In *Proceedings of the Twentieth National Conference on Artificial Intelligence* (AAAI-05), pp. 880-885, Pittsburgh, PA, July 2005.

Value Functions for RL-Based Behavior Transfer: A Comparative Study

Matthew E. Taylor, Peter Stone, and Yaxin Liu
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188
{mtaylor, pstone, yxliu}@cs.utexas.edu

Abstract

Temporal difference (TD) learning methods (Sutton & Barto 1998) have become popular reinforcement learning techniques in recent years. TD methods, relying on function approximators to generalize learning to novel situations, have had some experimental successes and have been shown to exhibit some desirable properties in theory, but have often been found slow in practice. This paper presents methods for further generalizing *across tasks*, thereby speeding up learning, via a novel form of *behavior transfer*. We compare learning on a complex task with three function approximators, a CMAC, a neural network, and an RBF, and demonstrate that behavior transfer works well with all three. Using behavior transfer, agents are able to learn one task and then markedly reduce the time it takes to learn a more complex task. Our algorithms are fully implemented and tested in the RoboCup-soccer keepaway domain.

Introduction

Temporal difference learning methods (Sutton & Barto 1998) have shown some success in different reinforcement learning tasks because of their ability to learn where there is limited prior knowledge and minimal environmental feedback. However, in practice, current TD methods are somewhat slow to produce near-optimal behaviors. Many techniques exist which attempt to speed up the learning process.

For example, Selfridge et al. (1985) used *directed training* to show that a learner can train faster on a task if it has first learned on a simpler variation of the task. In this paradigm the state transition function, which is part of the environment, can change between tasks. *Learning from easy missions* (Asada *et al.* 1994) is a technique that relies on human input to modify the starting state of the learner over time, making it incrementally more difficult for the learner. Both of these methods reduce the total training time required to successfully learn the final task. However, neither allow for changes to the state or action spaces between the tasks, limiting their applicability.

Reward shaping (Colombetti & Dorigo 1993; Mataric 1994) allows one to bias a learner's progress through the state space by adding in artificial rewards to the environmental rewards. Doing so requires sufficient knowledge about the environment a priori to guide the learner and must be done carefully to avoid unintended behaviors. While it is well understood how to add this type of guidance to a learner (Ng, Harada, & Russell 1999), we would prefer to allow the agent to learn faster by training on different (perhaps pre-existing) tasks rather than creating easier, artificial tasks.

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Value-based TD methods learn to estimate a *value function* for each possible state. The learner is then able to select the action which it believes will return the highest value in the long run. Over time the learned value function approaches the true value of each state by comparing the expected value of a state with the actual value received from that state. Value-based TD methods typically utilize a *function approximator* so that the value function can be approximated for novel situations. This approximation becomes critical as the number of situations the agents could be in grows, or becomes infinite.

In this paper we study the effect of *behavior transfer* (Taylor & Stone 2005) on the learning rates of value-based TD learners. Behavior transfer allows a TD learner trained on one task to learn significantly faster when training on another task with related, but different, state and action spaces. This method is more general than the previously referenced methods because it does not preclude the modification of the transition function, start state, or reward function. We will compare the efficacy of using behavior transfer to speed up learning in agents that utilize three different function approximators, a CMAC, a neural network, and an RBF, on a single reinforcement learning problem.

The key technical challenge of behavior transfer is mapping a value function in one representation to a meaningful value function in another, typically larger, representation. The mapping is necessarily dependent on both the task and the function approximator's representation of the value function, but we posit that the behavior transfer technique will be applicable to multiple tasks and function approximators. This paper establishes that behavior transfer is general enough to work effectively with multiple function approximators, in particular a neural network in addition to a CMAC. This paper also directly compares the efficacy of a CMAC and a neural network on the same complex learning task.

Behavior Transfer Methodology

To formally define behavior transfer we first review the reinforcement learning framework that conforms to the generally accepted notation for Markov decision processes (Puterman 1994). There is a set of possible perceptions of the current state of the world, S, and a learner has an initial starting state, $s_{initial}$. When in a particular state s, there is a set of actions, A, which can be taken. The reward function R maps each perceived state of the environment to a single number which is the instantaneous reward for the state. The transition function, T, takes a state and an action and returns the state of the environment after the action is performed. If transitions are non-deterministic the transition function is a probability distribution function. A learner is able to sense s, and typically knows A, but may or may not initially know S, R, or T.

A policy $\pi : S \mapsto A$ defines how a learner interacts with the environment by mapping perceived environmental states to actions. π is modified by the learner over time to improve performance, i.e. the expected total reward, and it completely defines the behavior of the learner in an environment. In the general case the policy can be stochastic. The success of an agent is determined by how well it maximizes the total reward it receives in the long run while acting under some policy π. An $optimal\ policy$, π^*, is a policy which does maximize this value (in expectation). Any reasonable learning algorithm attempts to modify π over time so that the agent's performance approaches that of π^*.

In this paper we consider the general case where $S_1 \neq S_2$, and/or $A_1 \neq A_2$ for two tasks. To use the learned policy from the first task, $\pi_{(1,final)}$, as the initial policy for a TD learner in a second task, we must transform the value function so that it can be directly applied to the new state and action space. A behavior transfer functional $\rho(\pi)$ will allow us to apply a policy in a new task. The policy transform functional ρ needs to modify the policy and its associated value function so that it accepts S_2 as inputs and allows for A_2 to be outputs. A policy generally selects the action which is believed to accumulate the largest expected total reward; the problem of transforming a policy between two tasks reduces to transforming the value function. In this paper we will therefore concentrate on transferring the state action values, Q, from one learner to another. Defining ρ correctly is the key technical challenge to enable general behavior transfer.

One measure of success in speeding up learning using this method is that given a policy $\pi_{(1,final)}$, the training time for π_2 to reach some performance threshold decreases when replacing the initial policy in task 2, $\pi_{(2,initial)}$, with $\rho(\pi_{(1,final)})$. This criterion is relevant when task 1 is given and is of interest in its own right or if $\pi_{(1,final)}$ can be used repeatedly to speed up multiple related tasks. A stronger measure of success is that the training time for both tasks using behavior transfer is shorter than the training time to learn the second task from scratch. This criterion is relevant when task 1 is created for the sole purpose of speeding up learning via behavior transfer and $\pi_{(1,final)}$ is not reused.

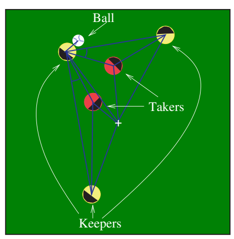

Figure 1: This diagram depicts the 13 state variables used for learning with 3 keepers and 2 takers. There are 11 distances to players and the center of the field, as well as 2 angles along passing lanes.

Testbed Domain

To test the efficacy of behavior transfer with different function approximators we consider the RoboCup simulated soccer keepaway domain using a setup similar to past research (Stone, Sutton, & Kuhlmann 2005). RoboCup simulated soccer is well understood as it has been the basis of multiple international competitions and research challenges. The multiagent domain incorporates noisy sensors and actuators, as well as enforcing a hidden state so that agents can only have a partial world view at any given time. While previous work has attempted to use machine learning to learn the full simulated soccer problem (Andre & Teller 1999; Riedmiller *et al.* 2001), the complexity and size of the problem have proven prohibitive. However, many of the RoboCup subproblems have been isolated and solved using machine learning techniques, including the task of playing keepaway.

Keepaway, a subproblem of RoboCup soccer, is the challenge where one team, the *keepers*, attempts to maintain possession of the ball on a field while another team, the *takers*, attempts to gain possession of the ball or force the ball out of bounds, ending an *episode*. Keepers that make better decisions about their actions are able to maintain possession of the ball longer and thus have a longer average episode length. Figure 1 depicts three keepers playing against two takers.

As more players are added to the task, keepaway becomes harder for the keepers because the field becomes more crowded. As more takers are added there are more players to block passing lanes and chase down any errant passes. As more keepers are added, the keeper with the ball has more passing options but the average pass distance is shorter. This reduced distance forces more passes and often leads to more errors because of the noisy actuators and sensors. For this reason keepers in 4 vs. 3 keepaway (i.e. 4 keepers and 3 takers) take longer to learn an optimal control policy than in 3 vs. 2. The hold time of the best policy for a constant field size also decreases when adding an equal number of keepers and takers. The time it takes to learn a policy which is near a handcoded solution roughly doubles as each additional keeper and taker is added (Stone, Sutton, & Kuhlmann 2005).

Learning Keepaway

The keepers use episodic SMDP Sarsa(λ) (Sutton & Barto 1998), a well understood temporal difference algorithm, to learn their task. In one implementation, we use linear tile-coding function approximation, also known as CMACs, which has been successfully used in many reinforcement learning systems (Albus 1981). A second implementation of our agents use neural networks, another method for function approximation that has had some notable past successes (Crites & Barto 1996; Tesauro 1994). The third implementation uses a radial basis function (RBF) (Sutton & Barto 1998). The keepers choose not from primitive actions (turn, dash, or kick) but higher-level actions implemented by the CMUnited-99 team (Stone, Riley, & Veloso 2000). A keeper without the ball automatically attempts to move to an open area (the receive action). A keeper in possession of the ball has the freedom to decide whether to hold the ball or to pass to a teammate.

Our CMAC and RBF agents are based on the keepaway benchmark players distributed by UT-Austin[1] which are described in (Stone *et al.* 2005). These benchmark players are built on the UvA Trilearn team (de Boer & Kok 2002) and the CMUnited-99 team (Stone, Riley, & Veloso 2000), whereas previous publications (Stone, Sutton, & Kuhlmann 2005) and our neural network players are built on the CMUnited-99 players alone. The newer benchmark players have better low-level functionality and are thus able to hold the ball for longer than the CMUnited-99 players, both before and after learning, but the learning and behavior transfer results are very similar to the older players.

CMACs allow us to take arbitrary groups of continuous state variables and lay infinite, axis-parallel tilings over them (see Figure 2). Using this method we are able to discretize the continuous state space by using tilings while maintaining the capability to generalize via multiple overlapping tilings. The number of tiles and width of the tilings are hardcoded and this dictates which state values will activate which tiles. The function approximation is learned by changing how much each tile

[1] Flash file demonstrations, source code, documentation, and mailing list can be found at *http://www.cs.utexas.edu/users/AustinVilla/sim/keepaway/*.

contributes to the output of the function approximator. By default, all the CMAC's weights are initialized to zero. This approach to function approximation in the RoboCup soccer domain is detailed by Stone and Sutton (2002).

An RBF is a generalization of the tile coding idea to a continuous function (Sutton & Barto 1998). In the one-dimensional case, an RBF approximator is a linear function approximator $\hat{f}(x) = \sum_i w_i f_i(x)$, where the basis functions have the form $f_i(x) = \phi(|x - c_i|)$, x is the current state, and c_i is the center of feature i. A CMAC is a degenerate case of RBF approximator with c_i's equally spaced and $\phi(x)$ a step function. Here we use Gaussian radial basis functions, where $\phi(x) = \exp(-\frac{x^2}{2\sigma^2})$, and the same c_i's as a CMAC. The learning for RBF networks is identical to that for CMACs except for the calculation of state-action values where the RBFs are used. As is the case for CMACs, the state-action values are computed as a sum of one-dimensional RBFs, one for each feature. By tuning σ, the experimenter can control the width of the Gaussian function and therefore the amount of generalization over the state space. In our implementation, a value of $\sigma = 0.25$ creates a Gaussian which roughly spans 3 CMAC tiles. We tried 3 different values for this parameter but more tuning may have reduced our learning times.

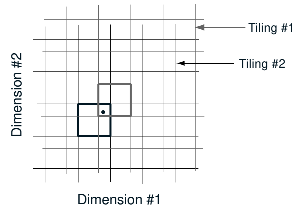

Figure 2: Tile-coding feature sets are formed from multiple overlapping tilings and state variables are used to determine the activated tile in each of the different tilings. Every activated tile contributes a weighted value to the total output of the CMAC for the given state. Increasing the number of tilings allows better generalization while decreasing the tile size allows more accurate representations of smaller details. Note that we primarily use one-dimensional tilings but that the principles apply in the n-dimensional case.

The neural network function approximator likewise allows a learner to select an action given a set of state variables. Each input to the neural network is set to the value of a state variable and each output corresponds to an action. The legal action with the highest activation is selected. We use a feed-forward network with a single hidden layer built with the Aspirin/MIGRANES 6.0 framework. Nodes in the hidden layer have a sigmoid transfer function and output nodes are linear. The network is then trained using standard backpropogation, modifying the weights connecting the nodes. To our knowledge, this work presents the first application of a function approximator other than a CMAC in the keepaway domain.

For the purposes of this paper, it is particularly important to note the state variables and action possibilities used by the learners. The keepers' states comprise distances and angles of the keepers $K_1 - K_n$, the takers $T_1 - T_m$, and the center of the playing region C (see Figure 1). Keepers and takers are ordered by increasing distance from the ball. Note that as the number of keepers *n* and the number of takers *m* increase, the number of state variables also increase so that the more complex state can be fully described. S must change (e.g. there are more distances to players to account for) and $|A|$ increases as there are more teammates for the keeper with possession of the ball to pass to. Full details of the keepaway domain and a player implementation similar to ours are documented elsewhere (Stone, Sutton, & Kuhlmann 2005).

Learning 3 vs. 2

On a 25m x 25m field, three keepers are initially placed in three corners of the field and a ball is placed near one of the keepers. The two takers are placed in the fourth corner. When the episode starts, the three keepers attempt to keep control of the ball by passing amongst themselves and moving to open positions. The keeper with the ball has the option to either pass the ball to one of its two teammates or to hold the ball. In this task A = {hold, passToTeammate1, passToTeammate2}. S is defined by 13 state variables, as shown in Figure 1. When a taker gains control of the ball or the ball is kicked out of the field's bounds the episode is finished. The reward to the Sarsa(λ) algorithm for the keeper is the number of time steps the ball remains in play after an action is taken. The episode is then reset with a random keeper placed near the ball.

All weights in the CMAC function approximator are initially set to zero. Similarly, all weights and biases in the neural network are set to small random numbers. We use a 13-20-4 network where the choice of 20 hidden units was chosen via experimentation. [2] As training progresses, the weights of the function approximator are changed by Sarsa(λ) so that the average hold time of the keepers increases. Throughout this process, the takers use a static hand-coded policy to attempt to capture the ball as quickly as possible. Policy evaluation is very noisy do to high environmental randomness.

Learning 4 vs. 3

Holding the field size constant we now add an additional keeper and an additional taker. R and T are essentially unchanged from 3 vs. 2 keepaway, but now A = {hold, passToTeammate1, passToTeammate2, passToTeammate3} and S is made up of 19 state variables due to the added players. The 4 vs. 3 task is harder and the learned average hold times after 20 hours of training with a CMAC function approximator learning from scratch decrease by roughly 32% from 3 vs. 2 to 4 vs. 3. The neural network used is a 19-30-5 network, where the use of 30 hidden units was chosen under the assumption that the 4 vs. 3 task is more complex than the 3 vs. 2 task, which had fewer inputs, hidden units, and outputs.[3]

In order to quantify how fast an agent in 4 vs. 3 learns, we set a target performance of 9.0 seconds for CMAC and neural network learners, while RBF learners have a target of 10.0 seconds. Thus, when a group of four CMAC keepers has learned to hold the ball for an average of 9.0 seconds over 1,000 episodes we say that the keepers have learned the 4 vs. 3 task. This threshold is chosen so that all trials need to learn for some nonzero amount of time and the majority of trials are able to reach the threshold before learning plateaus; because the RBF learners learn more quickly, they required a higher target performance. Note that our behavior transfer results hold for other (higher) threshold times as well. By averaging over many trials we can measure the effectiveness of learning in different situations.

Behavior Transfer in Keepaway

Learning on one task and transferring the behavior to a separate useful task can reduce the training time. In the keepaway domain, A and S are determined by the current keepaway task and thus differ from instance to instance. $s_{initial}$, R, and T, though formally different, are effectively constant across tasks. When S and A change, $s_{initial}$, R, and T change by definition. But in practice, R is always defined as 1 for every time step that the keepers maintain possession, and $s_{initial}$ and T are always defined by the RoboCup soccer simulation.

[2] Five different network sizes were tested, from 15 to 25 hidden nodes and the differences in performance were very small.

[3] Again, other networks with different numbers of hidden units were tried, but the differences in learning times were not significant.

In the keepaway domain we are able to intuit the mappings between states and actions in the two tasks, ρ, based on our knowledge of the domain. Our choice for the mappings is supported by empirical evidence showing that using this ρ with behavior transfer decreases training time. Other domains will not necessarily have such straightforward transforms between tasks of different complexity. Finding a general method to specify ρ is outside the scope of this paper and will be formulated in future work. One of the main future challenges will be identifying general heuristics for mapping states and actions between two related tasks. A primary contribution of this paper is demonstrating that there exist domains and function approximators for which ρ can be constructed and then used to successfully decrease learning times.

The naive approach of directly using the value function from $\pi_{(3vs2,final)}$ fails because S and A have changed. Keeping in mind that $\pi : S \mapsto A$, we see that the new state vectors which describe the learner's environment would not necessarily be correctly used, nor would the new actions be correctly evaluated by $\pi_{(3vs2,final)}$. To use the learned policy we modify it to handle the new actions and new state values in the second task so that the player can reasonably evaluate them.

The CMAC function approximator takes a state and an action and returns the expected total reward. The learner can evaluate each potential action for the current state and then use π to choose one. We construct a ρ_{cmac} and utilize it so that when we input a 4 vs. 3 action the weights for the activated tiles are not zero but instead are initialized by $\pi_{(3vs2,final)}$. To accomplish this, we copy weights from the tiles which would be activated for a similar action in 3 vs. 2 into the tiles activated for every new action in 4 vs. 3. The weights corresponding to the tiles that are activated for the "pass to teammate 2" action are copied into the weights for the tiles that are activated to evaluate the "pass to teammate 3" action. $\pi_{(4vs3,initial)}$ will initially be unable to distinguish between these two actions.

To handle new state variables we follow a similar strategy. The 13 state variables which are present in 3 vs. 2 are already handled by the CMAC's weights. The weights for tiles activated by the six new 4 vs. 3 state variables are initialized to values of weights activated by similar 3 vs. 2 state variables. For instance, weights which correspond to "distance to teammate 2" values in the state representation are copied into the weights for tiles corresponding to "distance to teammate 3" state values. This is done for all six new state variables. As a final step, any weights which have not been initialized are set to the average value of all initialized weights. This extra step provides an larger benifit when fewer 3 vs. 2 episodes are used and is studdied elsewhere (Taylor & Stone 2005). The 3 vs. 2 training was not exhaustive and therefore some weights which may be utilized in 4 vs. 3 would otherwise remain uninitialized. Tiles which correspond to every value in the new 4 vs. 3 state vector have thus been initialized to values determined via training in 3 vs. 2. See Table 1 for examples. Identifying similar actions and states between two tasks is essential for constructing ρ and may prove to be the main limitation when attempting to apply behavior transfer to different domains.

ρ_{RBF} is analogous to ρ_{CMAC}. The main difference between the RBF and CMAC function approximators are how weights are summed together to produces values, but the weights have similar structure in both function approximators. For a given state variable, a CMAC summs one weight per tiling. An RBF differs as it sums multiple weights for each tiling where weights are multiplied by the Gaussian function $\phi(x - c_i)$. We thus copy weights from actions and states in 3 vs. 2 into similar actions and states in 4 vs. 3, following the same schema as in ρ_{CMAC}.

Constructing ρ_{nnet} is similarly intuitive. The 13-20-4 network is augmented by adding 6 inputs, 10 hidden hidden nodes, and 1 output node. The weights connecting inputs 1-13 to hidden nodes 1-20 are copied over from the 13-20-4 network. Likewise, the weights from hidden nodes 1-20 to outputs 1-4 are copied over. The new weights between the input and hidden layers, i.e. those not present in the 13-20-4 network, are set to the average learned weight from the input to hidden layer. The new weights between the hidden and output layers are set to the average learned weights from the hidden to output layer. Every weight in the 19-30-5 network is set to an initial value based on the trained 13-20-4 network. Because ρ_{nnet} copies the average into the weights from the input to hidden layer, it is in some sense simpler than ρ_{cmac}, which initializes the weights for the new state variables to similar old state variables. We explore this simpler ρ_{cmac} to suggest that there are multiple ways to formulate ρ. Future work will attempt to determine a priori what type of ρ will work best for a given function approximator and pair of tasks.

Partial Description of ρ_{cmac}

4 vs. 3 state variable	3 vs. 2 state variable
$dist(K_1, C)$	$dist(K_1, C)$
$dist(K_2, C)$	$dist(K_2, C)$
$dist(K_3, C)$	$dist(K_3, C)$
$dist(\mathbf{K_4}, C)$	$dist(K_3, C)$
Min($dist(K_2, T_1)$, $dist(K_2, T_2)$, $dist(K_2, \mathbf{T_3})$)	Min($dist(K_2, T_1)$, $dist(K_2, T_2)$)
Min($dist(K_3, T_1)$, $dist(K_3, T_2)$, $dist(K_3, \mathbf{T_3})$)	Min($dist(K_3, T_1)$, $dist(K_3, T_2)$)
Min($dist(\mathbf{K_4}, T_1)$, $dist(\mathbf{K_4}, T_2)$, $dist(\mathbf{K_4}, \mathbf{T_3})$)	Min($dist(K_3, T_1)$, $dist(K_3, T_2)$)

Table 1: This table describes part of the ρ_{cmac} transform in keepaway. We denote the distance between a and b as $dist(a, b)$. Relevant points are the center of the field C, keepers K_1-K_4, and takers T_1-T_3. Keepers and takers are ordered in increasing distance from the ball and state values not present in 3 vs. 2 are in bold.

Having constructed ρs which handle the new states and actions for function approximators, we can now set $\pi_{(4vs3,initial)} = \rho(\pi_{(3vs2,final)})$ for all three sets of agents. We do not claim that these initial value functions are correct (and empirically they are not), but instead that they allow the learners to more quickly discover a near-optimal policy.

Results and Discussion

In Tables 2 and 3 we see that a CMAC, an RBF, and a neural network successfully allow independent players to learn to hold the ball from opponents when learning from scratch. However, the training times for agents that use neural networks is significantly longer. [4] Previous research has shown that a CMAC function approximator was able to successfully learn in this domain (Stone, Sutton, & Kuhlmann 2005), but to our knowledge no other function approximators had been tested in keepaway. This work confirms that other function approximators can be successfully used and that a CMAC is more efficient than a neural network, another obvious choice. We posit that this difference is due to the CMAC's property of *locality*.

CMAC Learning Results

# of 3 vs. 2 episodes	Ave. 4 vs. 3 time (hours)	Ave. total time (hours)
0	15.26	15.26
1	12.29	12.29
10	10.06	10.08
50	4.83	4.93
100	4.02	**4.22**
250	3.77	4.3
500	3.99	5.05
1000	3.72	5.85
3000	2.42	11.04
9000	1.38	50.49
18000	**1.24**	98.01

Table 2: Results showing that learning keepaway with a CMAC and applying behavior transfer can reduce training time. Minimum learning times are bold.

[4] Note that these neural network results use an older version of the agents than used by the CMAC or RBF. However, the newer version of the neural network players also learn much slower than the CMAC and RBF players.

Neural Network and RBF Learning Results

# of 3 vs. 2 episodes	Ave. NNET 4 vs. 3 time	Ave. NNET total time	Ave. RBF 4 vs. 3 time	Ave. RBF total time
0	397.16	397.16	12.02	12.02
10	283.55	283.57	7.78	7.80
50	240.07	240.17	7.09	**7.23**
100	**221.89**	**222.08**	7.53	7.79
250	296.73	297.22	7.26	7.97
500	392.82	393.88	7.14	8.62
1,000	357.32	359.45	**6.90**	10.13

Table 3: Results from learning keepaway with different amounts of 3 vs. 2 training time (in hours) indicates behavior transfer can reduce training time for neural network (9.0 second threshold) and RBF players (10.0 second threshold).

When a particular CMAC weight for one state variable is updated during training, the update will affect the output value of the CMAC for other nearby state variable values. The width of the CMAC tiles determines the generalization effect and outside of this tile width, the change has no effect. Contrast this with the non-locality of a neural network. Every weight is used for the calculation of a value function, regardless of how close two inputs are in state space. Any update to a weight in the neural network must change the final output of the network for every set of inputs. Therefore it may take the neural network longer to settle into an optimal configuration. The RBF function approximator had the best performance of the three. The RBF shares the CMAC's locality benefits, but is also able to generalize more smoothly due to the Gaussian summation of weights. When comparing the times of the RBF function approximator to that of the CMAC, it is important to note that the CMAC was only learning to hold the ball for an average of 9.0 second in 4 vs. 3. For example, when the 4 vs. 3 threshold is set to 10.0 seconds for the CMAC, behavior transfer from 1000 3v2 episodes takes 8.41 hours to learn 4 vs. 3, a 44% increase over the time to learn a 9.0 second hold time, and 20% longer than the equivalent RBF 4 vs. 3 training time.

To test the effect of using behavior transfer with a learned 3 vs. 2 value function, we train a set of keepers for a number of 3 vs. 2 episodes, save the function approximator's weights ($\pi_{(3vs2,final)}$) from a random 3 vs. 2 keeper, and use the weights to initialize all four keepers[5] in 4 vs. 3 so that $\rho(\pi_{(3vs2,final)}) = \pi_{(4vs3,initial)}$. Then we train on the 4 vs. 3 keepaway task until the average hold time for 1,000 episodes is greater than 9.0 seconds. Note that in our experiments we set the agents to have a 360^o field of view although agents do also learn with a more realistic 90^o field of view. Allowing the agents to see 360^o speeds up the rate of learning and increases the learned hold time, reducing data collection time.

Table 2 reports the average time spent training in 4 vs. 3 with CMAC players to achieve a 9.0 second average hold time for different amounts of 3 vs. 2 training. Column two reports the time spent training on 4 vs. 3 while the third column shows the total time to train 3 vs. 2 and 4 vs. 3. As can be seen from the table, spending time training in the simpler 3 vs. 2 domain can cause the time learning 4 vs. 3 to decrease. To overcome the high amounts of noise in our evaluation we run at least 45 independent trials for each data point reported.

Table 2 shows the potential of behavior transfer. We use a t-test to determine that the differences in the distributions of 4 vs. 3 training times and total training times when using behavior transfer are statistically significant ($p < 5.7 * 10^{-11}$) when compared to training 4 vs. 3 from scratch. Not only is the time to train the 4 vs. 3 task decreased when we first train on 3 vs. 2, but the total training time is less than the time to train 4 vs. 3 from scratch. We can therefore conclude that in the keepaway domain training first on a simpler task can increase the rate of learning enough that the total training time is decreased when using a CMAC function approximator.

To verify that the 4 vs. 3 CMAC players were benefiting from behavior transfer and not from having non-zero initial weights, we initialized CMAC weights uniformly to 1.0 in one set of experiments and then to random numbers from 0.0-0.5 in a second set of experiments. The learning time was *greater* than learning from scratch in both experiments. Haphazardly initializing CMAC weights may hurt the learner but systematically setting them through behavior transfer is beneficial.

Table 3 shows the average training time in 4 vs. 3 for different amount of 3 vs. 2 training using a neural network function approximator. All numbers reported are averaged over at least 35 independent trials. Not only is the 4 vs. 3 training time needed to reach the 9.0 second target performance reduced by using behavior transfer, but the total training time can also be reduced. A t-test confirms that the difference in total training times between using behavior transfer and training from scratch is statistically significant when using fewer than 500 3 vs. 2 episodes ($p < 1.2 * 10^{-2}$). Notice that the 4 vs. 3 training time increases as more 3 vs. 2 episodes are added. We posit this is due to overtraining, as the weights become more specific to the 3 vs. 2 task.[6] Table 3 also has results for the RBF players. All numbers reported are averaged over at least 29 independent trials; both 4 vs. 3 time and total time can be reduced with behavior transfer. A t-test confirms that all behavior transfer results differ from scratch ($p < 1.4 * 10^{-5}$).

We would like to be able to determine the optimal amount of time needed to train on an easier task to speed up a more difficult task. Determining these training thresholds for tasks in different domains is currently an open problem and will be the subject of future research, but our results suggest that the amount of time spent on the first task should be much smaller than the amount of time spent learning the second task.

Related Work

The concept of seeding a learned behavior with some initial simple behavior is not new. There have been approaches to simplifying reinforcement learning by manipulating the transition function, the agent's initial state, and/or the reward function, such as directed training (Selfridge, Sutton, & Barto 1985), learning from easy missions (Asada *et al.* 1994), and reward shaping (Colombetti & Dorigo 1993; Mataric 1994), as discussed in the Introduction. The "transfer of learning" approach (Singh 1992) applies specifically to temporally sequential subtasks. The subtasks must all be very similar in that they have the same state spaces, action spaces, and environment dynamics, although the reward function R may differ. While these four methods allow the learner to spend less total time training, they rely on a human modifying the task to create artificial problems to train on. We contrast this with behavior transfer where we allow the state and/or action spaces to change between actual tasks. This added flexibility permits behavior transfer to be applied to a wider range of domains as well as allowing independent modification of the transition function, the start state, or the reward function.

[5] We do so under the hypothesis that the policy of a single keeper represents all of the keepers' learned knowledge. Though in theory the keepers could be learning different policies that interact well with one another, so far there is no evidence that they do. One pressure against such specialization is that the keepers' start positions are randomized. In earlier informal experiments, there appeared to be some specialization when each keeper started in the same location every episode.

[6] Such an effect may be due to the particular neural network or player implementation used. We saw a similar effect before when using a CMAC with the CMUnited-99 players.

In some problems where subtasks are clearly defined by features, the subtasks can be automatically identified (Drummond 2002) and leveraged to increase learning rates. Learned subroutines have been successfully transfered in a hierarchical reinforcement learning framework (Andre & Russell 2002). By analyzing two tasks, subroutines may be identified which can be directly reused in a second task that has a slightly modified state space. For tasks which can be framed in a relational framework (Dzeroski, Raedt, & Driessens 2001), there is research (Morales 2003) which suggests ways of speeding up learning between two relational reinforcement learning tasks.

Imitation is another technique which may transfer knowledge from one learner to another (Price & Boutilier 2003). However, there is the assumption that "the mentor and observer have similar abilities" and thus may not be directly applicable when the number of dimensions of the state space changes or the agents have a qualitatively different action set. Other research (Fern, Yoon, & Givan 2004) has shown that it is possible to learn policies for large-scale planning tasks that generalize across different tasks in the same domain.

Another approach (Guestrin *et al.* 2003) uses linear programming to determine value functions for classes of similar agents. Using the assumption that T and R are similar among all agents of a class, class-based value subfunctions are used by agents in a new world that has a different number of objects (and thus different S and A). However, as the authors themselves state, the technique will not perform well in heterogeneous environments or domains with "strong and constant interactions between many objects (e.g. RoboCup)."

Conclusions

We have introduced the behavior transfer method of speeding up reinforcement learning and given empirical evidence for its usefulness. We have trained CMAC and neural network agents using TD reinforcement learning in related tasks with different state and action spaces and shown that not only is the time to learn the final task reduced, but that the total training time is reduced using behavior transfer when compared to simply learning the final task from scratch. In the future we will continue to explore how to apply behavior transfer to additional function approximators. Additionally, we will work on identifying tasks that are less directly related to each other but still benefit from behavior transfer.

Acknowledgments

We would like to thank Gregory Kuhlmann for his help with keepaway experiments described in this paper as well as Nick Jong, Raymond Mooney, and David Pardoe for helpful comments and suggestions. This research was supported in part by NSF CAREER award IIS-0237699, DARPA grant HR0011-04-1-0035, and the UT-Austin MCD Fellowship.

References

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

Andre, D., and Russell, S. J. 2002. State abstraction for programmable reinforcement learning agents. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 119–125.

Andre, D., and Teller, A. 1999. Evolving team Darwin United. In Asada, M., and Kitano, H., eds., *RoboCup-98: Robot Soccer World Cup II*. Berlin: Springer Verlag.

Asada, M.; Noda, S.; Tawaratsumida, S.; and Hosoda, K. 1994. Vision-based behavior acquisition for a shooting robot by using a reinforcement learning. In *Proc. of IAPR/IEEE Workshop on Visual Behaviors-1994*, 112–118.

Colombetti, M., and Dorigo, M. 1993. Robot Shaping: Developing Situated Agents through Learning. Technical Report TR-92-040, International Computer Science Institute, Berkeley, CA.

Crites, R. H., and Barto, A. G. 1996. Improving elevator performance using reinforcement learning. In Touretzky, D. S.; Mozer, M. C.; and Hasselmo, M. E., eds., *Advances in Neural Information Processing Systems 8*. Cambridge, MA: MIT Press.

de Boer, R., and Kok, J. R. 2002. The incremental development of a synthetic multi-agent system: The uva trilearn 2001 robotic soccer simulation team. Master's thesis, University of Amsterdam, The Netherlands.

Drummond, C. 2002. Accelerating reinforcement learning by composing solutions of automatically identified subtasks. *Journal of Artificial Intelligence Research* 16:59–104.

Dzeroski, S.; Raedt, L. D.; and Driessens, K. 2001. Relational reinforcement learning. *Machine Learning* 43:7–52.

Fern, A.; Yoon, S.; and Givan, R. 2004. Approximate policy iteration with a policy language bias. In Thrun, S.; Saul, L.; and Schölkopf, B., eds., *Advances in Neural Information Processing Systems 16*. Cambridge, MA: MIT Press.

Guestrin, C.; Koller, D.; Gearhart, C.; and Kanodia, N. 2003. Generalizing plans to new environments in relational mdps. In *International Joint Conference on Artificial Intelligence (IJCAI-03)*.

Mataric, M. J. 1994. Reward functions for accelerated learning. In *International Conference on Machine Learning*, 181–189.

Morales, E. F. 2003. Scaling up reinforcement learning with a relational representation. In *Proc. of the Workshop on Adaptability in Multi-agent Systems*.

Ng, A. Y.; Harada, D.; and Russell, S. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proc. 16th International Conf. on Machine Learning*.

Price, B., and Boutilier, C. 2003. Accelerating reinforcement learning through implicit imitation. *Journal of Artificial Intelligence Research* 19:569–629.

Puterman, M. L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc.

Riedmiller, M.; Merke, A.; Meier, D.; Hoffman, A.; Sinner, A.; Thate, O.; and Ehrmann, R. 2001. Karlsruhe brainstormers—a reinforcement learning approach to robotic soccer. In Stone, P.; Balch, T.; and Kraetszchmar, G., eds., *RoboCup-2000: Robot Soccer World Cup IV*. Berlin: Springer Verlag.

Selfridge, O.; Sutton, R. S.; and Barto, A. G. 1985. Training and tracking in robotics. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* 670–672.

Singh, S. P. 1992. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning* 8:323–339.

Stone, P.; Kuhlmann, G.; Taylor, M.; and Liu, Y. 2005. Keepaway soccer: From machine learning testbed to benchmark. In *Proceedings of RoboCup International Symposium*. To appear.

Stone, P.; Riley, P.; and Veloso, M. 2000. The CMUnited-99 champion simulator team. In Veloso, M.; Pagello, E.; and Kitano, H., eds., *RoboCup-99: Robot Soccer World Cup III*. Berlin: Springer. 35–48.

Stone, P.; Sutton, R. S.; and Kuhlmann, G. 2005. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*. To appear.

Sutton, R. S., and Barto, A. G. 1998. *Introduction to Reinforcement Learning*. MIT Press.

Taylor, M. E., and Stone, P. 2005. Behavior transfer for value-function-based reinforcement learning. In *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*. To appear.

Tesauro, G. 1994. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation* 6(2):215–219.