Adaptive Tile Coding for Value Function Approximation

Shimon Whiteson, Matthew E. Taylor, and Peter Stone
Department of Computer Sciences
University of Texas at Austin
1 University Station, C0500
Austin, TX 78712-0233
{shimon,mtaylor,pstone}@cs.utexas.edu

Abstract

Reinforcement learning problems are commonly tackled by estimating the optimal value function. In many real-world problems, learning this value function requires a function approximator, which maps states to values via a parameterized function. In practice, the success of function approximators depends on the ability of the human designer to select an appropriate representation for the value function. This paper presents *adaptive tile coding*, a novel method that automates this design process for tile coding, a popular function approximator, by beginning with a simple representation with few tiles and refining it during learning by splitting existing tiles into smaller ones. In addition to automatically discovering effective representations, this approach provides a natural way to reduce the function approximator's level of generalization over time. Empirical results in multiple domains compare two different criteria for deciding which tiles to split and verify that adaptive tile coding can automatically discover effective representations and that its speed of learning is competitive with the best fixed representations.

Introduction

In *reinforcement learning* (RL) problems, an agent must learn a policy for solving a sequential decision task. The agent never sees examples of correct behavior but instead receives positive or negative rewards for the actions it tries. From this feedback it must find a policy that maximizes reward over the long term. RL tasks are most commonly tackled with methods, such as *dynamic programming* (Bellman 1957), that learn *value functions*.

Value functions indicate, for some policy, the long-term expected value of a given state. The agent strives to learn the optimal value function, from which it can deduce an optimal policy. For small problems, the value function can be stored in a table. However, many real-world problems have large or continuous state spaces, rendering this approach infeasible. Such tasks usually require *function approximators*, which represent value estimates with parameterized functions.

However, using function approximators requires making crucial representational decisions (e.g. the number of hidden nodes in a neural network or the resolution of a state space discretization). Poor design choices can result in estimates that diverge from the optimal value function (Baird 1995) and agents that perform poorly. Even for reinforcement learning algorithms with guaranteed convergence (Baird & Moore 1999; Lagoudakis & Parr 2003), achieving high performance in practice requires finding an appropriate representation for the function approximator. Nonetheless, representational choices are typically made manually, based only on the designer's intuition.

The goal of our research is to automate this process by enabling an RL agent to autonomously find a good representation for its function approximator. Adaptive methods already exist for neural network function approximators (Whiteson & Stone 2006) and piecewise-linear representations based on kd-trees (Munos & Moore 2002). We contribute an adaptive method for *tile coding* (Albus 1981), a simple, computationally efficient function approximator that has enjoyed broad empirical success (Sutton 1996; Stone, Sutton, & Kuhlmann 2005). Tile coding, which forms a piecewise-constant approximation of the value function, requires a human designer to choose the size of each tile in each dimension of the state space.

This paper introduces *adaptive tile coding*, which automates this process by starting with large tiles and making them smaller during learning by splitting existing tiles in two. Beginning with simple representations and refining them over time is a strategy that has proven effective for other function approximators (Chow & Tsitsiklis 1991; Munos & Moore 2002; Whiteson & Stone 2006). In addition to automatically finding good representations, this approach gradually reduces the function approximator's level of generalization over time, a factor known to critically affect performance in tile coding (Sherstov & Stone 2005).

To succeed, adaptive tile coding must make smart decisions about which tiles to split and along which dimension. This paper compares two different criteria for prioritizing potential splits. The *value criterion* estimates how much the value function will change if a particular split occurs. By contrast, the *policy criterion* estimates how much the policy will change if a given split occurs.

Empirical results in two benchmark reinforcement learning tasks demonstrate that the policy criterion is more effective than the value criterion. The results also verify that adaptive tile coding can automatically discover representations that yield approximately optimal policies and that the adaptive approach's speed of learning is competitive with the best fixed tile-coding representations.

Background

This section reviews the reinforcement learning formalism and briefly describes tile coding representations and how they are used to approximate value functions.

Markov Decision Processes

An RL task is defined by a *Markov decision process* (MDP). For simplicity, this paper focuses on MDPs that are continuous but deterministic, though in principle the methods presented could be extended to stochastic domains. An MDP consists of a four-tuple $\langle S, A, T, R \rangle$ containing an infinite set of states S, a finite set of actions A, a transition function $T : S \times A \rightarrow S$ and a reward function $R : S \times A \rightarrow \Re$. An agent in state $s \in S$ that takes action $a \in A$ will transition to state $T(s, a)$ and receive reward $R(s, a)$.

The agent's goal is to learn an optimal policy $\pi^* : S \rightarrow A$ that maximizes the long-term discounted reward accrued. As in previous work on function approximation (Chow & Tsitsiklis 1991; Gordon 1995; Munos & Moore 2002), we assume the agent has a model of its environment (i.e. T and R are known). Given a model, the agent need only learn the optimal value function $V^* : S \rightarrow \Re$, where $V^*(s)$ is the long-term discounted reward the agent will receive starting in state s if it follows π^* thereafter. Once V^* is known, π^* is easily determined:

$$\pi^*(s) = \text{argmax}_a [R(s, a) + \gamma V^*(T(s, a))]$$

where $\gamma \in [0, 1]$ is a discount factor. To make function approximation feasible, we assume S is *factored* such that each state $s \in S$ is a vector of *state variables*: $s = \langle x_1, x_2, \ldots, x_k \rangle$ where $x_i \in \Re$.

Tile Coding

In tile coding (Albus 1981), a piecewise-constant approximation of the optimal value function is represented by a set of exhaustive partitions of the state space called *tilings*. Typically, the tilings are all partitioned in the same way but are slightly offset from each other. Each element of a tiling, called a *tile*, is a binary feature activated if and only if the given state falls in the region delineated by that tile. Figure 1 illustrates a tile-coding scheme with two tilings.

The value function that the tile coding represents is determined by a set of weights, one for each tile, such that

$$V(s) = \sum_{i=1}^{n} b_i(s) w_i$$

where n is the total number of tiles, $b_i(s)$ is the value (1 or 0) of the ith tile given state s, and w_i is the weight of that tile. In practice, it is not necessary to sum over all n tiles since only one tile in each tiling is activated for a given state. Given m tilings, we can simply compute the indices of the m active tiles and sum their associated weights.

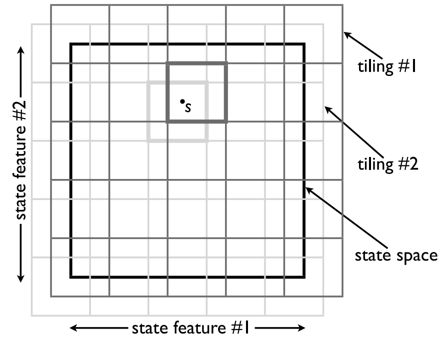

Figure 1: An example of tile coding with two tilings. Thicker lines indicate which tiles are activated for the given state s.

Given a model of the MDP as described above, we can update the value estimate of a given state s by computing $\Delta V(s)$ using dynamic programming:

$$\Delta V(s) = \max_a [R(s, a) + \gamma V(T(s, a))] - V(s)$$

and adjusting each weight so as to reduce $\Delta V(s)$:

$$w_i \leftarrow w_i + \frac{\alpha}{m} b_i(s) \Delta V(s)$$

where α is a learning rate parameter. As before, it is not necessary to update all n weights, only the m weights associated with tiles activated by state s. Algorithm 1 shows a simple way to learn an approximation of the optimal value function using tile coding. The function ACTIVE-TILE returns the tile in the given tiling activated by the given state. If only one tiling is used, then there is a trade-off between speed and precision of learning. Smaller tiles yield more precise value estimates but take longer to learn since those estimates generalize less broadly. Multiple tilings can avoid this trade-off, since more tilings improve resolution without reducing capacity for generalization.

Algorithm 1 TILE-CODING$(S, A, T, R, \alpha, \gamma, m, n)$

for $i \leftarrow 1$ to m **do**
  Initialize tiling i with n/m tiles
  for $j \leftarrow 1$ to n/m **do**
    Initialize tile j with zero weight
repeat
  $s \leftarrow$ random state from S
  $\Delta V(s) \leftarrow \max_a [R(s, a) + \gamma V(T(s, a))] - V(s)$
  for $i \leftarrow 1$ to m **do**
    $w \leftarrow$ weight of ACTIVE-TILE(s)
    $w \leftarrow w + \frac{\alpha}{m} \Delta V(s)$
until time expires

Method

Tile coding is a simple, computationally efficient method for approximating value functions that has proven effective (Sutton 1996; Stone, Sutton, & Kuhlmann 2005). However, it has two important limitations.

The first limitation is that it requires a human designer to correctly select the width of each tile in each dimension. While in principle tiles can be of any size and shape, they are typically axis-aligned rectangles whose widths are uniform within a given dimension. Selecting these widths appropriately can mean the difference between fast, effective learning and catastrophically poor performance. If the tiles are too large, value updates will generalize across regions in S with disparate values, resulting in poor approximations. If the tiles are too small, value updates will generalize very little and learning may be infeasibly slow.

The second limitation is that the degree of generalization is fixed throughout the learning process. Use of multiple tilings makes it possible to increase resolution without compromising generalization, but the degree of generalization never changes. This limitation is important because recent research demonstrates that the best performance is possible only if generalization is gradually reduced over time (Sherstov & Stone 2005). Intuitively, broad generalization at the beginning allows the agent to rapidly learn a rough approximation; less generalization at the end allows the agent to learn a more nuanced approximation.

This section presents *adaptive tile coding*, a novel function approximation method that addresses both of these limitations. The method begins with simple representations and refines them over time, a strategy that has proven effective for other function approximators, such as neural networks (Whiteson & Stone 2006), piecewise-linear representations based on kd-trees (Munos & Moore 2002), and uniform grid discretizations (Chow & Tsitsiklis 1991). Adaptive tile coding begins with a few large tiles, and gradually adds tiles during learning by splitting existing tiles. While there are infinitely many ways to split a given tile, for the sake of computational feasibility, our method considers only splits that divide tiles in half evenly. Figure 2 depicts this process for a domain with two state features.

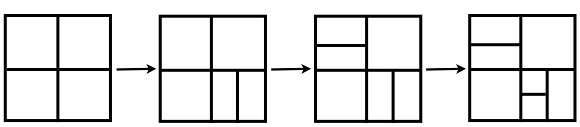

Figure 2: An example of how tiles might be split over time using adaptive tile coding.

By analyzing the current value function and policy, the agent can make smart choices about when and where to split tiles, as detailed below. In so doing, it can automatically discover an effective representation that devotes more resolution to critical regions of S, without the aid of a human designer. Furthermore, learning with a coarse representation first provides a natural and automatic way to reduce generalization over time. As a result, multiple tilings are no longer necessary: a single, adaptive tiling can provide the broad generalization needed early in learning and the high resolution needed later on. The remainder of this section addresses two critical issues: when and where to split tiles.

When to Split

Correctly deciding when to split a tile can be critical to performance. Splitting a tile too soon will slow learning since generalization will be prematurely reduced. Splitting a tile too late will also slow learning, as updates will be wasted on a representation with insufficient resolution to further improve value estimates. Intuitively, the agent should learn as much as possible with a given representation before refining it. Hence, it needs a way to determine when learning has plateaued.

One way to do so is by tracking *Bellman error* (i.e. ΔV). As long as V is improving, $|\Delta V|$ will tend to decrease over time. However, this quantity is extremely noisy, since updates to different tiles may differ greatly in magnitude and updates to different states within a single tile can move the value estimates in different directions. Hence, a good rule for deciding when to split should consider Bellman error but be robust to its short-term fluctuations.

In this paper, we use the following heuristic. For each tile, the agent tracks the lowest $|\Delta V|$ occurring in updates to that tile. It also maintains a global counter u, the number of updates occurring since the updated tile had a new lowest $|\Delta V|$ (each update either increments u or resets it to 0). When u exceeds a threshold parameter p, the agent decides that learning has plateaued and selects a tile to split. In other words, a split occurs after p consecutive updates fail to produce a new tile-specific lowest $|\Delta V|$.[1] Hence, the agent makes a *global* decision about when learning has finished, since $|\Delta V|$ may temporarily plateau in a given tile simply because the effects of updates to other tiles have not yet propagated back to it.

Where to Split

Once the agent decides that learning has plateaued, it must decide which tile to split and along which dimension.[2] This section presents two different approaches, one based on expected changes to the value function and the other on expected changes to the policy. Both require the agent to maintain *sub-tiles*, which estimate, for each potential split, what weights the resulting tiles would have. Since each state is described by k state features, each tile has $2k$ sub-tiles.

When a new tile is created, its sub-tile weights are initialized to zero. When the agent updates state s, it also updates the k sub-tiles that are activated by s, using the same rule as for regular weights, except that the update is computed by subtracting the relevant sub-tile weight (rather than the old value estimate) from the target value:

$$\Delta w_d(s) = \max_a[R(s,a) + \gamma V(T(s,a))] - w_d(s)$$

[1]There are many other ways to determine when learning has plateaued. For example, in informal experiments, we applied linear regression to a window of recent $|\Delta V|$ values. Learning was deemed plateaued when the slope of the resulting line dropped below a small threshold. However, this approach proved inferior in practice to the one described above, primarily because performance was highly sensitive to the size of the window.

[2]The agent splits only one tile at a time. It could split multiple tiles but doing so would be similar to simply reducing p.

where $w_d(s)$ is the weight of the sub-tile resulting from a split along dimension d activated by state s. Algorithm 2 describes the resulting method, with regular weight updates in lines 8–9 and sub-tile weight updates in lines 10–13. On line 19, the agent selects a split according to one of the criteria detailed in the remainder of this section.

Algorithm 2 ADAPTIVE-TILE-CODING($S, A, T, R, k, \alpha, \gamma, n, p$)

1: $u \leftarrow 0$
2: Initialize one tiling with n tiles
3: **for** $i \leftarrow 1$ to n **do**
4: Initialize ith tile and $2k$ sub-tile weights to zero
5: **repeat**
6: $s \leftarrow$ random state from S
7: $\Delta V(s) \leftarrow \max_a[R(s,a) + \gamma V(T(s,a))] - V(s)$
8: $w \leftarrow$ weight of tile activated by s
9: $w \leftarrow w + \alpha \Delta V(s)$
10: **for** $d \leftarrow 1$ to k **do**
11: $w_d \leftarrow$ weight of sub-tile w.r.t split along d activated by s
12: $\Delta w_d = \max_a[R(s,a) + \gamma V(T(s,a))] - w_d$
13: $w_d \leftarrow w_d + \alpha \Delta w_d$
14: **if** $|\Delta V| <$ lowest Bellman error on tile activated by s **then**
15: $u \leftarrow 0$
16: **else**
17: $u \leftarrow u + 1$
18: **if** $u > p$ **then**
19: Perform split that maximizes value or policy criterion
20: $u \leftarrow 0$
21: **until** time expires

Value Criterion Sub-tile weights estimate what values the tiles resulting from a potential split would have. Thus, the difference in sub-tile weights indicates how drastically V will change as a result of a given split. Consequently, the agent can maximally improve V by performing the split that maximizes, over all tiles, the value of $|w_{d,u} - w_{d,l}|$, where $w_{d,u}$ and $w_{d,l}$ are, respectively, the weights of the upper and lower sub-tiles of a potential split d. Using this *value criterion* for selecting splits will cause the agent to devote more resolution to regions of S where V changes rapidly (where generalization will fail) and less resolution to regions where it is relatively constant (where generalization is helpful).

Policy Criterion The value criterion will split tiles so as to minimize error in V. However, doing so will not necessarily yield maximal improvement in π. For example, there may be regions of S where V^* changes significantly but π^* is constant. Hence, the most desirable splits are those that enable the agent to improve π, regardless of the effect on V. To this end, the agent can estimate, for each potential split, how much π would change if that split occurred.

When updating a state s, the agent iterates over the $|A|$ possible successor states to compute a new target value. For each dimension d along which each successor state s' could be split, the agent estimates whether $\pi(s)$ would change if the tile activated by s' were split along d, by computing the expected change in $V(s')$ that split would cause:

$$\Delta V_d(s') = w_d(s') - V(s')$$

If changing $V(s')$ by $\Delta V_d(s')$ would alter $\pi(s)$, then the agent increments a counter c_d, which tracks *changeable actions* for potential split d in the tile activated by s' (see Figure 3). Hence, the agent can maximize improvement to π by performing the split that maximizes the value of c_d over all tiles. Using this *policy criterion*, the agent will focus splits on regions where more resolution will yield a refined policy.

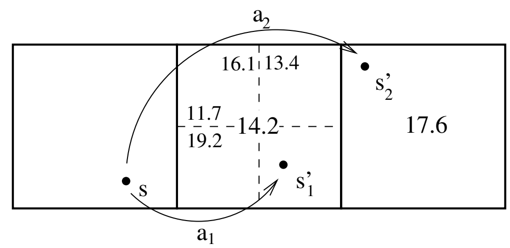

Figure 3: An agent updates state s, from which each action a_i leads to successor state s'_i. The figure shows the tiles, including weights, that these successor states fall in and shows sub-tile weights for the middle tile. Though $\pi(s) = 2$, a horizontal split to the middle tile would make $\pi(s) = 1$ (since $19.2 > 17.6$), incrementing c_d for that split.

Testbed Domains

This section describes mountain car and puddle world, two benchmark reinforcement learning domains that we use as testbeds for evaluating the performance of adaptive tile coding. Both domains have continuous state features and hence require value function approximation.

Mountain Car

In the mountain car task (Boyan & Moore 1995), depicted in Figure 4, an agent must drive a car to the top of a steep mountain. The car cannot simply accelerate forward because its engine is not powerful enough to overcome gravity. Instead, the agent must learn to drive backwards up the hill behind it, thus building up sufficient inertia to ascend to the goal before running out of speed.

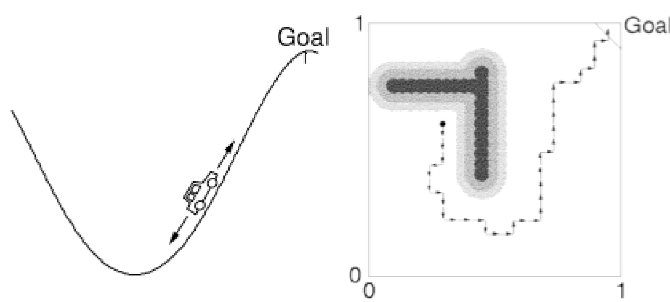

Figure 4: The mountain car and puddle world domains. These figures were taken from Sutton and Barto (1998) and Sutton (1996).

The agent's state consists of its current position $-1.2 \leq x \leq 0.5$ and velocity $-0.07 \leq v \leq 0.07$. The agent begins each episode in a state chosen randomly from these ranges. It receives a reward of -1 at each time step until reaching the goal. The agent's three available actions correspond to positive, negative, and zero throttle.

Puddle World

In the puddle world task (Sutton 1996), also depicted in Figure 4, a robot placed randomly in a two-dimensional terrain must navigate to a goal area in the upper right corner while avoiding the two puddles.

The agent's state consists of its x and y positions. It receives a reward of -1 at each time step until reaching the goal, plus additional penalties for being inside a puddle, equal to 400 times the distance inside the puddle. The agent's four available actions allow it to move up, down, left and right by 0.05, plus a small amount of Gaussian noise.[3]

Results and Discussion

To evaluate adaptive tile coding, we tested its performance in the mountain car and puddle world domains. The value and policy criteria were tested separately, with 25 independent trials for each method in each domain. In each trial, the method was evaluated during learning by using its current policy to control the agent in test episodes. The agent took one action for each update that occurred (i.e. one iteration of the repeat loop in Algorithms 1 and 2). Note that since the agent learns from a model, these test episodes do not affect learning; their sole purpose is to evaluate performance. The following parameter settings were used in all trials: $\alpha = 0.1$, $\gamma = 0.999$, $n = 4$ (2x2 initial tilings), and $p = 50$.

Next, we tested 18 different fixed tile-coding representations, selected by choosing three plausible values for the number of tilings $m \in \{1, 5, 10\}$ and six plausible values for the number of tiles n such that the tiles per feature $\sqrt[k]{n/m} \in \{5, 10, 25, 50, 100, 250\}$, where $k = 2$ is the number of state features in each domain. We tested each combination of these two parameters with $\alpha = 0.1$ and $\gamma = 0.999$ as before. We conducted 5 trials at each of the 18 parameter settings and found that only six in mountain car and seven in puddle world were able to learn good policies (i.e. average reward per episode > -100) in the time allotted.

Finally, we selected the three best performing fixed settings and conducted an additional 25 trials. Figure 5 shows the results of these experiments by plotting, for each domain, the uniform moving average reward accrued over the last 500 episodes for each adaptive approach and the best fixed approaches, averaged over all 25 trials for each method.

The variation in performance among the best fixed representations demonstrates that the choice of representation is a crucial factor in both the speed and quality of learning. Without *a priori* knowledge about what representations are effective in each task, both versions of the adaptive method consistently learn good policies, while only a minority of the fixed representations do so. Furthermore, when the policy criterion was used, the adaptive method learned approximately optimal policies in both domains, at speeds that are competitive with the best fixed representations.

While there are fixed representations that learn good policies as fast or faster than the adaptive approach (10x10 with 10 tilings in mountain car and 10x10 with 1 tiling in puddle world), those representations do not go on to learn approximately optimal policies as the adaptive approach does. Similarly, there are fixed representations that learn approximately optimal policies faster than the adaptive approach (50x50 with 10 tilings in mountain car and 25x25 with 1 tiling in puddle world), but those representations take significantly longer to learn good policies.

Furthermore, the fixed representations that learn good policies fastest are not the same as those that learn approximately optimal policies and are different in the two domains. By contrast, the adaptive method, with a single parameter setting, rapidly learns approximately optimal policies in both domains. Overall, these results confirm the efficacy of the adaptive method and suggest it is a promising approach for improving function approximation when good representations are not known *a priori*.

To better understand why the adaptive method works, we took the best representations learned with the policy criterion, reset all the weights to zero, and restarted learning with splitting turned off. The restarted agents learned much more slowly than the adaptive agents that began with coarse representations and bootstrapped their way to good solutions. This result suggests that the adaptive approach learns well, not just because it finds good representations, but also because it gradually reduces generalization, confirming the conclusions of Sherstov and Stone (2005).

The results also demonstrate that the policy criterion ultimately learns better policies than the value criterion. To understand why, we examined the structure of the final representations learned with each approach, as depicted in Figure 6. Lack of space prevents us from also depicting V and π. However, manual inspection of V confirms that in both domains the value criterion devotes more resolution to regions where V changes most rapidly. In mountain car, this region spirals outward from the center, as the agent oscillates back and forth to build momentum. In puddle world, this region covers the puddles, where reward penalties give V a sharp slope, and the area adjacent to the goal. However, those regions do not require fine resolution to represent approximately optimal policies. On the contrary, manual inspection reveals that π is relatively uniform in those regions.

By contrast, the policy criterion devotes more resolution to regions where the policy is not uniform. In mountain car, the smallest tiles occur in the center and near each corner, where π is less consistent. In puddle world, it devotes the least resolution to the puddle, where the policy is mostly uniform, and more resolution to the right side, where the "up" and "right" actions are intermingled. Hence, by striving to refine the agent's policy instead of just its value function, the policy criterion makes smarter choices about which tiles to split and consequently learns better policies.

[3]The presence of this noise means puddle world does not strictly satisfy our assumption that the environment is a deterministic MDP. However, the algorithms presented in this paper excel at this task anyway, as the results in the next section demonstrate.

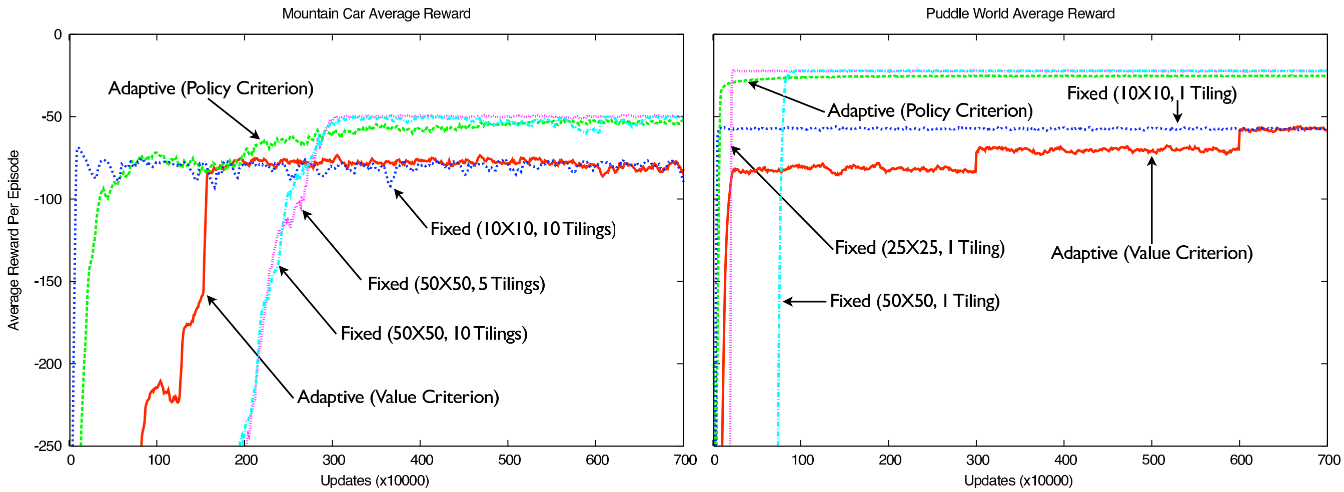

Figure 5: Average reward per episode in both mountain car and puddle world of the adaptive approach with value or policy criterion, compared to the best-performing fixed representations.

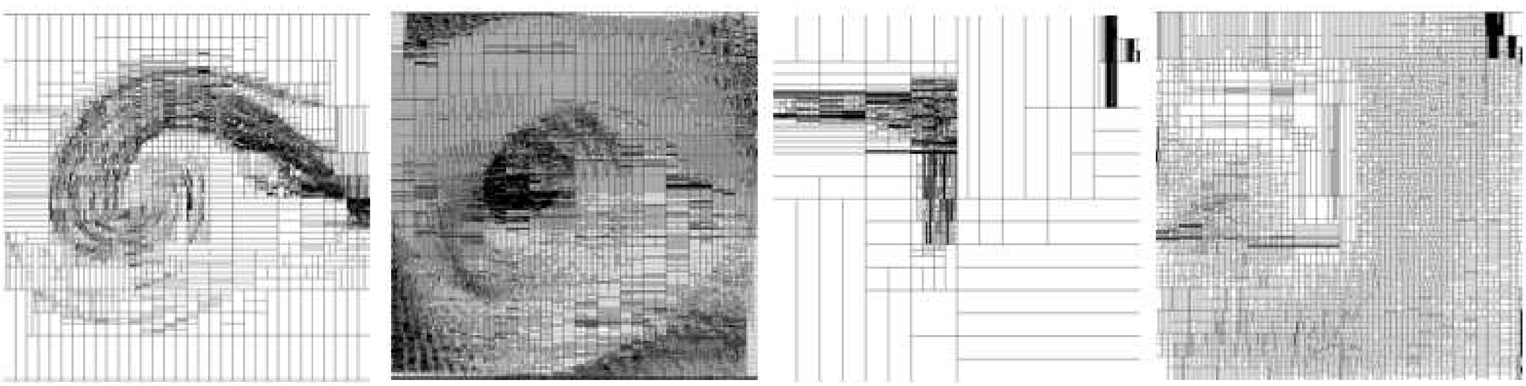

Figure 6: Examples of final tile coding representations learned by the adaptive methods. From left to right: value and policy criterion respectively in mountain car, and value and policy criterion respectively in puddle world.

Related and Future Work

A substantial body of previous work aims to automate the design of RL function approximators. Perhaps most related is the work of Munos and Moore (2002) on variable resolution function approximators. The primary difference is the use of piecewise-linear representations instead of tile coding. As a result, computing $V(s)$ once the right tile is located takes order of $k \ln k$ time instead of constant time. They propose a splitting rule that is similar to the value criterion used in this paper. They also propose examining the policy to determine where to split though their approach, unlike the policy criterion presented here, does not reason about subtile weights and works well only in conjunction with a criterion based on the value function. In addition, their method, by running dynamic programming to convergence between each split, may be computationally inefficient. Their empirical evaluations measure final performance at each resolution but do not consider, as we do, the speed of learning as measured in number of updates.

Sherstov and Stone (2005) demonstrate that reducing generalization during learning is key to good tile coding performance; they also present a method for automatically adjusting generalization, though their approach looks only at error in the value function without directly considering the policy. Chow and Tsitsiklis (1991) show how to compute the tile width of a uniform tiling necessary to learn an approximately optimal policy, though they make strong assumptions (e.g. that the transition probabilities are Lipschitz continuous). Lanzi et al. (2006) extend learning classifiers systems to use tile coding: evolutionary methods optimize a population of tile-coding function approximators, each of which covers a different region of the state space. The Parti-game algorithm (Moore & Atkeson 1995) automatically partitions state spaces but applies only to tasks with known goal regions and assumes the existence of a greedy local controller. Utile Suffix Memory (McCallum 1995) automatically learns a tree-based state representation, though the goal is coping with non-Markovian state, rather than efficient value function approximation. Other work on adaptive representations for function approximation includes evolving neural networks (Whiteson & Stone 2006), using cascade-correlation networks (Rivest & Precup 2003), or analyzing state space topologies to find basis functions for linear function approximators (Mahadevan 2005).

Natural extensions to this work include testing it in higher-dimensional spaces, developing extensions that work with model-free RL, and using ideas from *prioritized sweeping* (Moore & Atkeson 1993) to speed learning by focusing updates on tiles affected by recent splits.

References

Albus, J. S. 1981. *Brains, Behavior, and Robotics*. Peterborough, NH: Byte Books.

Baird, L., and Moore, A. 1999. Gradient descent for general reinforcement learning. In *Advances in Neural Information Processing Systems 11*. MIT Press.

Baird, L. 1995. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, 30–37. Morgan Kaufmann.

Bellman, R. E. 1957. *Dynamic Programming*. Princeton, NJ.: Princeton University Press.

Boyan, J. A., and Moore, A. W. 1995. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7*.

Chow, C.-S., and Tsitsiklis, J. N. 1991. An optimal one-way multigrid algorithm for discrete-time stochastic control. *IEEE Transactions on Automatic Control* 36(8):898–914.

Gordon, G. J. 1995. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, 261–268.

Lagoudakis, M. G., and Parr, R. 2003. Least-squares policy iteration. *Journal of Machine Learning Research* 4(2003):1107–1149.

Lanzi, P. L.; Loiacono, D.; Wilson, S. W.; and Goldberg, D. E. 2006. Classifier prediction based on tile coding. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, 1497–1504.

Mahadevan, S. 2005. Samuel meets Amarel: Automating value function approximation using global state space analysis. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*.

McCallum, A. R. 1995. Instance-based utile distinctions for reinforcement learning. In *Proceedings of the Twelfth International Machine Learning Conference*, 387–395.

Moore, A., and Atkeson, C. 1993. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning* 13:103–130.

Moore, A. W., and Atkeson, C. G. 1995. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning* 21(3):199–233.

Munos, R., and Moore, A. 2002. Variable resolution discretization in optimal control. *Machine Learning* 49:291–323.

Rivest, F., and Precup, D. 2003. Combining TD-learning with cascade-correlation networks. In *Proceedings of the Twentieth International Conference on Machine Learning*, 632–639. AAAI Press.

Sherstov, A. A., and Stone, P. 2005. Function approximation via tile coding: Automating parameter choice. In *Proceedings of the Symposium on Abstraction, Reformulation, and Approximation*, 194–205.

Stone, P.; Sutton, R.; and Kuhlmann, G. 2005. Reinforcement learning in robocup-soccer keepaway. *Adaptive Behavior* 13(3):165–188.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. Cambridge, Massachussets: MIT Press.

Sutton, R. 1996. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, 1038–1044.

Whiteson, S., and Stone, P. 2006. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research* 7(May):877–917.