In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems* (AAMAS 03), pp. 193-200, Melbourne, Australia, July 2003.

Concurrent Layered Learning

Shimon Whiteson and Peter Stone
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712 USA
{shimon,pstone}@cs.utexas.edu
http://www.cs.utexas.edu/~{shimon,pstone}

ABSTRACT

Hierarchies are powerful tools for decomposing complex control tasks into manageable subtasks. Several hierarchical approaches have been proposed for creating agents that can execute these tasks. Layered learning is such a hierarchical paradigm that relies on *learning* the various subtasks necessary for achieving the complete high-level goal. Layered learning prescribes training low-level behaviors (those closer to the environmental inputs) prior to high-level behaviors. In past implementations these lower-level behaviors were always frozen before advancing to the next layer. In this paper, we hypothesize that there are situations where layered learning would work better were the lower layers allowed to keep learning concurrently with the training of subsequent layers, an approach we call *concurrent layered learning*. We identify a situation where concurrent layered learning is beneficial and present detailed empirical results verifying our hypothesis. In particular, we use neuro-evolution to concurrently learn two layers of a layered learning approach to a simulated robotic soccer keepaway task. The main contribution of this paper is evidence that there exist situations where concurrent layered learning outperforms traditional layered learning. Thus, we establish that, when using layered learning, the concurrent training of layers can be an effective option.

Categories and subject descriptors: I.2.6 [Artificial Intelligence]: Learning—Connectionism and neural nets; I.2.8 [Artificial Intelligence]: Robotics—Autonomous vehicles.
General Terms: Algorithms, Experimentation.
Keywords: evolution, adaptation, learning.

1. INTRODUCTION

Hierarchies are powerful tools for decomposing complex control tasks into manageable subtasks. As a case in point, mammalian biology 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, etc. 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 (e.g. [2, 4]). Layered learning [19, 20] is such a hierarchical paradigm that relies on *learning* the various subtasks necessary for achieving the complete high-level goal. Layered learning is a bottom-up paradigm by which low-level behaviors (those closer to the environmental inputs) are trained prior to high-level behaviors.

The original implementation of layered learning [15] consisted of three learned layers. Lower-level behaviors were always trained and then frozen before advancing to the next layer. Once a subtask was learned, it was not allowed to change while subsequent subtasks were learned. This approach can aid learning by reducing the space of possible solutions we must search at a given time. However, it can also be restrictive. In our analogy to mammalian biology, that restriction is 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 this paper, we hypothesize that there are situations in which layered learning would work better were the lower layers allowed to keep learning concurrently with the training of subsequent layers. We identify such a situation and present detailed empirical results verifying our hypothesis. We refer to such concurrent training within the layered learning paradigm as *concurrent layered learning*.

Concurrent layered learning is consistent with the existing layered learning formalism. The main contribution of this paper is to establish that, when using layered learning, the concurrent training of layers can be an effective option.

The remainder of this paper is organized as follows. Section 2 explains layered learning as well as the substrate systems with which our implementation of layered learning is built, namely neuro-evolution and the robotic soccer keepaway testbed. Section 3 details our approach to applying traditional layered learning (without concurrent training) to the keepaway task and Section 4 shows how we modify this approach to use concurrent layered learning. Section 5 presents the results of our experiments verifying the advantage of the concurrent approach. Section 6 discusses our results and relates them to other research and Section 7 concludes.

2. BACKGROUND AND METHOD

Our particular implementation of layered learning uses the neuro-evolution ML algorithm [14] as the substrate learning approach. Our experiments are all conducted within a keep-away subtask of simulated robotic soccer. In the remainder of this section, we provide background on layered learning, neuro-evolution, and keepaway. We also introduce the tools necessary for implementing concurrent layered learning and mention some essential previous research. We discuss additional related work in Section 6.

2.1 Layered Learning

Table 1 summarizes the principles of the layered learning paradigm which are described in detail in this section[1].

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.

Table 1: The key principles of layered learning.

Principle 1

Layered learning is designed for domains that are too complex for learning a mapping directly from the input to the output representation. Instead, the layered learning approach consists of breaking a problem down into several task layers. At each layer, a concept needs to be acquired. A machine learning (ML) algorithm abstracts and solves the local concept-learning task.

Principle 2

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.

Principle 3

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

Principle 4

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.

Formalism

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:

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

$\boldsymbol{O_i}$ 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.

As stated in Principle 2 of layered learning, the definitions of the layers L_i are given *a priori*. Principle 4 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 $\forall i, j,\ F_i^j \in S \cup_{k=1}^{i-1} O_k$.

Layered learning was originally applied in a complex, multi-agent learning task, namely simulated robotic soccer in the RoboCup soccer server [10]. In this implementation [15], the learning of each h_i was completed before training layer L_{i+1}.

In the concurrent layered learning approach we propose, h_i is not frozen when we start to train L_{i+1}. Hence, the affect that h_i has on T_{i+1} is not fixed throughout the learning of L_{i+1}, but instead changes constantly as h_i continues to learn.

[1]This section is adapted from [20].

2.2 Neuro-Evolution

Our implementation of layered learning uses neuro-evolution as its ML algorithm M_i at each layer. Neuro-evolution is a machine learning technique that uses genetic algorithms to train neural networks [14]. In its simplest form, it 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 our domain and selectively reproducing the fittest individuals through crossover and mutation.

The Enforced Sub-Populations Method (ESP) [6] 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 1 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 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).

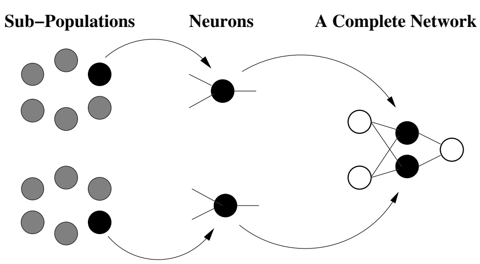

Figure 1: The Enforced Sub-Populations Method (ESP). The population of neurons is segregated into sub-populations shown here as clusters of grey circles. One neuron, shown in black, is selected from each sub-population. Each neuron consists of all the weights connecting a given hidden node to the input and output nodes, shown as white circles. The selected neurons together form a complete network which is then evaluated in the task.

In several benchmark sequential decision tasks, ESP outperformed other neuro-evolution algorithms as well as several reinforcement learning methods [6]. ESP is a promising choice for our task because the skills required in the keepaway game we consider are similar to those ESP has excelled at before.

2.2.1 Coevolution

The genetic algorithm is a natural ML technique with which to implement our modification of layered learning because it provides an elegant method for concurrent learning: coevolution. Coevolution consists of simultaneously evolving multiple components that perform different roles but are evaluated in a common domain. Coevolution can be competitive [7, 13], in which case these roles are adversarial and one component's gain is another's loss. Coevolution can also be cooperative [12], as when the various components share fitness scores. Multi-Agent ESP [23] 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 [23].

In concurrent layered learning, before training in L_{i+1} begins, we take the best network from L_i and use it to seed a new population. This new population continues to learn along with a separate population learning L_{i+1}. Hence, we evolve the two layers cooperatively using the same method as Multi-agent ESP, though each of our networks are not separate agents but rather components of the same agent. To perform a fitness evaluation, we take a network from the first population, which was seeded from the results of L_i, and evaluate it in T_{i+1} together with a network selected from the second population, which is learning L_{i+1} from scratch. The resulting score is shared by both networks.

2.2.2 Delta-Coding

To seed a population from the results of L_i, we use a technique called delta-coding [22]. When delta-coding, we take the result of a given layer, h_i, and use it to create a new population, each member of which is a perturbation of h_i. The network that will perform task L_i optimally in T_{i+1} is likely to be near h_i but occasionally may be radically different. Hence, we base the amount of perturbation on a Cauchy distribution, such that most of the new individuals are very similar to h_i but a few are significantly different.

Delta-coding is an effective method for preventing premature convergence to a local maxima by restoring diversity to the population. It is particularly well suited to helping populations adjust to sudden changes in their training environment [5]. Hence, it is an excellent way to seed a new population from the results of an earlier layer.

2.3 Keepaway

The experiments reported in this paper are all in a keepaway subtask of robotic soccer [18][2]. 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.

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 [16]. 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 [10], 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 2 shows how a game of keepaway is initialized. We place three keepers 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

[2]The definition and implementation of the keepaway domain in the SoccerBots environment is joint work with Nate Kohl and Risto Miikkulainen.

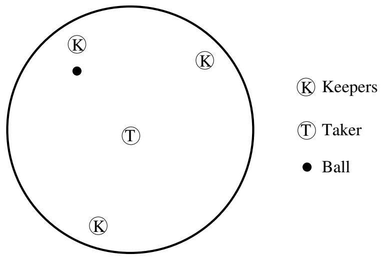

Figure 2: A game of keepaway after initialization.

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.

2.4 Previous Research

In previous research, we have succeeded in learning complete agents for a keepaway task in the SoccerBots environment [21]. Using neuro-evolution as the substrate learning method, we have employed layered learning to develop a suite of agents that perform this task.

The research presented in this paper is motivated by a particular question that arose during the course of that research. Despite our eventual success, we found that requiring the lower layers to be fixed can be restrictive because two adjacent layers are sometimes interdependent. For example, we cannot properly train an agent to pass unless we know how its teammates will try to receive its passes. But how can we train an agent to receive passes before we have a passer to kick to it? In situations like these, regardless of how we order the two layers, the training environment of the lower layer will be necessarily sub-optimal.

Here, we examine whether the effects of such imperfect training environments can be mitigated or eliminated by allowing the lower layer to continue to evolve. This question has not previously been examined in connection with the layered learning paradigm.

3. TRADITIONAL LAYERED LEARNING IN KEEPAWAY

In this section we detail a "traditional" layered learning approach to creating agents for the keepaway task. That is, each learned layer is frozen before learning the next higher layer. In Section 5 we experimentally compare this implementation to the concurrent layered learning implementation described in Section 4.

To control the keepers, we develop a set of three homogeneous agents, each of which can perform several heterogeneous roles. 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 given time. Unlike soccer, where a strong team will have forwards and defenders specialized for different roles, the symmetry of keepaway lends itself towards homogeneous teams. Having identical agents makes learning easier, since each agent learns from the experiences of its teammates as well as its own.

Figure 3 shows a simple decision tree for controlling each keeper in the keepaway task. If the agent is near 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.

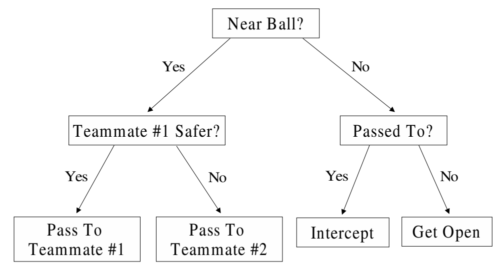

Figure 3: A decision tree for controlling keepers in the keepaway task. We implement the behavior at each of the leaves with layers from the learning hierarchy. Another layer, pass evaluation, is used to decide which teammate to pass to.

To implement this decision tree, the agents must master four different skills. Three of these skills correspond to the behaviors at the leaves of the tree: passing, intercepting, and getting open. The fourth skill, pass evaluation, is the ability to analyze the current game state and estimate the likelihood of successfully passing to a specific receiver. We use pass evaluation, not at a leaf of the tree, but at a branch, when deciding which teammate to pass to.

Figure 4 shows one way that agents can master these skills using a traditional layered learning approach. 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 it depends on have been learned, we start at the bottom, with intercept, and move up the hierarchy step by step. Each task is learned by training feed-forward neural networks via ESP, with sub-population sizes of 100. Each layer is described below.

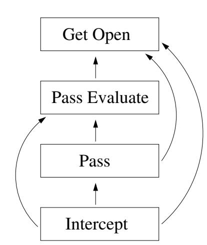

Figure 4: **A layered learning hierarchy for the keepaway task. Each box represents a layer and arrows indicate dependencies between layers. A layer cannot be learned until all the layers it depends on have been learned.**

$\mathbf{L_1}$: **Intercept:** The goal of this task is simply to get to the ball as quickly as possible. The obvious strategy, running directly towards the ball, is optimal only if the ball is motionless. When the ball has velocity, an ideal interceptor must anticipate where the ball is going. 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. The networks have 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: one controls the agent's heading and the other its speed. In all of our experiments, the taker continually uses the trained intercept behavior.

$\mathbf{L_2}$: **Pass:** In this task, we want the agent to kick the ball away at a specified angle. Passing is complicated by the fact that 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. To train the passer we again propel the ball towards the agent. We also randomly select the angle at which we want the agent to kick the ball. 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. The passer has three inputs: two for the ball's current position and one for the target angle. It has two hidden nodes and two outputs: one controls the agent's heading and the other its speed[3].

$\mathbf{L_3}$: **Pass Evaluate:** The pass evaluator's job is to analyze the current state of the game and assess the likelihood of successfully passing to a specific receiver. Because this layer and the one after it, get open, are the focus of this study, we describe them formally and in more detail than the other layers.

$\tilde{\mathbf{F}}_\mathbf{3} = \{\mathbf{Ball_r}, \mathbf{Ball_t}, \mathbf{Taker_r}, \mathbf{Taker_t}, \mathbf{Teammate_r}, \mathbf{Teammate_t}\}$: The agent learns to evaluate passes based on the current position of the ball, the taker, and the teammate whose potential as a receiver it is evaluating. Positions are represented relative to the agent as (r, t) in polar coordinates.

$\mathbf{O_3} = \{\mathbf{Confidence}\}$: The agent outputs a real number between 0 and 1 indicating its confidence that a pass to the given teammate would succeed.

$\mathbf{T_3}$: Figure 5 shows the pass evaluator's training environment. We place the ball in the center of the field and put the pass evaluator just behind it at various angles. We place two teammates 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, we run the evolving network twice, once for each teammate, and then pass, using L_2, to the teammate who received a higher evaluation. If the pass succeeds, the agent is rewarded. We evaluate each network fifty times and sum the scores.

$\mathbf{M_3}$ = **neuro-evolution:** Using ESP, we train a fully connected two-layer feed-forward neural network with 6 inputs, 2 hidden nodes, and 1 output. Figure 6 shows the architecture of this network. Each sub-population contains 100 neurons.

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

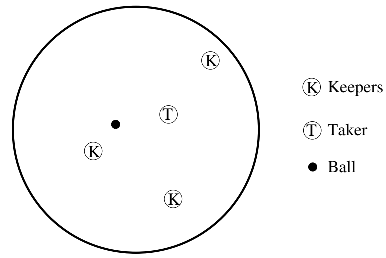

Figure 5: A typical training scenario for the pass evaluator. One keeper, the passer, must choose which teammate to kick to in order to prevent the taker from getting the ball.

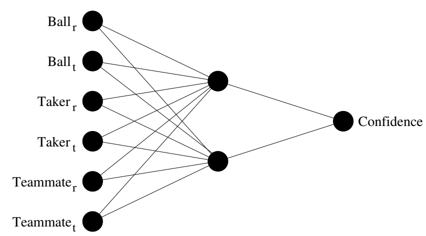

Figure 6: The fully connected two-layer feed-forward network h_3 for pass evaluation.

$\mathbf{L_4}$: **Get Open:** We use the get open behavior when a keeper does not have a ball and is not receiving a pass. Clearly, such an agent wants to get to a position where it can receive a pass. However, an optimal get open behavior will not necessarily position the agent where a pass is most likely to succeed. Instead, it will position the agent where a pass would be most strategically advantageous.

$\tilde{\mathbf{F}}_\mathbf{4} = \{\mathbf{Ball_r}, \mathbf{Ball_t}, \mathbf{Taker_r}, \mathbf{Taker_t}, \mathbf{Boundary_r}\}$: The agent receives as input the current position of the ball and the taker. It also knows how close it is to the field's bounding circle.

$\mathbf{O_4} = \{\mathbf{Heading}, \mathbf{Speed}\}$: The agent maneuvers on the field by altering its heading and its speed.

$\mathbf{T_4}$: 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 3 along with the evolved behaviors from L_1, L_2, and L_3. We evaluate each network in twenty games of keepaway and sum the scores.

$\mathbf{M_4}$ = **neuro-evolution:** Using ESP, we train a fully connected two-layer feed-forward neural network with 5 inputs, 2 hidden nodes, and 2 outputs. Figure 7 shows the architecture of this network. Each sub-population contains 100 neurons.

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

Once these four layers have been learned, we can fully implement the decision tree shown in Figure 3. With this

[3] The learning of L_1 and L_2 is joint work with Nate Kohl and Risto Miikkulainen.

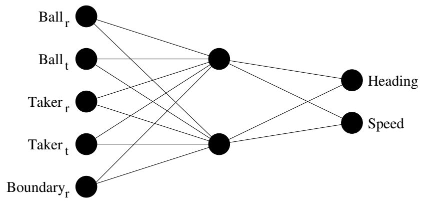

Figure 7: The fully connected two-layer feed-forward network h_4 for the get open behavior.

decision tree controlling each of the three keepers, we have a complete keepaway team. Note that the first layer is also used to control the taker, which continually tries to intercept the ball.

In the implementation described in this section, each layer L_i is learned and frozen before learning layer L_{i+1}. As such, there is no concurrent learning of the layers.

4. CONCURRENT LAYERED LEARNING IN KEEPAWAY

To test the potential of concurrent layered learning, we compare the traditional layered learning implementation described above to one in which L_3, pass evaluate, is permitted to coevolve with L_4, get open. We train L_1 through L_3 in exactly the same manner as described in Section 3. When we begin to train L_4, we use delta-coding to seed a new population from h_3 and use it to continue training L_3 concurrently with the population learning L_4. Both populations use episodes of the full keepaway task, T_4, as training examples.

Pass evaluation is an ideal task to try to coevolve with its successor because it typifies a common difficulty in layered learning: the training environment T_i often does not perfectly reflect the way h_i will be used in higher layers. In particular, the potential receivers in T_3 use the intercept behavior throughout the episode, but in a real game of keepaway, like T_4, those teammates will be using get open behavior until they have been passed to, as Figure 3 shows. We cannot easily remedy this discrepancy because when we train L_3, we do not have the get open behavior yet! h_3 is likely to be sub-optimal as a result and if we freeze it, h_4 and the resulting keepaway players will be sub-optimal too.

Note that inverting the order of these two layers does not solve this problem. If we trained get open first, we could use it to train the pass evaluator in a more representative scenario. However, this alternative just trades one problem for another. The goal of the get open behavior is to move to a location that is most strategically advantageous, which depends in part on how the passer chooses its receiver. Thus, if we train get open first, its training environment will be imperfect since no pass evaluator will yet exist.

In either case, the training of the lower layer will not be ideal. We hypothesize that concurrent layered learning, by allowing the lower layer to adjust to its new environment, will result in superior performance in the keepaway task.

5. EMPIRICAL RESULTS

In this section we present the results of experiments designed to test this hypothesis.

Our main result is that concurrent layered learning of L_3 and L_4 outperforms traditional layered learning on the keepaway task. Figure 8 depicts the average fitness of the entire population for each generation as we train L_4 in the keepaway task. Recall that each individual's fitness is the number of passes completed over twenty keepaway episodes. These results are averaged over seven runs, with L_3 retrained for each run. Using a t-test, we confirmed that, after generation 2, the difference between the two methods was statistically significant with 95% confidence. Concurrent layered learning significantly outperforms the traditional approach, confirming our hypothesis that concurrent layered learning can be advantageous.

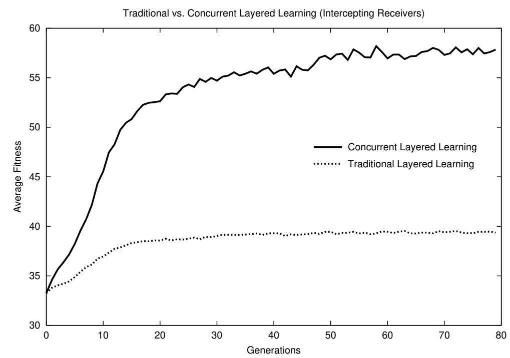

Figure 8: **Traditional vs. Concurrent Layered Learning (Intercepting Receivers). When learning L_3, the potential receivers use the intercept behavior as described in Section 3. These results are averaged over seven runs.**

As mentioned above, one hindrance to traditional layered learning in this case might be the discrepancy between T_3, when we train the pass evaluator, and T_4, when we use it. Since we do not have the get open behavior when we train L_3, we cannot make the potential receivers in T_3 behave exactly as they will in T_4. Instead, we make the receivers intercept in the hopes that this will approximate the scenarios the pass evaluator will see in an actual keepaway game.

To verify that the superior performance of concurrent layered learning is not due just to a poorly designed training environment for the pass evaluator, we consider another approximation of the get open behavior in T_3. In this alternative, the receivers are stationary until the passer decides to kick to one of them, at which point the selected receiver switches to interception. Since a successful get open strategy will likely keep the keepers away from each other at the edges of the field, we hypothesize that this behavior provides a more accurate environment for training the pass evaluator.

Figure 9 confirms that concurrent layered learning still outperforms the traditional approach in this modified environment. In fact, the divide is even greater. The differences are statistically significant after generation 3. The results of Figures 8 and 9 together suggest strongly that the superior performance of the concurrent approach is not due just to a poorly designed T_3 but to a limitation in the traditional method. In other words, concurrently training h_3 and h_4 after learning h_3 individually may be necessary for achieving the best possible results. Even if it is possible to design a training environment for L_3 that will produce an h_3 that is sufficient to match our best results (which we doubt), our method avoids the laborious effort of finely tuning the training of lower layers.

Interestingly, the traditional approach does worse in the modified environment but the concurrent approach does bet-

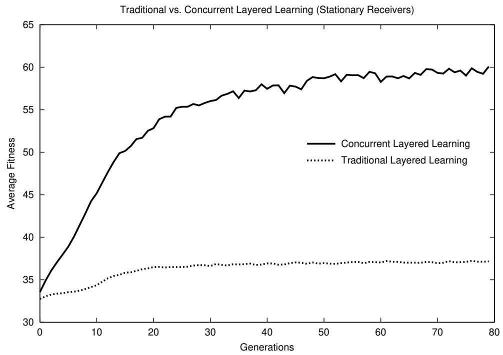

Figure 9: **Traditional vs. Concurrent Layered Learning in Keepaway (Stationary Receivers). When learning L_3 the potential receivers are stationary until the passer decides to kick to one them, at which point the selected receiver begins to intercept. These results are averaged over seven runs.**

ter. The h_3 we learn with stationary receivers, though a weak pass evaluator, seems to be nearer in the search space to a strong pass evaluator. Hence, this h_3 provides a better seed for a population that learns concurrently with L_4.

One question that remains is whether layered learning helps at all in this environment. Perhaps it is possible to achieve comparable results by simply coevolving L_3 from scratch along with L_4, without using h_3 as a seed. In other words, L_3 and L_4 could be conflated into a single layer in which we try to coevolve the pass evaluate and get open behaviors in the keepaway task given only h_1 and h_2. Figure 10 compares such an approach with our best results from concurrent layered learning (from Figure 9). The differences are statistically significant after generation 54. It is interesting to note that coevolving the two layers from scratch outperforms traditional layered learning, suggesting that these two layers happen to be strongly interdependent. Nonetheless, this comparison confirms that we can get the strongest results, not by replacing layered learning with coevolution, but by combining the two.

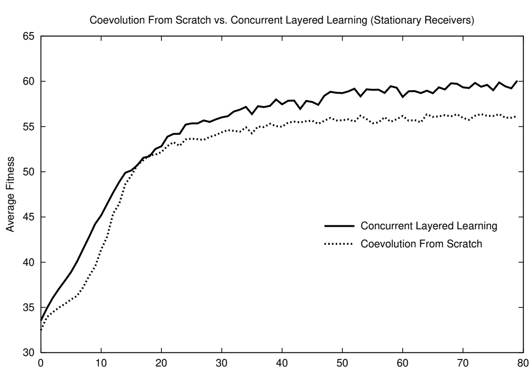

Figure 10: **Coevolution from Scratch vs. Concurrent Layered Learning (Stationary Receivers). These results are averaged over seven runs.**

6. DISCUSSION AND RELATED WORK

Our results demonstrate that concurrent layered learning can outperform traditional layered learning. In particular, we verified that concurrent learning achieves superior results when learning higher layers of SoccerBots keepaway agents.

We have focused here on a single instance of training concurrency involving just two layers of a complete layered learning implementation. While this single instance verifies that concurrent layered learning *can* be useful, we are not claiming that it should be used in all cases. Indeed, there may be instances in which traditional layered learning performs just as well as, or even outperforms, concurrent layered learning due to its more aggressive use of hierarchy. Nonetheless, our experience with traditional layered learning, including training the lower layers of the task described in this paper, suggests that there are also many instances in which it is not possible to create a perfect training environment for the lower layers.

For example, when training layers L_1 through L_3, we repeatedly found that our initial training environment in L_i was not sufficiently representative of the range of behaviors that were needed in L_{i+1}. In all of these cases, concurrent layered learning remedied the situation though we were able to achieve results that were equally good by finding a more clever training environment for L_i and retraining it prior to training L_{i+1}. Although concurrent layered learning does not provide a *performance* boost in those cases, it certainly would have saved us manual effort, which is one of the primary reasons for using machine learning.

Robotic soccer keepaway has been used as a testbed domain for several previous machine learning studies. A variant based on the the RoboCup soccer simulator was introduced for the purposes of studying multi-agent reinforcement learning [17]. In that 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 [11]. The work perhaps most related to ours uses two learned layers, each learned via genetic programming, for a keepaway task in a simplified abstraction of the SoccerBots environment [8]. This implementation uses the traditional layered learning approach of freezing the first layer (passing) before advancing to the next layer (the whole task).

The original implementation of the layered learning paradigm was on the full robotic soccer task in the RoboCup soccer simulator [15]. 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.

As indicated by the preceding example, 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 *2.2.1*, and hierarchical reinforcement learning.

Most hierarchical RL approaches use *gated* behaviors:

> 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. [9]

In some cases the behaviors are learned, in some cases the gating function is learned, and in some cases both are learned. In this last example, the behaviors are learned and fixed prior to learning the gating function. On the other hand, the MAXQ algorithm [3] does 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.

7. CONCLUSION AND FUTURE WORK

The main contribution of this paper is evidence that, when using layered learning, the concurrent training of layers is an effective option. Specifically, we have demonstrated one instance in which it is superior to traditional layered learning, where each layer is trained completely independently.

In ongoing research, we plan to identify additional instances, both in the keepaway task and in other tasks, where concurrent layered learning outperforms traditional layered learning, as well as situations in which the reverse is true. Ultimately, we aim to characterize and analyze the conditions under which concurrent layered learning is beneficial.

Acknowledgments

We thank Nate Kohl, Risto Miikkulainen, Daniel Whiteson, and Rena Whiteson for their helpful comments on initial versions of this paper. We also thank Nate Kohl for his active role in the creation of our testbed environment and participation in the work that motivated the line of research reported here.

8. REFERENCES

[1] T. Balch. Teambots domain: Soccerbots, 2000. `http://www-2.cs.cmu.edu/~trb/TeamBots/Domains/SoccerBots`.

[2] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23, 1986.

[3] T. G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, 1998.

[4] E. Gat. Three-layer architectures. *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, Menlo Park, CA, 1998.

[5] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342, 1997.

[6] 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.

[7] T. Haynes and S. Sen. Evolving behavioral strategies in predators and prey. In *Adaptation and Learning in Multiagent Systems*, pages 113–126. Springer Verlag, Berlin, 1996.

[8] 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.

[9] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, May 1996.

[10] 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.

[11] 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.

[12] M. A. Potter and K. A. D. Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8:1–29, 2000.

[13] 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.

[14] 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 *International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*, pages 1–37. IEEE Computer Society Press, 1992.

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

[16] 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.

[17] 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.

[18] P. Stone and R. S. Sutton. Keepaway soccer: a machine learning testbed. In *RoboCup-2001: Robot Soccer World Cup V*. Springer Verlag, Berlin, 2002.

[19] P. Stone and M. Veloso. A layered approach to learning client behaviors in the RoboCup soccer server. *Applied Artificial Intelligence*, 12:165–188, 1998.

[20] 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).

[21] S. Whiteson, N. Kohl, R. Miikkulainen, and P. Stone. Evolving robocup keepaway players through task decomposition. In *GECCO 2003: Proceedings of the Genetic and Evolutionary Computation Conference*, July 2003. To appear.

[22] D. Whitley, K. Mathias, and P. Fitzhorn. Delta-Coding: An iterative search strategy for genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 77–84, 1991.

[23] 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.