In *The Autonomous Agents and Multi-Agent Systems Conference* (AAMAS-07),
Honolulu, Hawaii, May 2007.

Transfer via Inter-Task Mappings in Policy Search Reinforcement Learning

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

ABSTRACT

The ambitious goal of transfer learning is to accelerate learning on a target task after training on a different, but related, source task. While many past transfer methods have focused on transferring value-functions, this paper presents a method for transferring policies across tasks with different state and action spaces. In particular, this paper utilizes transfer via inter-task mappings for policy search methods (TVITM-PS) to construct a transfer functional that translates a population of neural network policies trained via policy search from a source task to a target task. Empirical results in robot soccer Keepaway and Server Job Scheduling show that TVITM-PS can markedly reduce learning time when full inter-task mappings are available. The results also demonstrate that TVITM-PS still succeeds when given only incomplete inter-task mappings. Furthermore, we present a novel method for *learning* such mappings when they are not available, and give results showing they perform comparably to hand-coded mappings.

1. INTRODUCTION

In *reinforcement learning* (RL) [12] problems, agents take sequential actions with the goal of maximizing a reward signal, which may be time-delayed. Complex RL problems, such as robot control and game playing, in which the agents never have access to correctly labeled training examples, have been successfully learned. However, if RL agents begin from scratch without any assistance, mastering larger tasks may be infeasibly slow. Hence, a significant amount of research in RL focuses on improving performance by exploiting domain expertise. For example, if the designer can formulate advice about how the agent should behave [15] or can devise incrementally more challenging reward functions [2, 5], that agent may discover an effective policy more quickly.

Even if the designer has no expertise about how the agent should behave, he or she can still improve the agent's performance by specifying how a *target task* relates to a similar *source task*. Exploiting such information is a central goal of *transfer learning*. If the source task has already been learned, transfer learning need only reduce the training time on the target task to succeed. If the source task has not been learned, transfer learning can still be useful if it reduces the total training time. In other words, the time required to learn the source task and the target task, using transfer, must be less than the time required to learn the target task from scratch.

One successful transfer approach, *transfer via inter-task mapping for value function methods* (TVITM-VF) [13], utilizes a transfer functional to map a learned *value function* (i.e. a function that estimates the long-term expected reward of different states) from the source task to an initial value function in the target task. However, this approach applies only to methods which learn value functions, such as *temporal difference* (TD) methods [12]. Since *policy search* methods, which directly search the space of policies without learning value functions, can outperform TD methods on some tasks [9, 14], extending transfer learning to policy search methods is an important goal.

This paper presents *transfer via inter-task mapping for policy search methods* TVITM-PS, a method capable of transferring policies, represented as neural network action selectors, from a source to target task. We evaluate TVITM-PS empirically in two domains: robot soccer Keepaway, a standard multiagent RL benchmark domain [10], and Server Job Scheduling (SJS) [17], a probabilistic autonomic computing domain. Results show that TVITM-PS can reduce both target task training time and total training time compared to using the base policy search method without transfer.

Like other transfer learning methods (e.g. [8, 13, 15]), this approach can use hand-coded inter-task mappings to construct a transfer functional. However, this paper takes a further step by reducing the expertise required for successful transfer. First, we present additional empirical results that demonstrate that TVITM-PS succeeds even when only incomplete inter-task mappings are available. Second, we present a method for learning such mappings from experience. In this approach, mappings need not be available at all; only semantically distinct groups of state variables need to be identified. The resultant inter-task mappings are independent of the transfer method, RL algorithm, and internal agent representation. Results in both domains confirm that TVITM-PS succeeds with both incomplete and learned inter-task mappings.

Together, these results demonstrate that TVITM-PS is a promising approach for transfer learning with policy search methods, as it provides substantial performance improvements even when domain knowledge is limited.

2. TVITM-PS

To define the inter-task mappings utilized by TVITM-PS we use standard notation for Markov decision processes (MDPs) [6]. An agent's knowledge of the current state of its environment, $s \in S$, is a vector of k *state variables*, so that $s = x_1, x_2, \dots, x_k$. The agent has a set of actions, A, from which to choose. A reward function, $R : s \mapsto \mathbb{R}$, defines the instantaneous environmental reward of a state. A policy $\pi : S \mapsto A$ defines how an agent interacts with the environment. The success of an agent's policy is defined by how well it maximizes the total reward it receives in the long run while following that policy.

Transfer learning algorithms typically focus on a particular inter-

AAMAS'07, May 14-18, 2007, Honolulu, Hawaii.

nal agent representation. This work focuses on policies represented as neural network action selectors, which have k input nodes describing the agent's current state. There are $|A|$ output nodes and the agent chooses the action whose corresponding output node has the highest activation. We choose neural network action selectors because of their past successes in policy search (e.g. [9], [14], [17]). In this work we restrict the application of TVITM-PS to neural network action selectors, though there are no apparent obstacles to applying it to other policy search representations.

2.1 Constructing a Transfer Functional

This section shows how a hand-coded functional, ρ_H, is constructed for TVITM-PS such that $\rho_H(\pi_{source}) = \pi_{target}$. Thus ρ_H is able to transfer *policies* based on the relationship between state variables and action in a pair of tasks, whereas past transfer approaches [13] have focused on transferring value functions. To perform transfer with a neural network action selector, we must convert networks trained on the source task into networks suitable for training on the target task via ρ_H. However, we cannot simply copy the policy description unaltered because, in the general case, the state and actions spaces may differ between tasks, and therefore the policy function's inputs and outputs may differ.

Given an arbitrary pair of unknown tasks, one could not hope to correctly define such a transfer functional without sufficient domain knowledge or experience. When constructing ρ_H we assume that two hand-constructed inter-task mappings are provided, $\chi_{H,X}$ and $\chi_{H,A}$. $\chi_{H,X}$ maps each state variable in the target task to the most similar state variable in the source task: $\chi_{H,X}(x_{i,target}) = x_{j,source}$. Similarly, $\chi_{H,A}$ maps each action in the target task to the most similar action in the source tasks: $\chi_{H,A}(a_{i,target}) = a_{j,source}$. In this paper we transfer neural network action selectors so input nodes represent state variables while output nodes represent actions. Note that ρ_H transfers knowledge from the source task to the target task, while $\chi_{H,A}$ and $\chi_{H,X}$ are mappings from the target to the source.

Given $\chi_{H,X}$, $\chi_{H,A}$, and a trained network π_{source}, our goal is to create a new network π_{target} that can function in the target task. Initially, we define π_{target} to have no links, one input node for each state variable in the target task, one output node for each action in the target task, and the same number of hidden nodes as in π_{source}. We define the function δ to represent the 1-to-1 correspondence between these hidden nodes in the two networks: $\delta(h_{target}) = h_{source}$. Now each node n in π_{target} can be mapped back to a node in π_{source} via a function ψ which relies on δ and the hand-coded mappings:

$$\psi(n) = \begin{cases} \chi_{H,X}(n), & \text{if } n \text{ is an input node} \\ \chi_{H,A}(n), & \text{if } n \text{ is an output node} \\ \delta(n), & \text{if } n \text{ is a hidden node} \end{cases}$$

Using ψ, we can now generate π_{target} by copying the links that connect the corresponding nodes in π_{source}. For every pair of nodes n_i, n_j, in π_{target}, if a link exists between $\psi(n_i)$ and $\psi(n_j)$ in π_{source}, a new link with the same weight is created between n_i and n_j.[1] By applying this method to source task policies, we can initialize target task policies. All target task policies thus have structure and weights learned from the source task and we expect this knowledge to bias policies so that policy search methods can master the target task more quickly. Algorithm 1 summarizes this domain-independent process.

Algorithm 1 Constructing ρ_H for TVITM-PS

```
Construct a network $\pi_{target}$ where # of input and output nodes are determined by the target task
Add the same number of hidden nodes to $\pi_{target}$ as $\pi_{source}$
for each pair of nodes $n_i, n_j$ in $\pi_{target}$ do
    if link($\psi(n_i), \psi(n_j)$) in $\pi_{source}$ then
        Add link($n_i, n_j$) to $\pi_{target}$ with weight identical to link($\psi(n_i), \psi(n_j)$)
```

Note that there is no difficulty if the target policy has more or fewer inputs (state variables) or outputs (actions). Furthermore, for those policy search methods that can dynamically change the number of hidden nodes in a neural network (such as the one we use in our experiments), the constraint that the number of hidden nodes be the same in π_{source} and $\rho(\pi_{source})$ is not limiting (but for other policy search methods, it may be restrictive).

2.2 Incompletely Defined Inter-Task Mappings

Complete information relating the two tasks, as contained in $\chi_{H,X}$ and $\chi_{H,A}$, may not always be available. For example, when the target task is an extension of the source task, we may know which state variables and actions are identical in the two tasks but not know how to map *novel* state variables and actions back to the source task.

In such a scenario, instead of $\chi_{H,X}$ and $\chi_{H,A}$, we may define incomplete inter-task mappings, $\chi_{I,X}$ and $\chi_{I,A}$. Hence, we must use a partially defined ψ_I and the algorithm described above will sometimes be unable to add links. However, we can still use TVITM-PS with a modified transfer functional, ρ_I, generated using the same steps described in Algorithm 1 plus an additional step: fully connect any unconnected input/output pairs with a random weight.[2]

ρ_I exploits whatever information is available about the state variable and action correspondences, but all nodes in the network are still connected in the absence of this information. ρ_I tests whether transfer can succeed when only partial information about the correspondence of state variables and actions between the two tasks is available. While one would not expect transfer via ρ_I to work as well as when utilizing ρ_H, we will show that agents using ρ_I will still learn faster than starting from scratch (i.e. without transfer).

3. LEARNING INTER-TASK MAPPINGS

In Sections 2.1 and 2.2 we assumed knowledge of state variable and action relationships in a given pair of tasks. However, in some cases even partial mappings may not be available. This section addresses these cases by introducing a method to learn such mappings (domain-specific implementation details appear in Sections 5.3 and 6.2). To enable autonomous learning, we assume that state variables may be arranged into task-independent clusters which describe different objects in the domain. To discover appropriate inter-task mappings, the agent first observes its transitions in the source and target tasks. Given groupings of state variables plus gathered experience from source and target tasks, supervised learning methods then autonomously identify similarities between state variables and actions in the two tasks.

When learning in the source task, the agent records data samples of the form $(s_{source}, a_{source}, r, s'_{source})$, where r is the immediate reward; and s_{source} and s'_{source} are the state of the world (both vectors of k state variables) before and after the agent executes action a_{source}. These samples may be used to train classifiers that

[1] Alternatively, link weights could be to set such that the target network's activation for every output a, given $s_1 \dots s_k$, is the same as the source network's activation for $\chi_{H,A}(a)$, given $\chi_{H,X}(s_1) \dots \chi_{H,X}(s_k)$. However, informal results suggest this approach is less effective than directly copying weights.

[2] These random weights are chosen from the same distribution as used by the policy search algorithm when initializing networks from scratch.

predict the index of a particular state variable or action, given the rest of the data in the sample. For example, given a state, reward, and next state, classifier C_A predicts the action taken:

$$C_A(s_{source}, r, s'_{source}) = a_{source}.$$

Such a classifier may be used to define a learned inter-task mapping between actions, called $\chi_{L,A}$. If a sample gathered in the target task, $(s_{target}, a_{target}, r, s'_{target})$, is classified by C_A:

$$C_A(s_{target}, r, s'_{target}) = a_{source}$$

then a_{source} corresponds to a_{target} (i.e. $\chi_{L,A}(a_{target}) = a_{source}$).

In general, however, s_{target} and s'_{target} may have different state variables than s_{source} and s'_{source}. In such cases the state variables would have different semantics in different tasks and thus a classifier would be unlikely to produce useful mappings. To address this problem, we leverage our assumption and do not train classifiers on the full state, but on subsets of state variables as indicated by their given semantic groupings. Rather than training an action classifier with $2k + 1$ inputs, we will train multiple action classifiers, each with fewer inputs.

Suppose that there are T object types that define a domain's semantic groupings. For example, a logistics domain might have two object types: $T = \{trucks, locations\}$. If there are two trucks and two locations in the source task, then each state variable will belong to a particular object in $\{truck_A, truck_B, loc_A, loc_B\}$. Instead of learning one C_A, we learn a separate $C_{A,t}$ for each $t \in T$:

$$C_{A,t}(s_{i,source}, r, s'_{i,source}) = a_{source}$$

where $s_{i,source}$ contains the state variables associated with object i of type t. Thus the inputs for $C_{A,trucks}$ will be state variables associated with either $truck_A$ or $truck_B$. Each recorded data tuple (s, a, r, s') thus produces multiple training examples, one for each object.

Once trained, such classifiers can be used to define $\chi_{L,A}$. Each object j of type t in each sample gathered in the target task is input to the relevant $C_{A,t}$:

$$C_{A,t}(s_{j,target}, r, s'_{j,target}) = a_{source}.$$

Each classifier's output is interpreted as a "vote" for a correspondence between a_{target} and a_{source}; $\chi_{L,A}(a_{target})$ is defined as the action in the source task with the most votes. Continuing our example, states in the target task can be divided up so that the state variables $s_{j,target}$ and $s'_{j,target}$ that describe $truck_A$ in the target task are classified by $C_{A,truck}$, which counts as a vote for an a_{source} similar to the observed a_{target}. Likewise, state variables corresponding to $truck_B$ are classified by $C_{A,truck}$ to produce a second vote.

We may define a similar mapping between state variables, called $\chi_{L,X}$, by training classifiers to predict which object i is described in the input. Hence, we learn a separate $C_{X,t}$ for each of the t object types:

$$C_{X,t}(s_{i,source}, r, s'_{i,source}) = i$$

Once trained, these classifiers can be used to define $\chi_{L,X}$. Each object j of type t in each sample gathered in the target task is input to the relevant $C_{X,t}$:

$$C_{X,t}(s_{j,target}, r, s'_{j,target}) = i.$$

Again, each classifier's output is interpreted as a "vote" for a correspondence between object j in the target task and object i in the source task, and $\chi_{L,X}(s_{j,target})$ is defined by a winner-take-all method.

Given the inter-task mappings $\chi_{L,A}$ and $\chi_{L,X}$, we can then construct ρ_L via Algorithm 1. Note that if the action mapping were already known, each $C_{X,t}$ could utilize that χ_A to classify data in the form:

$$C_{X,t}(s_{j,target}, r, s'_{j,target}, \chi_A(a_{target})) = i$$

Likewise, a given χ_X could be leveraged to learn $\chi_{L,A}$. If one of the two classification tasks prove easier to learn, or one of the mappings is given but not the other, one inter-task mapping can be bootstrapped to learn the other.

Although we utilize these inter-task mappings with TVITM-PS, neural network action selectors, and policy search RL, this method is independent of the transfer method utilized, the agent's internal representation, and the base RL algorithm.

4. POLICY SEARCH WITH NEAT

Section 2 describes how to utilize TVITM-PS to initialize a neural network action selector for a target task given one for a source task. This transfer method is independent of the base learning algorithm, as long as it uses neural network action selectors. Neuroevolution methods [18, 20], which use *genetic algorithms* to evolve neural networks, are one such class of methods.

This paper uses *NeuroEvolution of Augmenting Topologies* (NEAT) [9] as a representative policy search method. Most neuroevolutionary systems require the network topology to be fixed and given (i.e. how many hidden nodes there are and how they are connected). By contrast, NEAT automatically evolves the topology by combining the search for network weights with evolution of the network structure. NEAT is a popular search and optimization method and is an appropriate choice for this paper because of past empirical successes on difficult RL tasks such as double pole balancing [9], Keepaway [14], and Server Job Scheduling [17].

In NEAT, a population of genomes, each of which describes a single neural network, is evolved over time: each genome is evaluated and the fittest individuals reproduce through crossover and mutation. NEAT begins with a population of simple networks with no hidden nodes and inputs connected directly to outputs. Two special mutation operators, *add hidden node* and *add link*, introduce new structure incrementally, but only structural mutations that improve performance tend to survive evolution. Thus NEAT can find an appropriate level of complexity for a given problem. NEAT is a general purpose optimization technique and can be applied to a wide variety of problems, but when applied to reinforcement learning problems it typically evolves action selectors.

5. KEEPAWAY

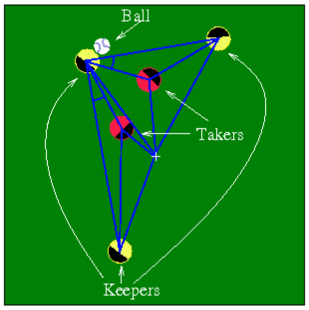

Figure 1: 3 vs. 2 Keepaway

To test the efficacy of TVITM-PS we consider the benchmark RoboCup simulated soccer Keepaway domain [11]. This multiagent domain has noisy sensors and actuators, and enforces a hidden state so that agents can perceive only a partial world view at any time. One team—the *keepers*—attempts to maintain possession of the ball on a field while another team—the *takers*—attempts to steal the ball or force it 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 longer average episodes. Figure 1 depicts three keepers playing against two takers.

The agents choose not from the simulator's primitive actions but from a set of higher-level macro-actions implemented as part of the player. These macro-actions can last more than one time step and the keepers make decisions only when a macro-action terminates. The macro-actions are Hold Ball, Get Open, Receive, and Pass [11]. The agents make decisions at discrete time steps, at which point macro-actions may be initiated and terminated. A keeper in 3 vs. 2 Keepaway in possession of the ball may choose to either hold the ball or pass to one of its teammates: A = {hold, passToTeammate1, passToTeammate2}. Keepers not in possession of the ball execute Receive, the macro-action in which the keeper who can reach the ball the fastest goes to the ball and the remaining players follow a hand-coded strategy to try to get open for a pass. Takers do not learn but follow a static hand-coded policy. Our players are built on version 0.6 of the benchmark players distributed by UT Austin [10], and our experiments use version 9.4.5 of the Soccer Server.

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 field, C. Keepers and takers are ordered by increasing distance from the ball and states are rotationally invariant. Note that as the number of keepers *n* and the number of takers *m* increase, the number of state variables also increases so that the more complex state can be fully described. The state variables must change (e.g. there are more distances to players to account for) and $|A|$ increases as there are more teammates to which the keeper with possession of the ball can pass.

In 3 vs. 2 Keepaway, three keepers are initially placed in three corners of a 25m × 25m field and a ball is placed near one of the keepers. Two takers are placed in the fourth corner. When an episode starts, the three keepers attempt to keep control of the ball by passing among themselves and moving to open positions. The agent's state is defined by 13 variables, as is shown by line segments and angles in Figure 1. The keepers receive a reward of +1 for every time step that the ball remains in play. The episode finishes when a taker gains control of the ball or the ball is kicked out of bounds. The episode is then reset with a random keeper placed near the ball. We used NEAT to evolve teams of homogeneous agents: in any given episode, copies of the same neural network control all three keepers.

Networks initially have 13 inputs, corresponding to the 13 state variables; 3 outputs, corresponding to the learnable macro-actions (i.e. A); 1 bias input; no hidden units; and random weights fully connecting all input and output nodes. Keepers always select the action with the highest activation, breaking ties randomly. The Keepaway task is stochastic and the evaluations are noisy; the optimal number of episodes to evaluate each NEAT organism for is difficult to establish *a priori*. We used 6,000 episodes per generation, as did previous research in this domain [14].

Keepaway becomes more challenging as more players are added because the field is more crowded. The keeper with the ball has more passing options but the average pass distance is shorter, which causes more passes and therefore more errors due to noise. Also, more takers are able to block passing lanes and chase down errant passes. For these reasons, keepers in 4 vs. 3 Keepaway take longer to learn an optimal control policy than in 3 vs. 2, and the asymptotic performance is lower.

The addition of an extra taker and keeper to the 3 vs. 2 task also results in a qualitative change. In 3 vs. 2, both takers must go towards the ball as both are needed to capture it from the keeper, but in 4 vs. 3 a third taker is free to roam the field and attempt to intercept passes. Hence, one keeper is often blocked from receiving a pass. Furthermore, the start state of one keeper is in the middle of the field, which has no analogue in 3 vs. 2.

In 4 vs. 3 Keepaway, A = {hold, passToTeammate1, passToTeammate2, passToTeammate3} and each state is composed of 19 state variables due to the added players. The number of evaluation episodes per generation was increased from 6,000 to 10,000 due to additional task complexity.

Partial Description of $\chi_{H,X}$

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 correspondences between state variables 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 bold.

5.1 Hand-Coded Keepaway Mappings

In this section we define the inter-task mappings $\chi_{H,X}$, $\chi_{H,A}$ used for transferring between 3 vs. 2 and 4 vs. 3 Keepaway. A and S change when the number of players is increased but we can intuitively define these mappings between states and actions to transfer knowledge between the two tasks.

The full hand-coded mapping between actions in the two tasks, $\chi_{H,A}$, identifies actions that have similar effects on the world state in both tasks. For the 3 vs. 2 and 4 vs. 3 tasks, the action "Hold ball" is equivalent, i.e. this action has a similar effect on the world in both tasks. Likewise, the action "Pass to closest keeper" is analogous in both tasks, as is "Pass to second closest keeper." We map the novel target action, "Pass to third closest keeper," to "Pass to second closest keeper" in the source task.

The state variable mapping, $\chi_{H,X}$, is handled with a similar strategy. Each of the 19 state variables in the 4 vs. 3 task is mapped to a similar state variable in the 3 vs. 2 task. For instance, "Distance to closest keeper" is the same in both tasks. "Distance to second closest keeper" in the source task is similar to "Distance to second closest keeper" in the target task, and also "Distance to third closest keeper" in the target task. See Table 1 for more examples of state variable mappings. These mappings are analogous to those previously used in Keepaway [13] and are used to construct ρ_H.

5.2 Incomplete Hand-Coded Mappings

$\chi_{I,X}$ is the same as $\chi_{H,X}$ except that the mapping for new state variables in 4 vs. 3 are not defined. For example, "Distance to second closest keeper" in the target task maps to "Distance to second closest keeper" in the source task, but $\chi_{I,X}$(*Distance to third closest keeper*) is undefined because this state variable is novel. Likewise, $\chi_{I,A}$ is the same as $\chi_{H,A}$, except $\chi_{I,A}$ (*Pass to third closest keeper*) is undefined. $\chi_{I,A}$ and $\chi_{I,X}$ can then be used to construct ψ_I and ρ_I, as in Section 2.1.

5.3 Learned Keepaway Inter-Task Mappings

As discussed in Section 3, when learning an inter-task mapping, domain knowledge is used to define how the state space should be semantically partitioned. In Keepaway it is natural to break the state space up into keepers, with four state variables each, and takers, defined by two state variables (see Table 2), similar to past transfer research in this domain [8]. To implement the learning method described in Section 3, we consider the Keepaway state at the time an action is executed and the state of the world, as perceived from that same keeper, after the action has successfully finished (i.e. the next time a keeper can select a macro-action).

To learn the inter-task mapping we train three classifiers using JRip, an implementation of RIPPER[1] included in Weka[19]. We selected JRip because it learns quickly and produces human understandable rules, but other classification methods in Weka had comparable results in informal experiments. The three classifiers learned are:

1. $C_{Keeper}(s_k, r, s'_k)$ = Source Keeper
2. $C_{Taker}(s_o, r, s'_o)$ = Source Taker
3. $C_{Action}(s_{3v2}, r, s'_{3v2})$ = Source Action

where s_k and s'_k are the subsets of state variables used to represent a single keeper (an object of type $keeper$) before and after an action has executed; s_o and s'_o describe a taker (an object of type $opponent$); s_{3v2} and s'_{3v2} describe an entire 3 vs. 2 Keepaway state; and r is the Keepaway reward accrued between actions (i.e. the number of timesteps elapsed).

Consider using a single $(s_{3vs2}, r, a, s'_{3vs2})$ tuple recorded in 3 vs. 2 used to train these three classifiers. This tuple yields two data points for training C_{Keeper}[3]:

- (s_{k_2}, r, s'_{k_2}), label = 2
- (s_{k_3}, r, s'_{k_3}), label = 3

where s_{k_2} and s_{k_3} are the state variables corresponding to keepers 2 and 3. Similarly, this tuple will produce two training examples for C_{Taker} and one for for C_{Action}.

Keeper State Variables
$dist(K_n, K_1)$
$dist(K_n, C)$
$min(dist(K_n, T_m))$
$min(ang(K_n, K_1, T_m))$
Taker State Variables
$dist(T_n, K_1)$
$dist(T_n, C)$

Table 2: The Keepaway state is partitioned into individual keepers and takers.

Once trained, the two classifiers are able to process data gathered from the world and label which keeper, taker, or action in the source task it is most similar to. Splitting data recorded from 3 vs. 2 into training and test sets allows us to check the correctness of the classifiers on the source task. We then utilize these classifiers to learn a mapping by applying them to data gathered in the target task.

In the target task we again assume that the state variables pertaining to keepers and takers can be identified. Each $(s_{4vs3}, r, a, s'_{4vs3})$ tuple recorded in the target task produces data for 3 keepers and 3 takers (again, assuming that K_1 is trivially identified). C_{Keeper} classifies sets of keeper state variables and C_{Taker} classifies the taker state variables, constructing $\chi_{L,X}$. Using $\chi_{L,X}$, the full state in the target task can be reformulated so that only information that was present in the source task is considered by C_{Action}. As in Section 5.1, multiple 4 vs. 3 keepers map to a single 3 vs. 2 keeper, and similarly for takers; one tuple recorded from the target task produces four examples for classification because s_{4vs3} may generate four[4] distinct s_{3v2}'s. $\chi_{L,A}$ is then constructed via C_{Action}.

For our experiments, we collected 1,000 tuples $(s_{3v2}, r, a_{3v2}, s'_{3v2})$ from 3 vs. 2 Keepaway and 100 $(s_{4v3}, r, a_{4v3}, s'_{4v3})$ tuples from 4 vs. 3 Keepaway. Note that collecting the tuples from the source task is "free," assuming that at least that many tuples were experienced during training. Recording the 100 tuples in 4 vs. 3 took about a minute of simulated time, which was negligible compared to the hours spent training. The three classifiers were able to correctly classify source players and actions, as tested with cross-validation. A representative confusion matrix for the target task keepers is shown in Table 3. The learned transfer functional is constructed via a winner-take-all scheme and the resulting mappings are shown in Table 4.

5.4 Keepaway Transfer Results

	Source K_2	Source K_3
Target K_2	286	17
Target K_3	234	69
Target K_4	37	266

Table 3: A representative Keeper confusion matrix, demonstrating that C_{Keeper} has been successfully learned.

In this section we compare learning times in Keepaway when learning from scratch or when using one of the three transfer functionals (see Table 4) to perform transfer. After a population of policies in 3 vs. 2 have been trained, we transfer the entire population[5] into 4 vs. 3 via TVITM-PS using ρ_H, ρ_I, or ρ_l. These initial policies in the 4 vs. 3 population will not be optimal but should enable evolution to more rapidly discover good policies in the target task.

Target	$\chi_{H,X}$	$\chi_{I,X}$	$\chi_{L,X}$
K_1	K_1	K_1	K_1
K_2	K_2	K_2	K_2
K_3	K_3	K_3	K_2
K_4	K_3	none	K_3
T_1	T_1	T_1	T_1
T_2	T_2	T_2	T_2
T_3	T_2	none	T_2
Target	$\chi_{H,A}$	$\chi_{I,A}$	$\chi_{L,A}$
Hold	Hold	Hold	Hold
Pass K_2	Pass K_2	Pass K_2	Pass K_2
Pass K_3	Pass K_3	Pass K_3	Pass K_2
Pass K_4	Pass K_3	none	Pass K_3

Table 4: This table enumerates the player correspondences defined by the three Keepaway χ's.

Recall that our first learning scenario focuses on the time required to learn in the target task. By setting a threshold level of performance in the target task, we are able to measure the amount of training time needed to achieve this performance. By this measure, transfer learning is effective if we learn the target task faster by utilizing policies trained in the source task than by learning from scratch. A second, and more difficult, scenario requires that the training time for the source and target tasks combined is shorter than the training time to learn the target task from scratch. We show in this section that TVITM-PS with NEAT meets both transfer goals by comparing learning with transfer and learning from scratch.

In Keepaway, noise in the sensors and actuators causes the evaluation of a policy to have high variance. Therefore, after learning finished we evaluated the best policy in each generation for 1,000 episodes to generate more accurate graphs. All times reported in this paper refer to simulator time. We report only sample complexity and not computational complexity because the running time for our learning methods is negligible compared to that of the Soccer Server. The machines used for our experiments allowed us to speed up the simulator by a factor of two so that the real experimental time required was roughly half that of the reported simulator time.

After setting a threshold performance values for the 4 vs. 3 task, we analyze the champions of each generation after learning and determine when the organism identified as the best by NEAT has learned to hold the ball for at least the threshold value, averaged over 1,000 episodes. If no policies reach the threshold value within

[3] We assume that the state variables corresponding to K_1, the keeper with the ball, are known, as the majority of distances are measured relative to it. Learning this keeper would be simple, but complicates the exposition. Likewise, we assume that the hold action is known; it is trivial to classify as it is the only action lasting a single timestep.

[4] s_{3v2} requires 3 objects of type keeper and 2 objects of type taker. K_1 will always be included in s_{3vs2} and K_4 has been correctly included in $\chi_{L,X}$, but the following additional player groupings are possible via $\chi_{L,X}$: (K_2, K_4, T_1, T_2), (K_3, K_4, T_1, T_2), (K_2, K_4, T_1, T_3) and (K_3, K_4, T_1, T_3).

[5] Transferring the entire trained population instead of a single policy allows search to begin in the target task from a variety of locations in policy space, increasing the chances of finding a good starting point for learning. Informal results show that this is more beneficial than transferring a few of the best policies.

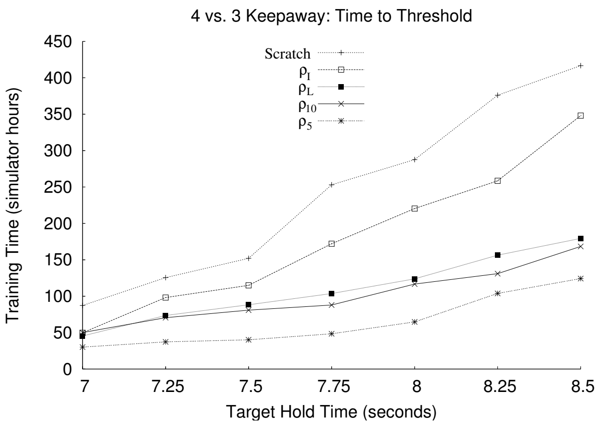

Figure 2: TVITM-PS **successfully reduces the average training time needed to reach a given performance level relative to scratch.**

500 hours, that trial is assigned a time of 500 hours.

Figure 2 shows the training time each method takes to reach threshold times of 7.0 to 8.5 seconds. Five learning curves were generated by averaging over 10 independent runs using four learning methods: learning from scratch, using ρ_H after training for 5 generations of 3 vs. 2 (ρ_5) or 10 generations of 3 vs. 2 (ρ_{10}), and using both ρ_I and ρ_L after training for 5 generations of 3 vs. 2. A Student's t-test confirms that both the difference between ρ_5 and scratch, as well as between ρ_{10} and scratch, are statistically significant at the 95% level for all points graphed. The differences between ρ_I and scratch, as well as ρ_L and scratch, are statistically significant at over half of the points graphed. These results clearly show that, in Keepaway, TVITM-PS reduces learning times in the target task.

When considering the total training time, learning curves in Figure 2 which use transfer are shifted up by the amount of time spent training in the source task. Figure 3 shows the target and total training times needed to reach a target threshold of 8.5 seconds. The differences between the total training times and scratch for all four TVITM-PS methods are statistically significant for roughly half of the target threshold times shown in Figure 2.

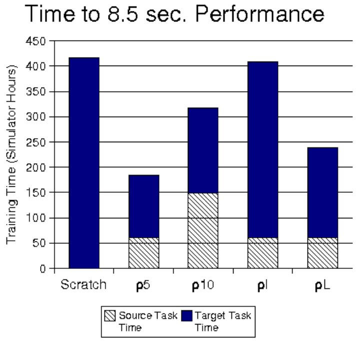

Figure 3: Average Keepaway training time needed to reach a target performance of 8.5 seconds. Both the target task time and total training time are successfully reduced compared to scratch.

6. SERVER JOB SCHEDULING

Server Job Scheduling (SJS) [17] is a complex probabilistic control task we use to assess whether TVITM-PS succeeds in a second, unrelated task. SJS is an *autonomic computing* task in which a server (e.g. a website's application server or database) must determine in what order to process jobs currently waiting in its queue. Its goal is to maximize the aggregate utility of all the jobs it processes. A *utility function* for each job type maps the job's completion time to the utility derived by the user [16]. The problem becomes challenging when these utility functions are non-linear and/or the server must process multiple types of jobs. Since selecting a particular job for processing necessarily delays the completion of all other jobs in the queue, the scheduler must weigh difficult trade-offs to maximize aggregate utility.

Each experiment in our simulator begins with 100 jobs preloaded into the server's queue and ends when the queue empties. During each timestep, the server removes and processes one job from its queue. During each of the first 100 timesteps, a new job of a randomly selected type is added to the end of the queue after the agent processes a job, forcing the agent to make decisions about which job to process as new jobs are arriving. Since one job is processed per timestep, each episode lasts 200 timesteps. For each job that completes, the scheduling agent receives an immediate reward determined by that job's utility function. Utility functions for the job types used in our experiments, which are not provided to the scheduling agent, are shown in Figure 4. The source task uses job types #1 and #2 while the target task uses all four and is therefore a prime candidate for TVITM-PS.

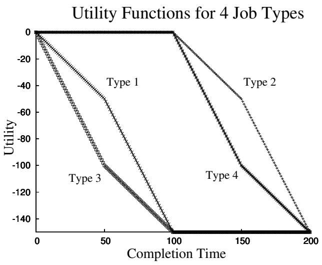

Figure 4: The four utility functions used in our SJS experiments.

The state and action spaces are discretized: the range of job ages from 0 to 200 is divided into four equal sections. The scheduler is told, at each timestep, how many jobs in the queue of each type fall in each range, resulting in 8 state variables in the source task and 16 in the target. The action space is similarly discretized to 8 or 16 distinct actions: rather than selecting a particular job, the scheduler specifies what type of job it wants to process and which of the four age ranges that job should lie in. Each NEAT network has 8 inputs and outputs in the source task and 16 in the target task. We use a population of 50 and evaluate each organism by averaging the reward from 5 episodes. After training is finished, the champion organisms are evaluated for an additional 95 episodes to reduce noise when graphing performance. Other NEAT parameters are the same as those reported for past research in this domain [17].

6.1 SJS Hand-Coded Mappings

In this section we define the inter-task mappings used for 2-job-type and 4-job-type scheduling. We utilize a similar heuristic to Keepaway: because job type 3 is most similar to job type 1, job type 1 in the source task corresponds to job types 1 and 3 in the target task. Likewise, job type 2 in the source task corresponds to job types 2 and 4 in the target task. The incomplete inter-task mappings are defined so that job types 1 and 2 in the target task correspond to job types 1 and 2 in the source task. The target task's novel job types, 3 and 4, are not given any correspondence to the source task and ρ_I will initialize their associated weights with random values.

We found that a small modification to Algorithm 1 improves performance. Instead of copying every link in the source task, $link(input, output)$, to every output node, we copy it for only half of the outputs. For instance, consider that in the source policy, state variable 1 for job 1 (an input) is connected to all 8 actions (outputs). Using $\chi_{H,X}$ and $\chi_{H,A}$, state feature 1 in the target should be connected to all 16 actions in the target policy. However, in the target network we found it sufficient to connect state feature 1 (an

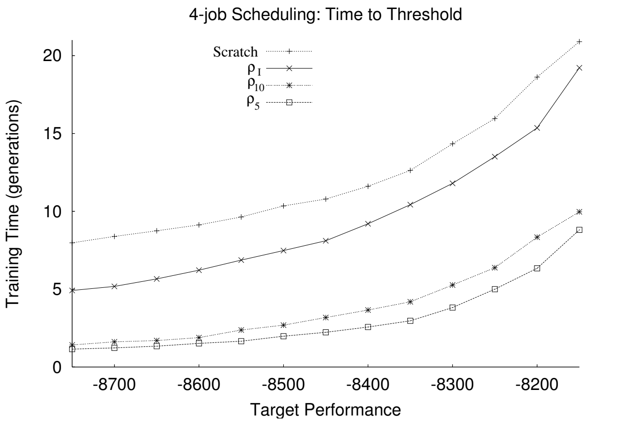

Figure 5: The average number of generations in SJS needed to attain a target performance is successfully reduced via TVITM-PS**.**

input for job type 1) to actions 1-8 (actions for job types 1 and 2). State feature 9 (an input for job type 3) was connected to actions 9-16 (actions for job types 3 and 4). This effectively redefines ψ so that the existing job features are connected to existing job actions, while novel job features are corrected to novel job actions, halving the number of links in the target network. We hypothesize that this performance increase is due to removing unnecessary complexity: learning in lower dimensional spaces is faster, assuming the complexity is sufficient to represent good policies.

6.2 Learning SJS Inter-Task Mappings

In SJS we group the state variables by job type. That is, a classifier C_{Job} will take as input state variables that specify the four job counts for a particular job type. s' is defined to be the state of the world immediately after a job has been removed but before a new job is added. Thus the two classifiers we use to learn $\chi_{L,X}$ and $\chi_{L,A}$ are defined as:

1. $C_{Job}(s_{job}, r, s'_{job}) =$ Source Job Type
2. $C_{Action}(s_{job}, r, jobType_{source}, s'_{job}) =$ Source Action

As in Keepaway, we first learn $\chi_{L,X}$ by learning the state variable correspondence between the source and target tasks. C_{Job} is trained with data in the source task, where every recorded tuple produces two data, one for each job type. Then data from the target task, four for each tuple, is used to map the four target job types into the two source job types. C_{Action} is trained on source data and two data are generated for each action taken. When using C_{Action} in the target task, since there are four job types, there are four data to classify for each recorded action in the environment. Actions in SJS are defined as removing a job from a particular job type and we utilize $\chi_{L,X}$ when using C_{Action}: $C_{Action}(s_{job}, r, \chi_{L,X}(jobType_{target}), s'_{job}) = A_{source}$. Thus C_{Action} is used to construct $\chi_{L,A}$ and ρ_L.

In our experiments, when using 10,000 tuples from the source task and 10,000 tuples from the target task, ρ_L was identical to ρ_H for all 4 job types and 16 actions, suggesting that this approach to learning inter-task mappings is effective compared to hand-coded mappings. On average, 50 episodes in the target tasks were used collecting data to generate ρ_L. This is still very small (one tenth of a generation) relative to total training times.

6.3 SJS Transfer Results

Figure 5 shows training time in the target task for server job scheduling. There are four learning curves as $\rho_L = \rho_H$ in this domain. Figure 6 shows the total training time needed to reach a performance where the average utility is at least -8150. Training times for all methods are averaged over 100 independent runs and champion performance is averaged over 100 episodes to reduce variance. If any NEAT run does not meet the specified threshold it is assigned a time of 100 generations. Differences between the number of target task episodes are significant at the 95% level for transfer and scratch for all thresholds graphed in Figure 5. The total number of generations used by ρ_5 and ρ_{10} are statistically different from scratch for all points graphed.

7. DISCUSSION

The previous sections empirically demonstrate that TVITM-PS can successfully transfer knowledge in both transfer scenarios, namely reducing both the target task training time and total training time. Transfer with ρ_I enables faster learning in the target task than from scratch in both domains, particularly at the beginning of learning, but ρ_H is even more beneficial. This suggests it is most effective to formulate a full transfer functional between all state variables and actions in the two tasks, but when one is not available, a partial inter-task mappings can still successfully enable transfer.

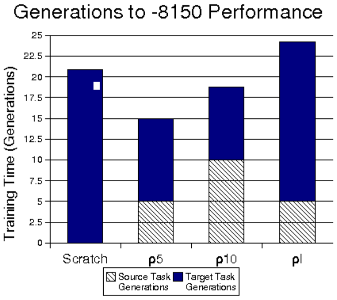

Figure 6: TVITM-PS **successfully reduces both the number of training generations and total generations needed to reach a target performance of -8150.**

In our experiments we found that ρ_L was able to outperform ρ_I because the learned transfer functionals were more similar to the hand-coded functionals. In domains where a complete inter-task mapping is unavailable, but classification is able to leverage similarities between the two tasks to correctly classify objects (i.e. cross-validation on the source data shows that the classifiers are correctly learning concepts), it may be more beneficial to use learned inter-task mappings rather than relying on an incomplete mapping. This is significant because much less domain knowledge is required when the inter-task mappings are learned; we believe this method represents a step towards fully autonomous transfer.

We hypothesize that ρ_5 outperforms ρ_{10} in both domains primarily due to two factors. Source networks trained for 10 generations have more links and nodes than those trained for 5, and more complex networks are likely to train more slowly. Secondly, ρ_{10} may have overfit the source task, as seen in other transfer work [13].

The significant benefit from TVITM-PS, particularly in SJS, is possible in part because of qualitative similarities in the source and target tasks, despite differing in S, A, and relative complexity. Commonalities between tasks can make it easier to reduce target task training time, but they will not necessarily make it easier to reduce *total* training time (our second goal), and may even make it harder. In the extreme case, the source could be identical to the target task, making the first transfer goal trivial but the second impossible. The fact that TVITM-PS meets *both* transfer learning goals is an important confirmation of this transfer method's effectiveness. Reducing the total time is possible, in part, because the source tasks are similar to, but easier to learn than, their respective target tasks.

8. RELATED AND FUTURE WORK

There have been many previous approaches to speeding up reinforcement learning. For instance, *directed training* [7] allows a researcher to modify the transition function over time and slowly make the task harder. Reward shaping [2, 5] implicitly guides agents by modifying the reward function. These methods generally assume S and A remain the same between pairs of tasks. TVITM-PS permits them to change, allowing transfer to be applied to a larger set of tasks.

Another related approach uses linear programming to determine value functions for classes of similar agents [4]. As the authors state, the technique will not perform well in heterogeneous environments or domains with "strong and constant interactions between many objects (e.g. RoboCup)." Automatically generated advice can also be used to speed up learning in transfer [15], allowing an RL agent to build up a model for the source task and extract general advice. A human provides a mapping for this advice into the new task, similar to ρ, to set relative preferences for different actions in different states. Other transfer work [3] allows speedup between tasks but does not allow S and A to differ between the two tasks. Homomorphisms may be leveraged to transfer between tasks with different S and A [8], and while χ_X is effectively unnecessary, χ_A is hand-coded and total training time is not reduced.

TVITM-PS is conceptually an extension of TVITM-VF [13]; we show in this paper that the transfer using inter-task mappings can be successfully applied to policy search methods and in multiple domains. Because TVITM-PS transfers policies rather than value functions, it is robust to changes in the reward structure between the two tasks that do not effect the optimal policy (i.e. multiplying the reward function by a positive constant). Differences in experimental setup prohibit direct comparisons between the two methods, but the percentage speedups from both transfer methods are similar.

An important direction for future work is to attempt to further reduce the required domain knowledge when learning $\chi_{L,X}$ and $\chi_{L,A}$, such as removing the requirement that the tasks contain identifiable object types. Our success when using simple classification suggests that more complex methods which can better leverage environmental experience may be able to remove the requirement for semantic knowledge. Additionally, it would expand the applicability of this method if inter-task mappings could be learned between domains with different reward structures (e.g. between Keepaway, where the reward is based on time, and Breakaway [15], where the reward is based on goals scored). It is possible that constructing transfer functionals that use a mixture of source state variables and actions rather than a winner-take-all approach will outperform our current formulation. Lastly, TVITM-PS assumes inter-task mappings between the two tasks can be provided or learned; it would be more beneficial to identify *a priori* which pairs of tasks are related *enough* so that TVITM-PS could succeed.

9. CONCLUSIONS

This paper introduces TVITM-PS for transfer with policy search methods and gives empirical evidence that it can significantly speed up learning in pairs of related RL tasks. We demonstrate how to construct transfer functionals from full, partial and learned inter-task mappings. We use NEAT, a popular policy search method, to master pairs of tasks in Keepaway and Server Job Scheduling that have different state and action spaces. Transferring learned policies between the two tasks reduces not only training time in the target task, but also the total training time required, an important confirmation of this transfer method's efficacy.

Acknowledgments

We would like to thank Cynthia Matuszek, Nick Jong, Lilyana Mihalkova, Shivaram Kalyanakrishnan, Joseph Reisinger, and anonymous reviewers for helpful comments and suggestions. This research was supported in part by DARPA grant HR0011-04-1-0035, NSF CAREER award IIS-0237699, and NSF award EIA-0303609.

10. REFERENCES

[1] W. W. Cohen. Fast effective rule induction. In *International Conf. on Machine Learning*, pages 115–123, 1995.

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

[3] F. Fernandez and M. Veloso. Learning by probabilistic reuse of past policies. In *Proc. of the 6th International Conference on Autonomous Agents and Multiagent Systems*, 2006.

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

[5] M. J. Mataric. Reward functions for accelerated learning. In *International Conference on Machine Learning*, 1994.

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

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

[8] V. Soni and S. Singh. Using homomorphisms to transfer options across continuous reinforcement learning domains. In *Proceedings of the Twenty First National Conference on Artificial Intelligence*, July 2006.

[9] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[10] P. Stone, G. Kuhlmann, M. E. Taylor, and Y. Liu. Keepaway soccer: From machine learning testbed to benchmark. In I. Noda, A. Jacoff, A. Bredenfeld, and Y. Takahashi, editors, *RoboCup-2005: Robot Soccer World Cup IX*, volume 4020, pages 93–105. Springer Verlag, Berlin, 2006.

[11] P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

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

[13] M. E. Taylor and P. Stone. Behavior transfer for value-function-based reinforcement learning. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 53–59, New York, NY, July 2005. ACM Press.

[14] M. E. Taylor, S. Whiteson, and P. Stone. Comparing evolutionary and temporal difference methods for reinforcement learning. In *Proc. of the Genetic and Evolutionary Computation Conf.*, pages 1321–28, July 2006.

[15] L. Torrey, T. Walker, J. Shavlik, and R. Maclin. Using advice to transfer knowledge acquired in one reinforcement learning task to another. In *Proceedings of the Sixteenth European Conference on Machine Learning*, 2005.

[16] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proc. of the International Conf. on Autonomic Computing*, pages 70–77, 2004.

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

[18] D. Whitley and J. Schaffer, editors. *International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*, Los Alamitos, CA, 1992. IEEE Computer Society Press.

[19] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[20] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.