Proceedings of the 2006 IEEE/RSJ
International Conference on Intelligent Robots and Systems
October 9 - 15, 2006, Beijing, China

Heterogeneous and Hierarchical Cooperative Learning via Combining Decision Trees

Masoud Asadpour * , Majid Nili Ahmadabadi † , and Roland Siegwart ‡
* Autonomous Systems Lab
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland. masoud.asadpour@epfl.ch
†Control and Intelligent Processing Center of Excellence
ECE Dept., University of Tehran, Iran. mnili@ut.ac.ir
‡ Autonomous Systems Lab
Swiss Federal Institute of Technology (ETHZ), Zurich, Switzerland. rsiegwart@ethz.ch

Abstract— **Decision trees, being human readable and hierarchically structured, provide a suitable mean to derive state-space abstraction and simplify the inclusion of the available knowledge for a Reinforcement Learning (RL) agent. In this paper, we address two approaches to combine and purify the available knowledge in the abstraction trees, stored among different RL agents in a multi-agent system, or among the decision trees learned by the same agent using different methods. Simulation results in non-deterministic football learning task provide strong evidences for enhancement in convergence rate and policy performance.**

I. Introduction

The existing learning methods suffer from the curse of dimensionality, requiring a large number of learning trials as state-space grows. RL [1], despite its strength in handling dynamic and non-deterministic environments, is an example of such learning methods. On the other hand, agents in a multi-agent system –although dealing with more dynamic and as a result tougher learning task– confront with two rich sources of knowledge which, if appropriately exploited, could greatly expand the horizon of the learning process: *the acquired knowledge from subtasks*, and *the learned knowledge from colleagues*. If agents encounter a multitude of subtasks over their entire learning time, there is not only an opportunity to transfer knowledge between them, but also a chance to generalize subtasks and construct a hierarchy among them.

It is believed that, the curse of dimensionality can be lessen, to a great extent, by implementation of state abstraction methods and hierarchical architectures. Moreover, incremental improvement of agent's performance becomes much simpler. Most of the proposed approaches require a pre-designed subtask hierarchy (HAM [2], MaxQ [3]). In addition to requiring intensive design effort for explicit formulation of the hierarchy and abstraction of the states, the designer should, to some extent, know how to solve the problem before s/he designs the hierarchy and subtasks. To simplify this process, automated state abstraction approaches use decision-trees and incrementally construct the abstraction hierarchy from scratch [4] [5] [6].

Every learning agent in a multi-agent system constructs it's own abstraction hierarchy, however due to confronting with different situations or heterogeneity in the abstraction method that each agent utilizes, those hierarchies might be quite different in essence. Moreover, wider spatial and temporal coverage of a group [7] brings a potential benefit out of sharing, adopting, adapting, and applying the stored knowledge in a way that, helps individuals to build more accurate abstractions.

Many researches have been inspired by cooperative learning schemes in human and animal society. Advice taking [8], imitation [9], ensemble learning [10], and strategy sharing [11] are among many recent researches. However, none of them address utilization of both hierarchical and cooperative learning methods. In this paper, we present two methods to combine the output or the structure of the abstraction trees. The trees are learned online by different RL agents in the society, or by the same agent but with different abstraction methods. We also introduce the methods to balance and prune the merged decision trees, in order to purify and condense the shared knowledge, and prepare it for further developments.

Our paper is structured as follows: The next section overviews the recent works on Cooperative RL, Hierarchical RL, and combining decision trees. In the third section, we introduce the necessary formalism and notations. The fourth section describes the new algorithms. The fifth section describes the simulation task and results. Conclusions and future works are discussed finally.

II. Literature Survey

A. Cooperative Reinforcement Learning

Cooperative Learning attempts to benefit from the knowledge, learned by different agents, through explicit and implicit exchange of the learned rules, gathered information, etc. Cooperative RL aggregates different approaches from sensor sharing to strategy sharing. In its very basic form, agents serve as a scout by sharing sensory data in order to augment their eyesight [12]. In *Episode Sharing* [12], agents share *(state, action, reward, next state)* triples letting their colleague explore in a virtual world. *Simple Strategy Sharing* [12] blends Q-table of agents into one unique table through averaging the corresponding cells. However it homogenizes the Q-tables that is unsuitable when the agents' levels of expertise are different.

Weighted Strategy Sharing (WSS) [11], [13], [14] tries to resolve the problem by introducing some *expertness* measures and assigning different weights to the Q-tables accordingly. The proposed expertness is evaluated as a function of the history of the received rewards and punishments, action-values (e.g. entropy functions [15]), or state transitions [16]. Extensions of WSS [15]–[17] try to discover the *areas of expertise* by measuring their expertness in state-level. Being too much elaborated, these methods becomes impractical when state space enlarges.

In this paper decision trees help us categorize similar states to one group by abstraction techniques. This provides the chance to enlighten the expertness assessment in state-level and decrease the amount of communication. Moreover, since decision trees are more human readable than Q-tables, combination of them would result in interpretable rule sets.

B. Hierarchical Reinforcement Learning

Techniques for non-uniform discretization of state space are already known e.g. Parti-game [18], G algorithm [19], and U-Tree [4]. U-Trees use decision tree to incrementally derive the abstraction hierarchy. Continuous U-Tree [5] extends U-Tree to work with continuous features. We show in [6] that the existing U-Tree based methods ignore explorative nature of RL. This imposes a bias on the distribution of the samples saved for introducing new splits in U-Trees. As a consequence, finding a proper split point becomes more and more difficult and the introduced splits are far from optimality. Moreover, since U-Tree-based techniques have been excerpted in essence from decision tree learning, the splitting criteria that they utilize are very general. By reformulation of state abstraction with different heuristics, some specialized criteria are derived in [6] and their efficiency is compared to widely used ones, like Kolmogorov-Smirnov and Information Gain Ratio tests.

C. Combining Decision Trees

Hall et al [20] describe an approach in which, decision trees are learned from disjoint subsets of a large data set. Then, in a combination process the learned trees are converted to rule sets, similar rules are combined into a more general one, and contradicting ones are resolved. Although suitable for off-line classification, it is not applicable to continuous online learning, since the final rule set is not (in general) convertible back to a decision tree. This is necessary in our case for further improvement and learning. Moreover, generalization and conflict resolution techniques are context-dependent and must be adapted to our case.

Another direction of combining multiple trees, known as ensemble methods, is to construct a linear combination of outputs of some model fitting methods, instead of using a single fit. Bagging [21] involves producing different trees from different sample sets of the problem domain in parallel and then aggregating the results on a set of test instances. Boosting [22] is another technique that involves generating a sequence of decision trees from the same data-set, whereby attention is paid to the instances that have caused error in the previous iterations. Ultimate output of the ensemble is specified by majority voting or weighted averaging.

We will apply both merge and ensemble techniques to state abstraction task. However, in our merge algorithm, generalization and conflict resolution will be solved by applying cooperative learning techniques to leaf combination procedure.

III. Formalism

We model the world as a MDP, which is a 6-tuple (S, A, T, γ, D, R) , where S is a set of *states* (here can be infinite), $A = \{a_1, \ldots, a_{|A|}\}$ is a set of *actions*, $T = \{P^a_{ss'}\}$ is a *transition model* that maps $S \times A \times S$ into probabilities in $[0, 1]$, $\gamma \in [0, 1)$ is a *discount factor*, D is the initial-state distribution from which the start state is drawn (shown by $s_0 \sim D$), and R is a *reward function* that maps $S \times A \times S$ into real-valued rewards. A policy π maps from S to A, a *value function*, V, on states or an *action-value function*(also called *Q-function*), Q, on state-action pairs. The aim is to find an optimal policy π^* (or equivalently, V^* or Q^*) that maximizes the expected discounted rewards of the agent. Each state s is a *sensory-input* vector $(x_1, \ldots, x_n)$, where x_i is a *feature* (called also a *state-variable*, or an *attribute*).

An *abstract state* $\bar{S}$ is a subset of state space S, such that all states within it have "close" values. *Value* of abstract state $\bar{S}$ is defined as the expected discounted reward return if an agent starts from a state in $\bar{S}$ and follows the policy π afterwards:

$$V^\pi(\bar{S}) = \sum_{s \in \bar{S}} \{P(s|\bar{S}) \sum_{a \in A} [\pi(s, a) \sum_{s'} P^a_{ss'} (R^a_{ss'} + \gamma V^\pi(s'))]\} \quad (1)$$

where $\pi(s, a)$ is the selection probability of action a in state s under policy π, $P^a_{ss'}$ is the probability that environment goes to state s' after doing action a in state s, and $R^a_{ss'}$ is the expected immediate reward after doing action a in state s and going to state s'. Action-value $Q^\pi(\bar{S}, a)$ is similarly defined:

$$Q^\pi(\bar{S}, a) = \sum_{s \in \bar{S}} [P(s|\bar{S}) \sum_{s'} P^a_{ss'} (R^a_{ss'} + \gamma V^\pi(s'))] \quad (2)$$

A. U-Tree

The U-Tree [4] [5] abstracts the state space incrementally. Each leaf L_i of the U-Tree corresponds to an abstract state $\bar{S}_i$. Leaves store the action-values $Q(\bar{S}_i, a_j)$ for all available actions a_j. The tree is initialized with a single leaf, assuming the whole world as one abstract state. New abstract states are added if necessary. Sub-trees of the tree represent subtasks of the whole task. Each sub-tree can have other sub-sub-trees that correspond to its sub-sub-tasks. The hierarchy breaks down to the leaves, that specify the primitive sub-tasks.

The procedure for construction of the abstraction tree loops through a two phase process: *sampling* and *processing*. During the sampling phase the algorithm behaves as a standard RL, with the added step of using the tree to translate sensory input to an abstract state. A history of the transition steps, i.e. $T_i = (s_i, a_i, r_i, s'_i)$ composed of the current state, the selected action, the received immediate reward, and the next state is recorded. The sample is assigned to a unique leaf based on the value of the current state. Each leaf have a list per action –with fixed capacity– for recording the sampled data-points.

After some learning episodes the processing phase starts. In this phase a value is assigned to each sample:

$$V(T_i) = r_i + \gamma V(\bar{s}_{i+1}) \ , \ V(\bar{s}_{i+1}) = \max_a Q(\bar{s}_{i+1}, a) \quad (3)$$

where $\bar{s}_{i+1}$ is the abstract state that s_{i+1} belongs to. If a significant difference among the distribution of sample-values within a leaf is found, the leaf is broken up to two leaves. To find the best split point, the algorithm loops over the features. The samples within a leaf are sorted according to a feature, and a trial split is virtually added between consecutive pairs of the feature values. This split divides the abstract state into two sub-sets. A *splitting criterion* compares the two sub-sets, and returns a number indicating the difference between their distributions. If the largest difference among all features is bigger than a confidence threshold, then the split is introduced. This procedure is repeated for all leaves.

B. Splitting Criteria

Instead of applying very general splitting criteria, our work incorporates three criteria that have been derived for Hierarchical RL [6]: SANDS (State Abstraction in Non-Deterministic Systems), SMGR (Softmax Gain Ratio), and VAR (Variance Reduction). These criteria offer different levels of heterogeneity in splits, since SANDS and SMGR is shown to have some sort of similarities [6].

1) SANDS: SANDS tries to maximize the expected reward return after introducing a split, by finding a point that well differentiates both value and selection probability of actions, before and after introducing the split. Given a U-Tree that defines a policy, π, it is enhanced to a policy, $\tilde{\pi}$, by splitting the partition $\bar{S}$ (one of the leaves) to partitions, $\bar{S}_1$ and $\bar{S}_2$, so that the expected reward return of the new tree is maximized. We show the best split should maximize the following term [6]:

$$SANDS(\bar{S}) = \max_{a \in A} \sum_{i=1}^{2} (\hat{\pi}_i^a - \hat{\pi}^a) \hat{\mu}_i^a \hat{\rho}_i^a \tag{4}$$

where $\hat{\mu}_i^a$ approximates $Q^{\tilde{\pi}}(\bar{S}_i, a)$ by averaged value of samples of action a that belong to $\bar{S}_i$ and $\hat{\rho}_i^a$ is the fraction of samples of action a that belong to $\bar{S}_i$. $\hat{\pi}_i^a$ and $\hat{\pi}^a$ are selection probabilities of action a for $s \in \bar{S}_i$ and $s \in \bar{S}$, respectively. They are approximated using $\hat{\mu}_i^a$ and Boltzmann distribution.

2) SMGR: SMGR tries to find the splits that results in more certainty on the selection probability of actions:

$$SMGR(\bar{S}) = \max_{a \in A} \frac{-\hat{\pi}^a \log \hat{\pi}^a + \sum_{i=1}^{2} \hat{\rho}_i^a \hat{\pi}_i^a \log \hat{\pi}_i^a}{-\sum_{i=1}^{2} \hat{\rho}_i^a \log \hat{\rho}_i^a} \tag{5}$$

where $\hat{\pi}^a$, $\hat{\pi}_i^a$, and $\hat{\rho}_i^a$ are defined as in SANDS. We prove that SMGR scales SANDS and combines it with a penalty term.

3) VAR: The VAR criterion is developed on the basis of Mean Square Error (MSE) reduction. We can prove that selecting the split that maximizes the following goal function, results in minimizing MSE on the action-value functions of the abstract state $\bar{S}$ over the whole action set A:

$$VAR(\bar{S}) = \max_{a \in A} \; \hat{\rho}_1^a \; \hat{\rho}_2^a \; (\hat{\mu}_1^a - \hat{\mu}_2^a)^2 \tag{6}$$

where $\hat{\rho}_i^a$ and $\hat{\mu}_i^a$ are defined like SANDS.

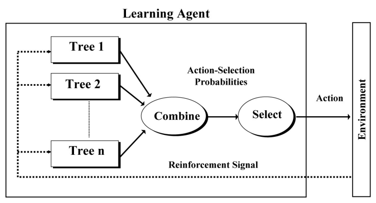


Fig. 1. Ensemble of abstraction trees jointly selects the next action

IV. Combining Abstraction Trees

Cooperative learning among abstraction trees could be realized by combining either their output or their structure. The former is called *Ensemble Learning*. For the latter, we introduce a new algorithm to merge, balance and prune the abstraction trees. It is called *CLASS*, which stands for *Combining Layered Abstractions of State Space*.

A. Ensemble Learning

We use the idea of Bagging [21] and Boosting [22], and let the learning agent construct multiple abstraction trees in parallel from the same learning episodes (Fig. 1). It is hoped that, the ensemble guide the abstraction trees to get specialized in different areas of state-space by introducing splits in different axis. In action-selection phase, the agent combines the probability distribution of actions, proposed by each tree by simple averaging. The action is, then, selected according to this combined distribution. Finally, the reinforcement signal is feeded back to all trees. Combining the output via averaging is reasonable, since all trees receive the same reinforcement signals and therefore their expertness measures are equal.

B. CLASS

The CLASS algorithm includes applying the sequence of merge, balance, and pruning procedures to the abstraction trees. The goal is to put the knowledge, acquired from different sources, together and purify it. However, the balance and the pruning algorithms are not constrained to cooperative learning. They could be applied to any abstraction tree in general.

The CLASS algorithm starts by merging the two abstraction trees. Then, the resulted tree is balanced and pruned. Another sequence could be to apply the balance and the pruning procedures twice, once to the abstraction trees before merge, and once to the resulted tree after merge. Our experiences show that, this way the resulted tree after merge is more lightweight. As a consequence, the balance procedure, which is the bottleneck of the algorithm, is accomplished faster.

1) Merge: Given the abstraction trees, T_1 and T_2, learned on the same state-space, a new decision tree $T_3 = T_1 + T_2$ is formed from pairwise intersection of the leaves of T_1 and T_2. The merge algorithm is non-commutative i.e. in general $T_1 + T_2 \neq T_2 + T_1$. However, the final partitions are same. Merging can be illustrated by simply overlaying the two partitions on top of each other, as shown in left column of Fig. 2 for a simple 2D state-space. T_1 and T_2 are different abstractions of the same state-space, and T_3 shows the overlayed partitions. The following pseudo-code describes the merge algorithm:

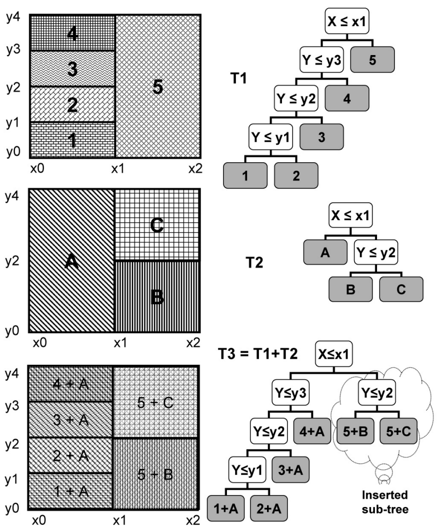


Fig. 2. An example for merging abstraction trees. top and middle: input abstraction trees, bottom: merge result

```
Node MergeTree(Node r1, Node r2){
  if (r1.Boundary overlaps with r2.Boundary) {
    if (r2 is Leaf) MergeLeaf(r1,(Leaf)r2);
    else {
      r1 = MergeIntNode(r1,(IntNode)r2);
      r1 = MergeTree(r1,((IntNode)r2).Left);
      r1 = MergeTree(r1,((IntNode)r2).Right);
  }}
  return r1;
} Node MergeIntNode(Node n1, IntNode n2) {
  if (n1.Boundary overlaps with n2.Boundary ) {
    if (n1 is Leaf) {
      if (n1.Boundary is splittable by n2.Split) {
        Create a new internal node
          with the same split as n2;
        Place n1 & its duplicate as children
          of the new internal node;
        return the new internal node;
      }
    }else{
      n1.Left = MergeIntNode(n1.Left,n2);
      n1.Right = MergeIntNode(n1.Right,n2);
  }}
  return n1;
} void MergeLeaf(Node n1, Leaf l2) {
  if (n1.Boundary overlaps with l2.Boundary) {
    if (n1 is Leaf)
      Combine n1.QFunctions with l2.QFunctions;
    else{
      MergeLeaf(n1.Left,l2);
      MergeLeaf(n1.Right,l2);
}}}
```

It is assumed in the algorithm that a *node* is either a *leaf* or an *internal node*. Every node has a *boundary* which specifies its corresponding *hypercube* in the state-space e.g. boundaries of the hypercube marked as 3 in Fig. 2 (top) is $[x_0 \rightarrow x_1, y_2 \rightarrow y_3]$. Hypercubes that correspond to leaves are *basic* hypercubes whereas the ones that correspond to internal nodes are *complex*. Internal nodes denote splits points. A *univariate* split, $X = a$, is a *hyperplane* that breaks a hypercube from point a along X axis to two smaller hypercubes (From now on, we refer to univariate splits as "split" simply). Internal nodes have two children specified by *left* ($X \leq a$) and *right* ($X > a$). A hypercube is *splittable* by the split, $X = a$, if the split point, a, is between (and not on) the start and the end points of the hypercube's boundary on the split axis, X.

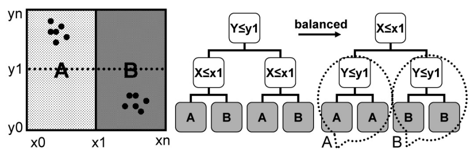


Fig. 3. An example to show efficiency of balance and pruning in state abstraction. left:actual partitions, middle:inefficient partitioning, right:balanced tree in terms of similarity between Q-functions of leaves.

The merge algorithm starts by traversing T_2 in pre-order (First the root, then the left sub-tree, finally the right sub-tree) and inserting the visited nodes one by one into T_1, if required (MergeTree function, see also Fig. 2-right column). If the boundary of a leaf of T_1 is splittable by a split in T_2, that split is initiated in T_1 by replacing the leaf with a sub-tree, consisting the split as root, and the leaf and its copy as children (MergeIntNode function, see also the inserted sub-tree in Fig. 2). Merging the leaves of T_2 is done by combining their Q-functions with the one of the overlapping leaves of T_1 (MergeLeaf function). Leaf combination process is carried out by applying WSS and expertness measures (sec. II-A).

2) Balance: Mapping from partitions to abstraction trees is not always one-to-one. While only one representation is possible for T_2 (Fig. 2-mid), multiple trees can represent T_1 (Fig. 2-top). Structure of the trees, produced by abstraction methods, depends on the algorithm and sequence of the incoming samples. The structure is not important for the final policy (all represent same policies). But, it is crucial for traversing or pruning the trees.

For decision trees that are constructed from off-line data, restructuring is not relevant. however, when constructing abstraction trees from online data, we must be able to restructure them, since the already introduced splits might be inefficient. Fig. 3 (left) shows a simple case where, the whole state-space is representable by a split at x_1 and two abstract states. Now if by chance, the sampled data points are distributed like the dark circles, the abstraction algorithm might introduce a split at y_1, followed by the splits at x_1 (Fig. 3-mid). In reality, this happens very often, since samples are very few comparing to the size of the state-space.

Utile Distinction Memory [23] partially solves this problem by virtually generating all potential splits. When the agent is insured of the efficiency of a virtual split, it is upgraded to an actual one. This method is not applicable to continuous features as the number of potential splits grows enormously. A possible solution would be the combination of *balance* and *prune* mechanisms. The balance algorithm restructures the tree so that, inefficient splits are shifted down to leaves and removed by the pruning procedure. The following pseudo-code describes the balance algorithm:

```
void Balance(Tree tree){
  NodeList[] list = new NodeList[tree.LeafCount];
  for(int i=0;i<tree.LeafCount;i++)
    list[0].Add(tree.Leaf[i]);
  for(int i=2;i<=tree.LeafCount;i++)
    for(int j=i/2;j>0;j--){
      NodeList n1=list[j-1], n2=list[i-j-1];
      if (n1!=n2)
        for(int p=n1.Count-1;p>=0;p--)
```

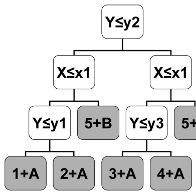


Fig. 4. Balanced version of T_3 in Fig. 2 in terms of depth

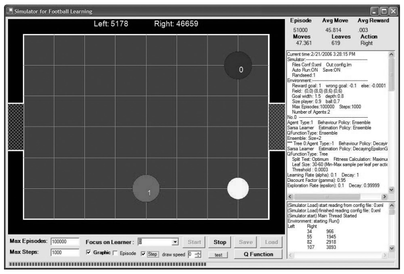
```
        for(int q=n2.Count-1;q>=0;q--)
          Combine(list,n1[p],n2[q]);
      else
        for(int p=n1.Count-1;p>0;p--)
          for(int q=p-1;q>=0;q--)
            Combine(list,n1[p],n2[q]);
    }
  Build tree recursively from list[tree.LeafCount-1][0];
} void Combine(NodeList[] list,Node n1,Node n2){
  if (n1 and n2 are combinable) {
    n12 = n1 combined to n2;
    best = list.FindNode(n12.Boundary)
    if (best != null){
      if (n12.BalanceFactor > best.BalanceFactor)
        list.Replace(best,n12);
    }else list.Add(n12);
}}
```

It tries every possible tree to find the best balanced one for the whole space. However, it does not recursively break the whole state-space in top-down manner. Instead, in order to reduce the time order of the algorithm, it combines the smaller hypercubes and build bigger ones in a bottom-up manner. In top-down manner, some sub-problems (i.e. balancing subsets of state-spaces) are solved multiple times, while in bottom-up manner, they are solved once, saved and reused afterwards.

The algorithm starts from leaves. They are tried in turn and combined with other combinable leaves to form complex hypercubes (Balance function). Two non-overlapping hypercubes are *combinable*, if their combination also forms a hypercube, i.e. they are neighbors and their boundaries are same in all except one axes, e.g. leaf 4 in Fig. 2 (top-left) has two neighbors: 3 and 5, however it is combinable only with 3.

A list records the root split for the sub-spaces that have been balanced up to now e.g. this list stores $X = x_1$ as the root split of $[x_0 \to x_2, y_0 \to y_4]$ in Fig. 2 (mid). If two hypercubes are combinable, then this list is checked to see whether their combination proceeds to a more balanced tree for the combined sub-space or not. If so, the new combination replaces the existing one (Combine function). Finally, the best root split for the whole space is turned out. Putting this split as the root of the balanced tree, we can recursively build the whole tree by finding the boundaries of the sub-spaces at left and right side of the root split. The list is recursively looked up to find the root split of these two sub-spaces and so on.

Different balance factors generate different types of balanced trees. By choosing the inverse of *average path-length from leaves to root* as the balance factor, the balanced tree would have the shortest depth among all possible trees. Fig. 4 shows the balanced version of T_3 in Fig. 2 in terms of this factor. Combining path-length with *the fraction of samples that belong to a specific node*, generates abstraction trees that are optimized for search purpose.



```
Episode    Avg Move    Avg Reward
 51000      45.814      .003
 Moves       Leaves      Action
  47.361     619         Right
```

```
Current time:2/21/2006 3:28:15 PM
Simulator:-----------------------
  Files Conf:0.xml   Out:config.lrn
  Auto Run:ON   Save:ON
  Randseed:1
Environment:---------------------
  Reward goal: 1   wrong goal: -0.1   else: -0.0001
  Field:  (0,0) (8,0) (8,6) (0,6)
  Goal width: 1.5   depth:0.8
  Size player: 0.9   ball:0.7
  Max Episodes:100000   Steps:1000
  Number of Agents:2
No.0 ----------------------------
Agent Type:1   Behaviour Policy: Ensemble
Sarsa Learner   Estimation Policy: Ensemble
QFunctionType: Ensemble
Ensemble: Size=2
*** Tree 0:Agent Type:-1   Behaviour Policy: Decayir
Sarsa Learner   Estimation Policy: DecayingEpsilonG
QFunctionType: Tree
  Split Test: Optimum   Fittness Calculation: Maximu
  Leaf Size: 30-60 (Min-Max sample per leaf per acti
  Threshold : 0.0003
Learning Rate (alpha): 0.1   Decay: 1
Discount Factor (gamma): 0.95
Exploration Rate (epsilon): 0.1   Decay: 0.99999
```

```
(Simulator.Load) start reading from config file: 0.xml
(Simulator.Load) finished reading config file: 0.xml
(Simulator.start) Main Thread Started
Environment: starting Run()
Left      Right
          34      966
          55      1945
          82      2918
          107     3893
```

Fig. 5. Simulation task and a screen-shot from the simulator

3) Prune: To restructure the abstraction trees for pruning, the balance factor could be defined as a *similarity measure* on the action-values of the sibling nodes. According to this measure, nodes with similar action-values are placed in vicinity. If the similarity between the sibling leaves is more than a threshold, they could be replaced (including their immediate parent) by a single leaf. In this paper, similarity of two nodes is defined as a binary function: one, if the greedy actions (The action with the highest action-value) of the nodes are equal, and zero otherwise. The balance factor for an internal node is recursively defined as: the similarity between its children (0 or 1), plus sum of the balance factors of its children.

The greedy action of a leaf is easily specified from its action-values. Since internal nodes do not hold the action-values, their equivalent values have to be computed. The action-values of an internal node, $\bar{S}$, is a weighted sum of the action-values of its children, $\bar{S}_1$ and $\bar{S}_2$: $Q(\bar{S}, a) = \sum_{i=1}^{2} P(\bar{S}_i|\bar{S}) Q(\bar{S}_i, a), \forall a \in A$ where the weights are estimated by the fraction of samples that belong to the child, out of the total samples in $\bar{S}$. Fig. 3 (right) shows the balanced version of the abstraction tree at its left, according to this balance factor. The split at y_1 is moved down and the more efficient split, x_1, is placed at root. Since the children of y_1 splits, both represent the same action-values, they can be replaced by one leaf.

V. Simulation Results

The learning task is a simplified football task in an 8×6 grid (Fig. 5). A learning agent plays against an intelligent opponent and learns to deliver the ball to the right goal. The opponent is programmed manually to move toward the ball, pick it and carry it to the left goal. Meanwhile, with certain probability (here 40%) it selects a wrong action, to leave a bit chance for the learner to score. Players can choose among 6 actions: {*Left, Right, Up, Down, Pick, Put*}. If a player selects an action that results in touching the boundaries or going to a pre-occupied position by another player, the action is ignored. If owner of the the ball hits the other player or the boundaries, the ball is freed. States of the learner consist of x and y coordinates of the ball and the players, plus the status of the ball from {*Free, Picked*}, and two additional states for left and right goal i.e. $85752 (= 2 \times (7 \times 5)^3 + 2)$ states.

Learning experiment consists of 100,000 *episodes*. The last 10,000 episodes are referred as *test phase*. Episodes start by placing the players and the ball in random (and unoccupied) positions. Episodes last up to 1000 *steps*. Steps include one

TABLE I

RESULTS OF INDIVIDUAL AND ENSEMBLE LEARNING

Method	leaf/ tree	action lrn	action tst	reward lrn	reward tst	score ratio lrn	score ratio tst
SANDS	1723	136.4	43.2	.679	.894	3.2	13.9
SMGR	1665	123.6	36.9	.718	.931	4.3	18.3
VAR	1827	179.2	54.7	.663	.919	3.3	17.9
ind. avg.	**1738**	**146.4**	**44.9**	**.687**	**.915**	**3.6**	**16.7**
SANDS-SANDS	1660	42.8	22.3	.798	.919	5.0	15.7
SMGR-SMGR	1729	57.2	23.9	.801	.942	5.5	22.1
VAR-VAR	1864	61.5	23.5	.784	.912	4.7	14.0
homog. avg.	**1751**	**53.8**	**23.2**	**.794**	**.924**	**5.1**	**17.3**
SANDS-SMGR	1666	37.5	21.4	.862	.956	7.5	27.3
SANDS-VAR	1740	45.5	21.2	.787	.920	4.7	15.4
SMGR-VAR	1780	40.1	21.4	.837	.943	6.4	22.8
hetrog. avg.	**1729**	**41.0**	**21.3**	**.829**	**.940**	**6.2**	**21.9**

movement by each player in turn. An episode is finished if, either players score or the maximum number of steps is passed. The learner receives +1 for correct scores, -0.1 for wrong scores or scores by the opponent, and -0.0001 otherwise. All methods use SARSA with decaying ϵ-greedy for sampling phase. Learning rate, α, is 0.1 and discount factor, γ, is 0.95. Exploration rate, ϵ, is initialized to 0.1 and is decayed by multiplying with 0.99999 after each episode. Leaf sample size is 360 (60 samples per action). For each state abstraction criteria, different confidence thresholds are tried, in order to find the best range of thresholds in terms of convergence. The best range is then divided to 10 equal-size sub-ranges and simulations are executed with these 10 thresholds.

A. Ensemble Learning Results

Table I shows the final results for *individual* and *ensemble* learning experiments, in both *homogeneous* and *heterogeneous* cooperation cases. The individual learning experiments refer to the experiments that engage only one abstraction tree. In this category, for every split criterion and confidence threshold, 10 simulation runs (with different random seeds) are executed. Ensembles engage two abstraction trees in parallel. In this category, all possible combinations of the splitting criteria and the confidence thresholds, that has been used in the individual experiments, are tried in turn. Heterogeneity, here, refers to the difference between the two splitting criteria used in the ensemble. For example, SMGR-VAR is considered as a heterogeneous cooperation between SMGR and VAR. While, VAR-VAR is considered as a homogeneous cooperation and refers to the ensembles in which, both trees use VAR criterion.

The first column of Table I shows the average number of leaves per trees at the end of learning. The rest show the number of actions, sum of the received rewards, and the score ratio, averaged per episodes for learning and test phases. These phases let us compare convergence rate and performance of the learned policies, respectively. *Score ratio* is the scores of the learner divided by the scores of the opponent. If a method effectively learns both attack and defence strategies, it would have higher ratios.

Table I shows the ensembles, almost in all cases, have better results than the individual methods, in terms of both convergence rate and performance of the learned policy. The average number of actions for the ensembles are less than the individual cases, both in learning and test phases. Also, the received rewards and the score ratios are all higher in both phases. The average number of leaves per trees are close to the individual cases (Note that the total number of leaves in the ensembles are bigger than individual cases), though, in heterogeneous category, they are slightly smaller. The ensembles that involve VAR criterion are exceptions. Although their convergence is faster than the individual cases, performance of the final policy degrades in some ensembles. In VAR-VAR and SANDS-VAR, the score ratios are smaller than VAR. Perhaps in some situations, multiple paths exist for catching the ball. If one policy tries e.g. to approach the ball first in x direction and then in y, while the other one tries the opposite, combining these conflicting policies could result in selecting an inefficient action and losing time. Meanwhile, the opponent catches the ball and scores it.

Heterogeneous ensembles have, in average, better results than homogeneous ones both in convergence rate and policy performance. SANDS-SMGR and SMGR-VAR are the two top methods among all. This can be due to the variety in the split axis. If the trees break a sub-space along different axis, the combined output is like a linear multi-variate split.

TABLE II

PERFORMANCE OF CLASS ALGORITHM

Method	merge	leaf prune	red.%	act-ion	rew-ard	score ratio
SANDS-SANDS	4551	2244	50.7	37.2	.904	17.6
SMGR-SMGR	4629	2254	51.3	32.9	.952	30.3
VAR-VAR	4281	2033	52.5	45.7	.929	22.8
homog. avg.	**4487**	**2177**	**51.5**	**38.6**	**.928**	**23.6**
SANDS-SMGR	6794	3175	53.3	33.6	.940	25.4
SANDS-VAR	5584	2454	56.1	39.5	.921	20.1
SMGR-VAR	7507	2964	60.5	36.9	.941	26.1
hetrog. avg.	**6628**	**2864**	**56.6**	**36.7**	**.934**	**23.9**

B. CLASS Results

The second category of tests concern the CLASS algorithm. All possible two-combinations of the trees, saved from individual learning experiments, are selected and the CLASS algorithm is executed on them. Since, agents in our simulation task have equal learning episodes, simple averaging is used for leaf combination (however, the expertness measures had similar results). Then, the resulted tree is tested in the same simulation task for 30,000 episodes. Table II displays the average number of leaves of the trees after merge, followed by the average number of leaves after balance and pruning the merged trees, and the percentage of reduction in the size of the trees. The next columns summarize the number of actions, sum of the received rewards, and score ratio averaged per episodes. Table II shows the combined trees by CLASS always find faster policies, that gain more scores and rewards than their individual versions (Table I), e.g. SANDS-VAR policies accomplish the task in 39.5 actions, that is quite faster than SANDS (43.2) and VAR (54.7).

Like ensembles, performance of the heterogeneous combinations of the splitting criteria is very close to the best of their homogeneous combinations, e.g. SMGR is the best method among the individual cases. Combining it with itself (SMGR-SMGR) results in the best method among all. However, its heterogeneous combination with SANDS or VAR, also have close performance to SMGR-SMGR. So, in cases where no idea about the performance of the individual methods exists, the best choice would be to mix them with a cooperative method, like Ensemble Learning or CLASS. The mixture would automatically benefit from their best. Thus, heterogeneous cooperative learning could simplify method selection or parameter tuning for learning algorithms.

Like ensembles, heterogeneous combinations have better performance than homogeneous ones, except in the number of leaves. In homogeneous cases, the input trees have more similar splits. As a result, the merged trees are smaller. Our analyzes emphasize that, the size of the merged trees are linearly increasing with the product of the size of the input trees. Pruning the trees reduce their size to less than half. The percentage of size reduction is higher in heterogeneous case, in which the trees are also bigger.

CLASS algorithm creates smaller trees than ensembles with almost better performance in terms of the received rewards and the score ratio. However, policies found by the ensembles are around 15 actions faster. This means, although their policies are fast in attack, they are inefficient in defence and let the opponent score more. CLASS results, however, have better defence and episodes are, thus, prolonged.

VI. Conclusion

In this paper two approaches to create cooperation between different sources of knowledge via ensembles of abstraction trees and via applying merge, balance, and pruning techniques were presented. The abstraction trees could come from different abstraction methods or different agents. Simulation results in non-deterministic football task provided strong evidences for faster convergence rate and more efficient policies compared to individual learning. Heterogeneous split criteria resulted in more efficient and faster converging ensembles and merged trees than homogeneous ones. Mixing heterogeneous methods with cooperative learning resulted in a system that its performance is close to the best method.

Ensemble learning and tree merging create bigger trees than individual learning. Merging the trees, that are created by homogeneous criteria, result in smaller trees than heterogeneous ones. Since homogeneous criteria generate similar splits, they can be factored out. Generating bigger trees could be fine as far as memory consumption does not matter e.g. in real-robot learning where faster convergence is more important.

We would like to apply these algorithms to a realistic soccer task. We look, also, for better abstraction methods that create still smaller trees. This is a big drawback of the current methods when they are applied to complex tasks.

Acknowledgment

This research is funded by the Future and Emerging Technologies programme (IST-FET) of the European Community, under grant IST-2001-35506. The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication. The Swiss participants are supported under grant 01.0573 by the Swiss Government.

References

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1996.

[2] R. Parr and S. Russell, "Reinforcement learning with hierarchies of machines," in *Advances in Neural Info. Processing Syst.*, M. Jordan, M. J. Kearns, and S. A. Solla, Eds., vol. 10. MIT Press, 1998.

[3] T. G. Dietterich, "Hierarchical reinforcement learning with the MAXQ value function decomposition," *Journal of Artificial Intelligence Research*, vol. 13, pp. 227–303, 2000.

[4] A. McCallum, "Reinforcement learning with selective perception and hidden state," Ph.D. dissertation, Computer Science Dept., Univ. of Rochester, 1995.

[5] W. T. Uther and M. M. Veloso, "Tree based discretization for continuous state space reinforcement learning," in *AAAI/IAAI*, 1998, pp. 769–774.

[6] M. Asadpour, M. N. Ahmadabadi, and R. Siegwart, "Reduction of learning time for robots using automatic state abstraction," in *Proc. 1st European Symp. on Robotics*, ser. Springer Tracts in Advanced Robotics, H. Christensen, Ed., vol. 22. Palermo, Italy: Springer-Verlag, March 2006, pp. 79–92.

[7] I. Riachi, M. Asadpour, and R. Siegwart, "Cooperative learning for very long learning tasks: A society inspired approach to persistence of knowledge," in *European Conference on Machine Learning, Cooperative Multi-Agent Learning workshop*, Porto, Portugal, 2005.

[8] R. Maclin and J. W. Shavlik, "Creating advice-taking reinforcement learners," *Machine Learning*, vol. 22, no. 1-3, pp. 251–281, 1996.

[9] G. Hayes and J. Demiris, "A robot controller using learning by imitation," in *Proc. of International Symposium on Intelligent Robotic Systems*, A. Borkowski and J. Crowley, Eds., vol. 1. Lifia Imag, Grenoble, France, 1994, pp. 198–204.

[10] T. G. Dietterich, "Ensemble methods in machine learning," *Lecture Notes in Computer Science*, vol. 1857, pp. 1–15, 2000.

[11] M. N. Ahmadabadi and M. Asadpour, "Expertness based cooperative q-learning," *IEEE Trans. Syst., Man, Cybern., Part B*, vol. 32, no. 1, pp. 66–76, 2002.

[12] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative learning," in *Readings in Agents*, M. N. Huhns and M. P. Singh, Eds. San Francisco, CA, USA: Morgan Kaufmann, 1997, pp. 487–494.

[13] M. N. Ahmadabadi, M. Asadpour, S. H. Khodaabakhsh, and E. Nakano, "Expertness measuring in cooperative learning," in *Proc. IEEE/RSJ Int. conf. on Intelligent Robots and Systems [IROS]*, vol. 3.

[14] M. N. Ahmadabadi, M. Asadpour, and E. Nakano, "Cooperative q-learning: the knowledge sharing issue," *Advanced Robotics*, vol. 15, no. 8, pp. 815–832, 2001.

[15] S. M. Eshgh and M. N. Ahmadabadi, "An extension of weighted strategy sharing in cooperative q-learning for specialized agents," in *Proc. 9th Int. Conf. on Neural Information [ICONIP]*, 2002, pp. 106–110.

[16] S.M.Eshgh, B.Araabi, and M.N.AhmadAbadi, "Cooperative q-learning through state transitions: A method for cooperation based on area of expertise," in *Proc. 4th Asia-Pacific Conf. on Simulated Evolution And Learning [SEAL]*, Singapore, 2002, pp. 61–65.

[17] S.M.Eshgh and M.N.Ahmadabadi, "On q-table based extraction of area of expertise for q-learning agents," in *World Automation Cong.*, 2004.

[18] A. W. Moore, "The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces," in *Advances in Neural Information Processing Systems*, J. D. Cowan, G. Tesauro, and J. Alspector, Eds., vol. 6. Morgan Kaufmann, 1994, pp. 711–718.

[19] D. Chapman and L. Kaelbling, "Input generalization in delayed reinforcement learning: An algorithm and performance comparisons," in *Proc. 12th Int. Joint Conf. on AI [IJCAI]*, 1991, pp. 726–731.

[20] L. O. Hall, N. Chawla, and K. W. Bowyer, "Decision tree learning on very large data sets," in *IEEE Conf. Syst., Man, Cybern.*, 1998.

[21] L.Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[22] Y.Freund and R.E.Schapire, "Experiments with a new boosting algorithm," in *Proc. 13th Int. Conf. Machine Learning*. San Francisco: Morgan Kaufmann, 1996, pp. 148–156.

[23] A. McCallum, "Overcoming incomplete perception with utile distinction memory," in *Int. Conf. on Machine Learning*, 1993, pp. 190–196.