Function Approximation in Hierarchical Relational Reinforcement Learning

Silvana Roncagliolo SILVANA@UCV.CL
Escuela de Ingenieria Informatica, Universidad Catolica de Valparaiso, Chile

Prasad Tadepalli TADEPALL@EECS.ORST.EDU
School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR, USA

Abstract

Recently there have been a number of different approaches developed for hierarchical reinforcement learning in propositional setting. We propose a hierarchical version of relational reinforcement learning (HRRL). We describe a value function approximation method inspired by logic programming which is suitable for HRRL.

1. Introduction

Hierarchical methods appear crucial to scale reinforcement learning to large real-world domains. The MAXQ value function decomposition is one of the ways in which hierarchical methods can be implemented in a principled manner. Decomposing the value function of a state into sub-value functions that belong to subtasks allows the learning algorithm to choose subtasks at each level using their own value functions. While this does not guarantee a globally optimal policy, or an optimal policy that can be represented hierarchically, it does learn a recursively optimal policy, i.e., a policy which is optimal given all the lower level subtasks are optimal. Extending hierarchical reinforcement learning to relational domains is thus an important problem.

The relative merits of policy-based and value-function-based methods have been a point of debate in the reinforcement learning community. This issue is particularly pressing in relational reinforcement learning (RRL), because it has been found that value-function based methods do not take advantage of the variable number of objects present in relational domains and hence do not generalize well. For example, the value function approximation algorithm called TILDE, which was used in relational Q-learning, is unable to generalize a policy from 2 blocks to 3 blocks. This led researchers to develop policy learning methods based on inductive logic programming and approximate policy iteration (Džeroski et al., 2001; Fern et al., 2003). Unfortunately, the rules learned by these methods can sometimes be complicated, or unable to represent a near-optimal policy.

One reason for the policies learned by policy-search methods to be complicated is that they are expressed as constraints on low level operators. In many complex domains, it is more natural and easier to express the target policy as constraints on higher level goal-subgoal decompositions. For example, to achieve `on(A,B)`, all that one needs to do is to first achieve `clear(A)` and `clear(B)`, and then apply the operator `puton(A,B)`. To achieve `clear(A)`, when `on(C,A)` is true, one has to first achieve `clear(C)`, which implies that the blocks above `C` must be recursively cleared. This gives a simple set of recursive rules that can always achieve `on(A,B)`. Goal-subgoal (or task-subtask) hierarchies also naturally and frequently arise in most other planning domains including logistics, information integration, cooking, etc.

In the current paper, we explore this hypothesis by presenting a natural extension of hierarchical reinforcement learning (Dietterich, 2000; Sutton et al., 1999; Parr & Russell, 1998; Kaelbling, 1993) to relational setting. We also present a relational function approximator in the form of Prolog-like rules with linear functions. We describe an algorithm to learn these rules and present some preliminary results in the blocks world domain. We conclude with some possible future extensions.

Appearing in *Proceedings of the ICML'04 workshop on Relational Reinforcement Learning*, Banff, Canada, 2004. The second author acknowledges the support of NSF under grant number IIS-0329278.

2. Hierarchical Relational MDPs

We consider a relational markov decision problem (RMDP) framework (O, P, S, A, T, R) where O is a set of objects, P is a set of predicates over objects, S is the set of all possible complete state specifications over O and P, A is the set of all possible instantiated actions, T is a state transition function that specifies the probability of the next state given the state and the (instantiated) action, and R is the immediate reward as a function of the current state and (instantiated) action. We assume that the set of objects O and the predicate symbols and their arities are known. Similarly, the action schema names, e.g., `puton`, and their numbers of parameters are known. The transition probabilities of actions and their immediate rewards are unknown. The goal is to optimize the expected total reward received.

We extend the above framework to a hierarchical setting similar to the MAXQ framework (Dietterich, 2000). We now have a task graph $G = (N, E)$ that specifies the task hierarchy. Each node $u \in N$ represents a parameterized task with subtasks $\{v|(u, v) \in E\}$. The subtasks are the possible actions at the disposal of the task, a set of subroutines it can call in some order. The graph G itself may be implicitly specified by a set of "decomposition rules" that describe how a task is to be decomposed into subtasks. For the purposes of this paper, we assume that this graph is an explicit input to the system.

Unlike the MAXQ framework of hierarchical RL (Dietterich, 2000), we do not restrict the task graph to be acyclic, thus facilitating recursion. A task can be achieved by repeatedly choosing an appropriate subtask in the current state that helps achieve the task. For example, the goal of `on(A,B)` may be achieved by achieving `clear(A)`, `clear(B)` and `puton(A,B)` in some order. The goal of `clear(A)` may be achieved by `clear(X)` and `puton(X,table)` in some order with suitable instantiations of the variable `X` (`X` should be above `A` for this move to be useful).

Consider a task i and its subtask j in the task graph. The Bellman equation for the value of task i, i.e., the expected total reward received during the task i, for state s with j as the first subtask is represented by the Q-function $Q(i, s, j)$, and is given by the following equation for expected total reward optimization.

$$Q(i, s, j) = E(V(s, j) + V(s', i)) \qquad (1)$$

where s' is the state where the subtask j terminates starting from s, $E(.)$ represents the expected value, and $V(s, j)$ is the expected value of completing task j starting from s and is given by:

$$V(s, j) = \begin{array}{ll} E(R(s, j)) & \textit{if } j \textit{ is primitive,} \\ \max_k Q(s, j, k) & \textit{otherwise.} \end{array} \qquad (2)$$

The above equation is analogous to the definition of the Q-function, except that the first term in the first equation represents the reward obtained in solving the subtask j, which in turn is expressed recursively as a function of the rewards obtained during its subtasks. It can be turned into an update equation in the following way:

$$Q(i, s, j) \leftarrow (1-\alpha)Q(i, s, j)+\alpha(V(s, j)+V(s', i)) \quad (3)$$

An average-reward version of the above method is described in (Seri & Tadepalli, 2002). It is found to be more efficient than the more standard MAXQ-Q learning in our experiments, partly because it caches the values of all subtasks and uses them to update the parent task's values rather than recursively descending all the way to the leaf nodes. Unfortunately, neither of these methods can work without proper abstraction of the state at each task level. In relational RL, the abstraction is more complicated because it relies on the relationships between different features rather than the presence or absence of a single feature. We make use of a full-fledged relational function approximator to make MAXQ-learning effective in relational domains.

3. Value Function Approximation

Previous experience with value function approximation in relational settings indicates that the value function generalizes poorly when it is not sufficiently expressive. For example, TILDE uses relational regression trees, whose leaves are assigned constant values (Džeroski et al., 2001). Thus, the value function is piecewise constant, an inappropriate choice for relational domains like blocks world, where the number of time steps to do something usually depends on the number of blocks that satisfy a condition, such as being above a certain block. For example, the number of steps necessary to clear a block is a linear function of the number of blocks above that block. Approximating it with a constant will not allow it to be applicable to a different number of blocks.

In many domains including the blocks world, it is perhaps more natural to learn piecewise *linear* functions. We represent the Q-values using a 3-place predicate, `Q(Task,Subtask,Val)` which means that the value of

```
q(clear(X),_,0) :- clear(X).
q(clear(X),clear(Y),V) :- on(Y,X),
   q(clear(Y),_,V1), V is V1-1.
q(on(X,Y),clear(Y),V) :- clear(X),
   q(clear(Y),_,V1), V is V1-1.
```

Table 1. A set of rules that compute the Q-values

achieving `Task` by achieving the `Subtask` is `Val`. The state is an implicit parameter in that all the predicates in the rule are evaluated at the same state. This predicate may in turn be expanded by a set of rules, which might include linear functions. For example, one such rule set is given in Table 1 in Prolog notation. Note that the reward for each step is assumed to be -1 and the reward for the goal state is 0.

The first rule in Table 1 says that the if a block `X` is already clear, then the value of the current state during the task of clearing it is 0. The second rule says that if a block `Y` is on `X`, then the total reward for clearing `X` is the total reward for clearing `Y` minus 1. The third rule says that if a block `X` is clear, then the the total reward for putting `X` on `Y` is the reward for clearing `Y` minus 1.

We have implemented a batch learning algorithm to learn rules of the above kind from user-given examples of Q-values. We have yet to integrate this batch function approximator with reinforcement learning. Our algorithm is similar to the FOR algorithm of (Karalic & Bratko, 1997) and works with a FOIL-like greedy search (Quinlan, 1990) coupled with linear regression.

Currently, the input to our algorithm is a set of examples, each of which consists of a fully specified state, the task, the first subtask, and the Q-value (see Table 2). The state is described as a conjunction of literals. The task and the subtask are currently single designated literals. The value is a real number that represents the total reward obtained in the given examples.

In addition, the input includes the task-subtask hierarchy. So for the blocks world domain, we have the hierarchy list (clear, clear), (on, on), (on, clear), meaning that "clear" only has "clear" as its subgoal, while "on" has both "on" and "clear" as subgoals. Also input to the algorithm is the maximum number of terms that can be part of the conditions of the learned rules.

The "basis functions" (features) for the value function of a task consists of numerical arguments as well as the results of value functions of its subtasks. The value function is assumed to be piece-wise linear in the basis functions and includes a bias (constant) term. Learning is done for the tasks in the hierarchy in a bottom-up manner.

State:	`[clear(a), on(a,b), on(b,c), on(c,d), on(d,table), clear(e), on(e,f), on(f,g), on(g,table), clear(table)]`
task:	`clear(c)`
subtask:	`clear(b)`
value:	-2
task:	`on(c,f)`
subtask:	`clear(c)`
value:	-4

Table 2. Two Examples for Function Approximation

```
Learn(Examples)
  For each task-subtask pair
    Let Exs := Examples for the current task-subtask
    Repeat
      Rule := LearnBestRule(Exs);
      Exs := Exs - {ex | ex matches Rule's condition }
    Until Exs is { }
```

Table 3. The top-level greedy algorithm

We use a greedy covering algorithm like FOIL (Quinlan, 1990) to learn the value function as a set of rules. It separates the examples for each task-subtask pair, and finds the best rule that minimizes the squared error with respect to those examples (see Table 3). Thus a list of rules is learned for each task-subtask pair. Each rule has an applicability condition (if part) which binds some variables, and a linear function of these variables (the then part) which predicts the value function of the state.

The best rule is found by incrementally adding conditions to the if-part of the rule, collecting all the bound numeric variables as features and then doing a linear regression on the resulting features (see Table 4). The appropriate condition literals include all the predicates applicable in the state as well as the value functions for the subtasks of the given task. The variables for doing linear regression include all the count variables that were bound in the if-part, and also the values of the subtasks of the given task. All possible extensions of the current if-part are considered, and for each possible extension, linear regression is performed on the resulting variables. The literal that yields the least possible regression error is finally chosen to be added to the condition, the examples are updated to match the rule constructed so far, and the algorithm continues to find

```
LearnBestRule(Examples)
  Rule := Empty;
  Repeat
    for each possible condition
      determine the possible features
      exs := the examples that satisfy the condition
      regressionError, linearFunction := Regress(exs,
                                  features)
      if regressionError < minError then
        minError := regressionError
        bestCondition := condition
        bestFunction := linearFunction
    end for;
    add bestCondition to the if-part of the rule;
    Examples := Examples that match bestCondition;
  until minError < ε
  add the bestFunction to the then-part of the rule.
```

Table 4. The Greedy Regression Algorithm

```
1  q(clear(X),clear(X),V):- clear(X), V is 0.
2  q(clear(X),clear(Z),V):- on(Z,X),
       q(clear(Z),_,V1), V is V1 - 1.
3  q(on(X,Y),clear(X),V):- clear(Y),
       q(clear(X),_,V1), V is V1 - 1.
4  q(on(X,Y),clear(X),V):- below (Y,X),
       q(clear(Y),_,V1), V is V1 - 1.
5  q(on(X,Y),clear(X),V):- q(clear(X),_,V1),
       q(clear(Y),_,V2), V is V1 + V2 - 1.
6  q(on(X,Y),clear(Y),V):- below(X,Y),
       q(clear(X),_,V1), V is V1 - 1.
7  q(on(X,Y),clear(Y),V):- clear(X),
       q(clear(Y),_,V1), V is V1 - 1.
8  q(on(X,Y),on(X,Y),V):- on(X,Y), V is 0.
```

Table 5. The Rules Learned by Greedy Regression

the next literal to be added. If the regression error is less than a preset parameter ϵ, the algorithm terminates the condition part of the rule and adds the best linear function found to the then-part of the rule.

4. Results and Future Work

So far we only have done preliminary experiments in the blocks world domain. We generated all possible examples from 1, 2, and 3 states. The total number of examples ranged from 70 in the 1-state case to about 130 in the case of 3 states. We then ran our function approximation algorithm on it to learn a set of rules. Our system was able to learn the rules shown in Table 5 for the goals of "clear" and "on" in the blocks world.

Note that some of the rules, e.g., rule 4 and 6 use the high-level predicate "`below`". All predicates are given to the system as prior knowledge. Rule 4 says that if `Y` is `below X`, then putting `X` on `Y` takes only one more step after clearing `Y`. Rule 5 covers the default case of having to `clear` both blocks and then put one on the other.

There is much that remains to be done. We need to do a bigger experimental study in the blocks world and other domains and evaluate the algorithm more thoroughly. It appears that the condition selection can be made more efficient by adding heuristics. Rather than having to specify the first subtask explicitly, we would like to learn it from the examples. This seems possible by the strategies we have previously explored to learn goal decomposition rules from user-given examples (Reddy et al., 1996; Reddy & Tadepalli, 1999). We assume that predicates like "`below`" are already known to the system. Introducing such useful new predicates automatically is an important open problem. Finally, we need to incorporate this algorithm into a full reinforcement learner that generates its own examples rather than being supplied with solved examples. Generalizing the algorithms to stochastic domains is another important direction.

References

Dietterich, T. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research, 13*, 227–303.

Džeroski, S., De Raedt, L., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning, 43*, 7–52.

Fern, A., Yoon, S., & Givan, R. (2003). Approximate policy iteration with a policy language bias. *Advances in Neural Information Processing Systems, 16*.

Kaelbling, L. (1993). Hierarchical learning in stochastic domains: Preliminary results. *Proceedings of the Tenth International Conference on Machine Learning* (pp. 167–173).

Karalic, A., & Bratko, I. (1997). First order regression. *Machine Learning, 26*, 147–176.

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems, 10*.

Quinlan, J. (1990). Learning logical definitions of from relations. *Machine Learning, 5*, 239–266.

Reddy, C., & Tadepalli, P. (1999). Learning Horn definitions: Theory and an application to planning. *New Generation Computing*, *17*, 77–98.

Reddy, C., Tadepalli, P., & Roncagliolo, S. (1996). Theory-guided empirical speedup learning of goal-decomposition rules. *Proceedings of the 13th International Conference on Machine Learning* (pp. 409–417). Bari, Italy: Morgan Kaufmann.

Seri, S., & Tadepalli, P. (2002). Model-based hierarchical average-reward reinforcement learning. *Proceedings of International Machine Learning Conference*. Sydney, Australia: Morgan Kaufmann.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*, 181–211.