To appear in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2003.

Evolving Keepaway Soccer Players through Task Decomposition

Shimon Whiteson, Nate Kohl, Risto Miikkulainen, and Peter Stone

Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, Texas 78712-1188
{shimon,nate,risto,pstone}@cs.utexas.edu
http://www.cs.utexas.edu/~{shimon,nate,risto,pstone}

Abstract. In some complex control tasks, learning a direct mapping from an agent's sensors to its actuators is very difficult. For such tasks, decomposing the problem into more manageable components can make learning feasible. In this paper, we provide a task decomposition, in the form of a decision tree, for one such task. We investigate two different methods of learning the resulting subtasks. The first approach, layered learning, trains each component sequentially in its own training environment, aggressively constraining the search. The second approach, coevolution, learns all the subtasks simultaneously from the same experiences and puts few restrictions on the learning algorithm. We empirically compare these two training methodologies using neuro-evolution, a machine learning algorithm that evolves neural networks. Our experiments, conducted in the domain of simulated robotic soccer keepaway, indicate that neuro-evolution can learn effective behaviors and that the less constrained coevolutionary approach outperforms the sequential approach. These results provide new evidence of coevolution's utility and suggest that solution spaces should not be over-constrained when supplementing the learning of complex tasks with human knowledge.

1 Introduction

One of the goals of machine learning algorithms is to facilitate the discovery of novel solutions to problems, particularly those that might be unforeseen by human problem-solvers. As such, there is a certain appeal to "tabula rasa learning," in which the algorithms are turned loose on learning tasks with no (or minimal) guidance from humans. However, the complexity of tasks that can be successfully addressed with tabula rasa learning given current machine learning technology is limited.

When using machine learning to address tasks that are beyond this complexity limit, some form of human knowledge must be injected. This knowledge simplifies the learning task by constraining the space of solutions that must be considered. Ideally, the constraints simply enable the learning algorithm to find the best solutions more quickly. However there is also the risk of eliminating the best solutions from the search space entirely.

In this paper, we consider a multi-agent control task that, given current methods, seems infeasible to learn via a tabula rasa approach. Thus, we provide some structure via a task decomposition in the form of a decision tree. Rather than learning the entire task from sensors to actuators, the agents now learn a small number of subtasks that are combined in a predetermined way.

Providing the decision tree then raises the question of how training should proceed. For example, 1) the subtasks could be learned sequentially, each in its own training environment, thereby adding additional constraints to the solution space. On the other hand, 2) the subtasks could be learned simultaneously from the same experiences. The latter methodology, which can be considered coevolution of the subtasks, does not place any further restrictions on the learning algorithms beyond the decomposition itself.

In this paper, we empirically compare these two training methodologies using neuro-evolution, a machine learning algorithm that evolves neural networks. We attempt to learn agent controllers for a particular domain, namely keepaway in simulated robotic soccer. Our results indicate that neuro-evolution can learn effective keepaway behavior, though constraining the task beyond the tabula rasa approach proves necessary. We also find that the less constrained coevolutionary approach to training the subtasks outperforms the sequential approach. These results provide new evidence of coevolution's utility and suggest that solution spaces should not be over-constrained when supplementing the learning of complex tasks with human knowledge.

The remainder of the paper is organized as follows. Section 2 introduces the keepaway task as well as the general neuro-evolution methodology. Section 3 fully specifies the different approaches that we compare in this paper. Detailed empirical results are presented in Section 4 and are evaluated in Section 5. Section 6 concludes and discusses future work.

2 Background

This section describes simulated robotic soccer keepaway, the domain used for all experiments reported in this paper. We also review the fundamentals of neuro-evolution, the general machine learning algorithm used throughout.

2.1 Keepaway

The experiments reported in this paper are all in a keepaway subtask of robotic soccer [15]. In keepaway, one team of agents, the *keepers*, attempts to maintain possession of the ball while the other team, the *takers*, tries to get it, all within a fixed region. Keepaway has been used as a testbed domain for several previous machine learning studies. For example, Stone and Sutton implemented keepaway in the RoboCup soccer simulator [14]. They hand-coded low-level behaviors and applied learning, via the Sarsa(λ) method, only to the high-level decision of when and where to pass. Di Pietro et al. took a similar approach, though they used genetic algorithms and a more elaborate high-level strategy [8]. Machine learning was applied more comprehensively in a study that used genetic programming, though in a simpler grid-based environment [6].

We implement the keepaway task within the SoccerBots environment [1]. SoccerBots is a simulation of the dynamics and dimensions of a regulation game in the RoboCup small-size robot league [13], in which two teams of robots maneuver a golf ball on a field built on a standard ping-pong table. SoccerBots is smaller in scale and less complex than the RoboCup simulator [7], but it runs approximately an order of magnitude faster, making it a more convenient platform for machine learning research.

To set up keepaway in SoccerBots, we increase the size of the field to give the agents enough room to maneuver. To mark the perimeter of the game, we add a large bounding circle around the center of the field. Figure 1 shows how a game of keepaway is initialized. Three keepers are placed just inside this circle at points equidistant from each other. We place a single taker in the center of the field and place the ball in front of a randomly selected keeper.

After initialization, an episode of keepaway proceeds as follows. The keepers receive one point for every pass completed. The episode ends when the taker touches the ball or the ball exits the bounding circle. The keepers and the taker are permitted to go outside the bounding circle.

In this paper, we evolve a controller for the keepers, while the taker is controlled by a fixed intercepting behavior. The keepaway task requires complex behavior that integrates sensory input about teammates, the opponent, and the ball. The agents must make high-level decisions about the best course of action and develop the precise control necessary to implement those decisions. Hence, it forms a challenging testbed for machine learning research.

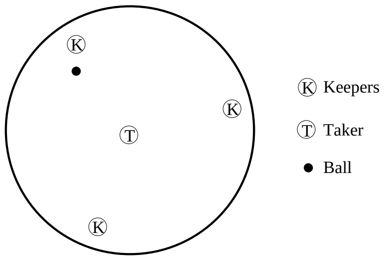

Fig. 1: A game of keepaway after initialization. The keepers try to complete as many passes as possible while preventing the ball from going out of bounds and the taker from touching it.

2.2 Neuro-Evolution

We train a team of keepaway players using neuro-evolution, a machine learning technique that uses genetic algorithms to train neural networks [11]. In its simplest form, neuro-evolution strings the weights of a neural network together to form an individual genome. Next, it evolves a population of such genomes by evaluating each one in the task and selectively reproducing the fittest individuals through crossover and mutation.

The Enforced Sub-Populations Method (ESP) [4] is a more advanced neuro-evolution technique. Instead of evolving complete networks, it evolves sub-populations of neurons. ESP creates one sub-population for each hidden node of the fully connected two-layer feed-forward networks it evolves. Each neuron is itself a genome which records the weights going into and coming out of the given hidden node. As Figure 2 illustrates, ESP forms networks by selecting one neuron from each sub-population to form the hidden layer of a neural network, which it evaluates in the task. The fitness is then passed back equally to all the

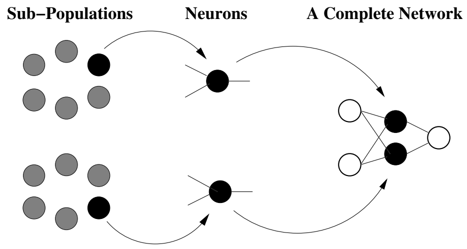

Fig. 2: The Enforced Sub-Populations Method (ESP). The population of neurons is segregated into sub-populations, shown here as clusters of grey circles. One neuron, shown in black, is selected from each sub-population. Each neuron consists of all the weights connecting a given hidden node to the input and output nodes, shown as white circles. The selected neurons together form a complete network which is then evaluated in the task.

neurons that participated in the network. Each sub-population tends to converge to a role that maximizes the fitness of the networks in which it appears. ESP is more efficient than simple neuro-evolution because it decomposes a difficult problem (finding a highly fit network) into smaller subproblems (finding highly fit neurons).

In several benchmark sequential decision tasks, ESP outperformed other neuro-evolution algorithms as well as several reinforcement learning methods [2–4]. ESP is a promising choice for the keepaway task because the basic skills required in keepaway are similar to those at which ESP has excelled before.

3 Method

The goals of this study are 1) to verify that neuro-evolution can learn effective keepaway behavior, 2) to show that decomposing the task is more effective than tabula rasa learning, and 3) to determine whether coevolving the component tasks can be more effective than learning them sequentially.

Unlike soccer, in which a strong team will have forwards and defenders specialized for different roles, keepaway is symmetric and can be played effectively with homogeneous teams. Therefore, in all these approaches, we develop one controller to be used by all three keeper agents. Consequently, all the agents have the same set of behaviors and the same rules governing when to use them, though they are often using different behaviors at any time. Having identical agents makes learning easier, since each agent learns from the experiences of its teammates as well as its own. In the remainder of this section, we describe the three different methods that we consider for training these agents.

3.1 Tabula Rasa Learning

In the tabula rasa approach, we want our learning method to master the task with minimal human guidance. In keepaway, we can do this by training a single "monolithic" network. Such a network attempts to learn a direct mapping from the agent's sensors to its actuators. As designers, we need only specify the network's architecture (i.e. the inputs, hidden units, outputs, and their connectivity) and neuro-evolution does the rest. The simplicity of such an approach is appealing though, in difficult tasks like keepaway, learning a direct mapping may be beyond the ability of our training methods, if not simply beyond the representational scope of the network.

To implement this monolithic approach with ESP, we train a fully connected two-layer feed-forward network with nine inputs, four hidden nodes, and two outputs, as illustrated in Figure 3. This network structure was determined, through experimentation, to be the most effective. Eight of the inputs specify the positions of four crucial objects on the field: the agent's two teammates, the taker, and the ball. The ninth input represents the distance of the ball from the field's bounding circle. The inputs to this network and all those considered in this paper are represented in polar coordinates relative to the agent. The four hidden nodes allow the network to learn a compacted representation of its inputs. The network's two outputs control the agent's movement on the field: one alters its heading, the other its speed. All runs use sub-populations of size 100.

Since learning a robust keepaway controller directly is so challenging, we facilitate the process through incremental evolution. In incremental evolution, complex behaviors are learned gradually, beginning with easy tasks and advancing through successively more challenging ones. Gomez and Miikkulainen showed that this method can learn more effective and more general behavior than direct evolution in several dynamic control tasks, including prey capture [2] and non-Markovian double pole-balancing [3].

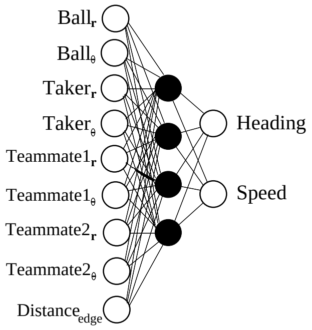

Fig. 3: The monolithic network for controlling keepers. White circles indicate inputs and outputs while black circles indicate hidden nodes.

We apply incremental evolution to keepaway by changing the taker's speed. When evolution begins, the taker can move only 10% as quickly as the keepers. We evaluate each network in 20 games of keepaway and sum its scores (numbers of completed passes) to obtain its fitness. When the population's average fitness exceeds 50 (2.5 completed passes per episode), the taker's speed is incremented by 5%. This process continues until the taker is moving at full speed or the population's fitness has plateaued.

3.2 Learning with Task Decomposition

If learning a monolithic network proves infeasible, we can make the problem easier by decomposing it into pieces. Such task decomposition is a powerful, general principle in artificial intelligence that has been used successfully with machine learning in the full robotic soccer task [12]. In the keepaway task, we can replace the monolithic network with several smaller networks: one to pass the ball, another to receive passes, etc.

To implement this decomposition, we developed a decision tree, shown in Figure 4, for controlling each keeper. If the agent is near the ball, it kicks to

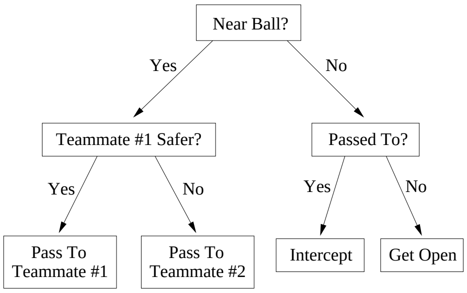

Fig. 4: A decision tree for controlling keepers in the keepaway task. The behavior at each of the leaves is learned through neuro-evolution. A network is also evolved to decide which teammate the agent should pass to.

the teammate that is more likely to successfully receive a pass. If it is not near the ball, the agent tries to get open for a pass unless a teammate announces its intention to pass to it, in which case it tries to receive the pass by intercepting the ball. The decision tree effectively provides some structure (based on human knowledge of the task) to the space of policies that can be explored by the learners.

To implement this decision tree, four different networks must be trained. The networks, illustrated in Figure 5, are described in detail below. As in the monolithic approach, these network structures were determined, through experimentation, to be the most effective.

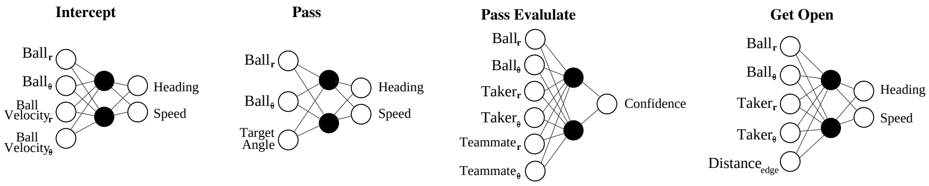

Fig. 5: The four networks used to implement the decision tree shown in Figure 4. White circles indicate inputs and outputs while black circles indicate hidden nodes.

Intercept: The goal of this network is to get the agent to the ball as quickly as possible. The obvious strategy, running directly towards the ball, is optimal only if the ball is not moving. When the ball has velocity, an ideal interceptor must anticipate where the ball is going. The network has four inputs: two for the ball's current position and two for the ball's current velocity. It has two hidden nodes and two outputs, which control the agent's heading and speed.

Pass: The pass network is designed to kick the ball away from the agent at a specified angle. Passing is difficult because an agent cannot directly specify what direction it wants the ball to go. Instead, the angle of the kick depends on the agent's position relative to the ball. Hence, kicking well requires a precise "wind-up" to approach the ball at the correct speed from the correct angle. The pass network has three inputs: two for the ball's current position and one for the target angle. It has two hidden nodes and two outputs, which control the agent's heading and speed.

Pass Evaluate: Unlike the other networks, which correspond to behaviors at the leaves of the decision tree, the pass evaluator implements a branch of the tree: the point when the agent must decide which teammate to pass to. It analyzes the current state of the game and assesses the likelihood that an agent could successfully pass to a specific teammate. The pass evaluate network has six inputs: two each for the position of the ball, the taker, and the teammate whose potential as a receiver it is evaluating. It has two hidden nodes and one output, which indicates, on scale of 0 to 1, its confidence that a pass to the given teammate would succeed.

Get Open: The get open network is activated when a keeper does not have a ball and is not receiving a pass. Clearly, such an agent should get to a position where it can receive a pass. However, an optimal get open behavior would not just position the agent where a pass is most likely to succeed. Instead, it would position the agent where a pass would be most strategically advantageous (e.g. by considering future pass opportunities as well). The get open network has five inputs: two for the ball's current position, two for the taker's current position, and one indicating how close the agent is to the field's bounding circle. It has two hidden nodes and two outputs, which control the agent's heading and speed.

After decomposing the task as described above, we need to evolve networks for each of the four subtasks. These networks can be trained in sequence, through layered learning, or simultaneously, through coevolution. The remainder of this section details these two alternatives.

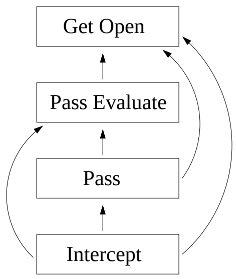

Fig. 6: A layered learning hierarchy for the keepaway task. Each box represents a layer and arrows indicate dependencies between layers. A layer cannot be learned until all the layers it depends on have been learned.

Layered Learning. One approach to training the components of a task decomposition is layered learning, a bottom-up paradigm in which low-level behaviors are learned prior to high-level ones [16]. Since each component is trained separately, the learning algorithm optimizes over several small solution spaces, instead of one large one. However, since some sub-behaviors must be learned before others, it is not usually possible to train each component in the actual domain. Instead, we must construct a special training environment for each component. The hierarchical nature of layered learning makes this construction easier: since the components are learned from the bottom-up, we can use the already completed sub-behaviors to help construct the next training environment. In the original implementation of layered learning, each sub-task was learned and frozen before moving to the next layer [16]. However, in some cases it is beneficial to allow some of the lower layers to continue learning while the higher layers are trained [17]. For simplicity, here we freeze each layer before proceeding.

Figure 6 shows one way in which the components of the task decomposition can be trained using layered learning. An arrow from one layer to another indicates that the latter layer depends on the former. A given task cannot be learned until all the layers that point to it have been learned. Hence, learning begins at the bottom, with intercept, and moves up the hierarchy step by step. The training environment for each layer is described below.

Intercept: To train the interceptor, we propel the ball towards the agent at various angles and speeds. The agent is rewarded for minimizing the time it takes to touch the ball. As the interceptor improves, the initial angle and speed of the ball increase incrementally.

Pass: To train the passer we propel the ball towards the agent and randomly select at which angle we want it to kick the ball. The agent employs the intercept behavior learned in the previous layer until it arrives near the ball, at which point it switches to the pass behavior being evolved. The agent's reward is inversely proportional to the difference between the target angle and the ball's actual direction of travel. As the passer improves, the range of angles at which it is required to pass increases incrementally.

Pass Evaluate: To train the pass evaluator, the ball is placed in the center of the field and the pass evaluator is placed just behind it at various angles. Two teammates are situated near the edge of the bounding circle on the other side of the ball at a randomly selected angle. A single taker is placed similarly but nearer to the ball to simulate the pressure it exerts on the passer. The teammates and the taker use the previously learned intercept behavior. We run the evolving network twice, once for each teammate, and pass to the teammate who receives the higher evaluation. The agent is rewarded only if the pass succeeds.

Get Open: When training the get open behavior, the other layers have already been learned. Hence, the get open network can be trained in a complete game of keepaway. Its training environment is identical to that of the monolithic approach with one exception: during a fitness evaluation the agents are controlled by our decision tree. The tree determines when to use each of the four networks (the three previously trained components and the evolving get open behavior).

At each layer, the results of previous layers are used to assist in training. In this manner, all the components of the task decomposition can be trained and assembled into an effective keepaway controller. However, the behaviors learned with this method are optimized for their training environment, not the keepaway task as a whole. It may sometimes be possible to learn more effective behaviors through coevolution, which we discuss next.

Coevolution. A much less constrained method of learning the keepaway agents' sub-behaviors is to evolve them all simultaneously, a process called coevolution. In general, coevolution can be competitive [5, 10], in which case the components are adversaries and one component's gain is another's loss. Coevolution can also be cooperative [9], as when the various components share fitness scores.

In our case, we use an extension of ESP designed to coevolve several cooperating components. This method, called Multi-Agent ESP, has been successfully used to master multi-agent predator-prey tasks [18]. In Multi-Agent ESP, each component is evolved with a separate, concurrent run of ESP. During a fitness evaluation, networks are formed in each ESP and evaluated together in the task. All the networks that participate in the evaluation receive the same score. Therefore, the component ESPs coevolve compatible behaviors that together solve the task.

The training environment for this coevolutionary approach is very similar to that of the get open layer described above. The decision tree still governs each keeper's behavior though the four networks are now all learning simultaneously, whereas three of them were fixed in the layered approach.

4 Empirical Results

To compare monolithic learning, layered learning, and coevolution, we ran seven trials of each method, each of which evolved for 150 generations. In the layered approach, the get open behavior, trained in a full game of keepaway, ran for 150 generations. Additional generations were used to train the lower layers. Figure 7 shows what task difficulty (i.e. taker speed) each method reached during the course of evolution, averaged over all seven runs. This graph shows that decomposing the task vastly improves neuro-evolution's ability to learn effective controllers for keepaway players. The results also demonstrate the efficacy of coevolution. Though it requires fewer generations to train and less effort to implement, it achieves substantially better performance than the layered approach in this task.

How do the networks trained in these experiments fair in the hardest version of the task? To determine this, we tested the evolving networks from each method against a taker moving at 100% speed. At every fifth generation, we selected the strongest network from the best run of each method and subjected it to 50 fitness evaluations, for a total of 1000 games of keepaway for each network (recall that one fitness evaluation consists of 20 games of keepaway). Figure 8, which shows the results of these tests, further verifies the effectiveness of coevolution. The learning curve of the layered approach appears flat, indicating that it was unable to significantly improve the keepers' performance through training the get open network. However, the layered approach outperformed the monolithic method, suggesting that it made substantial progress when training the lower layers. It is essential to note that neither the layered nor monolithic approaches trained at this highest task difficulty, whereas the best run of coevolution did. Nonetheless, these tests provide additional confirmation that neuro-evolution can truly master complex control tasks once they have been decomposed, particularly when using a coevolutionary approach.

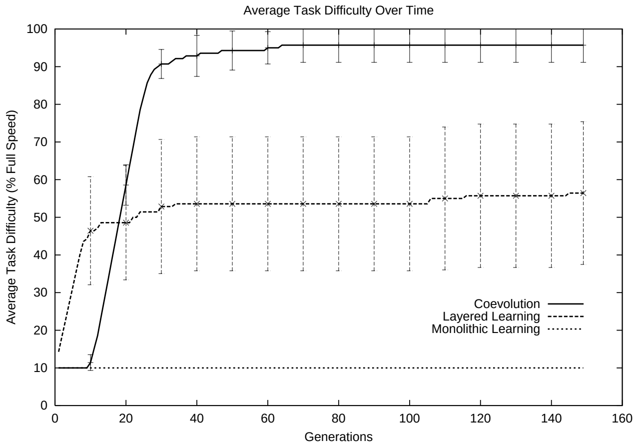

Fig. 7: Task difficulty (i.e. taker speed) of each method over generations, averaged over seven runs. Task decomposition proves essential for reaching the higher difficulties. Only coevolution reaches the hardest task.

5 Discussion

The results described above verify that given a suitable task decomposition neuro-evolution can learn a complex, multi-agent control task that is too difficult to learn monolithically. Given such a decomposition, layered learning developed a successful controller, though the less-constrained coevolutionary approach performed significantly better.

By placing fewer restrictions on the solution space, coevolution benefits from greater flexibility, which may contribute to its strong performance. Since coevolution trains every sub-behavior in the target environment, the components have the opportunity to react to each other's behavior and adjust accordingly. In layered learning, by contrast, we usually need to construct a special training environment for most layers. If any of those environments fail to capture a key aspect of the target domain, the resulting components may be sub-optimal. For example, the interceptor trained by layered learning is evaluated only by how quickly it can reach the ball. In keepaway, however, a good interceptor will approach the ball from the side to make the agent's next pass easier. Since the coevolving interceptor learned along with the passer, it was able to learn this superior behavior, while the layered interceptor just approached the ball directly. Though it is possible to adjust the layered interceptor's fitness function to encourage this indirect approach, it is unlikely that a designer would know *a priori* that such behavior is desirable.

The success of coevolution in this domain suggests that we can learn complex tasks simply by providing neuro-evolution with a high-level strategy. However, we suspect that in extremely difficult tasks, the solution space will be too large for coevolution to search effectively given current neuro-evolution techniques. In these cases, the hierarchical features of layered learning, by greatly reducing the solution space, may prove essential to a successful learning system.

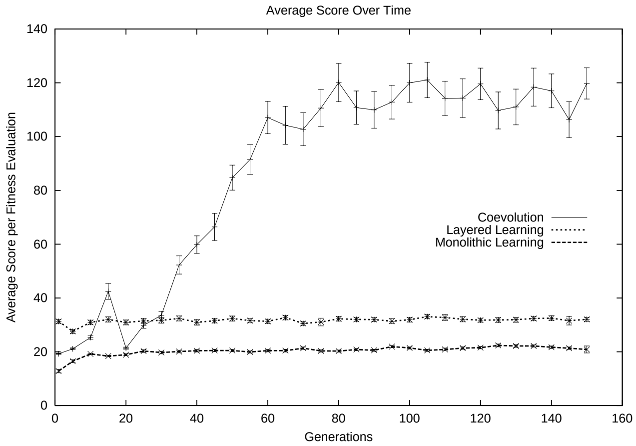

Fig. 8: Average score per fitness evaluation for the best run of each method over generations when the taker moves at 100% speed. These results demonstrate that task decomposition is important in this domain and that coevolution can effectively learn the resulting subtasks.

Layered learning and coevolution are just two points on a spectrum of possible methods which differ with respect to how aggressively they constrain learning. At one extreme, the monolithic approach tested in this paper places very few restrictions on learning. At the other extreme, layered learning confines the search by directing each component to a specific sub-goal. The layered and coevolutionary approaches can be made arbitrarily more constraining by replacing some of the components with hand-coded behaviors. Similarly, both methods can be made less restrictive by requiring them to learn a decision tree, rather than giving them a hand-coded one.

6 Conclusion and Future Work

In this paper we verify that neuro-evolution can master keepaway, a complex, multi-agent control task. We also show that decomposing the task is more effective than training a monolithic controller for it. Our experiments demonstrate that the more flexible coevolutionary approach learns better agents than the layered approach in this domain.

In ongoing research we plan to further explore the space between unconstrained and highly constrained learning methods. In doing so, we hope to shed light on how to determine the optimal method for a given task. Also, we plan to test both the layered and coevolutionary approaches in more complex domains to better assess the potential of these promising methods.

Acknowledgments

This research was supported in part by the National Science Foundation under grant IIS-0083776, and the Texas Higher Education Coordinating Board under grant ARP-0036580476-2001.

References

1. T. Balch. Teambots domain: Soccerbots, 2000. `http://www-2.cs.cmu.edu/~trb/TeamBots/Domains/SoccerBots`.
2. F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342, 1997.
3. F. Gomez and R. Miikkulainen. Solving non-Markovian control tasks with neuroevolution. Denver, CO, 1999.
4. F. Gomez and R. Miikkulainen. Learning robust nonlinear control with neuroevolution. Technical Report AI01-292, The University of Texas at Austin Department of Computer Sciences, 2001.
5. T. Haynes and S. Sen. Evolving behavioral strategies in predators and prey. In G. Weiß and S. Sen, editors, *Adaptation and Learning in Multiagent Systems*, pages 113–126. Springer Verlag, Berlin, 1996.
6. W. H. Hsu and S. M. Gustafson. Genetic programming and multi-agent layered learning by reinforcements. In *Genetic and Evolutionary Computation Conference*, New York,NY, July 2002.
7. I. Noda, H. Matsubara, K. Hiraki, and I. Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12:233–250, 1998.
8. A. D. Pietro, L. While, and L. Barone. Learning in RoboCup keepaway using evolutionary algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1065–1072, New York, 9-13 July 2002. Morgan Kaufmann Publishers.
9. M. A. Potter and K. A. D. Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8:1–29, 2000.
10. C. D. Rosin and R. K. Belew. Methods for competitive co-evolution: Finding opponents worth beating. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 373–380, San Mateo,CA, July 1995. Morgan Kaufman.
11. J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In D. Whitley and J. Schaffer, editors, *International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*, pages 1–37. IEEE Computer Society Press, 1992.
12. P. Stone. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press, 2000.
13. P. Stone, (ed.), M. Asada, T. Balch, M. Fujita, G. Kraetzschmar, H. Lund, P. Scerri, S. Tadokoro, and G. Wyeth. Overview of RoboCup-2000. In *RoboCup-2000: Robot Soccer World Cup IV*. Springer Verlag, Berlin, 2001.
14. P. Stone and R. S. Sutton. Scaling reinforcement learning toward RoboCup soccer. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 537–544. Morgan Kaufmann, San Francisco, CA, 2001.
15. P. Stone and R. S. Sutton. Keepaway soccer: a machine learning testbed. In *RoboCup-2001: Robot Soccer World Cup V*. Springer Verlag, Berlin, 2002.
16. P. Stone and M. Veloso. Layered learning. In *Machine Learning: ECML 2000*, pages 369–381. Springer Verlag, Barcelona,Catalonia,Spain, May/June 2000. Proceedings of the Eleventh European Conference on Machine Learning (ECML-2000).
17. S. Whiteson and P. Stone. Concurrent layered learning. In *Second International Joint Conference on Autonomous Agents and Multiagent Systems*, July 2003. To appear.
18. C. H. Yong and R. Miikkulainen. Cooperative coevolution of multi-agent systems. Technical Report AI01-287, The University of Texas at Austin Department of Computer Sciences, 2001.