Transfer Learning in Reinforcement Learning Problems Through Partial Policy Recycling*

Jan Ramon, Kurt Driessens, and Tom Croonenborghs

K.U. Leuven, Dept. of Computer Science, Celestijnenlaan 200A, B-3001 Leuven

Abstract. We investigate the relation between transfer learning in reinforcement learning with function approximation and supervised learning with concept drift. We present a new incremental relational regression tree algorithm that is capable of dealing with concept drift through tree restructuring and show that it enables a Q-learner to transfer knowledge from one task to another by recycling those parts of the generalized Q-function that still hold interesting information for the new task. We illustrate the performance of the algorithm in several experiments.

1 Introduction

Inductive transfer or transfer learning is concerned with the connection between learning in different but related contexts, e.g. learning to drive a bus after having learned to drive a car. In a machine learning context, transfer learning is concerned with the added benefits that learning one task can have on a different, but related task. More specifically, in a reinforcement learning (RL) context, the added effects of transfer learning can help the learning agent to learn a new (but related) task faster, i.e., with a smaller amount of training experience.

Concept drift [14] refers to changes over time in the concept under consideration. Examples include socioeconomic phenomena such as fashion, but also more technical subjects such as computer-fraud and -intrusion. In a machine learning context, concept drift is usually seen as an added difficulty in a learning task and has given rise to the field of theory revision.

In principle, the two notions of transfer learning and concept drift are very similar. Both deal with a change in the target hypothesis, but both also assume that there will be large similarities between the old and the new target concept. The biggest difference between the two problem definitions is that for transfer learning, it is usually known when the context change takes place. For concept drift, this change is usually unannounced.

In this paper, we investigate the possibility of using methods from theory revision in a transfer learning context. While reinforcement learning, we will try to recycle from one task, those parts of a learned policy that still hold relevant information for a new, related task. We do this by introducing a new incremental

* Research supported by Research Foundation-Flanders (FWO-Vlaanderen), by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen) and by GOA 2003/08 "Inductive Knowledge Bases".

relational regression tree algorithm that uses a number of tree restructuring operators that are suited for first-order regression trees. The algorithm does not store past training experience but instead stores statistical information about past experience that allows it to decide when to use which operator. We will illustrate the performance both on a relational task with concept drift and for transfer learning in relational reinforcement learning tasks.

2 Transfer Learning and Theory Revision

Most current work in transfer learning does not incorporate ideas from theory revision. A lot of transfer learning approaches use a mapping to relate the new task to the task for which a policy was already learned. Some approaches use the learned knowledge to aid exploration in the new task [7,9]. Although guided exploration is well suited to overcome the problem of sparse rewards, we believe that a lot of knowledge is lost if one only uses earlier experience to steer exploration. In the context of concept drift or theory revision it is common to try to use knowledge learned previously to facilitate learning the new concept.

The approach of Torrey et al. [12] incorporates knowledge about the Q-values of the first task into the construction of the Q-function of the new task as soft-constraints to the linear optimization problem that approximates the Q-function. However, we feel that a lot of knowledge about the structure of the Q-function is lost. Taylor et al. [11] take a first step into this direction by reusing policies represented by neural networks. In a general game playing context, there is the approach of Banerjee and Stone [1] where game independent features that encode search knowledge are learned that can be used in new tasks.

However, we would like to re-use partial policies to a larger extend. As is common in theory revision approaches, we will conserve structures in the target function that were discovered for the first task when they apply to the second and expand on them or, if necessary delete them from the learned model. Building an adapted model of the target concept based on already discovered structural knowledge about related tasks should facilitate learning and thus create a new form of transfer learning for reinforcement learning.

3 Incremental Tree Learning and Restructuring

We will be using logical decision trees as our target models. Because we will be re-using learned models from related but different tasks, we need to be able to learn models that generalize well over tasks which share some properties. For this, relational or first-order representations are very well suited. We will use decision trees because the built-in modularity of trees allows easy access to different parts of the learned theory.

Chapman and Kaelbling [3] proposed an incremental regression tree algorithm designed with reinforcement learning in mind. On a high level, the G-algorithm stores the current decision tree and, for each leaf, statistics for all tests that can be used to split that leaf further. Except for the leaf-splitting, i.e. building the tree incrementally, no tree-restructuring operators are used. TG [5] upgrades the G-algorithm to a first-order context and uses a similar approach to build first-order decision trees. A first-order decision tree is a binary decision tree in which internal nodes contain tests which are a conjunction of first-order literals. A constraint placed on the first-order literals is that a variable that is introduced in a node (i.e., it does not occur in higher nodes) does not occur in the right subtree of the node. This constraint stems from the fact that variables in the tests of internal nodes are existentially quantified. Suppose a node introduces a new variable X. Where the left subtree of a node corresponds to the fact that a substitution for X has been found to make the conjunction true, the right side corresponds to the situation where no substitution for X exists, i.e., there is no such X. Therefore, it makes no sense to refer to X in the right subtree.

Probably the best known tree-restructuring algorithm is the ITI-algorithm [13]. It stores statistics about the performance of all splitting criteria for all the nodes in the tree and incorporates operators for tree-restructuring such as tree-transposition or slewing cut-points for numerical attributes. The tree-transposition operator switches tests between parent and child nodes and copies the statistical information from the original tree to the restructured one. Recursive tree-transpositioning can be used to install a designated test at any internal node of a propositional tree. Recently, Dabney and McGovern [4] developed relational UTrees (which incorporate relational tests in nodes) and an ITI-based incremental learner. However, this system has several drawbacks. First, it requires all training examples to be remembered. Second, the performance of nodes is measured by a set of randomly generated trees, which are of limited depth and whose generation is computationally expensive.

We will introduce the TGR algorithm, an incremental relational decision tree algorithm that employs tree restructuring operators and does not need to store all past learning experience. It extends the TG algorithm mentioned above, the difference being the availability of four tree-restructuring operators:

Splitting a leaf. This operator splits a leaf into two subleafs, using the best suited test. This is the only operator used by standard (non-restructuring) top down induction of decision trees (TDIDT) algorithms such as TG .

Pruning a leaf. This is the inverse operator of the first. When predictions made in two sibling-leafs (leafs connected to the same node) become similar, it will join the two leafs and remove the internal node.

Revising an internal node. This is a bit more complex than the two previous ones. It is illustrated on the left side of Figure 1. When it turns out that a different test from the one originally chosen at an internal node becomes significantly better, the dependencies between tests in first-order trees make it impossible to make a straightforward swap of the two tests. Instead, we construct a new node with the new test and repeat on both sides the original tree (Figure 1). This avoids any information loss, and it can be hoped that redundant nodes will be further pruned away by the other operators.

Pruning a subtree. This operator is related to the previous one, but will shrink the tree instead of enlarging it. It is illustrated at the right side

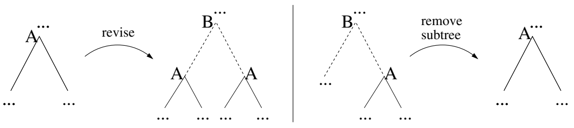

Fig. 1. Revising a test (left) and pruning a subtree (right)

of Figure 1. This operator is used when a node can (or should) be deleted. Because of the dependencies between tests in a first-order tree, the choice of subtrees to replace the original tree starting at the deleted node is limited to the right-side one. As before, the left subtree can contain references to variables introduced by the test used in the deleted node (e.g. B in Figure 1).

Although these restructuring operators differ from those used in ITI, the resulting trees are quite similar. One difference is that the recursive application of the transposition operator changes the bottom most tests in the tree and our revision operator does not. Just like the TG algorithm, TGR will not store learning examples, but discard them after the update of the statistics in the tree. The major difference between the two is that TGR stores the statistics that TG only stores in the leafs for all the nodes in the tree. On top of this, to be able to use the pruning operator, TGR stores the predictive performance of both the entire subtree and just the right subtree for each node in the tree.

4 Experimental Evaluation

4.1 Bongard Problems with Concept Drift

In a first series of experiments we will use Bongard problems [2] to evaluate the performance in the context of concept drift. One Bongard learning example consists of a (varying) number of objects and relations between them. For our experiments, we created a dataset of Bongard examples (with every example having randomly 0-4 circles, 0-4 triangles and a number of *in*/2 relations depending on the concept) and feed them to the learner one by one, together with the classification (positive or negative) for a chosen concept. After a certain number of learning examples, we change the concept. We will show the predictive accuracy (tested on 500 examples) for both TG and TGR on all problems. The results are averaged over 10 runs, evaluated once per 500 training examples.

In the first experiment we interleave two target concepts. We start with the concept "Is there a triangle inside a circle?" (A) for 2000 learning examples, then change it to "Is there a circle inside a triangle?" (B) and alternate the two in an $ABABAB$ fashion every 5000 learning examples with in the end 5000 examples extra to show further convergence. Figure 2 (left) shows that TGR is able to keep up with the concept changes while TG adapts much more slowly. In Figure 2 (right), one can see that although the tree size of the theory built by TGR can grow suddenly when TGR decides to swap one of the topmost tests in the tree using the third revision operator and thereby almost doubling the size

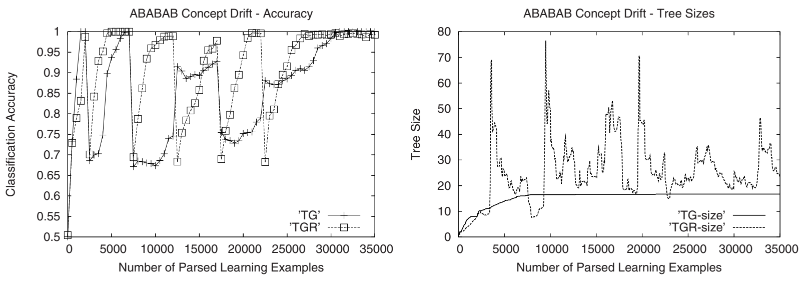

Fig. 2. Bongard problem with $ABABABB$ concept drift

of the tree, TGR usually recovers quickly by deleting appropriate parts of the tree. The tree does however stay larger than the TG one most of the time. This experiment also shows the independence of the TGR algorithm to the number of already processed examples.

In a second experiment, we increase the difficulty of the concept change during the experiment. We again start with concept A and change it into concept B after 5000 learning examples, but after 10 000 examples we change it into "Is there a circle in a circle?" and after 20 000 learning examples into "Is there a triangle in a triangle?". The concept changes are ordered by difficulty, i.e. by the size of the subtree that will have to be changed. Between the first two concepts, there is only a change in the relationship of the two objects involved. The built tree will still have to verify the existence of both a circle and a triangle, but will have to change the test on the $in/2$ relation. The second change will require the change of both the $in/2$ relation and one of the existence checks. The last step changes the entire concept.

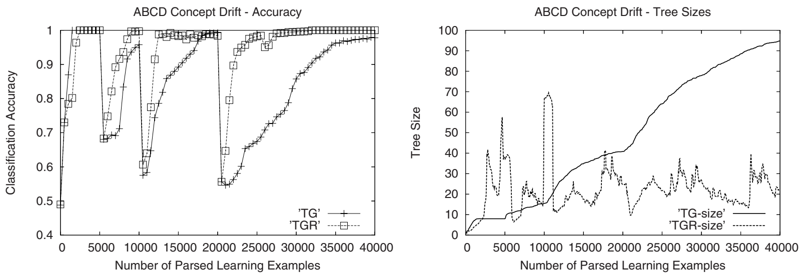

Fig. 3. Bongard problem with increasingly difficult concept changes

Figure 3 shows that TGR adapts to concept changes better and faster than the TG algorithm. TG quickly starts to slow down and, after a while, is not able to react to the concept changes within a reasonable amount of learning examples. The right graph clearly shows the advantage of TGR with respect to the size of the resulting theory. Although this is difficult to show, TGR succeeds

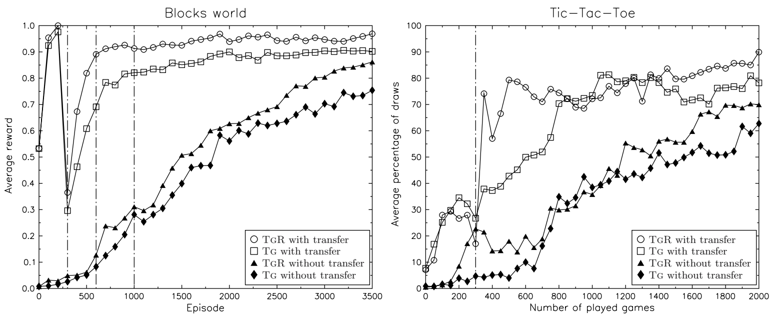

Fig. 4. The performance of RRL with and without inductive transfer

in transferring the usable part of its tree through the concept changes. The dip in performance that can be seen for TGR is largely caused by the time it takes TGR to actually notice the concept change, before it starts rebuilding its model. The fact that the second and especially the last concept change require more changes in the tree cause TGR to take a bit more time before the new concepts are learned, but the differences are minimal compared to TG.

4.2 Relational Reinforcement Learning with Transfer of Knowledge

We also present experiments with inductive transfer for relational reinforcement learning problems. In relational reinforcement learning as presented by Džeroski et al. [6], a relational regression algorithm is used to learn a generalized Q-function. We performed experiments in the blocks world [10], an often used test bed for relational reinforcement learning and in the tic-tac-toe game as an example application from the general game playing challenge [8]. In the experiments, we use two versions of the RRL system, one that uses TG as a regression algorithm to approximate the Q-function and one that uses the new TGR algorithm. We will refer to these two systems with RRL-TG and RRL-TGR respectively.

Blocks World. In the blocks world experiments, we added transfer learning to the regular blocks world setting by allowing the agent to train in worlds with easier goal-concepts before learning the target task. The agent only receives a reward when it reaches the desired goal in the minimal number of steps. If the goal is not reached in that time, the episode ends without any rewards.

We consider the $on(A, B)$-goal, i.e. block A needs to be directly on top of block B, as target task in a blocks world with 13 blocks. In the setting with inductive transfer, the agent's goal in the first 300 episodes is to clear a certain block ($clear(A)$) in a world with 4 blocks, then the goal is changed so that the agent only receives a reward iff 2 target blocks are clear at the same time ($clear(A), clear(B)$) in a world with 7 blocks. In episode 600 the goal is changed to $on(A, B)$ with 10 blocks and finally to the target goal-concept of $on(A, B)$ in an environment with 13 blocks at episode 1000. For both RRL-TG and RRL-TGR we use a language bias similar to previously reported experiments performed with RRL-TG in the blocks world [5].

Figure 4 shows the learning graph for RRL-TG and RRL-TGR both with and without transfer of knowledge. The shown received reward was obtained by freezing the Q-function approximation and testing its policy on 100 test-episodes, every 100 episodes. The results are averaged over 10 runs. The horizontal lines indicate where the goal-concept is changed. Hence the reader should compare the performance in the target task for the experiment without transfer with that for the experiment with transfer, starting after 1000 training episodes.

The performance without transfer is tested on the target goal from the start. In this setting the learning behavior of RRL-TGR is slightly better than that of RRL-TG since concept drift is by nature unavoidable in reinforcement learning. As the learning agent explores more of the world and gets a better idea of the optimal strategy, the utility (e.g. Q)-values that it computes will automatically increase and earlier underestimates should be forgotten. In the setting with inductive transfer, a good policy for the target task is learned a lot faster. Although inductive transfer also helps RRL-TG , the concept changes make it harder to learn an optimal policy.

Making a direct comparison of the RRL-TGR system to the Relational UTrees of [4] is difficult, both because they did not perform any experiments involving changing goals and because they only report results obtained by an ϵ-greedy policy on the $on(A, B)$ task in a blocks world with three blocks. We can state however that on $on(A, B)$ in a world with three blocks, RRL-TGR clearly learns much faster (converging to an optimal policy after 200-300 learning episodes with a maximum length of 3 steps) than the results reported in [4] where an "epsilon-optimal" policy is only reached after learning on 20.000 (state,action)-pairs.

Tic-tac-toe. We also present an experiment using tic-tac-toe, an application from the general game playing challenge. This is a well known two player game. If both players play optimally it results in a draw. The game is very asymmetric and although it is relatively easy to learn to draw against an optimal player using reinforcement learning when one is allowed to start the game, it becomes really hard to do this when the opposing player is allowed to act first. In fact, against a starting player that optimizes his game strategy against a random player, the probability of playing a draw with a random strategy is only 0.52%. This sparsity of the reward makes it very hard to build a good policy starting from scratch against an optimal player. We therefor devised an experiment which allowed RRL to start playing against a starting player which performs 1-step-look-ahead and transfer the learned knowledge when learning against the optimal player. The 1-step-look-ahead player will play winning moves if they exist and counter winning moves of the opponent. However, in states where neither exists, it will play randomly and therefor will not be immune to the generation of forks.

The language used by TG and TGR includes both non-game-specific knowledge (e.g. search related features) and game-specific features that can be automatically extracted from the specification of the game in the general game description language as used in the general game playing challenge.

Figure 4 shows the percentage of draws (the optimal result) RRL achieves against an optimal player both with and without inductive transfer averaged over 10 runs. In the experiment without transfer, RRL-TGR does slightly better than RRL-TG . In the case where the agent can start learning against an easier player and transfer the learned knowledge, it learns to draw much faster against the optimal player. For these experiments we let RRL practice against the 1-step-look-ahead player for the first 300 games. The first seven reported results in the graph are draws against the 1-step-look-ahead player. In fact, RRL-TGR learned to win against that player in 76% of its games after 300 training games (56% for RRL-TG). After game 300, RRL is allowed to train against the optimal player and using the results from the first 300 games, learns to achieve draws against the optimal player a lot faster. In this setting, RRL-TGR can adapt his policy a lot faster while RRL-TG needs more time to adjust its policy to the new player.

5 Conclusions and Future Work

We introduced inductive transfer in reinforcement learning through partial re-use of previously learned policies, by using an incremental learner capable of dealing with concept drift. We designed a first-order decision tree algorithm that uses four tree-restructuring operators suited for first-order decision trees to adapt its theory to changes in the target concept. These operators take the dependencies that occur between the tests in different nodes in first-order trees into account and can be applied without the need to store the entire past training experience.

Experimental results from a Bongard learning problem with concept drift showed that the resulting algorithm is indeed capable of dealing with changes in the target concept using partial theory revision. Experiments with relational reinforcement learning show both that the new algorithm allows the RRL system to react to goal changes more rapidly and to benefit from the re-use of parts of previously learned policies when the learning task becomes more difficult.

The TGR algorithm currently does not know when the agent changes to a new task. Exploiting this knowledge is an important direction for future work.

References

1. Banerjee, B., Stone, P.: General game learning using knowledge transfer. In: The 20th International Joint Conference on Artificial Intelligence, pp. 672–677 (2007)
2. Bongard, M.: Pattern Recognition. Spartan Books (1970)
3. Chapman, D., Kaelbling, L.: Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In: Proceedings of the 12th International Joint Conference on Artificial Intelligence, pp. 726–731 (1991)
4. Dabney, W., McGovern, A.: Utile distinctions for relational reinforcement learning. In: Proc. of IJCAI'07, pp. 738–743 (2007)
5. Driessens, K., Ramon, J., Blockeel, H.: Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In: Flach, P.A., De Raedt, L. (eds.) ECML 2001. LNCS (LNAI), vol. 2167, pp. 97–108. Springer, Heidelberg (2001)

6. Džeroski, S., De Raedt, L., Driessens, K.: Relational reinforcement learning. Machine Learning 43, 7–52 (2001)
7. Fernández, F., Veloso, M.: Probabilistic policy reuse in a reinforcement learning agent. In: AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, pp. 720–727. ACM Press, New York (2006)
8. Genesereth, M., Love, N.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005)
9. Madden, M., Howley, T.: Transfer of Experience Between Reinforcement Learning Environments with Progressive Difficulty. AI Rev. 21(3-4), 375–398 (2004)
10. Slaney, J., Thiébaux, S.: Blocks world revisited. AI Jour. 125(1-2), 119–153 (2001)
11. Taylor, M., Whiteson, S., Stone, P.: Transfer via inter-task mappings in policy search reinforcement learning. In: AAMAS'07 (2007)
12. Torrey, L., Shavlik, J., Walker, T., Maclin, R.: Skill acquisition via transfer learning and advice taking. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 425–436. Springer, Heidelberg (2006)
13. Utgoff, P.: Decision tree induction based on efficient tree restructuring. Machine Learning 29(1), 5–44 (1997)
14. Widmer, G., Kubat, M.: Learning in the presence of concept drift and hidden contexts. Machine Learning 23(2), 69–101 (1996)