In *Machine Learning*,
59(1):5-30, 2005.

Evolving Soccer Keepaway 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. Complex control tasks can often be solved by decomposing them into hierarchies of manageable subtasks. Such decompositions require designers to decide how much human knowledge should be used to help learn the resulting components. On one hand, encoding human knowledge requires manual effort and may incorrectly constrain the learner's hypothesis space or guide it away from the best solutions. On the other hand, it may make learning easier and enable the learner to tackle more complex tasks. This article examines the impact of this trade-off in tasks of varying difficulty. A space laid out by two dimensions is explored: 1) how much human assistance is given and 2) how difficult the task is. In particular, the neuroevolution learning algorithm is enhanced with three different methods for learning the components that result from a task decomposition. The first method, coevolution, is mostly unassisted by human knowledge. The second method, layered learning, is highly assisted. The third method, concurrent layered learning, is a novel combination of the first two that attempts to exploit human knowledge while retaining some of coevolution's flexibility. Detailed empirical results are presented comparing and contrasting these three approaches on two versions of a complex task, namely robot soccer keepaway, that differ in difficulty of learning. These results confirm that, given a suitable task decomposition, neuroevolution can master difficult tasks. Furthermore, they demonstrate that the appropriate level of human assistance depends critically on the difficulty of the problem.

Keywords: Coevolution, Neural Networks, Genetic Algorithms, Robot Soccer.

1. Introduction

Hierarchical task decomposition is a powerful method for tackling complex problems. As a case in point, mammalian morphology is a composition of hierarchically organized components, each able to perform specialized subtasks. These components span many levels of behavior ranging from individual cells to complex organs, and culminating in the complete organism. Even at the purely behavioral level, organisms have distinct subsystems, including reflexes, the visual system, and so on. It is difficult to imagine a monolithic entity that would be capable of the range and complexity of behaviors that mammals exhibit.

Similarly, hierarchical approaches have been proposed to help create agents for complex control tasks (Brooks, 1986; Gat, 1998; Dietterich, 1998). Typically, these approaches require a task decomposition as input, thus raising the question of how much human knowledge should be used to help learn the resulting components. On one hand, encoding human knowledge requires manual effort and may incorrectly constrain the learner's hypothesis space or guide it away from the best solutions. On the other hand, it may reduce the learning complexity and enable the learner to tackle significantly more difficult tasks than would be otherwise possible. This article examines the impact of this tradeoff in tasks of varying difficulty. A space laid out by two dimensions is explored: 1) how much human assistance is given to the learning method and 2) how difficult the task faced by the learning method is.

All of our experiments rely on neuroevolution, a learning algorithm which uses genetic algorithms to train neural networks. To explore the first dimension, we enhance neuroevolution with three different methods for learning the components that result from a task decomposition. At one extreme of this dimension lies the paradigm known as *coevolution* (Rosin and Belew, 1995; Potter and Jong, 2000; Ficici and Pollack, 1998; Stanley and Miikkulainen, 2004), in which no additional information is given to the learner beyond the task decomposition itself. At the other extreme lies *layered learning* (Stone and Veloso, 1998, 2000), a bottom-up paradigm by which low-level behaviors (those closer to the environmental inputs) are trained and fixed prior to training high-level behaviors. In our analogy to mammalian morphology, coevolution is akin to evolving cells, complex organs, and higher-level behaviors all simultaneously, whereas layered learning is more akin to requiring that cells completely evolve and remain fixed prior to evolving organs, which in turn must remain unchanged as high-level behaviors develop. In addition, this paper introduces a new approach, called *concurrent layered learning*, that combines layered learning's aggressive use of human knowledge with the flexibility of coevolution.

The human assistance provided to learning algorithms typically consists of either *constraints* or *guidance*. Constraints, which are usually more restrictive, prevent the learner from exploring certain parts of the hypothesis space. In contrast, guidance leaves the entire space open but encourages exploration in certain parts, by initializing the learner with a specific hypothesis or supplying an intermediate fitness function that favors part of the space. The task decomposition on which all three of the above methods rely is a form of constraint, since the learner is not allowed to explore solutions that do not consist of the components provided by the decomposition. Layered learning adds additional constraints by fixing the solutions to components that have already been learned. It also adds guidance, by supplying intermediate fitness functions for the components to maximize. Concurrent layered learning, by sometimes allowing those previously learned components to continue training, converts some of those constraints into less restrictive guidance.

To examine the second dimension, the difficulty of the learning task, we develop two versions of a complex task called robot soccer keepaway. In the easier version, the learner is asked to train low-level components that will be controlled by a hand-coded, high-level strategy. In the more difficult version, the hand-coded component is removed and the learner is asked to train a component that implements a high-level strategy in addition to the low-level components it governs.

Detailed empirical tests were conducted that compare coevolution, layered learning, and concurrent layered learning on both versions of the keepaway task. The learning algorithm in all cases is neuroevolution, which has been shown to be effective in learning control policies and behavior strategies in similar domains (Schaffer et al., 1992; Gomez and Miikkulainen, 2001; Yao, 1999). Our results demonstrate that:

- given a suitable task decomposition, neuroevolution can master a complex, multi-agent control task at which it otherwise fails,

- when training the components that result from such a task decomposition, the correct level of human assistance to supply to the learning method depends critically on the difficulty of the task, and

- on difficult tasks, our novel method, concurrent layered learning, offers a more effective balance between human assistance and flexibility.

The remainder of this paper is organized as follows. Section 2 explains the substrate systems on which our experiments are built, namely the robot soccer keepaway testbed and neuroevolution. Section 3 specifies the coevolution and layered learning approaches that we use, introduces concurrent layered learning, and details our application of these methods to the keepaway domain. Section 4 presents the results of our detailed experiments while Sections 5 and 6 discuss the implications of our experiments and related work respectively. Section 7 discusses future work and Section 8 concludes.

2. Background

This section describes simulated robot soccer keepaway, the domain used for all experiments reported in this paper. We also review the fundamentals of neuroevolution, the machine learning algorithm used throughout.

2.1. Keepaway

Keepaway is a subtask of robot soccer, where 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 (Stone and Sutton, 2002). Our implementation of the keepaway task is based on the SoccerBots environment (Balch, 2000), which simulates the dynamics and dimensions of a regulation game in the RoboCup small-size robot league (Stone et al., 2001). In this league, two teams of robots maneuver a golf ball on a small field. SoccerBots is smaller in scale and less complex than the RoboCup simulator (Noda et al., 1998), but it runs approximately an order of magnitude faster, making it a more practical platform for machine learning research.

In SoccerBots, each robot receives noise-free sensory input describing the current state of the game. All these inputs (described in detail in Section 3) are scaled to $[-1, 1]$ and presented in relative coordinates. The actuators for each robot consist of a throttle, a directional output that specifies a turn direction relative to the player's current orientation, and a small paddle that allows the players to "kick" the ball. A simple physics engine models collisions between robots and the ball.

Keepaway in SoccerBots is played on a circular field (Figure 1). Three keepers are placed just inside the circle at points equidistant from each other, and a single taker is placed in the center of the field. The ball is then placed in front of a randomly selected keeper and the game begins.

During the game, the keepers try to complete as many passes to each other as possible, while the taker does its best to steal the ball.

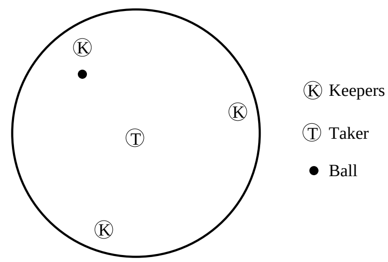

Figure 1. A game of keepaway after initialization.

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. All keepers are controlled by the same evolved neural network controller, while the taker is controlled by a fixed intercepting algorithm.

A number of factors make this implementation of the keepaway task challenging:

– The keepers are relatively large when compared to the playing area, which makes moving and positioning difficult around the ball.

– The ball does not move much faster than the players, which prevents the keepers from being able to quickly make passes around the taker.

– The keepers do not possess any abilities for handling the ball. They are modeled as simple cylinders, and lack any way to “grab” the ball and move with it. If they run into the ball, the ball will bounce away.

The keepaway task therefore 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.

2.2. Neuroevolution

The team of keepaway players is trained using neuroevolution, a machine learning technique that uses genetic algorithms to train neural networks (Schaffer et al., 1992). In its simplest form, neuroevolution strings the weights of a neural network together to form an individual genome. A population of such genomes is then evolved by evaluating each one in the task and selectively reproducing the fittest individuals through crossover and mutation.

The Enforced Subpopulations Method (ESP; Gomez and Miikkulainen, 2001, 2003) is a more advanced neuroevolution technique. Instead of evolving complete networks, it evolves subpopulations of neurons. ESP creates one subpopulation 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 subpopulation 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 neurons that participated in the network. Each subpopulation tends to converge to a role that maximizes the fitness of the networks in which it appears. ESP is more efficient than simple neuroevolution because it decomposes a difficult problem (finding a highly fit network) into smaller subproblems (finding highly fit neurons). Below we provide a step-by-step description of the ESP algorithm.[1]

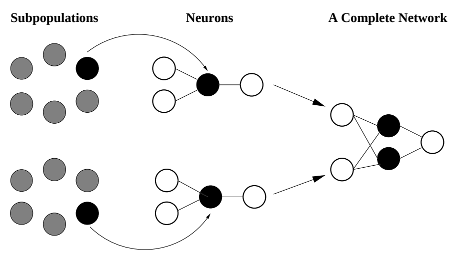

Figure 2. The Enforced Subpopulations Method (ESP). The population of neurons is segregated into subpopulations, shown here as clusters of grey circles. One neuron, shown in black, is selected from each subpopulation. Each neuron consists of all the weights connecting a given hidden node to the input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

1. Initialization. The number of hidden units u in the networks that will be formed is specified and a subpopulation of neuron chromosomes is created. Each chromosome encodes the input and output connection weights of a neuron with a random string of real numbers.

2. Evaluation. A set of u neurons is selected randomly, one neuron from each subpopulation, to form the hidden layer of a feed-forward network. The network is submitted to a *trial* in which it is evaluated

[1] This description is adapted from (Gomez and Miikkulainen, 2001)

on the task and awarded a fitness score. The score is added to the *cumulative fitness* of each neuron that participated in the network. This process is repeated until each neuron has participated in an average of t trials.

3. Recombination. The average fitness of each neuron is calculated by dividing its cumulative fitness by the number of trials in which it participated. Neurons are then ranked by average fitness within each subpopulation. Each neuron in the top quartile is recombined with a higher-ranking neuron using 1-point crossover and mutation at low levels. The resulting offspring replace the lowest-ranking half of the subpopulation.

4. The Evaluation-Recombination cycle is repeated until an optimal solution is found or until a threshold time limit is reached.

When performance begins to stagnate (i.e. the score of the best network from each generation has not improved in 20 generations), ESP applies a diversification technique called delta-coding (Whitley et al., 1991) in order to prevent premature convergence. Delta-coding selects the strongest individual from a population and uses it to seed a new population. Each member of this new population is a perturbation of the selected seed. Because the seed is highly fit, optimal networks are likely to be similar to it but occasionally may be radically different. Hence, the amount of perturbation is based on a Cauchy distribution, such that most of the new individuals are very similar to the seed but a few are significantly different. The diversification that delta-coding provides can significantly improve the performance of genetic algorithms (Whitley et al., 1991).

For particularly difficult problems, ESP can be coupled with a process called incremental evolution. In incremental evolution, complex behaviors are learned gradually by exposing the agents to a series of increasingly difficult training environments. The agents initially learn very easy tasks and advance to more difficult ones as their performance improves. The target domain is the last training environment in this sequence. 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 and non-Markovian double pole-balancing (Gomez and Miikkulainen, 1997, 1999).

Neuroevolution has repeatedly been shown to be an effective reinforcement learning method for non-linear control tasks (Gruau et al., 1996; Moriarty and Miikkulainen, 1996; Stanley and Miikkulainen, 2004; Gomez, 2003). In particular, 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 (i.e. learning multi-agent behaviors in the predator-prey domain and being able to execute the fine-grained control necessary for pole-balancing and robot control). In these benchmark sequential decision tasks, ESP was shown to outperform other neuroevolution algorithms as well as several reinforcement learning methods (Gomez and Miikkulainen, 1997, 1999, 2001, 2003).

3. Method

This section describes several methods for training agents in complex control tasks using neuroevolution. These methods vary in the degree to which human knowledge is used to focus the learning. First, we describe the baseline tabula rasa approach. Second, we describe how complex tasks can be decomposed into more manageable pieces. Finally, we present three different methods for learning the components that result from such a task decomposition: coevolution, layered learning, and concurrent layered learning.

This section also describes in detail how we apply each of the above methods to keepaway, the testbed we use in all of our experiments. Though all these techniques for mastering keepaway are very different, they share the same general approach: to develop one controller for use by all three keeper agents. This approach is feasible because keepaway is a symmetric task that can be played effectively with homogeneous teams. Since the keepers share a controller, they have the same set of behaviors and the same rules governing when to use them, though they are often using different behaviors at a given time. Having identical agents makes learning easier, since each agent learns from the experiences of its teammates as well as its own. However, it does eliminate the possibility of learning asymmetric policies in which the individual agents learn specialized roles. We leave the exploration of heterogeneous agents to future work.

The experiments reported in this section all involve the keepers learning against a single, fixed taker. This taker single-mindedly attempts to intercept the ball using the same interception behavior learned by the keepers in the layered learning approach described in Section 3.2.4.

3.1. Tabula Rasa Learning

In the tabula rasa approach, the learning method is required to master the task with minimal human guidance. With neuroevolution, this means training a single "monolithic" network from scratch to perform the entire task. 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 number of inputs, hidden units, outputs, and their connectivity) and neuroevolution 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 power of available training methods.

The tabula rasa approach can be implemented for keepaway by training a single, monolithic network for controlling a given keeper. ESP is used to train a fully connected two-layer feed-forward network with nine inputs, four hidden nodes ($u = 4$), and two outputs, as shown in Figure 3. This network structure was determined, through experimentation, to be the most effective.[2]

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 agent always knows the location of these objects even if they are behind the agent or far away. 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 network's two outputs control the agent's movement on the field: one alters its heading, the other its speed. All runs use subpopulations of size 100. The number of trials t is 10.

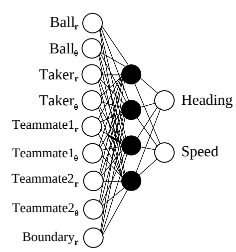

Figure 3. The monolithic network for controlling keepers. White circles indicate inputs and outputs while black circles indicate hidden nodes.

Incremental evolution was used to alter the taker's speed as follows. When evolution begins, the taker can move only 10% as quickly as the keepers. Each network is evaluated in 20 games of keepaway and its scores (numbers of completed passes) are summed to obtain its fitness. When the population's average fitness exceeds 40 (two completed passes per episode), the taker's speed is incremented by 5% of the keepers'

[2] In an attempt to improve performance, several different network architectures with differing numbers of hidden nodes and inputs were examined. For example, we were able to reduce the number of inputs by providing the difference between two angles rather than the angles themselves.

speed. This process continues until the taker is moving at the same speed as the keepers or the population's fitness has plateaued.

3.2. Task Decomposition

If learning a monolithic network proves infeasible, it may be possible to make the problem tractable by decomposing it into some number of components. In particular, if the task can be broken into independent subtasks, each subtask can be learned separately, and combined into a complete solution. Task decomposition is a powerful, general principle in artificial intelligence that has been used successfully with machine learning in tasks like the full robot soccer task (Stone, 2000).

3.2.1. *Keepaway with a Decision Tree*

The keepaway task can be decomposed by replacing the monolithic network with several smaller networks: one to pass the ball, another to receive passes, etc. A decision tree, shown in Figure 4, is used to implement this decomposition and controls each keeper. If the agent is within three player-lengths of the ball, it kicks 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.

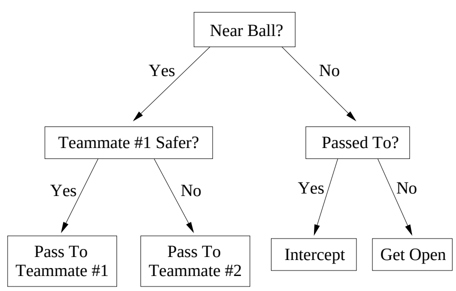

Figure 4. A decision tree for controlling keepers in the keepaway task. The behavior at each of the leaves is learned through neuroevolution. A network is also evolved to decide to which teammate the agent should pass. The "Near Ball" and "Passed To" nodes in this tree are hand-coded predicates, and are not evolved.

To implement this decision tree, four different networks must be trained. As in the monolithic approach, several different network architectures were tested, varying the number of inputs and hidden nodes. The networks shown in Figure 5 were determined, through experimentation, to be the most effective. The networks perform the following tasks:

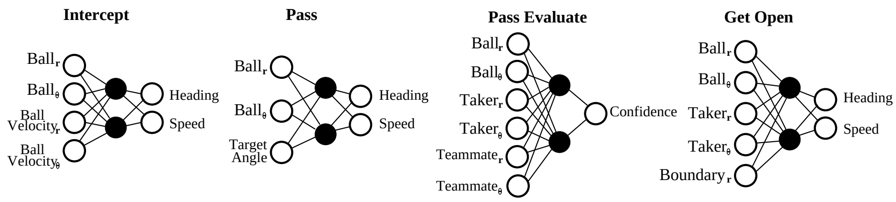

Figure 5. The four networks used to implement the decision tree shown in Figure 4. White circles indicate inputs and outputs while black circles indicate hidden nodes.

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 where the agent must decide to which teammate to pass. It analyzes the current state of the game and assesses the likelihood that the 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 in terms of future pass opportunities. 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.

Once these four networks are trained, they can be combined via a decision tree into a single strategy.

3.2.2. *Keepaway with a Switch Network*

How would the challenges of learning keepaway change if, as system designers, we lacked the time or expertise to create and fine-tune a decision tree like the one described above? To answer this question, we consider in our experiments a more difficult version of the keepaway task in which the hand-coded decision tree is not available. Instead, we add a fifth learned component which must determine when to employ each of the other four components.

Figure 6 shows the structure of the switch network that is used to control each keepaway player. Since this network must make high-level decisions about the state of the game, it naturally requires more inputs than the other components. At each time step, the switch network receives input about the position of the ball, both of its teammates, and the taker. It is also given its distance from the field's bounding circle and the output of two calls to the pass evaluate network, evaluating how likely a pass to each teammate is to succeed. The switch network's four outputs correspond to each of the low-level behaviors it can employ: passing to the first teammate, passing to the second teammate, intercepting, and getting open. Each agent's behavior is determined by the output with the highest activation.

Both of the task decompositions described above (one with a decision tree and one with a switch network) leave us with several components that need to be learned. The remainder of this section details three methods for learning these components. They differ dramatically in how much human knowledge they supply to the learning process.

3.2.3. *Coevolution*

To avoid injecting human knowledge into the learning process beyond that provided by the task decomposition, we can train all the components simultaneously in the target domain. This process is particularly straightforward when neuroevolution is used as the learning method because the various components can then be simply coevolved. Coevolution consists of simultaneously evolving multiple components that

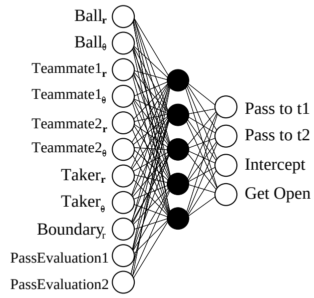

Figure 6. The switch network for controlling keepers. White circles indicate inputs and outputs while black circles indicate hidden nodes.

perform different roles but are evaluated in a common domain. Coevolution can be competitive, in which case these roles are adversarial and one component's gain is another's loss (Haynes and Sen, 1996; Rosin and Belew, 1995). Coevolution can also be cooperative, as when the various components share fitness scores (Potter and Jong, 2000). Multi-Agent ESP (Yong and Miikkulainen, 2001) is an extension of ESP that allows multiple components to coevolve cooperatively. In this system, each component is evolved with a separate, concurrent run of ESP. For each fitness evaluation, Multi-Agent ESP forms a network from each ESP and then evaluates these networks together in the task, all of which receive the same score when the evaluation completes. Multi-Agent ESP has been successfully used to master multi-agent predator-prey tasks (Yong and Miikkulainen, 2001), and is used as the coevolution framework for the keepaway task in this paper as well.

Given our decision tree and the task decomposition described above, applying coevolution to keepaway is straightforward. Evolution consists of four simultaneous runs of ESP, one for each of the networks we need to train. Each run uses subpopulations of 100 neurons. In each fitness evaluation, one network is selected from each population (i.e. one passing network, one intercepting network, etc.) and assembled to form a keeper. As the keepaway game proceeds, the decision tree determines how the networks are employed to control each keeper. The resulting score is passed back to all four networks. Incremental evolution controls the speed of the taker, just as in the tabula rasa approach. In the absence of a decision tree, we can simply add a fifth concurrent run of ESP which trains switch networks too.

Unlike layered learning (described below), the coevolutionary approach to learning keepaway makes no attempt to develop specific training tasks for each component. Instead, components are evaluated only in the target domain (actual games of keepaway) and only as part of a complete system. Hence, this method gives evolution a lot of flexibility to discover the most useful behaviors. Furthermore, since the components are always evaluated together, they can easily adapt to each other and coordinate effectively.

3.2.4. *Layered Learning*

Coevolution provides a simple and flexible framework for learning several subtasks. However, it offers the learning algorithm little assistance beyond that of the decomposition itself. On more difficult tasks, it may be useful or even necessary to learn components in a more structured, sequential fashion. Layered learning is a bottom-up, hierarchical paradigm for doing just that.

The main principles of the layered learning paradigm (summarized in Table I) are:[3]

Table I. The key principles of layered learning.

1. A mapping directly from inputs to outputs is not tractably learnable.
2. A bottom-up, hierarchical task decomposition is given.
3. Machine learning exploits data to train and/or adapt. Learning occurs separately at each level.
4. The output of learning in one layer feeds into the next layer.

Tractability: Layered learning is designed for domains that are too complex for learning a mapping directly from the input to the output representation. Instead, the problem is broken down into several task layers. At each layer, a concept needs to be acquired, and any appropriate machine learning (ML) algorithm can be used for this purpose.

Decomposition: Layered learning uses a bottom-up incremental approach to hierarchical task decomposition. Starting with low-level subtasks, the process of creating new ML subtasks continues until the high-level tasks, that deal with the full domain complexity, are reached. The appropriate learning granularity and subtasks to be learned are determined as a function of the specific domain. The task decomposition in layered learning is not automated. Instead, the layers are defined by the ML opportunities in the domain.

Learning: Machine learning is used as a central part of layered learning to exploit data in order to *train* and/or *adapt* the overall

[3] This section is adapted from (Stone and Veloso, 2000).

system. ML is useful for training functions that are difficult to fine-tune manually. It is useful for adaptation when the task details are not completely known in advance or when they may change dynamically. Like the task decomposition itself, the choice of machine learning method depends on the subtask.

Interactions: The key defining characteristic of layered learning is that each learned layer directly affects the learning at the next layer. A learned subtask can affect the subsequent layer by:

- constructing the set of training examples;
- providing the features used for learning; and/or
- pruning the output set.

Layered learning can be formally defined as follows. Consider the learning task of identifying a hypothesis h from among a class of hypotheses H which map a set of state feature variables S to a set of outputs O such that, based on a set of training examples, h is most likely (of the hypotheses in H) to represent unseen examples.

When using the layered learning paradigm, the complete learning task is decomposed into hierarchical subtask layers $\{L_1, L_2, \ldots, L_n\}$ with each layer defined as

$$L_i = (\vec{F_i}, O_i, T_i, M_i, h_i)$$

where:

$\vec{\boldsymbol{F_i}}$ is the input vector of state features relevant for learning subtask L_i. $\vec{F_i} = <F_i^1, F_i^2, \ldots>$. $\forall j,\ F_1^j \in S$.

$\boldsymbol{O_i}$ is the set of outputs from among which to choose for subtask L_i. $O_n = O$.

$\boldsymbol{T_i}$ is the set of training examples used for learning subtask L_i. Each element of T_i consists of a correspondence between an input feature vector $\vec{f} \in \vec{F_i}$ and $o \in O_i$.

$\boldsymbol{M_i}$ is the ML algorithm used at layer L_i to select a hypothesis mapping $\vec{F_i} \mapsto O_i$ based on T_i.

$\boldsymbol{h_i}$ is the result of running M_i on T_i. h_i is a function from $\vec{F_i}$ to O_i.

Note that a layer describes more than a task; it also describes an approach to solving that task and the resulting solution.

As stated in the Decomposition principle of layered learning, the definitions of the layers L_i are given *a priori*. The Interaction principle is addressed via the following stipulation. $\forall i < n$, h_i directly affects L_{i+1} in at least one of three ways:

- h_i is used to construct one or more features F_{i+1}^k.
- h_i is used to construct elements of T_{i+1}; and/or
- h_i is used to prune the output set O_{i+1}.

It is noted above in the definition of $\vec{F_i}$ that $\forall j,\ F_1^j \in S$. Since $\vec{F_{i+1}}$ can consist of new features constructed using h_i, the more general version of the above special case is that $\forall i, j,\ F_i^j \in S \cup_{k=1}^{i-1} O_k$.

When training a particular component, layered learning freezes the components trained in previous layers, thereby adding additional constraints to the learning process. It also adds guidance, by training each layer in a special environment intended to prepare it well for the target domain.

To apply layered learning to keepaway, we must decide in what sequence to learn the components and develop special training environments for each one. Figure 7 shows one way of arranging the layers. An arrow from one layer to another indicates that the latter layer depends on the former. Since a layer cannot be learned until all the layers on which it depends have been learned, the learning process starts at the bottom, with intercept, and moves up the hierarchy step by step. Below is a description of each layer using layered learning's formal notation, including a description of the special training environment of each layer.

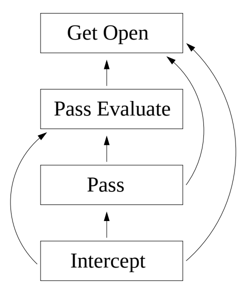

Figure 7. A layered learning hierarchy for the keepaway task. Each box represents a layer and arrows indicate dependencies between layers. A layer cannot be learned until all the layers it depends on have been learned.

$\mathbf{L_1}$: **Intercept** :

$\vec{\mathbf{F_1}} = \{\mathbf{Ball_r}, \mathbf{Ball_\theta}, \mathbf{BallVelocity_r}, \mathbf{BallVelocity_\theta}\} \in \Re^\mathbf{4}$

$\mathbf{O_1} = \{\mathbf{Heading}, \mathbf{Speed}\} \in \Re^\mathbf{2}$

$\mathbf{T_1}$: To train the interceptor, the ball is propelled towards the agent at various angles and speeds. The agent is rewarded for minimizing the time it takes to touch the ball.

$\mathbf{M_1}$ = **neuroevolution:** ESP (with a subpopulation size of 100) is used to train a network with 4 inputs, 2 hidden nodes, and 2 outputs (see Figure 5).

$\mathbf{h_1}$ = a trained interceptor.

$\mathbf{L_2}$: **Pass** :

$\vec{\mathbf{F_2}} = \{\mathbf{Ball_r}, \mathbf{Ball_\theta}, \mathbf{TargetAngle}\} \in \Re^{\mathbf{3}}$

$\mathbf{O_2} = \{\mathbf{Heading}, \mathbf{Speed}\} \in \Re^{\mathbf{2}}$

$\mathbf{T_2}$: To train the passer the ball is again propelled towards the agent. The angle at which the agent should kick the ball is randomly chosen. When the simulation begins, the agent employs the intercept behavior learned in L_1 until it arrives near the ball, at which point it switches to the evolving pass behavior. The agent's reward is inversely proportional to the difference between the target angle and the ball's actual direction of travel.

$\mathbf{M_2}$ = **neuroevolution:** ESP (with a subpopulation size of 100) is used to train a network with 3 inputs, 2 hidden nodes, and 2 outputs (see Figure 5).

$\mathbf{h_2}$ = a trained passer.

$\mathbf{L_3}$: **Pass Evaluate** :

$\vec{\mathbf{F_3}} = \{\mathbf{Ball_r}, \mathbf{Ball_\theta}, \mathbf{Taker_r}, \mathbf{Taker_\theta}, \mathbf{Teammate_r}, \mathbf{Teammate_\theta}\} \in \Re^{\mathbf{6}}$

$\mathbf{O_3} = \{\mathbf{Confidence}\} \in \Re$

$\mathbf{T_3}$: The ball is placed in the center of the field and the pass evaluator is stationed just behind it at various angles. Two teammates are placed 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 intercept behavior from L_1. When training the pass evaluator, the evolving network is run twice, once for each teammate. The pass evaluator then passes, using L_2, to the teammate who received a higher evaluation. If the pass succeeds, the evaluator is rewarded. Each network is evaluated fifty times and rewarded with the sum of the scores.

$\mathbf{M_3}$ = **neuroevolution:** ESP (with a subpopulation size of 100) is used to train a network with 6 inputs, 2 hidden nodes, and 1 output (see Figure 5).

$\mathbf{h_3}$ = a trained pass evaluator.

$\mathbf{L_4}$: **Get Open** :

$\vec{\mathbf{F_4}} = \{\mathbf{Ball_r}, \mathbf{Ball_\theta}, \mathbf{Taker_r}, \mathbf{Taker_\theta}, \mathbf{Boundary_r}\} \in \Re^{\mathbf{5}}$

$\mathbf{O_4} = \{\mathbf{Heading}, \mathbf{Speed}\} \in \Re^{\mathbf{2}}$

$\mathbf{T_4}$: The training environment for the get open behavior is an actual game of keepaway, described above. The taker uses the intercept behavior evolved in L_1 and the keepers use the decision tree described in Figure 4 along with the evolved behaviors from L_1, L_2, and L_3. Each network is evaluated in 20 games of keepaway and rewarded with the sum of the scores.

$\mathbf{M_4}$ = **neuroevolution:** ESP (with a subpopulation size of 100) is used to train a network with five inputs, two hidden nodes, and two outputs (see Figure 5).

$\mathbf{h_4}$ = a trained get open behavior.

Once these four layers have been learned, they can be combined with the decision tree to form a complete keepaway player. Note that the L_1 intercept behavior trained in this scenario is used by the taker for all of our experiments.

If a decision tree is not available and a switch network must be learned instead, we can add a fifth layer to the hierarchy, as shown in figure Figure 8. Since it would be infeasible to train a switch network in the absence of the low-level behaviors it controls, it appears at the top of the hierarchy. This decision requires us to change T_4, the training environment for get open. The original T_4 trained get open networks in actual games of keepaway, with each agent controlled by the decision tree. Since the decision tree is not available in this scenario and the switch network is not learned until L_5, we must construct a new training environment T_4' which requires neither a hand-coded nor a learned high-level strategy. Below are the details of the new get open layer including a description of this new training environment.

$\mathbf{L_4'}$: **Get Open** :

$\vec{\mathbf{F_4}}' = \{\mathbf{Ball_r}, \mathbf{Ball_\theta}, \mathbf{Taker_r}, \mathbf{Taker_\theta}, \mathbf{Boundary_r}\} \in \Re^{\mathbf{5}}$

$\mathbf{O_4'} = \{\mathbf{Heading}, \mathbf{Speed}\} \in \Re^{\mathbf{2}}$

$\mathbf{T_4'}$: To evolve a get open behavior, two keepers are placed on the field along with a taker. One keeper begins near the ball and uses the passing behavior learned in L_2 to try to kick the ball past the taker to the other keeper, which is controlled by the

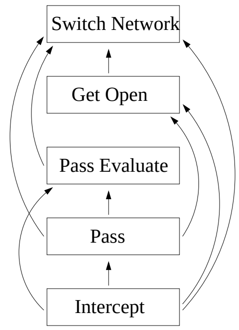

Figure 8. A layered learning hierarchy for the more difficult version of keepaway in which a hand-coded decision tree is not available. Each box represents a layer and arrows indicate dependencies between layers. Note that there is no arrow from pass evaluate to get open because T_4', the new training environment for get open, does not use a pass evaluate network.

evolving get open behavior. This keeper's reward is based on how well it moves to positions that maximize the likelihood of a successful pass.

$\mathbf{M_4'}$ = **neuroevolution:** Using ESP, we train a fully connected two-layer feed-forward neural network with 5 inputs, 2 hidden nodes, and 2 outputs.

$\mathbf{h_4'}$ = a trained get open behavior.

Once the four low-level components are completed, a switch network can be trained in an actual game of keepaway. The details of L_5 are below.

$\mathbf{L_5}$: **Switch Network** :

$\vec{\mathbf{F_5}} = \{\mathbf{Ball_r}, \mathbf{Ball_\theta}, \mathbf{Teammate1_r}, \mathbf{Teammate1_\theta}, \mathbf{Teammate2_r}, \mathbf{Teammate2_\theta}, \mathbf{Taker_r}, \mathbf{Taker_\theta}, \mathbf{Boundary_r}, \mathbf{PassEvaluation1}, \mathbf{PassEvaluation2}\} \in \Re^{\mathbf{11}}$

$\mathbf{O_5} = \{\mathbf{Pass1}, \mathbf{Pass2}, \mathbf{Intercept}, \mathbf{GetOpen}\} \in \Re^{\mathbf{4}}$

$\mathbf{T_5}$: The training environment for the switch network is an actual game of keepaway, described above. Each keeper uses the switch network to determine which of the previously evolved behaviors h_1, h_2, and h_4 it should use. The pass evaluator h_3 is run twice (once for each teammate) and the results are presented as input to the switch network. Each network is evaluated in 20 games of keepaway and rewarded with the sum of the scores.

$\mathbf{M_5}$ = **neuroevolution:** Using ESP, we train a fully connected two-layer feed-forward neural network with 11 inputs, two hidden nodes, and four outputs (see Figure 6).

$\mathbf{h_5}$ = a trained switch network.

3.2.5. *Concurrent Layered Learning*

Coevolution provides no human assistance beyond the task decomposition, thereby minimally restricting the search space. In contrast, layered learning provides a good deal of assistance, constraining and guiding the learners' search space much more. Concurrent layered learning is an approach which occupies an intermediate range on this spectrum. It retains the guidance of special training environments for lower layers but, by not always freezing lower layers, adds some of the flexibility of coevolution.

One difficulty with traditional layered learning is that, no matter how carefully the special training environments for the lower layers are designed, there are bound to be imperfections. Discrepancies will inevitably exist between the behaviors that those environments encourage and the behaviors that are optimal in the target domain. Concurrent layered learning tries to correct for those discrepancies by allowing certain lower layers to continue to adapt while higher layers are being trained.

Layered learning was originally developed for the complex, multi-agent learning task of simulated robot soccer (Noda et al., 1998; Stone, 2000). In the original implementation, the learning of each component was completed before any subsequent layer was trained. Concurrent layered learning relaxes this restriction. When learning a layer L_i, we select from all the previously learned hypotheses some subset $P \subseteq \{h_1, h_2, ..., h_{i-1}\}$ that we want to continue to train in the current environment T_i. The effect that such hypotheses have on T_i is no longer fixed throughout the learning of L_i, but instead changes constantly as those hypotheses continue to learn.

For each $h_k \in P$, the best network is taken from L_k and used to seed a new population before training in T_i begins. These new populations continue to learn along with a separate population learning L_i. Hence, the layers are evolved cooperatively using Multi-agent ESP. To perform a fitness evaluation, a network is taken from each population that was seeded with h_k and evaluated in T_i, together with a network selected from the population that is learning L_i from scratch. The resulting score is shared by all the networks that participate.

To seed a population from the results of L_k, delta-coding is used as described in Section 2.2. Since delta-coding is particularly well suited to helping populations adjust to sudden changes in their training environment (Gomez and Miikkulainen, 1997), it is an excellent way to seed a new population from the results of an earlier layer.

Concurrent layered learning supplements layered learning by applying coevolution in a restricted form. Hence, it represents a middle ground between those two methods. It preserves layered learning's hierarchical structure but also offers some of coevolution's flexibility, as layers are continually allowed to adapt to each other.

Concurrent layered learning provides a general framework for combining layered learning and coevolution but does not specify which layers should remain unfrozen. Hence, to apply concurrent layered learning to keepaway, we must decide, at each layer, which lower layers to leave frozen and which to allow to continue to adapt. In this paper, rather than search this space of possible choices, we consider one simple way that concurrent layered learning can be applied to keepaway.

In this implementation, each of the lower layers L_1, L_2, and L_3 are trained exactly as in the traditional layered learning approach described above. Any previously learned components are fixed in these training environments. However, when beginning to train L_4, all the hypotheses h_1, h_2, and h_3 that have already been learned are unfrozen. Hence, each of the lower components learn initially in their own special training environment but then are given the opportunity to fine-tune their behavior in T_4, which is our target domain of keepaway.

If a switch network must be learned instead of relying on a decision tree, we maintain the simple approach of freezing all lower layers until the top of the hierarchy is reached. Hence, layers L_1, L_2, L_3, and L'_4 are trained exactly as in the traditional layered learning approach. When training of L_5 begins, all four lower components are unfrozen and fine-tuned concurrently with L_5, which learns from scratch.

4. Empirical Results

In a series of experiments, the tabula rasa, coevolution, layered learning, and concurrent layered learning approaches were compared in the keepaway task.[4] Each method was evaluated in seven trials, consisting of 250 generations each. Recall that each subpopulation contains 100 neurons, each of which is evaluated an average of 10 times per generation. Since each evaluation consists of 20 games of keepaway, each generation requires simulating 20,000 games.

In layered and concurrent layered learning, additional generations were used to train the lower layers. Specifically, the intercept, pass, and pass evaluate layers trained for 40, 100, and 60 generations, respectively. In the switch network version of the task, the get open layer was trained

[4] Video of these results is available at:
`http://nn.cs.utexas.edu/keyword?keepaway`

for 125 generations in its special training environment. The additional computational cost of using layered or concurrent layered learning is not as great as it may seem since the fitness evaluations for most of the lower layers run in a fraction of the time required for the target domain. Furthermore, the most important characteristics of the learning process are the quality of the keepaway players that result from these methods and the human effort required to implement them, rather than the speed at which they are discovered.

Figure 9 shows what task difficulty (i.e. taker speed) each method reached during the course of evolution, averaged over the seven runs. Recall that the taker starts at 10% of the keepers' speed and accelerates by 5% of the keeper's speed each time the keepers achieve a target performance level. The tabula rasa method fails to make any task transitions, whereas even the weakest task decomposition methods are able to make several. With a hand-coded decision tree, the less assisted methods (i.e. coevolution and concurrent layered learning) do remarkably better than the most assisted one, i.e. layered learning. When required to learn a switch network, however, the relative performance of these methods changes. Coevolution becomes one of the worst methods, and the more assisted methods based on layered learning do much better. Concurrent layered learning, which retains some of coevolution's flexibility, performs the best. For the purposes of comparison, a hand-coded player is also evaluated. [5] For the purpose of comparison, the hand-coded player was evaluated in the same incremental manner as the learned players, and was able to advance to a task difficulty of 50%. Note that all of the methods were able to outperform the hand-coded solution when given a decision tree, and concurrent layered learning was able to outperform the hand-coded solution even when required to learn a switch network.

How do the networks trained in these experiments fare in the target domain of the complete keepaway task? To determine this for the methods that used a hand-coded decision tree, the evolving networks from each method were tested against a taker moving at 100% speed. At every fifth generation, the strongest network from the best run of each method was subjected 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 10, which shows the results of these tests, offers dramatic confirmation of the effectiveness of the less assisted learning methods.

[5] The hand-coded player was given a strategy of running directly to the ball and passing to whichever teammate was less blocked by the taker. When not involved in a pass, each hand-coded player would attempt to maximize the distance between itself and all other players, without going out of bounds.

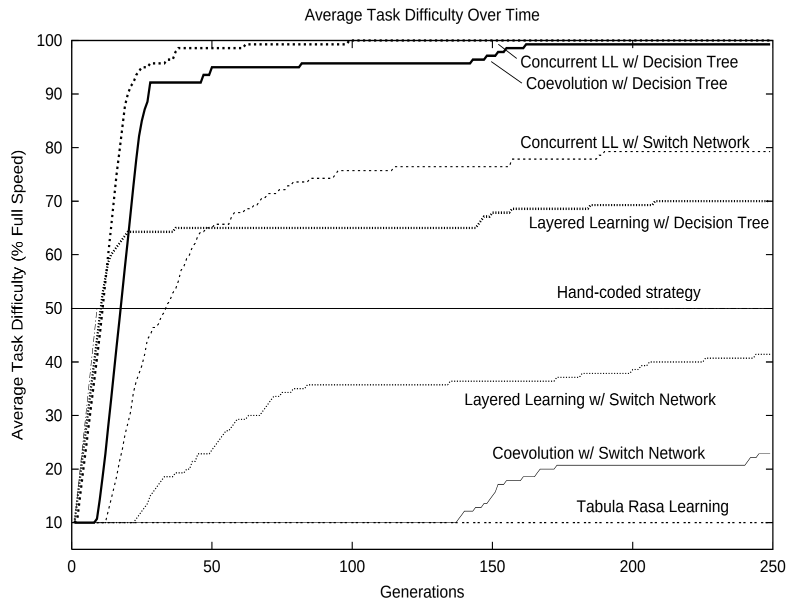

Figure 9. Task difficulty (i.e. taker speed) of each method over generations, averaged over seven runs. Task decomposition proves essential for reaching the higher difficulties. Only coevolution with a decision tree and concurrent layered learning with a decision tree reach the hardest task.

When given a decision tree, coevolution and concurrent layered learning perform much better than both the other learning methods as well as the hand-coded strategy.

Next, we examined how the different learning methods performed in the keepaway task when they were required to learn a switch network instead of using a hand-coded decision tree. Since learning without the aid of a decision tree is more difficult, all the methods tested in this manner performed at a lower level. Therefore, we tested them against a taker moving at 50% speed. While this introduces some minor discrepancies between training and testing (e.g. T_3 uses a taker moving at full speed), it better eludicates the differences between the various methods, since none of them performed well against a full speed taker in the absence of a decision tree. All other parameters are the same as those used to test the decision tree methods.

Figure 11 highlights how much more difficult it is to learn both high-level and low-level behaviors at the same time. Without the aid of a hand-coded decision tree, most of the learning methods were not able to match the performance of the hand-coded strategy. One exception is concurrent layered learning, which does substantially better than the hand-coded approach. Methods receiving less human assistance, like

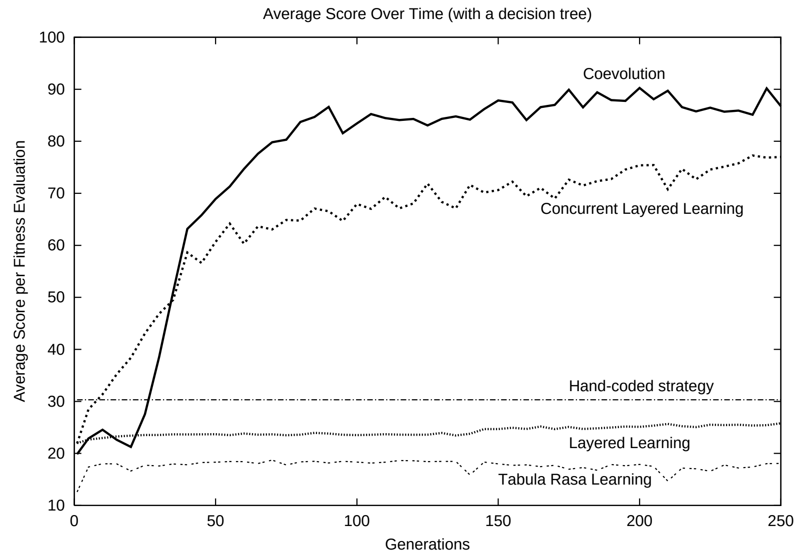

Figure 10. Average score per fitness evaluation for each method over generations when a decision tree is supplied. These results demonstrate that task decomposition is important in this domain and that less assisted methods of learning are more effective at learning the resulting subtasks.

coevolution, which did well when given a decision tree, now perform rather poorly. Hence, the value of human assistance increases as the task becomes more difficult.

5. Discussion

The success of several of the methods described above, particularly that of coevolution and concurrent layered learning with a decision tree, indicates that neuroevolution can master a complex control task like keepaway and offer a striking improvement over the hand-coded approach. The extremely poor performance of the tabula rasa method confirms that providing a suitable task decomposition is essential to this success.

The results suggest that it is important to strike a balance between the flexibility of methods like coevolution and the guidance and constraints of more structured methods like layered learning. Where that balance lies depends on the difficulty of the problem.

Providing the learner with a hand-coded decision-tree enables the learner to master the task without additional human assistance, as the success of coevolution with a decision tree indicates. The additional constraints and guidance offered by layered learning are superfluous

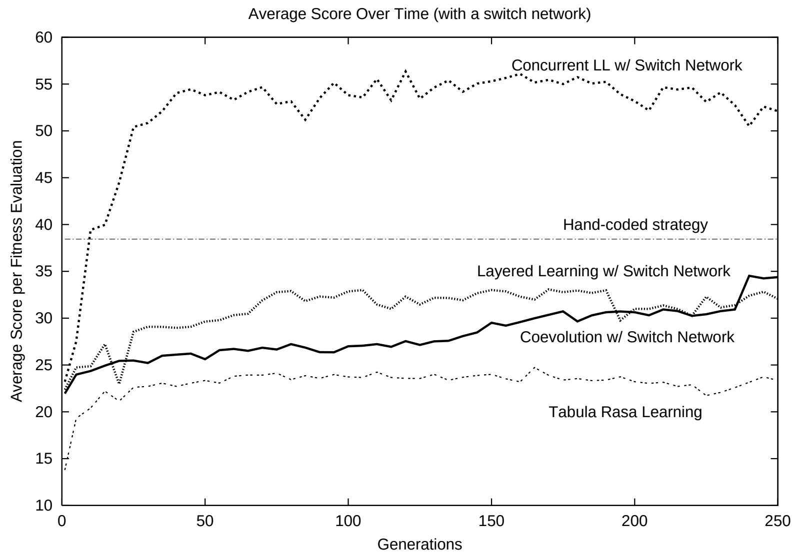

Figure 11. Average score per fitness evaluation for each method over generations when a switch network is also learned. These results show that in this harder version of the keepaway task, the best solution is that which strikes a balance between human assistance and flexibility.

and even detrimental in this scenario because they prevent certain high performing solutions from being learned. For example, one component where the flexibility of coevolution proves especially useful is the intercept task. Instead of learning to move to the ball as quickly as possible, the coevolved interceptor learns to approach the ball from an angle. While this behavior does not minimize the amount of time required to intercept the ball, it does put the keeper in a better position to complete the next pass, thus improving overall score. Since the interceptor trained by layered learning is rewarded only by how quickly it can get to the ball, it is unable to learn this strategy.

Concurrent layered learning, which imposes layered learning's structure but also preserves some of coevolution's flexibility, does nearly as well as coevolution when given a decision tree. However, coevolution is much easier to implement since it does not require the designer to develop special training environments for each component. Hence it is the preferable method in this case.

If the task is made more challenging by removing the decision tree and asking the methods to learn a high-level strategy also, the balance between flexibility and human assistance shifts. As Figure 11 indicates, an unassisted method like coevolution does very poorly when asked to learn a switch network along with the other four components. Without the aid of the decision tree, this method has little to guide it through a very large search space and does not fare much better than the tabula rasa method. In this more difficult version of the task, the structure offered by layered learning pays substantial dividends. However, even in this scenario we can still gain some benefit from coevolution's flexibility if we apply it more sparingly. The superior performance of concurrent layered learning to traditional layered learning when using a switch network confirms this conclusion.

The failure of the tabula rasa method and of coevolution with a switch network indicates that the learning methods are not yet sophisticated enough to simultaneously learn both the high and low level parts of the task without some human assistance. However, the assistance we offer need not be large if we choose what form it takes wisely. In fact, the best performing method overall, coevolution with a decision tree, is also one of the easier approaches to implement. It requires us to hand-code a high-level strategy but, once we do so, the lower components learn with remarkably little human help.

If the time or expertise necessary to hand-code a decision tree is not available, the task can still be mastered if we provide the learner with the additional structure of layered learning. Even in this scenario, constraints should be applied to the learner sparingly. The success of concurrent layered learning with a switch network highlights the importance of retaining some of coevolution's flexibility even when the task is made more difficult.

6. Related Work

In this section, the references made throughout the text are supplemented with more detailed comparisons to previous research along three dimensions: layered learning, task decomposition, and keepaway.

6.1. Layered learning

The original implementation of the layered learning paradigm was on the full robot soccer task in the RoboCup soccer simulator (Stone, 2000). First, a neural network was used to learn an interception behavior. This behavior was used to train a decision tree for pass evaluation, which was in turn used to generate the input representation for a reinforcement learning approach to pass selection. Lower-level behaviors were always trained and then frozen before advancing to the next layer. That is, once a subtask was learned, it was not allowed to change while subsequent subtasks were learned.

A subsequent application of layered learning uses two learned layers, each learned via genetic programming, for a keepaway task in a simplified abstraction of the TeamBots environment (Hsu and Gustafson, 2002a). This implementation uses the traditional layered learning approach of freezing the first layer (passing) before advancing to the next layer (the whole task).

Concurrent layered learning, our enhancement of the traditional approach, is consistent with the existing layered learning formalism. In a preliminary version of this article (Whiteson and Stone, 2003), we first demonstrated that concurrent layered learning can outperform traditional layered learning by allowing just two of the higher layers to be learned concurrently for SoccerBots keepaway agents. Here we add subsequent evidence to that effect in a slightly modified scenario such that four layers are learned concurrently.

Some other previous work, not explicitly following the layered learning paradigm, is nonetheless related in its methodology and motivation. *Bootstrap learning* (Kuipers and Beeson, 2002) is used to enable a mobile robot to recognize places. It uses clustering techniques to learn "distinctive states" in the environment. This clustering then feeds into a causal/topological map based on history that is used to disambiguate distinctive states. Finally, the topological map feeds into a layer that uses labeled images to learn a mapping from sensory images to distinctive states. Like traditional layered learning, this method freezes the lower layers before moving on to the higher layers of learning.

We found that an appropriate level of flexibility in layered learning is essential for progress. Less constrained types of learning, like concurrent layered learning, can offer significant benefits over standard layered learning.

6.2. Task Decomposition

As illustrated by its initial implementation which made use of neural networks, decision trees, and a reinforcement learning algorithm, layered learning makes no commitment to any particular learning algorithm, and indeed can combine several different algorithms across the different layers. There have also been some hierarchical approaches proposed that are specific to individual learning algorithms, most notably coevolution, as summarized in Section 3.2.3, and hierarchical reinforcement learning.

Most hierarchical RL approaches use *gated* behaviors (Kaelbling et al., 1996):

> There is a collection of behaviors that map environment states into low-level actions and a gating function that decides, based on

> the state of the environment, which behavior's actions should be switched through and actually executed. (Kaelbling et al., 1996)

In some cases the behaviors are learned (Mahadevan and Connell, 1991), in some cases the gating function is learned (Maes and Brooks, 1990), and in some cases both are learned (Lin, 1993). In this last example, the behaviors are learned and fixed prior to learning the gating function. On the other hand, feudal Q-learning (Dayan and Hinton, 1993) and the MAXQ algorithm (Dietterich, 1998) learn at all levels of the hierarchy simultaneously. In all of these approaches, the behaviors and the gating function are all control tasks with similar inputs and actions (sometimes abstracted). Layered learning, both traditional and concurrent, allows for conceptually different tasks, such as pass evaluation and get open, at the different layers.

In another algorithm-specific technique, *Many-layered learning* (Utgoff and Stracuzzi, 2002) learns from an input stream how to choose the layers in a feed-forward neural network. Once a concept is learned, it is used as input to things that are still unlearned.

There are, of course, many other examples of the successful use of hierarchy in the literature, especially when one broadens one's focus beyond learning approaches. As just one example, the field of robotics has seen many hierarchical approaches, including the subsumption architecture (Brooks, 1986) and three-layered architectures (Gat, 1998). Like the task decompositions for learning discussed in this article, these approaches rely on a manual task decomposition. Some of the lessons presented in this paper regarding the tradeoffs between imposing more or less structure on the task may be applicable to that field as well.

6.3. Keepaway

Robot soccer keepaway has been used as a testbed domain for several previous machine learning studies (including one described above (Hsu and Gustafson, 2002b)). A variant based on the the RoboCup soccer simulator was introduced for the purposes of studying multi-agent reinforcement learning (Stone and Sutton, 2001). In this research, the low-level behaviors were hand-coded; only the high-level decision of when and where to pass was learned. An evolutionary learning approach has been successfully used for the same task, but again with only a single learned layer (Pietro et al., 2002).

Previous work has also explored the use of keepaway techniques for full soccer (Stone and McAllester, 2001). The ATT-CMUnited-2000 team successfully incorporated a solution to the 11 vs. 11 keepaway problem on a full-sized field to control the behavior of the player in possession of the ball. The motivation behind this work was to facilitate learning the policies in a principled way. However all of the reported work used hand-coded policies.

The keepaway domain used for the research described in this paper incorporates the aspects of multi-agent strategy described above with learning on all levels of a task decomposition.

7. Future Work

In ongoing research, we are exploring different ways of implementing concurrent layered learning. By doing so, we aim at discovering a systematic method for deciding, at each layer, which subset of previously learned hypotheses should be allowed to continue training.

Applying the lessons learned from this research to other domains is an interesting avenue for future research. One possibility would be to test coevolution and concurrent layered learning in an extremely complex domain, such as the full robot soccer task. It would be interesting to observe how the ideal amount of human assistance changes when the difficulty of the task increases drastically.

Other multiagent gaming domains, such as the Legion-I domain (Bryant and Miikkulainen, 2003) have many similarities with robot soccer keepaway and might provide a worthwhile testbed for the methods described in this paper. Similarly, the domain of automated driving (Pyeatt and Howe, 1998) could also be used to test the generality of the methods. Both of these domains feature tasks that are easily decomposable: an effective city-guarding strategy for a legion might involve finding the nearest cities, trading places with other legions, and patrolling for barbarians; a decomposition for automated driving might include accelerating into traffic, following a lane, and avoiding obstacles. Given these task decompositions, an analysis similar to that presented above could be performed to determine the ideal level of human assistance. Such a study would allow us to both examine the effects of injecting human knowledge in different tasks and to test how well coevolution and concurrent layered learning perform in different domains.

8. Conclusion

The main contributions of this paper are 1) verification that, given a suitable task decomposition, neuroevolution can master a complex, multi-agent control task at which it otherwise fails, 2) empirical evidence that, when training the components that result from such a task decomposition, the correct level of human assistance to apply to the learning method depends critically on the difficulty of the task, and 3) introduction of a novel method, concurrent layered learning, which, on difficult tasks, offers a more effective balance between constraint and flexibility.

Methods for learning decomposed components hold enormous promise for mastering challenging learning problems. By injecting human knowledge into the process, these techniques can leverage current learning algorithms to tackle more difficult classes of problems. The manual effort involved need not be burdensome. In fact, this research demonstrates that the best results are obtained when the constraints and guidance supplied by human designers are applied sparingly.

Acknowledgments

We would like to thank the anonymous reviewers of this paper for their valuable feedback. This work is supported in part by NSF grants IIS-0083776 and EIA-0303609, THECB ARP-003658-476-2001, and NSF CAREER award IIS-0237699.

References

Balch, T.: 2000, 'TeamBots Domain: SoccerBots'. `http://www-2.cs.cmu.edu/~trb/TeamBots/Domains/SoccerBots`.

Brooks, R. A.: 1986, 'A Robust Layered Control System for a Mobile Robot'. *IEEE Journal of Robotics and Automation* **RA-2**, 14–23.

Bryant, B. D. and R. Miikkulainen: 2003, 'Neuroevolution for Adaptive Teams'. In: *Proceedings of the 2003 Congress on Evolutionary Computation*, Vol. 3. pp. 2194–2201.

Dayan, P. and G. E. Hinton: 1993, 'Feudal reinforcement learning'. In: S. J. Hanson, J. D. Cowan, and C. L. Giles (eds.): *Advances in Neural Information Processing Systems 5*. San Mateo, CA: Morgan Kaufmann, pp. 271–278.

Dietterich, T. G.: 1998, 'The MAXQ method for hierarchical reinforcement learning'. In: *International Conference on Machine Learning*. pp. 118–126, Morgan Kaufmann.

Ficici, S. G. and J. B. Pollack: 1998, 'Challenges in Coevolutionary Learning: Arms-Race Dynamics, Open-Endedness, and Mediocre Stable States'. In: Adami, Belew, Kitano, and Talor (eds.): *Proceedings of the Sixth International Conference on Artificial Life*. Cambridge, Massachusetts, USA, pp. 238–247, MIT Press.

Gat, E.: 1998, 'Three-Layer Architectures'. In: D. Kortenkamp, R. P. Bonasso, and R. Murphy (eds.): *Artificial Intelligence and Mobile Robots*. Menlo Park, CA: AAAI Press, pp. 195–210.

Gomez, F. and R. Miikkulainen: 1997, 'Incremental Evolution of Complex General Behavior'. *Adaptive Behavior* **5**, 317–342.

Gomez, F. and R. Miikkulainen: 1999, 'Solving Non-Markovian Control Tasks with Neuroevolution'. In: *Proceedings of the International Joint Conference on Artificial Intelligence*. Denver, CO, pp. 1356–1361, Kaufmann.

Gomez, F. and R. Miikkulainen: 2001, 'Learning Robust Nonlinear Control with Neuroevolution'. Technical Report AI01-292, The University of Texas at Austin Department of Computer Sciences.

Gomez, F. J.: 2003, 'Robust Non-Linear Control through Neuroevolution'. Ph.D. thesis, University of Texas at Austin. Technical Report AI-TR-03-303.

Gomez, F. J. and R. Miikkulainen: 2003, 'Active Guidance for a Finless Rocket Using Neuroevolution'. In: E. Cantu-Paz, J. A. Foster, K. Deb, L. D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, K. Dasgupta, M. A. Potter, A. C. Schultz, K. A. Dowsland, and N. J. J. Miller (eds.): *Genetic and Evolutionary Computation - GECCO 2003*. Chicago, pp. 2084–2095, Springer Verlag.

Gruau, F., D. Whitley, and L. Pyeatt: 1996, 'A Comparison Between Cellular Encoding and Direct Encoding for Genetic Neural Networks'. In: J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.): *Genetic Programming 1996: Proceedings of the First Annual Conference*. pp. 81–89, MIT Press.

Haynes, T. and S. Sen: 1996, 'Evolving Behavioral Strategies in Predators and Prey'. In: G. Weiß and S. Sen (eds.): *Adaptation and Learning in Multiagent Systems*. Berlin: Springer Verlag, pp. 113–126.

Hsu, W. H. and S. M. Gustafson: 2002a, 'Genetic Programming and Multi-Agent Layered Learning by Reinforcements'. In: *Genetic and Evolutionary Computation Conference*. New York,NY.

Hsu, W. H. and S. M. Gustafson: 2002b, 'Genetic Programming and Multi-Agent Layered Learning by Reinforcements'. In: *Genetic and Evolutionary Computation Conference*. New York,NY, pp. 764–771.

Kaelbling, L. P., M. L. Littman, and A. W. Moore: 1996, 'Reinforcement Learning: A Survey'. *Journal of Artificial Intelligence Research* **4**, 237–285.

Kuipers, B. and P. Beeson: 2002, 'Bootstrap Learning for Place Recognition'. In: *Proceedings of the Eighteenth National Conference on Artificial Intelligence*.

Lin, L.-J.: 1993, 'Reinforcement Learning for Robots Using Neural Networks'. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Maes, P. and R. A. Brooks: 1990, 'Learning to coordinate behaviors'. In: *Proceedings of the Eighth National Conference on Artificial Intelligence*. pp. 796–802, Morgan Kaufmann.

Mahadevan, S. and J. Connell: 1991, 'Scaling reinforcement learning to robotics by exploiting the subsumption architecture'. In: *Proceedings of the Eighth International Workshop on Machine Learning*. pp. 328–332.

Moriarty, D. E. and R. Miikkulainen: 1996, 'Efficient Reinforcement Learning Through Symbiotic Evolution'. *Machine Learning* **22**, 11–32.

Noda, I., H. Matsubara, K. Hiraki, and I. Frank: 1998, 'Soccer Server: A Tool for Research on Multiagent Systems'. *Applied Artificial Intelligence* **12**, 233–250.

Pietro, A. D., L. While, and L. Barone: 2002, 'Learning In RoboCup Keepaway Using Evolutionary Algorithms'. In: W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska (eds.): *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*. New York, pp. 1065–1072, Morgan Kaufmann Publishers.

Potter, M. A. and K. A. D. Jong: 2000, 'Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents'. *Evolutionary Computation* **8**, 1–29.

Pyeatt, L. and A. Howe: 1998, 'Learning to Race: Experiments with a Simulated Race Car'. In: D. J. Cook (ed.): *Proceedings of the 11th International Florida Artificial Intelligence Research Society Conference*. Florida, pp. 357–361.

Rosin, C. D. and R. K. Belew: 1995, 'Methods for Competitive Co-evolution: Finding Opponents Worth Beating'. In: S. Forrest (ed.): *Proceedings of the Sixth International Conference on Genetic Algorithms*. San Mateo,CA, pp. 373–380, Morgan Kaufman.

Schaffer, J. D., D. Whitley, and L. J. Eshelman: 1992, 'Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art'. In: D. Whitley and J. Schaffer (eds.): *International Workshop on Combinations of Genetic Algorithms and Neural Networks*. Los Alamitos, CA, pp. 1–37, IEEE Computer Society Press.

Stanley, K. O. and R. Miikkulainen: 2004, 'Competitive Coevolution through Evolutionary Complexification'. *Journal of Artificial Intelligence Research* pp. 63–100.

Stone, P.: 2000, *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press.

Stone, P., (ed.), M. Asada, T. Balch, M. Fujita, G. Kraetzschmar, H. Lund, P. Scerri, S. Tadokoro, and G. Wyeth: 2001, 'Overview of RoboCup-2000'. In: P. Stone, T. Balch, and G. Kraetszchmar (eds.): *RoboCup-2000: Robot Soccer World Cup IV*. Berlin: Springer Verlag, pp. 1–28.

Stone, P. and D. McAllester: 2001, 'An Architecture for Action Selection in Robotic Soccer'. In: *Proceedings of the Fifth International Conference on Autonomous Agents*. pp. 316–323.

Stone, P. and R. S. Sutton: 2001, 'Scaling Reinforcement Learning toward RoboCup Soccer'. In: *Proceedings of the Eighteenth International Conference on Machine Learning*. pp. 537–544, Morgan Kaufmann, San Francisco, CA.

Stone, P. and R. S. Sutton: 2002, 'Keepaway Soccer: a Machine Learning Testbed'. In: A. Birk, S. Coradeschi, and S. Tadokoro (eds.): *RoboCup-2001: Robot Soccer World Cup V*. Berlin: Springer Verlag, pp. 214–223.

Stone, P. and M. Veloso: 1998, 'A Layered Approach to Learning Client Behaviors in the RoboCup Soccer Server'. *Applied Artificial Intelligence* **12**, 165–188.

Stone, P. and M. Veloso: 2000, 'Layered Learning'. In: R. L. de Mántaras and E. Plaza (eds.): *Machine Learning: ECML 2000 (Proceedings of the Eleventh European Conference on Machine Learning)*. Barcelona,Catalonia,Spain: Springer Verlag, pp. 369–381.

Utgoff, P. E. and D. J. Stracuzzi: 2002, 'Many-Layered Learning'. *Neural Computation* **14**, 2497–2529.

Whiteson, S. and P. Stone: 2003, 'Concurrent Layered Learning'. In: *AAMAS 2003: Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems*. pp. 193–200.

Whitley, D., K. Mathias, and P. Fitzhorn: 1991, 'Delta-Coding: An Iterative Search Strategy for Genetic Algorithms'. In: R. K. Belew and L. B. Booker (eds.): *Proceedings of the Fourth International Conference on Genetic Algorithms*. pp. 77–84.

Yao, X.: 1999, 'Evolving Artificial Neural Networks'. *Proceedings of the IEEE* **87**(9), 1423–1447.

Yong, C. H. and R. Miikkulainen: 2001, 'Cooperative Coevolution of Multi-Agent Systems'. Technical Report AI01-287, The University of Texas at Austin Department of Computer Sciences.