Induced states in a decision tree constructed by Q-learning

Kao-Shing Hwang\textsuperscript{a,}\textsuperscript{*}, Yu-Jen Chen\textsuperscript{b}, Wei-Cheng Jiang\textsuperscript{b}, Tsung-Wen Yang\textsuperscript{b}

\textsuperscript{a}Department of Electrical Engineering, National Sun Yat-sen University, Taiwan
\textsuperscript{b}Department of Electrical Engineering, National Chung Cheng University, Taiwan

\textbf{A B S T R A C T}

This paper develops a tree-construction method based on the framework of reinforcement learning (RL). The induction of a decision tree is regarded as a problem of RL, where the optimal policy should be found to obtain the maximal accumulated information gain. The proposed approach consists of two stages. At the first stage, the emulation/demonstration stage, sensory-action data in a mechatronic system or samples of training patterns are generated by an operator or stimulator. The records of these emulation data are aggregated into components of the state space represented by a decision tree. State aggregation for decartelization of a state space consists of two phases: split estimation and tree growing. In the split estimation phase, an inducer estimates long-term evaluations of splits at visited nodes. In the second phase, the inducer grows the tree by the predicted long-term evaluations, which is approximated by a neural network model. At the second stage, the learned behavior or classifier is shaped by the RL scheme with a discretized state space constructed by the decision tree derived from the previous stage. Unlike the conventional greedy procedure for constructing and pruning a tree, the proposed method casts the sequential process of tree induction to policy iterations, where policies for node split are evaluated and improved repeatedly until an optimal or near-optimal policy is obtained. The splitting criterion regarded as an action policy is based on long-term evaluations of payoff instead of using immediate evaluations on impurity. A comparison with CART (classification and regression tree) and C4.5 on several benchmark datasets is presented. Furthermore, to show its applications for learning control, the proposed method is applied further to construct a so-called tree-based reinforcement learning method, where the mechanism works with a discrete state space derived from the proposed method. The results show the feasibility and high performance of the proposed system as a state partition by comparison with the renowned Adaptive Heuristic Critic (AHC) model.

© 2012 Elsevier Inc. All rights reserved.

\textbf{1. Introduction}

The process of modeling a target system by observing the given input-output data pairs and then establishing the mapping relationship is generally referred to as emulation/demonstration learning. The purposes of this type of learning are: (1) to predict a system’s output when seeing new input data; (2) to extract the relationship between the input and output of the system; and (3) to develop a more sophisticated controller derived this rapid prototype [1]. To process the observations from emulation/demonstration into experiences, an appropriate repository structure is needed to represent its different characteristics. Currently, numerous structures have been proposed to fit various needs, such as neural networks [31] and decision trees [19,20,24,30]. In some cases, for example data mining, interpretability and the ability to understand the learned model

\textsuperscript{*} Corresponding author.
\textit{E-mail address:} hwang@ccu.edu.tw (K.-S. Hwang).

0020-0255/5 - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2012.06.009
is important [10]. Decision trees often yield simpler solutions that are easier to interpret than neural networks since decision trees can be transformed into sets of nested premises and consequences to improve human readability.

Decision tree clustering is a technique that uses tree structures to solve pattern classification or identification problems. Decision trees contain internal and leaf nodes. Each internal node runs a test to determine the child node that will be the next to be visited. The process repeats until a leaf node associated with the numerical values or equations is reached. Therefore, given an input, the corresponding output can be calculated by traversing a decision tree from the root to a leaf node which contains the most appropriate response to the input. In terms of state aggregation, the receptive fields are grouped into exhaustive partitions of the input space. Each such partition is called a node split, and each element of the partition is represented by a leaf node, each of which is assigned a predicted output value or function.

The process of constructing a decision tree, also known as tree induction, can be viewed as the problem of how to classify features in a decision tree. In general, tree induction is a recursive procedure that assigns an optimal test to each visited node. It has been shown that the problem of constructing optimal decision trees that have a minimal number of tests required to classify an unknown sample is an NP-complete problem [12]. Most tree induction algorithms are typically based on a top-down greedy strategy that always makes locally optimal decisions at each node. Nevertheless, due to the greedy and local nature of such decisions, an optimal decision at the current node may result in more subsequent tests than a non-optimal one. This is because a greedy decision is near-sightedely attracted by a better imminent benefit than a mediocre choice with much better payoff but more distant in time.

Some top-down algorithms such as U-tree and G-learning, usually retain historical information and occasionally perform batch processing to determine what states (tree nodes) need to be split by evaluating the Kolmogorov-Smirnov test and sum-squared error [19,26]. These methods have been used in conjunction with dynamic programming methods or Q-learning, to estimate the value function across the state space. Briefly speaking, these algorithms focus more on handling discrepancy of data types; for example, G-learning handles binary data, U-Trees handle discrete data, and Pyeatt and Howe’s approach handles continuous data. Moreover, these methods always induce a tree with deeper layers since the splitting decision is based on the temporal estimated evaluation instead of the true value reflecting the reality of current situations. Therefore, various ways of evaluating in various ways for tree induction and implementation has been a major focus for improving decision tree performance [25]. Although non-greedy approaches such as look-ahead methods have been adopted by some investigators [7,18], these methods still cannot ensure the production of better structures of decision trees.

In this paper tree induction problems are regarded as a sequence or process of decisions that can be tailored into the framework of policy search in RL. The proposed self-organizing decision tree induction method aims to tackle the greedy problems occurring in proceeding tree induction methods. In the proposed method, split selection considers not only the local evaluation-impurity calculation, but also the accumulated evaluations before a split is selected. In other words, it applies a reinforcement learning technique to the process of tree construction, which is assumed to be episodic. That is, if an unsatisfactory tree structure is obtained, the process will be set back to the initial condition and start over again. Therefore, the tree structure is refined by experiences through repeated processes of policy iterations on split, instead of one-shot induction.

The proposed tree induction method consists mainly of two tasks. In the first task, the inducer traverses the subtree space to update long-term evaluation estimates of splits at each visited node. The update procedure transfers the local evaluation back to the preceding subtrees. In the second task, nodes proliferate by a split carrying a maximal long-term evaluation. The concise structure of the induced tree can be further applied to the problem of classifications, such a pattern clustering, recognition, or state decoder. In this paper, the proposed method is embedded into an actor-critic model as a state decoder to improve learning capability of the original model. Benefitting from the self-organizing attributes of the proposed method, the problem of uneven density of state occurrences in a large state space encountered by an actor-critic model is hence tackled. Generalization among states can also be achieved to certain degree. This paper is organized as follows. In Section 2, basic knowledge and techniques of decision tree and reinforcement learning is briefly described. Section 3, depicts details of the proposed self-organizing decision tree method based on reinforcement learning. The experimental results for clustering several benchmark data sets are presented in Section 4. In Section 5, comparisons are made between the tree-based actor-critic and Adaptive Heuristic Critic (AHC) based on a series of simulation results. Finally, conclusions are drawn in Section 6.

2. Decision tree

A decision tree is a tree structure that can approximate a function from a given labeled data set. It consists of two kinds of nodes: internal nodes and leaf nodes. Each internal node contains a test that determines which child node should be visited. Each leaf node has a predicted label, value, or equation for the input patterns falling in the node. Given an input pattern, its output is determined as follows. Starting from the root node, it traverses down the tree branch to a child node according to the outcome of the test. This process is then repeated until a leaf node is reached. The output is the label or value of the visited leaf node.

Decision Tree Induction is a procedure that constructs an optimal decision tree from a given labeled data set. Two of the common optimality criteria are minimum error rate and minimum number of leaf nodes. It has been shown that
constructing an optimal decision tree that requires minimal tests is an NP-complete problem [12]. Thus finding efficient heuristics for constructing near-optimal decision trees is necessary.

Top-down induction way is a typical algorithm for the decision tree induction, which contains two procedures, growing and pruning. The tree growing procedure is based on a recursive process, known as “divide and conquering” [23]. At each recursive iteration, the inducer considers the partition of the data set that corresponds to a node and assigns an appropriate split for the node according to some splitting measures. The data set is then divided into two smaller subsets by the split. Two child nodes are also created to represent the subsets. The tree growing procedure is then applied to the child nodes recursively. Top-down approaches involve three tasks: (1) selection of a splitting criterion, (2) the deciding which nodes are terminal, and (3) the assignment of each terminal node to a label value.

There are various selections for the splitting criterion. The typical splitting criterion, impurity evaluation, is described in the following. The measure of the performance of a split \( \tau \) is defined as the impurity change due to the split. The impurity change is expressed as

\[
\Delta E(\phi, \tau) = E(\phi) - p_lE(\phi_l) - p_rE(\phi_r),
\]

where \( \phi \) is the node being split; \( E(\phi) \) is the impurity of the node \( \phi \); \( E(\phi_l) \) and \( E(\phi_r) \) are the impurities of left and right branch nodes; and \( p_l \) and \( p_r \) are the percentages of cases at node \( \phi \) that branch leftward and rightward, respectively. The optimal split \( \tau^* \) is a split that gives the largest decrease in the impurity as

\[
\Delta E(\phi, \tau^*) = \max_{\tau} \Delta E(\phi, \tau).
\]

In classification problems, the two most common impurity functions are the entropy function and the Gini index [2], as follows

\[
E(\phi) = E(p_1, \ldots, p_j) = -\sum_{j=1}^{J} p_j \cdot \ln(p_j)
\]

\[
E(\phi) = E(p_1, \ldots, p_j) = 1 - \sum_{j=1}^{J} p_j^2
\]

where \( J \) is the number of classes; \( p_j \) is the percentage of cases that belong to class \( j \) in the node \( \phi \). In regression problems, the impurity function is usually taken as the squared error of fitting the local model to the data set in the node:

\[
E(\phi) = \min \sum_{i=1}^{N_{\phi}} (y_i - f(x_i, \phi))^2
\]

where \( x_i, y_i \) is a labeled data point in the node \( \phi \); \( f(x_i, \phi) \) is a local model for node \( \phi \); \( N(\phi) \) is the number of the data points in the node \( \phi \). However, the decision tree should grow trivially according to the previous procedure. In CART and C4.5, the decision tree is pruned using an acceptable error rate.

3. Reinforcement learning

3.1. Reinforcement learning problem

The reinforcement learning problem deals with a mode of learning from an interaction to achieve a goal. The problem contains two components: the agent and the environment [4,5,9,13,14,32]. The agent interacts with the environment at each sequence of discrete time steps. At each time step \( t \), the agent observes a representation of the environment’s state, \( s_t \in S \), where \( S \) is the set of all possible states, and it selects and takes an action, \( a_t \in A(s_t) \) based on a policy \( \pi_t \), where \( A(s_t) \) is the set of actions available in state \( s_t \). Later the environment changes to a new state \( s_{t+1} \) and sends a feedback called reward, \( r_{t+1} \in \mathbb{R} \) to the agent.

Policy \( \pi_t \) is a mapping from states to probabilities of selecting each possible action. The reward feedback is a means of telling the agent what it should achieve, but not how to achieve it. The objective of learning can be said to be maximizing the expected return \( R_t \). The return \( R_t \) is defined as

\[
R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \sum_{i=0}^{T} \gamma^i r_{t+i+1},
\]

where \( \gamma \) is a discount factor, \( 0 \leq \gamma \leq 1 \). The discount factor determines how the future rewards affect the agent’s behavior. If \( \gamma < 1 \), the infinite sum will have a finite value when \( T = \infty \). If \( \gamma = 0 \), the agent will be greedy in the sense that it is concerned only with maximal immediate rewards. As \( \gamma \) approaches 1, the agent will consider future rewards more strongly.

Reinforcement learning method is based on approximated value functions to estimate how good it could be for an agent to be in a given state [11,15,27,29]. There are two kinds of value functions: the state-value function \( V^\pi(s) \) and the action-value function \( Q^\pi(s, a) \). The state value \( V^\pi(s_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \) is defined as the expected return when starting from \( s_t \) and following policy \( \pi \) thereafter. The action value \( Q^\pi(s_t, a_t) = r_{t+1}(s_t, a_t) + \sum_{i=1}^{\infty} \gamma^i r_{t+i+1} \) is defined as the expected return when starting from \( s_t \), taking action \( a_t \) and following policy \( \pi \) thereafter.
Solving a reinforcement learning problem is to find a policy that receives a maximum return in the long run. The optimal policy \( \pi^* \) is defined as \( V^*(s) \equiv V^*(s) \) for all \( s \in S \) and \( \pi' \). The optimal state-value and action-value functions are then defined as

\[
V^*(s_t) = \max_\pi V^*(s_t), \quad Q^*(s_t, a_t) = \max_\pi Q^*(s_t, a_t)
\]

Note that we can write \( Q^* \) in terms of \( V^* \) as

\[
Q^*(s_t, a_t) = r(s_t, a_t) + \gamma V^*(s_{t+1}).
\]

3.2. Reinforcement learning algorithm

(1) **Q-learning**: This is one of the best-known reinforcement learning algorithms, and it learns the optimal action-value function \( Q^*(s, a) \) [28]. The action–value function maps each state-action pair to a real value that is the maximum accumulative reward that can be received from taking action \( a \) in the state \( s \) to the future. The optimal policy is determined from the optimal action-value function as: \( \pi^* (s) = \arg \max_a Q^*(s, a) \).

Given that the optimal policy is followed after action \( a_t \) is taken, \( Q^*(s_t, a_t) \) is the immediate reward of taking that action from state \( s_t \) plus the maximum action-value of the next state. Therefore, the action-value function satisfies \( Q^*(s_t, a_t) = r(s_t, a_t) + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a) \). Temporal Difference (TD) is used to incrementally construct the action-value function as:

\[
Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha (r(s_t, a_t) + \gamma \max_{a_{t+1}} Q_t(s_{t+1}, a) - Q_t(s_t, a_t)).
\]

(2) **Adaptive Heuristic Critic (AHC)**: AHC is a well-known actor-critic model, which consists of two adaptive elements: the adaptive critic element (ACE) and the associative search element (ASE) [3]. ASE implements a policy function \( \pi(s_t) \) that maps each state vector \( s_t \) into an action, whereas ACE represents a state-value function \( v(s_t) \) that evaluates the goodness of each state vector. Fig. 1 shows the learning system structure.

The state variables of the environment are quantized into an \( n \)-component binary vector \( \overline{X}(t) = (x_1(t), x_2(t), \ldots, x_n(t)) \) by a decoder. The ASE contains two kinds of weights: the long-term trace weight \( w_l \) that determines a control action and the short-term trace weight \( \hat{w}_s \), which is used to update the long-term trace. The ASE’s output \( Y(t) \) is determined from \( \overline{X}(t) \) as

\[
\text{net} = \sum_{i=1}^{n} w_l(t)x_i(t)
\]

\[
Y(t) = \text{sign}(\text{net} + \text{noise}(t)) = \begin{cases} 1 & \text{if } \text{net + noise}(t) \geq 0 \\ -1 & \text{if } \text{net + noise}(t) < 0 \end{cases}
\]

The update rules of ASE are expressed as \( \hat{w}_s(t + 1) = \gamma \hat{w}_s(t) + (1 - \gamma)Y(t)x_i(t) \) and \( w_l(t + 1) = w_l(t) + \alpha \hat{w}_s(t) \hat{e}_i \), where \( \delta \) is the ASE trace decay rate, \( \alpha \) is the positive learning rate, \( \hat{r}(t) \) is the internal reinforcement signal, and \( \hat{e}_i(t) \) is the eligibility function at time \( t \) of input pathway \( i \). The trace \( \hat{e}_i \) stores information both about how long ago the input path \( i \) was triggered and also about what action was taken. The internal reinforcement signal \( \hat{r}(t) \) is provided by the ACE. The output of the ACE is the inner product of the quantized state vector and the state value vector \( v_l \) expressed as \( \hat{r}(t) = \sum_{i=1}^{n} v_l(t)x_i(t) \). The internal reinforcement signal is calculated by \( \hat{r}(t + 1) = (r(t + 1) + \gamma \hat{r}(t + 1)) - p(t) \), where \( \gamma \), \( 0 < \gamma \leq 1 \), is a positive constant called the discount factor. The update rules for ACE are expressed as:

\[
x_i(t + 1) = \lambda x_i(t) + (1 - \lambda)x_i(t)
\]

\[
v_l(t + 1) = v_l(t) + \beta \hat{r}(t)x_i(t),
\]

where \( \lambda \) is the ACE trace decay rate and \( \beta \) is the positive learning rate.

In this work, in addition to developing a self-organizing decision tree method where tree induction is conducted through repetitive reinforcement learning, the proposed method is applied to the framework of a critic-actor model, so-called tree-based reinforcement learning. In this type of learning, state decoding for state space partitioning of the model is executed by the tree learned, to improve the learning performance.

![Fig. 1. AHC learning system structure (redrawn from [3]).](image-url)
4. Decision tree induction through reinforcement learning

The proposed self-organizing decision tree method consists of two phases: split estimation and tree growing. In the first phase, the incremental tree induction procedure, where new training samples become available over time [8], is modeled as a reinforcement learning problem. The inducer traverses the subtree space to estimate the long-term evaluations of the splits of the visited nodes. In the second phase, the inducer merely grows the tree by choosing the split resulting in the maximum long-term evaluation among all available splits such that there is no pruning executed for the constructing tree. The next sections present detail on how decision tree induction can be modeled as a reinforcement learning problem and how the key elements of the reinforcement learning problem are formulated in.

4.1. Top-down tree induction as decision processes

Top-down tree induction is a procedure that recursively partitions the input space into non-overlapping subspaces. Starting from the root node that contains all the data, the inducer makes sequential decisions about how to split each visited node into two child nodes. A slightly different description of the induction procedure is as follows. A repository keeps the subtrees that have been visited but not yet assigned a split by the inducer. In an iteration, the inducer takes a subtree $T$ from the repository and specifies a split for the root node of the subtree or terminates the subtree as a leaf node. One of the child subtrees caused by the split is pushed into the repository. The other one is taken for the next iteration. The process is repeated until the repository is empty.

A subtree space is defined as a space that contains all possible subtrees. Fig. 2a illustrates a subtree space that the inducer traverses in order to specify a test for each visited node sequentially. Each circle represents a subtree. Each triangle represents an available split of a subtree, and the upper subtree is partitioned into the two following exclusive subtrees. In the figure, the inducer starts from the node $\phi_i$ and chooses a split $\tau_i$ among the available split set $\{\tau_i \mid 1 \leq i \leq N_{\text{split}}(\phi_i)\}$, where $N_{\text{split}}(\phi_i)$ is the number of available splits in the node $\phi_i$. Then the inducer moves down to one of the child subtrees, $\phi_i + 1$. The other one is pushed into the repository for later processing. The thick line presents the path through which the inducer traverses. Fig. 2b shows the subspaces that correspond to the subtrees. Dotted rectangles denote subspaces. The dashed line across the subspace denotes the split that partitions the subspace into two smaller subspaces.

4.2. Reinforcement learning model

By representing the tree induction procedure as decision processes, tree growing can be modeled as a reinforcement learning problem. Fig. 3 shows the functional structure of decision tree growth with reinforcement learning mechanism in the decision tree induction problem. The agent receives the state representation $s_t$ of the current visited node $\phi_t$ and chooses a split $\tau_t$ from the available split set according to the policy $\pi_t$. The split divides the node into two child nodes, $\phi_k$ and $\phi_r$. A local evaluation $r_{t+1}$, called a reward, for the split corresponding to the next state $s_{t+1}$ is received. The agent moves randomly to one of the child nodes and traverses the subtree space of the child node continually until it reaches a node that satisfies the following stopping conditions: (1) the data size is smaller than a threshold; (2) the depth of the node is greater than a threshold; or (3) the inducer meets a terminal node.

The decision tree learning consists of several episodes. An episode always starts from the state that contains all the training data and ends at the state that satisfies the stopping conditions. A subtree may be visited several times in order to gather the estimation of the splits. The key elements in the reinforcement learning problem are formulated as follows:

![Fig. 2. A subtree space that the inducer traverses.](a) (b)
4.2.1. State representation

State representation is a mapping from subtrees to state vectors. The root node of each subtree corresponds to a subspace that encloses the data set of the node. A subspace is defined as \( \{ x \in \mathbb{R}^{N_{\text{attr}}} | l_i \leq x_i \leq u_i, i = 1 \sim N_{\text{attr}} \} \) where \( N_{\text{attr}} \) is the number of attributes for the data set. Two diagonal points of a subspace are denoted as \( x^a \) and \( x^b \), which can determine the bounding of the subspace by projecting orthogonally onto the hypercubic of the space, respectively. The state vector of a subtree \( s \) contains a no-split action and all possible splits at the state. The no-split action is similar to the stopping criterion. It assigns a node to be a terminal node. Typically the splitting point is the center between two adjacent data points picked up at random. For a data set of size \( M \), the number of available split for a numerical variable is less than or equal to \( M-1 \). For simplicity, the \( j \)-th split value \( \tau_q \) of the \( i \)-th numerical variable, \( x_i \), at a node is formally defined as \( \tau_q = x^a_i + (x^b_i - x^a_i) j / (N_{\text{split}} + 1) \) where \( j = 1 \sim N_{\text{split}} \); \( N_{\text{split}} \) is the number of splits of a single variable.

4.2.2. Action set

The action set \( A(s) \) of state \( s \) contains a no-split action and all possible splits at the state. The no-split action is similar to the stopping criterion. It assigns a node to be a terminal node. Typically the splitting point is the center between two adjacent data points picked up at random. For a data set of size \( M \), the number of available split for a numerical variable is less than or equal to \( M - 1 \). For simplicity, the \( j \)-th split value \( \tau_q \) of the \( i \)-th numerical variable, \( x_i \), at a node is formally defined as \( \tau_q = x^a_i + (x^b_i - x^a_i) j / (N_{\text{split}} + 1) \) where \( j = 1 \sim N_{\text{split}} \); \( N_{\text{split}} \) is the number of splits of a single variable.

4.2.3. Reward function

The reward function \( r(s, a) \) is designed as the impurity change due to split \( a \) when the action is a split action. The evaluation is used in several tree induction methods. The main difference is that in the reinforcement learning, the inducer will choose the split with the maximum return instead of the maximum local evaluation, i.e. reward. Note that the return is the accumulative reward received in the long run. The reward function \( r(s, a) \) is formally defined as

\[
r(s, a) = \begin{cases} 
\Delta E(s, a), & \text{if } a \text{ is a split action} \\
-E(s), & \text{if } a \text{ is a no-split action}
\end{cases}
\]

where \( E(s) \) is the impurity function at the terminal state \( s \), and \( \Delta E(s, a) \) is the impurity change due to the split \( a \) in state \( s \).

4.2.4. Action-Value function

The action-value function \( Q^\pi(s, a) \) is defined as the action value by taking action \( a \) in state \( s \) under a policy \( \pi \). In this case, the action-value function is equal to the reward plus the sum of maximum action-values of the child nodes under policy \( \pi \). We redefine the action-value function as:

\[
Q^\pi(s, a) = r(s, a) + \gamma \sum_{i \in \{1 \sim \tau_q \}, a' \in A(s)} p_i Q^\pi(s'_i, a')
\]

where \( s'_i \) corresponds to the state vectors of the child nodes caused by the split \( a \); \( p_i \) is the percentages of cases at state \( s'_i \), and \( \gamma \) is the discount factor. Since the objective of the proposed RL method is to construct an accurate and small decision tree, the influence of future rewards on splitting should be equivalent to the one of the immediate reward. We adopt the Q-learning method to learn optimal action-value function. The updating rule is expressed as

\[
Q_{t+1}(s, a) = (1 - \alpha) Q_t(s, a) + \alpha \left[ r(s, a) + \gamma \sum_{i \in \{1 \sim \tau_q \}, a' \in A(s)} p_i \max_{a'} Q_t(s'_i, a') \right]
\]

The split estimation phase and tree growing phase are shown in Figs. 4 and 5, respectively.
5. Experiment

The experiment compares the performance of the proposed method, CART [2], and C4.5 [22]. CART is a top-down heuristic method that consists of two phases: tree growing and tree pruning. In our CART implementation, the splitting criterion is based on the entropy function, and the pruning method is based on the minimum cost-complexity principle described in [1].

The methods are evaluated on three benchmark datasets from the UCI Machine Learning Repository [17]. Each dataset is divided into two categories, one for tree induction and the other for calculating the error rate for performance measurement by tree induction methods, such as CART, C4.5, and the proposed method. In other words, each decision tree induced by the aforementioned methods is constructed from the training set and then is tested on the test set. We use the ten-fold cross-validation procedure to estimate the error rate. The threshold of data size of the proposed method is set to 5, and the maximum depth of the proposed decision tree is 50. We compare the classification accuracy and trees sizes of these methods. The size of a decision tree is the number of its leaves. The error rate is the number of samples that are classified into wrong classes over all samples. The results of datasets are listed in Table 1. The C4.5 result is taken from Ref. [16]. Compared with the other methods, the proposed RL method, induced many fewer nodes in a tree without deteriorating the error rate. Although the error rate in the StatLog vehicle silhouette by the RL method is slightly worse than the ones produced by the other methods, the result is acceptable when considering the resource of memory consumption.
6. Simulations by tree-based critic-actor model

The objective of the tree-based critic actor model is to apply the method to a critic-actor model algorithm to achieve automatic state partitioning for state value approximation. The state space is partitioned into boxes of different sizes by a decision tree, which is constructed by the RL decision tree induction during the learning procedure. The sensory input from the environment is mapped to a leaf node by the decision tree.

Two concepts of states, the sensory input denoted as $I$ and the discrete state denoted as $s$ corresponding to a leaf node in the self-organizing tree, have been defined in [26]. In other words, a discrete state is transformed from a sensory input by a decision tree. The method is executed in two repeated cycles: data gathering and tree refinement [21]. Each leaf node is associated with a pair of actor-critic traces denoted by $w_i$ and $v_i$. At the beginning of learning, the initial sizes of subspaces may be large, and in this situation, the agent may take several steps to cross a subspace to another neighbor. If the weights are updated at each time step with the state of learning agent staying at the same leaf node, an undesired problem arises. The internal reinforcement signal is defined at time $t$ as

$$\hat{r}(t) = r(t+1) + \gamma p(t+1) - p(t)$$

where $\gamma$ is a discount rate and $p(t)$ is the state estimating value at time $t$.

When the state change does not occur two successive times, the trace activation remains the same, such that $p(t) = p(t+1)$ and the internal reinforcement signal becomes $\hat{r}(t) = r(t+1) + \gamma p(t) - p(t) = r(t+1) + (\gamma - 1)p(t)$. In this case, if $r(t+1)$ is zero, then $\hat{r}(t) = (\gamma - 1)p(t)$. It should be noted that a positive $\hat{r}(t)$ represents the current action that seems better than previously expected. This situation misleads the direction of adjustment of the action policy.

Therefore, the weights $w_i$ and $v_i$ are modified only when the state of an agent falls on a different leaf. Consequently, the calculation of the internal reinforcement learning is described as follows. Assume the agent enters a state $s$ at time step $t$, and then leaves the state to enter another state $s'$ at time step $t+k$. The sequence $(l_t, l_{t+1}, \ldots, l_{t+k})$ is the sensory inputs received from time step $t$ to time step $t+k$. The internal reinforcement signal at time $t+k$ considering the average sojourn time in the state [6] is defined as

$$\hat{r}(t+k) = \sum_{i=0}^{k-1} V(l_{t+i})/k - p(t)$$

where

$$V(l_{t+i}) = \begin{cases} r(t+i) + \gamma V(l_{t+i+1}) & \text{for } i = 0 \text{ to } k-2 \\ r(t+k-1) + \gamma V(l_{t+k}) = r(t+k-1) + \gamma V(s') & \text{for } i = k-1 \end{cases}$$

The data gathering cycle is depicted in Fig. 6. After the gathering cycle, the tree refinement cycle described in Section 3 is then performed. Completion of the data gathering cycle and gathering cycle are called an epoch, which should be repeated to perform until the end of the learning task.

The control problem of a cart-pole balance system depicted in [3] is simulated for comparison. Without a priori knowledge of the dynamics of the cart-pole system, the proposed approach learns to balance the pole as long as possible. The simulation consists of a sequence of trials. Each trial begins at the initial state $x = 0, \dot{x} = 0, \theta = 0, \dot{\theta} = 0$, and ends with a failure signal when $\theta$ exceeds the range of $[-12, 12]$ or $x$ leaves the interval of $[-2.4, 24]$. The results of simulations are shown in Fig. 7, where the $x$-axis denotes the number of trials and the $y$-axis denotes the number of steps. The arrows point at the ends of the epochs. In addition, an epoch is terminated while the rate of weight change is less than a certain threshold. At the very beginning, the tree is constructed by dividing the state space evenly into nine partitions for the data gathering phase of the first epoch. All the trace variables are reset to zero at the start of each trial. An epoch consists of a data gathering phase, where the cart-pole is eventually under sufficient control to avoid the failure of balancing, and a tree refinement process occurs right after the end of an epoch as failure occurs. In other words, the tree refinement process is executed before proceeding to the data gathering phase of the next epoch. The upper figure in Fig. 7 shows all trails in the simulation, and the lower one shows the first 34 trails. To save memory in the recording data, the sampling rate is set to be one sample for every 100 steps; that is, each tic on the $y$-axis represents 100 control cycles. The arrows indicate the end of the epochs. The first epoch ends at time step 11 since the weight updating falls below the threshold, and a new epoch begins immediately after

<table>
<thead>
<tr>
<th>Datasets</th>
<th>RL method</th>
<th>CART</th>
<th>C4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Error rates</td>
<td>Tree sizes</td>
<td>Error rates</td>
</tr>
<tr>
<td>BUPA liver disorder</td>
<td>0.301</td>
<td>19</td>
<td>0.4</td>
</tr>
<tr>
<td>PIMA Indian diabetes</td>
<td>0.207</td>
<td>40</td>
<td>0.27</td>
</tr>
<tr>
<td>StatLog vehicle silhouette</td>
<td>0.368</td>
<td>32</td>
<td>0.38</td>
</tr>
</tbody>
</table>

### Table 1

Experimental comparisons.
the tree refinement process. With a new state partition tree, the second epoch lasts until time step 21. Fig. 8 shows the averaged weights changes in each trial. The first two epochs are terminated due to a slow rate of weight change (update). The whole process of the simulation adjusts the tree twice before it succeeds in balancing the pole for a long period.

Compared to the outcomes obtained in [1], the number of nodes of refined trees in our simulation is always below thirty, and it is far less than 162 regions, which the AHC method clusters the state space into. Although the process goes through two refinement processes before obtaining a successful epoch, where it takes 17 trials to train the critic-actor model, the number of the accumulated trials from the beginning trial of the first epoch to the successful trial is 37, which is still less than the 75 trials in [3].

Fig. 6. Data gathering cycle.
7. Conclusion

A self-organizing decision tree induction method based on reinforcement learning is proposed, and the performance of the proposed method is compared with CART and C4.5 in the data clustering. This approach explores more compact state representation, accommodating the generalization over the input states, which is lacking in non-overlapping state space partitioning methods. Instead of applying the pruning process to generalization after tree growing, the self-organizing method constructs a tree based on the accumulated predicted evaluation associated with each internal node.

However, although a split with the maximal long-term evaluation value is taken to avoid the greedy problem; inappropriate selection of the reward function may still induce a non-optimal decision tree. Meanwhile, the mapping from sensory input to the agent's state representation can be many-to-many in a complex system environment. In other words, a single external node may represent multiple sensory inputs that will introduce the problem of perceptual aliasing. This always leads to failure in dexterous or precise control tasks, where high resolution for partitioning is possibly needed in only a small fraction of the state space. Therefore, the concept of neighborhood functions can be implemented for state representation in credit assignment problems of the actor-critic model.

References
