions are arranged according to similarity in only one or two dimensional space. However, the performance of the hierarchical aproach would be strongly affected with the assignment of the action dimension to the decision layer in higher dimensional action space. It is possible to arrange independent binary trees for each action dimension. Finding an appropriate structure of the binary tree would also be a learning problem to solve. The design of the binary tree action selector for high dimensional action space is a future work.

7. Conclusion

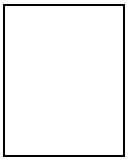
We proposed a binary-tree stochastic policy representation, that is a generalization technique of (or using) action space for probability of action selection. The efficiency of our method is owing to the property of the environments that neighboring actions will almost result in similar state transitions. The computational expense for selectiong action and finding eligibilities is quite little since the agent

can perform it using only local information. This feature is advantageous to the implementation for large problems. Replacing the flat action selection with the binary tree action selection achieves not only the effective learning but also to omit parameter tuning that is coming from the number of actions. The proposed action selector is a theoretically sound and simple stochastic-policy representation. Therefore, it can be applied to all the policy gradient methods such as actor-critic, REINFORCE, or VAPS algorithms.

References

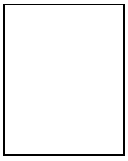
1) Barto, A.G., Sutton, R.S. & Anderson, C.W.: Neuronlike adaptive elements that can solve difficult learning control problems, *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC–13, no.5, pp. 834–846 (1983).
2) Takanori Fukao, Norikatsu Ineyama and Norihiko Adachi: Reinforcement Learning Using Regularization Theory to Treat the Continuous States and Actions, Trans. of Institute of Systems, Control and Information Engineers, Vol.11, No.11, pp.593–599 (1998) in Japanese.
3) Horiuchi, T., Fujino, A., Katai, O. & Sawaragi, T.: Fuzzy Interpolation-Based Q-learning with Continuous Inputs and Outputs, Transactions of the Society of Instrument and Control Engineers, Vol.35, No.2, pp.271–279 (1999) in Japanese.
4) Kimura, H. & Miyazaki, K. & Kobayashi, S.: A Guideline for Designing Reinforcement Learning Systems, Journal of the Society of Instrument and Control Engineers, Vol.38, No.10, pp.618–623 (1999) in Japanese
5) Kimura, H. & Kobayashi, S.: An Analysis of Actor/Critic Algorithms using Eligibility Traces: Reinforcement Learning with Imperfect Value Functions Journal of Japan Society of Artificial Intelligence, Vol.15, No.2, pp.267–275 (2000) in Japanese.
6) Peshkin, L. & Meuleau, N. & Kaelbling, L. P.: Learning Policies with External Memory, *Proceedings of the 16th International Conference on Machine Learning*, pp.307–314 (1999).
7) Sutton, R.S.: Generalization in reinforcement learning: Successful examples using sparse coarse coding, *Advances in Neural Information Processing Systems 8 (NIPS8)*, pp.1038–1044 (1996).
8) Sutton, R. S. & Barto, A.: Reinforcement Learning: An Introduction, *A Bradford Book*, The MIT Press (1998).
9) Sutton, R. S., McAllester, D., Singh, S. & Mansour, Y.: Policy Gradient Methods for Reinforcement Learning with Function Approximation, *Submitted to Advances in Neural Information Processing Systems 12 (NIPS12)*, (2000).
10) Tsitsiklis, J. N., & Roy, B. V.: An Analysis of Temporal-Difference Learning with Function Approximation, IEEE Transactions on Automatic Control, Vol.42, No.5, pp.674–690 (1997).
11) Watkins, C. J. C. H. & Dayan, P.: Technical Note: *Q*-Learning, *Machine Learning 8*, pp.279–292 (1992).
12) Williams, R. J.: Simple Statistical Gradient Following Algorithms for Connectionist Reinforcement Learning, *Machine Learning 8*, pp. 229–256 (1992).

Hajime Kimura (Member)



He received Ph.D. degree in intelligence science from Tokyo Institute of Technology in 1997. He was empolyed as a PD researcher in Japan society for the promotion of science in 1997. He belonged to the interdisciplinary graduate school of science and engineering in Tokyo Institute of Technology as a research associate from 1998 to 2004. He has been an associate professor of the graduate school of engineering of Kyushu university. The research interest is artificial intelligence, especially reinforcement learning.

Shigenobu Kobayashi (Member)



He received Ph.D. degree in decision science and technology from Tokyo Institute of Technology in 1974. He joined the department of controll engineering in Tokyo institute of Technology as a research associate in 1974. He moved to the interdisciplinary graduate school of science and engineering in Tokyo Institute of Technology as an associate professor in 1981, and is currently a professor there since 1990. His research interest is algorithms, inferences, controll and machine learning.

Reprinted from Trans. of the SICE
Vol. 37 No. 12 1147/1155 2001