The Carli Architecture–Efficient Value Function Specialization for Relational Reinforcement Learning

Mitchell Keith Bloch
University of Michigan
Ann Arbor, MI, USA
bazald@umich.edu

John Edwin Laird
University of Michigan
Ann Arbor, MI, USA
laird@umich.edu

Abstract

We introduce Carli–a modular architecture supporting efficient value function specialization for relational reinforcement learning. Using a Rete data structure to support efficient relational representations, it implements an initially general hierarchical tile coding and specializes it over time using a fringe. This hierarchical tile coding constitutes a form of linear function approximation in which conjunctions of relational features correspond to weights with non-uniform generality. This relational value function lends itself to learning tasks which can be described by a set of relations over objects. These tasks can have variable numbers of both features and possible actions over the course of an episode and goals can vary from episode to episode. We demonstrate these characteristics in a version of Blocks World in which the goal configuration changes between episodes. Using relational features, Carli can solve this Blocks World task, while agents using only propositional features cannot generalize from their experience to solve different goal configurations.

1 Introduction

Many kinds of problems can be formulated as reinforcement learning problems. Simple reinforcement learning algorithms (such as both Sarsa(λ) and Q(λ)) directly suffer from the curse of dimensionality. The number of Q-values to be learned scales directly with the product of the size of the state-space, $|D_1 \times D_2 \times D_3 \times \dots|$, and the size of the action-space, $|\{A_1, \dots, A_n\}|$. As the state-space and action-space grow large for more complex problems, it becomes difficult to learn efficiently. The issue is a lack of some capacity to generalize.

Linear function approximation is one approach to adding the ability to generalize to these temporal difference methods. The simple value function can be replaced by n basis functions, $\phi_1, \dots, \phi_n$. The Q-function is estimated by $Q(s,a) = \sum_{i=1}^{n} \phi_i(s,a) w_i$, where $\phi_i(s,a)$ is commonly 1 or 0 to indicate whether the feature is currently active for a given state and action, or not, and where the weight, w_i, represents the value contribution of the feature for the action under consideration, or a partial Q-value. That each feature applies to many states provides the agent with the ability to learn about states which the agent has yet to visit. For example, the feature (*stack*, matches, *goal–stack*) applies to many actions throughout the state-space and not just to one specific state-action pair.

A second approach to support generalization from experience is to start with a coarse tile coding, and to break the tiles into smaller, more specific tiles over time. As the agent specializes its value function, it learns a more refined policy, but without the full cost associated with starting with a maximally specialized value function from the beginning. It is possible to combine this with linear function approximation by keeping coarser, more general tiles in the system and allowing general learning to continue even as the smaller, more specialized tiles are added [Bloch and Laird, 2013]. This strategy assumes that the hierarchy accurately captures the structure of the true value function to some degree, allowing more general patterns in the value function to be captured by the more general tiles and requiring the more specific tiles to capture the finer details.

Relational reinforcement learning is a third approach to support generalization from experience [Tadepalli *et al.*, 2004; Džeroski *et al.*, 2001]. Given relations and conjunctions thereof, it is possible to condense both the state-space and action-space for certain tasks. For example, if the agent is concerned that (*stack*, matches, *goal–stack*), it can learn about all actions in which that match is present, regardless of the labels of any of the blocks in either the *stack* or the *goal–stack*. As these features do not always depend on the size of the state-space and action-space, they can be particularly effective for scenarios with arbitrarily large state-spaces.

As relational reinforcement learning allows an agent to reason about objects more abstractly, it allows the agent to generalize from its experience to solve problems from outside its training set. The alternative of propositionalizing the relational representation tends to create far more features and has more difficulty transferring to similar problems.

We develop the Carli[1] architecture using all three of these methods to provide generalization. The intent is to explore not only how to efficiently implement a relational reinforcement learning agent with dynamically variable specialization, but to investigate how best to do this specialization and learn with this kind of system. The manner in which the Carli architecture maps states and actions to weights and specializes that mapping over time is critical. As explored by Bloch and Laird [2013], Whiteson *et al.* [2007], and McCallum [1996], Carli uses a tile coding that begins with large, general tiles and then specializes it over time to generate smaller tiles and more specific weights. As done by McCallum [1996], a fringe is stored alongside each leaf weight to allow the agent to make informed decisions about which features to use next when specializing a tile. However, instead of collecting instances, we store fixed-size metrics to be used by different specialization criteria. And like the system explored by Bloch and Laird [2013], it keeps more general tiles in the system as more specific ones are added, allowing more general learning to continue as the policy is refined. The major contribution of Carli is its unique use of the Rete algorithm to implement a computationally efficient value function, providing a specializable, hierarchical tile coding that supports relational features.

2 Related Work

Efficient rule matching is a well established problem in the context of rule-based systems. We map the problem of providing weights for linear function approximation onto this matching problem. The Soar cognitive architecture [Laird, 2012] and Soar-RL [Nason and Laird, 2004]–its implementation of reinforcement learning–were the primary inspiration for this work. Soar would require significant modification to support efficient value function specialization as implemented in Carli.

Driessens *et al.* [2001] implemented the RRL-TG algorithm. They implemented query packs as a way to use more memory to allow common parts of queries to be represented only once in their data structure. This optimization enforces a similar structure to that of Rete, sharing earlier tests in order to save work. RRL-TG requires specializations to be binary splits based on truth tests, which Carli does not require.

Adlin being the name of their Icarus-based architecture, Asgharbeygi *et al.* [2005] noted that "when the time constraint is extremely strict or the environment is changing too rapidly, Adlin does not provide a satisfying performance gain." The goal of their relational reinforcement learning system is to guide inference rather than to provide near optimal control, but one solution they considered to deal with their performance problem as part of their future work was "to incorporate the simple idea behind truth maintenance systems and Rete matchers," which is related to what we have done in our work.

In order to support online, relational reinforcement learning, it is necessary to support efficient detection of conjunctions of relational features specifically. A system explored by Bloch and Laird [2013] that used a trie to store the value function did not support the use of relational features. Their trie implementation was efficient but ill suited for supporting relational tests. It may have been possible to modify the implementation to support a variable number of actions, but any relational features would have needed to be be propositionalized. This propositionalization would have increased the number of features considerably and, due to the lack of a notion of variable binding, additionally made it difficult to support conjunctions of relational features that refer to the same variables.

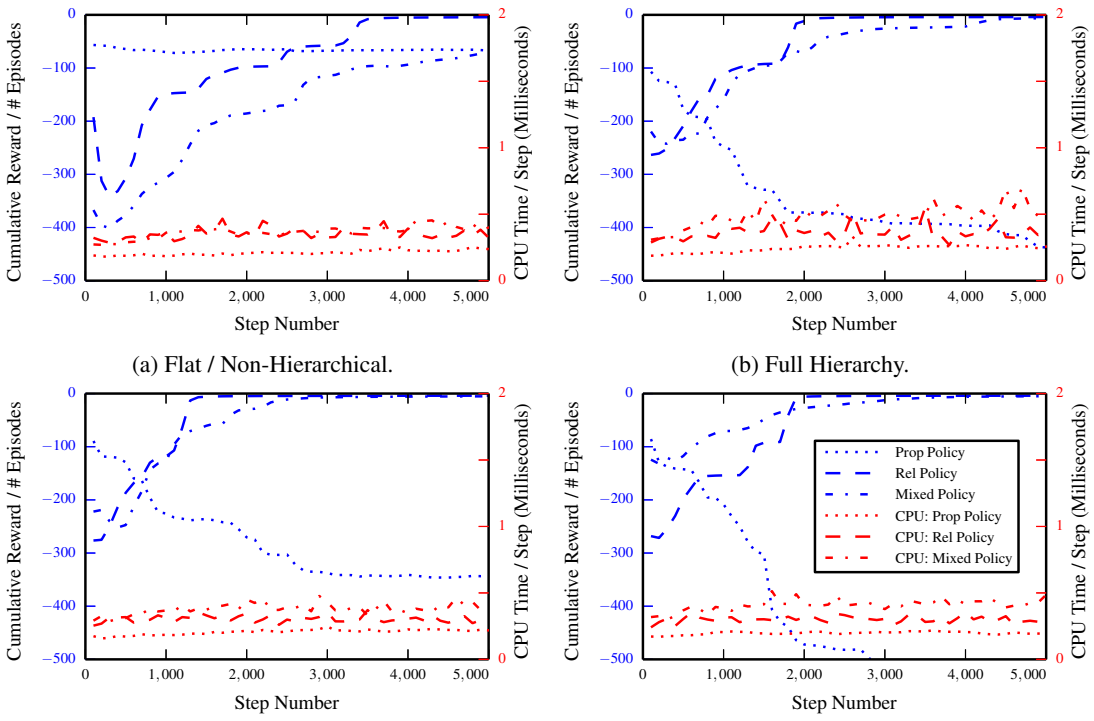
3 Carli

We use Rete [Forgy, 1979] to provide efficient rule matching and weight retrieval for our agents. Rete provides the relational structure, variable bindings, and efficiency we need to support relational features. Hierarchical tile codings are implemented in the Rete as a set of rules sharing the same tests up until feature specialization occurs, and the work to fire these shared tests is effectively free in the Rete. The cost of minimally firing and retracting rules to match changes in the environment is linear in the number of changes rather than in the size of the state-space. Efficient implementation of the Rete algorithm has been explored in detail by Doorenbos [1995].

One way to specialize a tile coding over time is to maintain a fringe of possible future specializations and to conditionally (or periodically) choose from among those possible specializations to expand the value function. As part of that expansion, it is necessary to create a new fringe for the new tiles which have been added to the tile coding.

The hierarchical structures of both Rete and this kind of tile coding line up in a way that makes their joint implementation very convenient, albeit non-trivial. We found it useful to define a rule grammar to specify the combinations of features. The rules adhering to this grammer together constitute our tile coding. Given our grammar, we can extract details about the types of features the rules are adding to the system. At the time of fringe expansion, we can separate the new features from the rules (in the case that they correspond to fringe tiles) and modify other rules to incorporate the new features in order to create a new fringe. Similarly, it is possible to collapse a tile coding to undo earlier specializations should it be deemed useful to do so.

[1] Carli can be downloaded from https://github.com/bazald/carli.



(a) Flat / Non-Hierarchical.

(b) Full Hierarchy.

(c) Value Criterion.

(d) Policy Criterion.

Figure 1: These plots display task performance (in blue) and total system CPU time (in red) for Carli agents in Blocks World. Propositional features are dotted and relational features are dashed. The legend in figure 1d applies to all figures 1a through 1d.

4 Blocks World

Blocks World presents a configuration of blocks, each with a unique label. The agent is tasked with moving one block at a time from the top of a stack of blocks to either the top of a different stack or the table. The agent completes the task when the blocks configuration matches the goal configuration.

The Blocks World task that interests us exposes the goal configuration to the agent and changes the goal from episode to episode. Agents using a standard propositional representation (dotted in figure 1) of this formulation of the Blocks World task should be unable to learn a near optimal strategy for solving new goal configurations because they must ultimately capture some details of the task specific to a particular goal configuration in order to function at all. Agents using relational representations (dashed in figure 1) should can capture connections between stacks of blocks in both the current configuration and the goal configuration, generalizing to different goal configurations. Agents using both representations together (dash-dotted in figure 1) should still be able to learn to solve the task, though they may require more time to do so.

The propositional features available to our agents are: whether the destination block is in the right place or not (a propositionalized relational feature that is equivalent to the second relational feature below); the name of the block to be moved; and the name of the destination block.

The relational features available to our agents are: whether the source stack of blocks matches a goal stack or not; a stack that is a subset of a goal stack would be considered matching; placing a block on top of such a stack would cause it to no longer match; whether the destination stack matches a goal stack or not; whether the top block of the source stack matches the top of the destination stack or not (a top block being considered to match the top of the destination stack if and only if the destination stack matches a goal stack and the additional block will allow it to continue to match); and whether the destination (stack) is the table.

We train our Carli agents on-policy with a discount rate of 0.9 and an eligibility trace decay rate of 0.3 using Blocks World instances with 4 blocks. Figures 1a through 1d plot the cumulative reward divided by the number of episodes solved and the current computational time required per step of the environment.

Figures 1a and 1b depict performance of agents doing all specialization before the first step. Both sets of agents specialize maximally before the first step, choosing features at random. The flat agents discard more general features from the system while the hierarchical agents keep the more general features for linear function approximation. It is clear that the hierarchical tile coding is advantageous to the agents that include the relational features, although it also allows divergence in the agent using only propositional features.

Figures 1c and 1d depict performance of agents doing online specialization after gaining some experience, but still early in their lifetimes. The agents using the value criterion attempt to minimize error when specializing the value function by choosing feature axes in the fringes which present large value discrepancies between Q-values. The agents using the policy criterion attempt to be more tolerant of error in the value function by choosing only feature axes that are likely to alter their policies when specializing the value function. These specialization criteria are derived from those presented by Whiteson *et al.* [2007]. The performance pattern for agents using the different sets of features predicted at the beginning of this section holds.

The memory utilization differs depending primarily on which features are included. The agents using only the propositional features tend to include approximately 52 weights in their value function. The agents using only the relational features tend to include approximately 25. And agents using both sets of features tend to include between 300 and 350 weights (304 for the value criterion, and 350 for the policy criterion, and 390 for the full hierarchy). As shown in the figures, the computational cost scales sublinearly with memory usage, although more selective criteria may be needed to keep memory costs down in large domains. Computing the relational features is more expensive than computing the propositional features, but their increased power is clearly valuable.

5 Future Work

We intend to explore additional domains in the future, including Infinite Mario [Togelius *et al.*, 2010]. However, the number of distinct variables is currently fixed by the features initially provided to the system. One extension would be to allow Carli agents to reason about a number of different blocks, enemies, and pits in the environment without enumerating a fixed number of block, enemy, and pit variables ahead of time. Its unclear whether this higher order fringe grammar would have sufficient value to be worth the complications involved in its implementation.

Adding support for Greedy-GQ(λ) [Maei, 2011], a temporal difference method that is guaranteed to converge when using linear function approximation, would probably improve the performance of Carli agents.

We could benefit from better tests to determine which features to include, in addition to our value and policy criteria. There is a tradeoff between specializing quickly and specializing slowly. If our system specializes more quickly it can begin learning a more specialized policy faster, but it may be more likely to geneate a suboptimal Rete structure.

Our Rete value function implementation supports efficient despecialization to undo suboptimal specializations, but we have yet to develop criteria to allow us to decide when to despecialize or respecialize with different features tests higher up in the Rete.

References

[Asgharbeygi *et al.*, 2005] Nima Asgharbeygi, Negin Nejati, Pat Langley, and Sachiyo Arai. Guiding inference through relational reinforcement learning. In *Proceedings of the 15th International Conference on Inductive Logic Programming*, ILP'05, pages 20–37, Berlin, Heidelberg, 2005. Springer-Verlag.

[Bloch and Laird, 2013] Mitchell Keith Bloch and John Edwin Laird. Online value function improvement. In *RLDM 2013: Extended Abstracts*, pages 111–115, 2013.

[Doorenbos, 1995] Robert B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Pittsburgh, PA, USA, 1995. UMI Order No. GAX95-22942.

[Driessens *et al.*, 2001] Kurt Driessens, Jan Ramon, and Hendrik Blockeel. Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In *Machine Learning: ECML 2001*, pages 97–108. Springer, 2001.

[Džeroski *et al.*, 2001] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine learning*, 43(1-2):7–52, 2001.

[Forgy, 1979] Charles Lanny Forgy. *On the Efficient Implementation of Production Systems.* PhD thesis, Pittsburgh, PA, USA, 1979. AAI7919143.

[Laird, 2012] John E. Laird. *The Soar Cognitive Architecture*. The MIT Press, 2012.

[Maei, 2011] Hamid Reza Maei. *Gradient Temporal-Difference Learning Algorithms*. PhD thesis, 2011.

[McCallum, 1996] Andrew Kachites McCallum. *Reinforcement learning with selective perception and hidden state*. PhD thesis, 1996. AAI9618237.

[Nason and Laird, 2004] Shelley Nason and John E. Laird. Soar-rl: Integrating reinforcement learning with soar. In *Cognitive Systems Research*, pages 51–59, 2004.

[Tadepalli *et al.*, 2004] Prasad Tadepalli, Robert Givan, and Kurt Driessens. Relational reinforcement learning: An overview. In *Proceedings of the ICML-2004 Workshop on Relational Reinforcement Learning*, pages 1–9, 2004.

[Togelius *et al.*, 2010] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 mario ai competition. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

[Whiteson *et al.*, 2007] Shimon Whiteson, Matthew E. Taylor, and Peter Stone. Adaptive tile coding for value function approximation, 2007.