In *Artificial Intelligence* (AIJ), 254:21--43, January 2018.
https://doi.org/10.1016/j.artint.2017.09.001

Available online at www.sciencedirect.com

ScienceDirect

Artificial Intelligence 254 (2018)

Journal Logo

Overlapping Layered Learning☆

Patrick MacAlpine, Peter Stone

Department of Computer Science
The University of Texas at Austin
Austin, TX 78701, USA

{*patmac, pstone*}@*cs.utexas.edu*

Abstract

Layered learning is a hierarchical machine learning paradigm that enables learning of complex behaviors by incrementally learning a series of sub-behaviors. A key feature of layered learning is that higher layers directly depend on the learned lower layers. In its original formulation, lower layers were frozen prior to learning higher layers. This article considers a major extension to the paradigm that allows learning certain behaviors independently, and then later stitching them together by learning at the "seams" where their influences overlap. The UT Austin Villa 2014 RoboCup 3D simulation team, using such overlapping layered learning, learned a total of 19 layered behaviors for a simulated soccer-playing robot, organized both in series and in parallel. To the best of our knowledge this is more than three times the number of layered behaviors in any prior layered learning system. Furthermore, the complete learning process is repeated on four additional robot body types, showcasing its generality as a paradigm for efficient behavior learning. The resulting team won the RoboCup 2014 championship with an undefeated record, scoring 52 goals and conceding none. This article includes a detailed experimental analysis of the team's performance and the overlapping layered learning approach that led to its success.

Keywords: layered learning, reinforcement learning, robot soccer, CMA-ES, robot skill learning, hierarchical machine learning

1. Introduction

Task decomposition is a popular approach for learning complex control tasks when monolithic learning—trying to learn the complete task all at once—is difficult or intractable [4, 5, 6]. Layered learning [7] is a hierarchical task decomposition machine learning paradigm that enables learning of complex behaviors by incrementally learning a series of sub-behaviors. A key feature of layered learning is that higher layers directly depend on the learned lower layers. In its original formulation, lower layers were frozen prior to learning higher layers. Freezing lower layers can be restrictive, however, as doing so limits the combined behavior search space over all layers. Concurrent layered learning [8] reduced this restriction in the search space by introducing the possibility of learning some of the behaviors simultaneously by "reopening" learning at the lower layers while learning the higher layers. A potential drawback of increasing the size of the search space, however, is an increase in the dimensionality and thus possibly the difficulty of what is being learned.

This article considers an extension to the layered learning paradigm, known as *overlapping layered learning*, that allows learning certain parameterized behaviors independently, and then later stitching them together by learning at the "seams" where their influences overlap. Overlapping layered learning aims to provide a middle ground between

☆This article is based on previous conference publications [1, 2, 3].

reductions in the search space caused by freezing previously learned layers and the increased dimensionality of concurrent layered learning. Additionally, for complex tasks where it is difficult to learn one subtask in the presence of another, it reduces the dimensionality of the parameter search space by focusing only on parts responsible for subtasks working together.

The UT Austin Villa 2014 RoboCup 3D simulation team, using overlapping layered learning, learned a total of 19 layered behaviors for a simulated soccer-playing robot, organized both in series and in parallel. To the best of our knowledge this is more than three times the number of layered behaviors in any prior layered learning system. Furthermore, the complete learning process is repeated on four different heterogeneous robot body types, showcasing its generality as a paradigm for efficient behavior learning. The resulting team won the RoboCup 2014 championship with an undefeated record, scoring 52 goals and conceding none.

Primary contributions of this article are twofold. First, we introduce the overlapping layered learning paradigm, and present general scenarios where its use is beneficial. Second, we provide a detailed description and analysis of our machine learning approach, incorporating overlapping layered learning, to create a large and complex control system that was a core component of the 2014 RoboCup 3D simulation league championship team as well as three subsequent championship teams.

The remainder of this article is organized as follows. Section 2 provides background information on the original layered learning paradigm which is the basis for this work. Section 3 specifies and motivates the overlapping layered learning paradigm while contrasting it with traditional and concurrent layered learning. In Section 4 we introduce the RoboCup 3D simulation domain in which we evaluate this research. Section 5 details the overlapping layered learning approach of the 2014 UT Austin Villa team and in Section 6 we provide detailed analysis of its performance. Section 7 discusses related work while Section 8 concludes.

2. Layered Learning Paradigm

Table 1 summarizes the principles of the original 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 reaching the high-level task that deal with the full domain complexity. 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.

[1]This section is adapted from [7].

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. In the former case, learning can be done off-line and frozen for future use. In the latter, on-line learning is necessary: since the learner needs to adapt to unexpected situations, it must be able to alter its behavior even while executing its task. 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

$$L_i = (\vec{F}_i, O_i, T_i, M_i, H_i, h_i)$$

where:

$\boldsymbol{\vec{F}_i}$ 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 policy representation mapping $\vec{F}_i$ to O_i.[2]

$\boldsymbol{h_i}$ is the result of running M_i on T_i. h_i is a specific instantiation of H_i and is a function from $\vec{F}_i$ to O_i.

As set out 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 $\forall i, j, F_i^j \in S \cup_{k=1}^{i-1} O_k$.

In the context of this work all learned behaviors—or hypotheses—are represented by parameterized policies, where a parameterized policy with k parameters for layer L_i is the set of parameters $H_i = \{H_i^1, \ldots, H_i^k\}$, and the hypothesis is the corresponding set of the parameters' learned values $h_i = \{h_i^1, \ldots, h_i^k\}$.

[2]Previous work [7, 8] included H_i within M_i, however we separate out H_i for ease of notation.

3. Overlapping Layered Learning Paradigm

As explained in Section 2, layered learning is a hierarchical learning paradigm that enables learning of complex behaviors by incrementally learning a series of sub-behaviors—each learned sub-behavior is a layer in the learning progression. Higher layers depend on lower layers for learning. This dependence can include providing features for learning, such as initial seed values for parameters when H is a parameterized policy, as well as a previous learned layer's behavior being incorporated into the learning task for the next layer to be learned. In layered learning's original sequential formulation, layers are learned in a sequential bottom-up fashion and, after a layer is learned, it—the learned hypothesis h of the layer—is frozen before beginning learning of the next layer.

Concurrent layered learning, on the other hand, purposely does not freeze newly learned layers, but instead keeps them open during learning of subsequent layers: h_i is not frozen before the training of L_{i+1} begins. Thus, the effect that h_i has on T_{i+1} is not fixed during learning of L_{i+1}, and in fact changes as h_i continues to be learned. We denote the continued learning of h_i as $H_i \subseteq H_{i+1}$, meaning that H_i—the set of of parameters that the values of h_i are assigned to—is a subset of H_{i+1}, and thus the parameters H_i are fully included in what is learned for the hypothesis h_{i+1}—the parameter values h_i are re-learned—in the subsequent layer of learning L_{i+1}. Previously learned layers' behaviors are left open so that learning may enter areas of the behavior search space that are closer to the combined layers' optimum behavior as opposed to being confined to areas of the joint layer search space where the behaviors of previously learned layers are fixed. While concurrent layered learning does not restrict the search space in the way that freezing learned layers does, the increase in the search space's dimensionality can make learning slower and more difficult.

Overlapping layered learning seeks to find a tradeoff between freezing each layer once learning is complete and leaving previously learned layers open. It does so by *keeping some, but not necessarily all, parts of previously learned layers open during learning of subsequent layers.* The part of previously learned layers left open is the "overlap" with the next layer being learned. In this regard concurrent layered learning can be thought of as an extreme of overlapping layered learning with a "full overlap" between layers. Additionally, overlapping layered learning allows for behaviors to be *learned independently* in parallel—not just sequentially in series—and then *later stitched together* by learning at the "seams" where their influences overlap.

Overlapping Layered Learning: Layered learning both in series and parallel in which some, but not necessarily all, parts of previously learned layers are left open during learning of subsequent layers.

A layer of learning can be partially left open by freezing only a subset of its learned behavior's policy's parameters during subsequent layers of learning. We denote a set of parameters H' as being open and learned in a layer of learning L_i if the set of parameters are included in the layer's learned policy representation H_i (i.e. $H' \subseteq H_i$), and conversely a set of previously learned values h' for parameters H' are frozen if they are still part of the layer of learning—in the context of this work previously learned and frozen parameter values serve to define behavior used in the training task T_i which we denote as $h' \prec T_i$—but are not included in the layer's learned policy representation (i.e. $H' \cap H_i = \emptyset$). A previously learned set of parameter values h' may also be used to initialize or seed values of H_i. We denote such a relationship as $h' \dashrightarrow H_i$

The following are several general scenarios, depicted in the bottom row of Figure 1, for overlapping layered learning that help to clarify the learning paradigm and identify situations in which it is useful:

Combining Independently Learned Behaviors (CILB): Two or more behaviors are learned independently in the same or different layers, and then are combined together for a joint behavior at a subsequent layer by relearning some subset of the behaviors' parameters or "seam" between the behaviors. Let $L_{i,b}$ represent learning of the bth independent behavior in the ith layer of learning. In the case of two independent behaviors learned in parallel in the same layer, which we notate as $h_{i,1}$ learned in $L_{i,1}$ and $h_{i,2}$ learned in $L_{i,2}$, they are combined in a subsequent layer of learning $L_{j,b}$, where $j > i$, to learn a joint behavior $h_{j,b}$ containing parts or all of either or both of $H_{i,1}$ and $H_{i,2}$ (i.e. $H_{i,1} \in L_{j,b}$, $H_{i,2} \in L_{j,b}$, and $\exists H' \subseteq \{H_{i,1} \cup H_{i,2}\}$ s.t. both $H' \neq \emptyset$ and $H' \subseteq H_{j,b}$). This scenario is best when subtask behaviors are too complex and/or potentially interfere with each other during learning, such that they must be learned independently, but ultimately need to work together for a combined task. *Example: A basketball playing robot that must be able to dribble the ball across the court and shoot it in the basket. The*

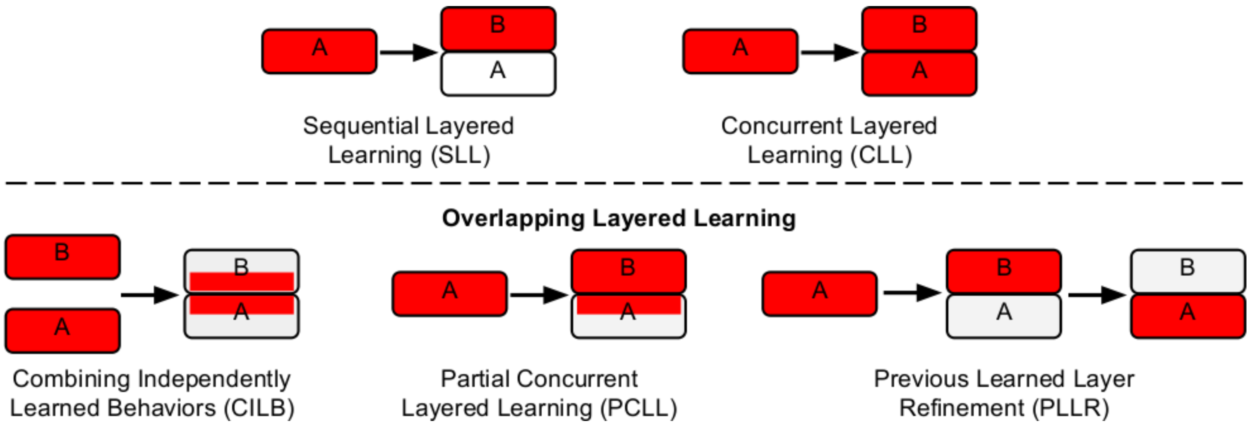

Figure 1. Paradigms for layered learning. Boxes represent different behaviors with the behaviors or parts of behaviors being learned shown in red. The arrows represent the transition from one layer of learning to the next.

tasks of dribbling and shooting are too complex to attempt to learn them together, but after the tasks are learned independently they can be combined by re-optimizing parameters that control the point on the court at which the robot stops dribbling and the angle at which the robot shoots the ball.

Partial Concurrent Layered Learning (PCLL): Only part, but not all, of a previously learned layer's behavior parameters are left open when learning a subsequent layer with new parameters. Thus there exists a non-empty proper subset of parameters H_i' of a learned behavior h_i that are re-learned in the behavior h_{i+1} as part of the subsequent layer of learning L_{i+1} (i.e. $\exists H_i' \subset H_i$ s.t. $H_i' \neq \emptyset$ and $H_i' \subset H_{i+1}$). The part of the previously learned layer's parameters left open is the "seam" between the layers. Partial concurrent learning is beneficial if full concurrent learning unnecessarily increases the dimensionality of the search space to the point that it hinders learning, and completely freezing the previous layer diminishes the potential behavior of the layers working together. *Example: Teaching one robot to pick up and hand an object to another robot. First a robot is taught to pick up an an object and then reach out its arm and release the object. The second robot is then taught to reach out its arm and catch the object released by the first robot. During learning by the second robot to catch the object, the part of the previously learned behavior of the first robot to hand over the object is left open so that the first robot can adjust its release point of the object to a place that the second robot can be sure to reach.*

Previous Learned Layer Refinement (PLLR): After a layer is learned and frozen, and then a subsequent layer is learned, part or all of the previously learned layer is then unfrozen and relearned to better work with the newly learned layer that is now fully or partially frozen. We consider re-optimizing a previously frozen layer under new conditions as a new learned layer behavior with the "seam" between behaviors being the unfrozen part of the previous learned layer. Thus there exists a non-empty subset of parameters H_i' of a learned behavior h_i that is frozen in a subsequent layer of learning L_j, where $j > i$, but then is eventually unfrozen and re-learned in the behavior h_k as part of a later layer of learning L_k, where $k > j$ (i.e. $\exists H_i' \subseteq H_i$ s.t. $H_i' \neq \emptyset, h_i' \prec T_j$ but $H_i' \cap H_j = \emptyset$ *(learned values h_i' for parameters H_i' are frozen in L_j)*, and $H_i' \subseteq H_k$ *(learned values h_i' for parameters H_i' are unfrozen in L_k)*). This scenario is useful when a subtask is required to be learned before the next subsequent task layer can be learned, but then refining or relearning the initial learned subtask layer's behavior to better work with the newly learned subsequent task layer's behavior provides a benefit. *Example: Teaching a robot to walk. First the robot needs to learn how to stand up so that if it falls over it can get back up and continue trying to walk. Eventually the robot learns to walk so well that it barely if ever falls over during training. Later, when the robot does eventually fall over, it is found that the walking motion learned by the robot is not stable if the robot tries to walk right after standing up. The robot needs to relearn the standing up behavior layer such that after doing so it is in a stable position to start walking with the learned walking behavior layer.*

Figure 2. A screenshot of the Nao humanoid robot (left), and a view of the soccer field during a 11 versus 11 game (right).

4. Testbed Domain: 3D Simulated RoboCup Soccer

Robot soccer has served as an excellent testbed for learning scenarios in which multiple skills, decisions, and controls have to be learned by a single agent, and agents themselves have to cooperate or compete. There is a rich literature based on this domain addressing a wide spectrum of topics from low-level concerns, such as perception and motor control [9, 10], to high-level decision-making [11]. The domain serves both as the motivation for the research presented in this paper and the evaluation testbed. We therefore introduce it in this section.

The RoboCup 3D simulation environment is based on SimSpark [12, 13],[3] a generic physical multiagent systems simulator. SimSpark uses the Open Dynamics Engine[4] (ODE) library for its realistic simulation of rigid body dynamics with collision detection and friction. ODE also provides support for the modeling of advanced motorized hinge joints.

The robot agents in the simulation are modeled after the Nao robot, which has a height of about 57 cm and a mass of 4.5 kg. The agents interact with the simulator by sending torque commands and receiving perceptual information. Each robot has 22 degrees of freedom: six in each leg, four in each arm, and two in the neck. Joint perceptors provide the agent with noise-free angular measurements every simulation cycle (20 ms), while joint effectors allow the agent to specify the torque and direction in which to move a joint. Although there is no intentional noise in actuation, there is slight actuation noise that results from approximations in the physics engine and the need to constrain computations to be performed in real-time.

Visual information about the environment is given to an agent every third simulation cycle (60 ms) through noisy measurements of the distance and angle to objects within a restricted vision cone (120°). Agents are also outfitted with noisy accelerometer and gyroscope perceptors, as well as force resistance perceptors on the sole of each foot. Additionally, a single agent can communicate with the other agents every other simulation cycle (40 ms) by sending messages limited to 20 bytes.

In addition to the standard Nao robot model, four additional variations of the standard model, known as heterogeneous types, are available for use. The variations from the standard model include changes in leg and arm length, hip width, and also the addition of toes to the robot's foot. Figure 2 shows a visualization of the standard Nao robot and the soccer field during a game.

5. Overlapping Layered Learning Applied to Robot Soccer

UT Austin has been a perennial participant in the annual RoboCup soccer competitions, and has won the championship six times between 2011 and 2017. The team began using sequential layered learning in 2011, but introduced overlapping layered learning in 2014, and has used it since. In this section, we therefore focus on that year's team.

[3] http://simspark.sourceforge.net/
[4] http://www.ode.org/

The 2014 UT Austin Villa team introduced an extensive layered learning approach to learn skills for the robot such as getting up, walking, and kicking. This approach includes sequential layered learning where a newly learned layer is frozen before learning of subsequent layers, as well as overlapping layers where parts of previously learned layers are re-optimized as part of the current layer being learned.

In total over 500 parameters were optimized during the course of layered learning. All parameters were optimized using the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) algorithm [14], which has been successfully applied previously to learning skills in the RoboCup 3D simulation domain [15]. CMA-ES is a policy search algorithm that successively generates and evaluates sets of candidates—with each candidate being a set of parameter values—sampled from a multivariate Gaussian distribution. Once CMA-ES generates a group of candidates, each candidate is evaluated with respect to an objective function—also known as a *fitness* measure—that returns a real number value indicating how well a candidate performed on the training task being learned. When all the candidates in the group are evaluated, the mean of the multivariate Gaussian distribution is recalculated as a weighted average of the candidates with the highest returned values. The covariance matrix of the distribution is also updated to bias the generation of the next set of candidates toward directions of previously successful search steps.

A total of 705,000 learning trials were performed during the process of optimizing 19 behaviors. As CMA-ES is a parallel search algorithm, optimization was performed on a Condor [16] distributed computing cluster allowing for many jobs to be run in parallel. Running the complete optimization process took about 5 days, and we calculated it could theoretically be completed in as little as 49 hours assuming no job queuing delays on the computing cluster, and all possible parallelism during the optimization process is exploited. Note that this same amount of computation, when performed sequentially on a single computer,[5] would take approximately 561 days, or a little over 1.5 years, to finish.

The following subsections document the overlapping layered learning parts of the approach used by the team. Full details of all of the learned behavior layers are provided in Appendix A,[6] and a diagram of how all the different layered learning behaviors fit together during the course of learning can be seen in Figure 3.

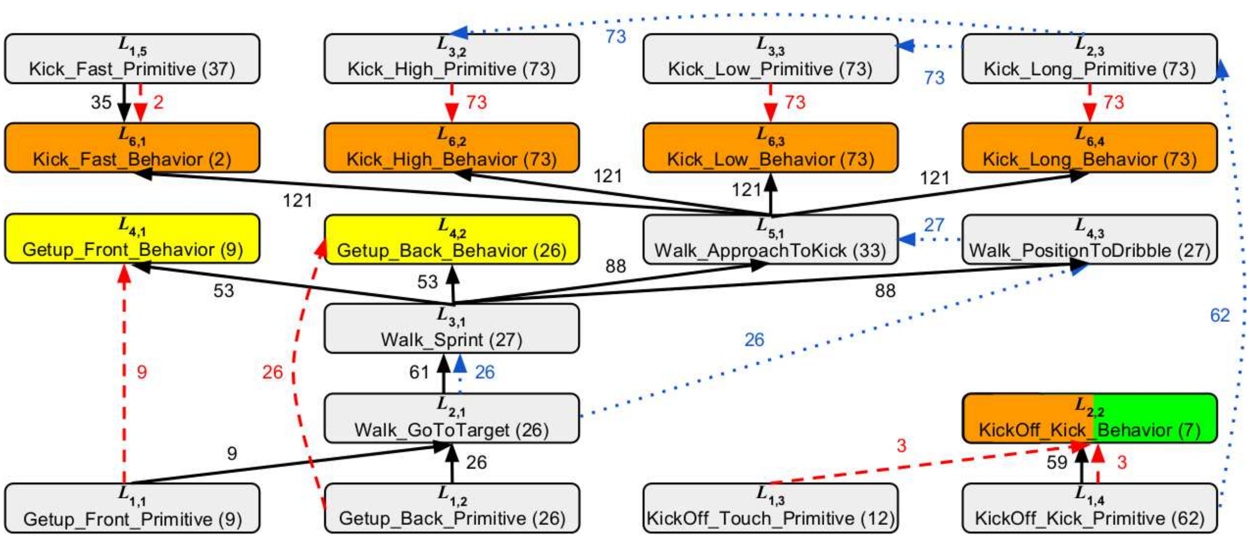

Figure 3. Different layered learning behaviors with the number of parameters optimized for each behavior shown in parentheses (best viewed in color). Solid black arrows show number of learned and frozen parameters passed from previously learned layer behaviors, dashed red arrows show the number of overlapping parameters being passed and relearned from one behavior to another, and the dotted blue arrows show the number of parameter values being passed as seed values to be used in new parameters at the next layer of learning. Overlapping layers are colored with CILB layers in orange, PCLL in green, and PLLR in yellow. Descriptions of the layers are provided in Appendix A.

[5]As measured on an Intel(R) Xeon(R) CPU E31270 @ 3.40GHz.

[6]Videos of some of the behaviors being learned are available at
http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/overlappingLayeredLearning.html

5.1. Getup and Walking using PLLR

The UT Austin Villa team employs an omnidirectional walk engine using a double inverted pendulum model to control walking. The walk engine has many parameters that need to be optimized in order to create a stable and fast walk including the length and frequency of steps as well as center of mass offsets. Full details of the walk engine and its parameters are given in Appendix B. Instead of having a single set of parameters for the walk engine, which in previous work we found to limit performance [1], walking is broken up into different subtasks for each of which a set of walk engine parameters is learned.

Before optimizing parameters for the walk engine, "getup" behaviors are learned so that if the robot falls over it is able to stand back up and start walking again. Getup behaviors are necessary for faster learning during walk optimizations, as without the ability to get up after falling, a walk optimization task would have to be terminated as soon as the robot fell over. There are two such behaviors for getting up: *GetUp_Front_Primitive* for standing up from lying face down and *Getup_Back_Primitive* for standing up from lying face up. Each getup behavior consists of different joint angle movements that form a fixed series of poses. The poses themselves are specified in a parameterized skill description language. Information about the skill description language is provided in Appendix D. During learning, getup behaviors are evaluated based on how quickly the robot is able to stand up [17].

After the getup primitive behaviors are learned, we start optimizing the first walk engine parameter set in the next layer of learning. This is the *Walk_GoToTarget* behavior which is used for walking to different target locations on the soccer field. Learning this walk is accomplished by having the robot walk to a series of target points on the field in the form of an obstacle course, and the robot is rewarded for how quickly it can complete the obstacle course while being penalized for every time it falls over. After the *Walk_GoToTarget* parameter set is learned and fixed, the *Walk_Sprint* walk engine parameter set used to quickly walk straight forward—to targets within 15° of the robot's current heading—is then optimized in the third layer of learning using the same obstacle course optimization task. Full details of how these walk parameter sets are optimized can be found in Appendix C. Note that the optimization tasks used for learning different walk parameter sets purposely transition between all previously learned walk parameter sets and the current one being learned to ensure that the robot can smoothly transition between them without losing stability.

After learning both the *Walk_GoToTarget* and *Walk_Sprint* walk engine parameter sets, we re-optimize the getups by learning the *GetUp_Front_Behavior* and *GetUp_Back_Behavior* behaviors in the fourth layer of learning. *GetUp_Front_Behavior* and *GetUp_Back_Behavior* are overlapping layered learning behaviors as they contain the same parameters as the previously learned *GetUp_Front_Primitive* and *GetUp_Back_Primitive* behaviors respectively. The getup behavior parameters are re-optimized from their primitive behavior values through the same optimization as the getup primitives, but with the addition that right after completing a getup behavior the robot is asked to walk in different directions and is penalized if it falls over while trying to do so. Unlike the getup primitive behaviors, which were learned in isolation, the relearned getup behaviors are stable transitioning from standing up and then almost immediately walking. One might think that the walk parameter sets learned would be stable transitioning from the original learned getups due to the getup primitive behaviors being used in the walk parameter optimization tasks, however this is not always the case. During learning, walks become stable such that toward the end of optimizing a walk parameter set the robot almost never falls, and thus rarely uses the getup primitive behaviors. Relearning the getup behaviors is an example of previous learned layer refinement (PLLR).

5.2. Kicking using CILB

Four primitive kick behaviors were learned by the 2014 UT Austin Villa team (*Kick_Long_Primitive*, *Kick_Low_Primitive*, *Kick_High_Primitive*, and *Kick_Fast_Primitive*). Each kick primitive, or kicking motion, was learned by placing the robot at a fixed position behind the ball and having it optimize joint angles for a fixed set of key motion frames defined by a skill description language. Information about the skill description language is provided in Appendix D. Note that initial attempts at learning kicks directly with the walk, instead of learning kick primitives independently, proved to be too difficult due to the variance in stopping positions of the walk as the robot approached to kick the ball.

While the kick primitive behaviors work quite well when the robot is placed in a standing position behind the ball, they are very hard to execute when the robot tries to walk up to the ball and kick it. One reason for this difficulty is that when the robot approaches the ball to kick it using the *Walk_ApproachToKick* walk parameter set—used for approaching and stopping at a precise position behind the ball before executing a kick—the precise offset position from the ball that the kick primitives were optimized to work with do not match that of the position the robot stops at after walking up to the ball. In order to allow the robot to transition from walking to kicking, full kick behaviors for all the kicks are optimized (*Kick_Long_Behavior*, *Kick_Low_Behavior*, *Kick_High_Behavior*, *Kick_Fast_Behavior*). Each full kick behavior is learned by having the robot walk up to the ball and attempt to kick it from different starting positions—as opposed to having the robot just standing behind the ball as was done when optimizing the kick primitive behaviors.

The full kick behaviors are overlapping layered learning behaviors because they re-optimize previous learned parameters. In the case of *Kick_Fast_Behavior*, only the x and y kick primitive offset position parameters from the ball, which is the target position for the walk to reach for the kick to be executed, are re-optimized. The fast kick is quick enough that it almost immediately kicks the ball after transitioning from walking, and thus just needs to be in the correct position near the ball to do so. A comparison of overlapping layered learning paradigms for learning *Kick_Fast_Behavior* is shown in Figure 4. The above overlapping layered approach of first independently learning the walk approach and kick, and then learning the two position parameters, does better than both the sequential layered learning approach where all kick parameters are learned after freezing the approach, and the concurrent layered learning approach where both approach and kick parameters are learned simultaneously.

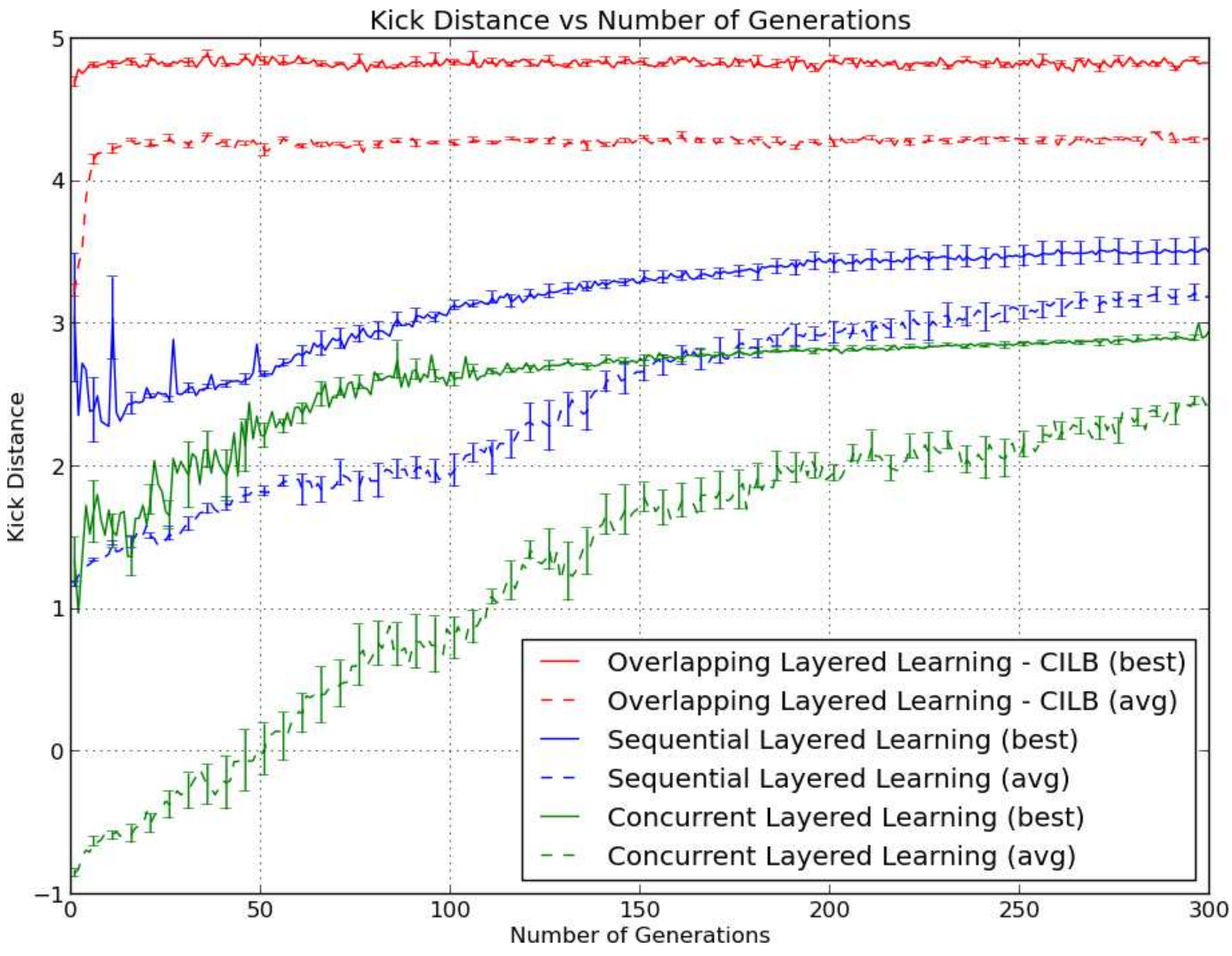

Figure 4. Performance of different layered learning paradigms across generations of CMA-ES when optimizing *Kick_Fast_Behavior*. Results are averaged across five optimization runs and error bars show the standard error.

For the other full kick behaviors, all kick parameters from their respective kick primitive behaviors are re-optimized. Unlike the fast kick, there is at least a one second delay between stopping, walking, and kicking the ball, during which the robot can easily become destabilized and fall over. By opening up all kicking parameters the robot has the necessary freedom to learn kick motions that maintain its stability between stopping after walking and making contact with the ball. Learning the kick behaviors by combining them with the *Walk_ApproachToKick* behavior for walking up to the ball are all examples of combining independently learned behaviors (CILB).

5.3. KickOff using both CILB and PCLL

For kickoffs the robot is allowed to "teleport" itself to a starting position next to the ball before kicking it, and thus does not need to worry about walking up to the ball. Scoring directly off a kickoff is not allowed, however, as another robot must first touch the ball before it goes into the opponent's goal. In order to score on a kickoff we perform a multiagent task where one robot touches the ball before another kicks it.

The first behavior optimized for scoring off the kickoff is *KickOff_Kick_Primitive* in which a robot kicks the ball from the middle of the field. The robot is rewarded for kicking the ball as high and as far as possible as long as the ball crosses the goal line below the height of the goal. In parallel a behavior for another robot is learned to lightly touch the ball called *KickOff_Touch_Primitive*. Here a robot is rewarded for touching the ball lightly and, after ensuring that the robot has made contact with the ball, that the ball moves as little as possible. Finally an overlapping layered behavior called *KickOff_Kick_Behavior* is learned which re-optimizes x, y, and θ angle offset positions from the ball from both the *KickOff_Kick_Primitive* and *KickOff_Touch_Primitive* behaviors. Re-optimizing these positioning parameters together is important so that the robots do not accidentally collide with each other and also so that the kicking robot is at a good position to kick the ball after the first agent touches it. Learning *KickOff_Kick_Behavior* is another example of combining independently learned behaviors (CILB).

In addition to the positioning parameters of both robots being re-optimized for *KickOff_Kick_Behavior*, a new parameter that determines the time at which the first robot touches the ball is optimized. This synchronized timing parameter is necessary so that the robots are synced with each other and the kicking robot does not accidentally try to kick the ball before the first robot has touched it. As a new parameter is optimized along with a subset of previously learned parameters, learning *KickOff_Kick_Behavior* is also an example of partial concurrent layered learning (PCLL).

Further information about the kickoff, including how a seed for the kick was learned through observation, can be found in [18].

6. Results and Analysis

At the 2014 RoboCup 3D simulation competition—the first year the UT Austin Villa team used overlapping layered learning—UT Austin Villa finished first among 12 teams while scoring 52 goals and conceding none across 15 games. Considering that most of the team's strategy layer—including team formations using a dynamic role assignment and formation positioning system [19]—remained unchanged from that of the previous year's second place finishing team, a key component to the 2014 team's improvement and success at the competition was the new approach incorporating overlapping layered learning used to learn the team's low level behaviors.

After every RoboCup competition teams are required to release the binaries that they used during the competition. In order to analyze the performance of the different components of our overlapping layered learning approach before the 2014 competition, we played 1000 games with different versions of the UT Austin Villa team against each of the top three teams from the RoboCup 2013 competition. The following subsections provide analysis of game results when turning on and off the kickoff and kicking components learned though an overlapping layered learning approach. Additionally, to demonstrate the generality of our overlapping layered learning approach, we provide data that isolates the performance of our complete overlapping layered learning approach applied to different robot models.

6.1. Overall Team Performance

Table 2 shows the average goal difference across all games against each opponent achieved by the complete 2014 UT Austin Villa team. Against all opponents the team had a significantly positive goal difference, and in fact out of the 3000 games played the team only lost one game (to AustinVilla2013). These game results show the effectiveness of the team's overlapping layered learning approach in dramatically improving the performance of the team from the previous year in which the team achieved second place at the competition—the 2014 team is able to beat the previous year's team by an average of 1.525 goals.

Table 2. Full game results, averaged over 1000 games. Each row corresponds to one of the top three finishing teams at RoboCup 2013. Entries show the average goal difference achieved by the 2014 UT Austin Villa team versus the given opponent team. Values in parentheses are the standard error. Total number of wins, losses, and ties across all games was 2852, 1, and 147 respectively.

Opponent	Average Goal Difference
Apollo3D	2.726 (0.036)
AustinVilla2013	1.525 (0.032)
FCPortugal	3.951 (0.049)

Data provided in Appendix E, showing the overall team's performance when playing against the released 2014 teams' binaries, corroborates the overall team's strong performance. When playing 1000 games against each of the eleven 2014 opponents UT Austin Villa did not lose a single game out of the 11,000 played, and had at least an average goal difference of 2 against every opponent. Additionally, bolstered by the team's strong set of skills developed through overlapping layered learning techniques, UT Austin Villa won all games it played at the RoboCup 2015 [20], 2016 [21], and 2017 competitions.

6.2. *KickOff Performance*

To isolate the performance of the learned multiagent behavior to score off the kickoff, we disabled this feature and instead just had the robot taking the kickoff kick the ball toward the opponent's goal to a position as close as possible to one of the goal posts without scoring. Table 3 shows results from playing against the top three teams at RoboCup 2013 without attempting to score on the kickoff.

Table 3. Full game results, averaged over 1000 games. Each row corresponds to one of the top three finishing teams at RoboCup 2013. Entries show the average goal difference achieved by a version of the 2014 UT Austin Villa team *not attempting to score on a kickoff* versus the given opponent team. Values in parentheses are the standard error. Total number of wins, losses, and ties across all games was 2644, 5, and 351 respectively.

Opponent	Average Goal Difference
Apollo3D	2.059 (0.038)
AustinVilla2013	1.232 (0.032)
FCPortugal	3.154 (0.046)

By comparing results in Table 3 to that of Table 2 we see a significant drop in performance when not attempting to score on kickoffs. This result is not surprising as we found that the kickoff was able to score around 90% of the time against Apollo3D and FCPortugal, and over 60% of the time against the 2013 version of UT Austin Villa. The combination of using both CILB and PCLL overlapping layered learning led to a large boost to the team's performance.

6.3. *Kicking Performance*

To isolate the performance of kicking learned through an overlapping layered learning approach we disable all kicking (except for on kickoffs where we once again have a robot kick the ball as far as possible toward the opponent's goal without scoring) and used an "always dribble" behavior. Data from playing against the top three teams at the RoboCup 2013 competition when only dribbling is shown in Table 4.

Table 4. Full game results, averaged over 1000 games. Each row corresponds to one of the top three finishing teams at RoboCup 2013. Entries show the average goal difference achieved by a version of the 2014 UT Austin Villa team *using a dribble only strategy* versus the given opponent team. Values in parentheses are the standard error. Total number of wins, losses, and ties across all games was 2480, 15, and 505 respectively.

Opponent	Average Goal Difference
Apollo3D	1.790 (0.033)
AustinVilla2013	0.831 (0.023)
FCPortugal	1.593 (0.028)

Here we see another significant drop in performance when comparing Table 4 to Table 3. Kicking provided a large gain in performance, nearly doubling the average goal difference against FCPortugal, compared to only dribbling. This result is in stark contrast to when UT Austin Villa won the 2011 RoboCup competition, in which the team tried to incorporate kicking skills without using an overlapping layered learning approach, and found that kicking actually hurt the performance of the team [22].

6.4. Different Robot Models

At the 2014 RoboCup competition teams were given the option of using five different robot types with the requirement that at least three different types of robots must be used on a team and no more than seven of any one type. The five types of robots available were the following:

Type 0: Standard Nao model

Type 1: Longer legs and arms

Type 2: Quicker moving feet

Type 3: Wider hips and longest legs and arms

Type 4: Added toes to foot

We applied our overlapping layered learning approach for learning behaviors to each of the available robot types. Game data from playing against the top three teams at RoboCup 2013 is provided in Table 5 for each robot type.

Table 5. Full game results, averaged over 1000 games. Each row corresponds to one of the top three finishing teams at RoboCup 2013. Entries show the average goal difference achieved by a version of the 2014 UT Austin Villa team *using different heterogeneous robot types* versus the given opponent team.

	Avg. Goal Difference per Robot Type				
Opponent	**Type 0**	**Type 1**	**Type 2**	**Type 3**	**Type 4**
Apollo3D	1.788	1.907	1.892	1.524	2.681
AustinVilla2013	0.950	0.858	1.152	0.613	1.104
FCPortugal	2.381	2.975	3.331	2.716	3.897

While there are some differences in performance between the different robot types, likely due to the differences in their body models, all of the robot types are able to reliably beat the top teams from the 2013 RoboCup competition. This shows the efficacy of our overlapping layered learning approach and its ability to generalize to different robot models. During the 2014 competition the UT Austin Villa team used seven type 4 robot models as they showed the best performance, two type 0 robot models as they displayed the best performance on kickoffs, and one each of the type 1 and type 3 robot models as they were the fastest at walking [3].

6.5. Summary of Results

Results from the data we collected in Section 6.1 and Appendix E provide evidence of the 2014 team's overall improvement and success gained by incorporating overlapping layered learning into the approach used to learn the team's low level behaviors. In particular, kicks learned through overlapping layered learning techniques boosted the performance of the team as shown by the data isolating the kicks' contributions to game results in Sections 6.2 and 6.3. Finally, the generality of our learning approach is demonstrated by its success when applied to different robot models in Section 6.4.

7. Related Work

Within RoboCup soccer domains there has been previous work in using layered learning approaches to learn complex agent behaviors. Stone used layered learning to train three behaviors for agents in the RoboCup 2D simulation domain and specified an additional two that could be learned as well [7]. Gustafson et al. used two layers of learning when applying genetic programming to the keep away subtask within the RoboCup 2D simulation domain [23]. Whiteson and Stone later introduced concurrent layered learning within the same keepaway domain during which four layers were learned. Cherubini et al. used layered learning for teaching AIBO robots soccer skills that included six behaviors [24]. Layered learning has also been applied to non-RoboCup domains such as Boolean logic [25], non-playable characters in video games [26], and concept synthesis in road traffic simulations [27]. To the best of our knowledge our overlapping layered learning approach, containing 19 learned behaviors, has more than three times the behaviors of any previous layered learning systems.

Work by Mondesire has discussed the concept of learned layers overlapping, and focuses on a concern of information needed to perform a subtask being lost or forgotten as it is replaced during the learning of a task in a subsequent layer [28]. Our work differs in that we are not concerned with the performance of individual subtasks in isolation, but instead are interested in maximizing the performance of subtasks when they are combined.

Our use of CMA-ES as the optimization algorithm allows for a high degree of parallelization during learning, and recent work by Salimans et al. has corroborated the success and scalability of evolution strategies when applied to reinforcement learning tasks [29]. While our implementation of overlapping layered learning uses CMA-ES for learning the component skills, its hierarchical nature also bears some resemblance to, and shares some motivation with, classic approaches to hierarchical reinforcement learning for learning complex behaviors. Most hierarchical reinforcement learning approaches use gated behaviors: a gating function decides which among a collection of behaviors should be executed, with each behavior mapping potential environment states to low-level actions [30]. In these approaches, the behaviors and gating function are all control tasks with similar inputs and actions (sometimes abstracted). Layered learning, on the other hand, allows for conceptually different tasks, such as a soccer player evaluating the probability of a pass being successful and moving to get open for a pass [8], at the different layers.

One widely used approach to hierarchical reinforcement learning is the MAXQ algorithm [31]. The MAXQ algorithm learns at all levels of a hierarchy simultaneously. MAXQ converges to a recursively optimal policy in which learned subtasks are locally optimal as opposed to being hierarchically optimal—the learned substasks' policies are not necessarily optimal when taking into account transitions to and from other subtasks. Also, unlike MAXQ, layered learning allows for the flexibility of using different machine learning algorithms at each level of the hierarchy.

Another popular approach to hierarchical reinforcement learning is the options framework [32]. Options, or temporally extended actions, can be thought of as learned policies with initiation and termination conditions used to complete subtasks. Layers in our behavior hierarchy can be seen as options in the sense that they are policies that run for limited periods of time within the overall behavior. In the context of robot soccer one could learn two separate options for both walking and kicking a ball, and—after finding out that the robot is unable to transition from the walk option to the kick option without falling—then learn a third option as a bridge to transition between walking and kicking that stops and stabilizes the robot before it kicks the ball. However, it could be inefficient and slow to need to execute three option behaviors to walk to and kick the ball. Instead, it may be possible to learn a better policy through overlapping layered learning and CILB that is more efficient and faster to execute: a policy consisting of a new behavior that combines the previously learned walking and kicking subtask behaviors without needing to stop and stabilize the robot before kicking the ball.

Progressive neural networks [33] follow some of the concepts of layered learning. These networks have been successfully used to learn policies for a series of related reinforcement learning tasks. Progressive neural networks are sequentially trained on individual tasks, with only the weights for a single column of the network being learned for the current task the network is being trained on. After each task is learned the weights of the neural network are frozen, and a new column with open weights is added to the network before learning the next task. The outputs from the layers of the previously learned task's network column is used as inputs to the layers of the current task's network column that is being learned. The architecture of progressive neural networks—where the output of a previously learned task is used as input to the next task being learned—is similar in spirit to layered learning. The architecture differs from overlapping layered learning, however, in that learned weights of the network are never unfrozen and relearned. Furthermore, progressive neural networks leverage task similarities to learn successive tasks through transfer learning [34], in which they focus on learning policies for tasks that are performed in isolation from each other, where as overlapping layered learning is better suited for developing subtasks that work well together and can smoothly transition between each other.

8. Summary and Discussion

This article introduces overlapping layered learning methodologies and motivates general scenarios for their use. The article also includes a detailed description and experimental analysis of the extensive overlapping layered learning approach used by the UT Austin Villa team in winning the 2014 RoboCup 3D simulation competition.[7] Furthermore, the complete learning process is repeated on four different robot body types, showcasing its generality as a paradigm

[7] Team video highlights from the competition can be found on the team's homepage at http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/#2014

for efficient behavior learning. While first introduced in 2014, overlapping layered learning has continued to be used successfully by the UT Austin Villa team in all years since then as well.

Currently the overlapping layered learning methodologies require a person to select which parameters to freeze and leave open during each successive layer of learning. Additionally, learning of complex skills is manually segmented into different layers of learning. Future work in the area of layered learning includes the automated determination of appropriate subtasks for layered learning, as well as automated identification and selection of useful layer overlap or "seams" to use with overlapping layered learning methodologies. If these selection processes can be automated it would lessen the burden and potential need for someone with expert domain knowledge when performing optimizations.

A base code release of the 2015 UT Austin Villa RoboCup 3D simulation agent [35], which includes hooks for optimizing the skills and behaviors presented in this work, can be found at http://github.com/LARG/utaustinvilla3d.

Acknowledgments

We thank Mike Depinet for his contributions to learning both kicks and the behavior to approach the ball before a kick. This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. LARG research is supported in part by NSF (CNS-1330072, CNS-1305287, IIS-1637736, IIS-1651089), ONR (21C184-01), AFOSR (FA9550-14-1-0087), Raytheon, and Lockheed Martin. Peter Stone serves on the Board of Directors of, Cogitai, Inc. The terms of this arrangement have been reviewed and approved by the University of Texas at Austin in accordance with its policy on objectivity in research.

[1] P. MacAlpine, S. Barrett, D. Urieli, V. Vu, P. Stone, Design and optimization of an omnidirectional humanoid walk: A winning approach at the RoboCup 2011 3D simulation competition, in: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI), 2012.

[2] P. MacAlpine, M. Depinet, P. Stone, UT Austin Villa 2014: RoboCup 3D simulation league champion via overlapping layered learning, in: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI), Vol. 4, 2015, pp. 2842–48.

[3] P. MacAlpine, M. Depinet, J. Liang, P. Stone, UT Austin Villa: RoboCup 2014 3D simulation league competition and technical challenge champions, in: RoboCup-2014: Robot Soccer World Cup XVIII, Lecture Notes in Artificial Intelligence, Springer Verlag, Berlin, 2015.

[4] S. P. Singh, Transfer of learning by composing solutions of elemental sequential tasks, Machine Learning 8 (3-4) (1992) 323–339.

[5] S. Whitehead, J. Karlsson, J. Tenenberg, Learning multiple goal behavior via task decomposition and dynamic policy merging, in: Robot learning, Springer, 1993, pp. 45–78.

[6] S. Whiteson, N. Kohl, R. Miikkulainen, P. Stone, Evolving soccer keepaway players through task decomposition, Machine Learning 59 (1-2) (2005) 5–30.

[7] P. Stone, Layered learning in multiagent systems: A winning approach to robotic soccer, MIT Press, 2000.

[8] S. Whiteson, P. Stone, Concurrent layered learning, in: Second Int. Joint Conf. on Autonomous Agents and Multiagent Systems, ACM Press, New York, NY, 2003, pp. 193–200.

[9] S. Behnke, M. Schreiber, J. Stückler, R. Renner, H. Strasdat, See, walk, and kick: Humanoid robots start to play soccer, in: Proc. of the Sixth IEEE-RAS Int. Conf. on Humanoid Robots (Humanoids 2006), IEEE, 2006, pp. 497–503.
URL http://www.ais.uni-bonn.de/nimbro/papers/Humanoids06_Behnke.pdf

[10] M. Riedmiller, T. Gabel, R. Hafner, S. Lange, Reinforcement learning for robot soccer, Autonomous Robots 27 (1) (2009) 55–73.
URL http://ml.informatik.uni-freiburg.de/_media/publications/gr_09.pdf

[11] S. Kalyanakrishnan, P. Stone, Learning complementary multiagent behaviors: A case study, in: RoboCup 2009: Robot Soccer World Cup XIII, 2010.
URL http://www.cs.utexas.edu/~shivaram/papers/ks_rs_2009.pdf

[12] J. Boedecker, M. Asada, Simspark–concepts and application in the robocup 3d soccer simulation league, Autonomous Robots (2008) 174–181.

[13] Y. Xu, H. Vatankhah, Simspark: An open source robot simulator developed by the robocup community, in: S. Behnke, M. Veloso, A. Visser, R. Xiong (Eds.), RoboCup 2013: Robot World Cup XVII, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 632–639. doi:10.1007/978-3-662-44468-9_59.

[14] N. Hansen, S. Müller, P. Koumoutsakos, Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es), Evolutionary Computation 11 (1) (2003) 1–18.

[15] D. Urieli, P. MacAlpine, S. Kalyanakrishnan, Y. Bentor, P. Stone, On optimizing interdependent skills: A case study in simulated 3d humanoid robot soccer, in: Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011), Vol. 2, IFAAMAS, 2011, pp. 769–776.

[16] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the Condor experience., Concurrency - Practice and Experience 17 (2-4) (2005) 323–356.

[17] P. MacAlpine, N. Collins, A. Lopez-Mobilia, P. Stone, UT Austin Villa: RoboCup 2012 3D simulation league champion, in: RoboCup-2012: Robot Soccer World Cup XVI, Lecture Notes in Artificial Intelligence, Springer Verlag, Berlin, 2013.

[18] M. Depinet, P. MacAlpine, P. Stone, Keyframe sampling, optimization, and behavior integration: Towards long-distance kicking in the robocup 3d simulation league, in: RoboCup-2014: Robot Soccer World Cup XVIII, Lecture Notes in Artificial Intelligence, Springer Verlag, Berlin, 2015.

[19] P. MacAlpine, E. Price, P. Stone, SCRAM: Scalable collision-avoiding role assignment with minimal-makespan for formational positioning, in: Proc. of the Twenty-Ninth AAAI Conf. on Artificial Intelligence (AAAI), 2015.

[20] P. MacAlpine, J. Hanna, J. Liang, P. Stone, UT Austin Villa: RoboCup 2015 3D simulation league competition and technical challenges champions, in: L. Almeida, J. Ji, G. Steinbauer, S. Luke (Eds.), RoboCup-2015: Robot Soccer World Cup XIX, Lecture Notes in Artificial Intelligence, Springer Verlag, Berlin, 2016.

[21] P. MacAlpine, P. Stone, UT Austin Villa: RoboCup 2016 3D simulation league competition and technical challenges champions, in: S. Behnke, D. D. Lee, S. Sariel, R. Sheh (Eds.), RoboCup 2016: Robot Soccer World Cup XX, Lecture Notes in Artificial Intelligence, Springer, 2016.

[22] P. MacAlpine, D. Urieli, S. Barrett, S. Kalyanakrishnan, F. Barrera, A. Lopez-Mobilia, N. Ştiurcă, V. Vu, P. Stone, UT Austin Villa 2011: A champion agent in the RoboCup 3D soccer simulation competition, in: Proc. of 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS), 2012.

[23] S. M. Gustafson, W. H. Hsu, Layered learning in genetic programming for a cooperative robot soccer problem, Springer, 2001.

[24] A. Cherubini, F. Giannone, L. Iocchi, Layered learning for a soccer legged robot helped with a 3d simulator, in: RoboCup 2007: Robot Soccer World Cup XI, Vol. 5001 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 385–392. doi:10.1007/978-3-540-68847-1_39.
URL http://dx.doi.org/10.1007/978-3-540-68847-1_39

[25] D. Jackson, A. Gibbons, Layered learning in boolean gp problems, in: Genetic Programming, Vol. 4445 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 148–159. doi:10.1007/978-3-540-71605-1_14.
URL http://dx.doi.org/10.1007/978-3-540-71605-1_14

[26] S. Mondesire, R. Wiegand, Evolving a non-playable character team with layered learning, in: Computational Intelligence in Multicriteria Decision-Making (MDCM), 2011 IEEE Symposium on, 2011, pp. 52–59. doi:10.1109/SMDCM.2011.5949283.

[27] S. H. Nguyen, J. Bazan, A. Skowron, H. S. Nguyen, Layered learning for concept synthesis, in: Transactions on Rough Sets I, Springer, 2004, pp. 187–208.

[28] S. C. Mondesire, Complementary layered learning, Ph.D. thesis, Univ. of Central Florida Orlando, Florida (2014).

[29] T. Salimans, J. Ho, X. Chen, I. Sutskever, Evolution strategies as a scalable alternative to reinforcement learning, arXiv preprint arXiv:1703.03864.

[30] L. P. Kaelbling, M. L. Littman, A. W. Moore, Reinforcement learning: A survey, Journal of artificial intelligence research 4 (1996) 237–285.

[31] T. G. Dietterich, Hierarchical reinforcement learning with the maxq value function decomposition, J. Artif. Intell. Res.(JAIR) 13 (2000) 227–303.

[32] R. S. Sutton, D. Precup, S. Singh, Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning, Artificial intelligence 112 (1-2) (1999) 181–211.

[33] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, R. Hadsell, Progressive neural networks, arXiv preprint arXiv:1606.04671.

[34] M. E. Taylor, P. Stone, Transfer learning for reinforcement learning domains: A survey, Journal of Machine Learning Research 10 (Jul) (2009) 1633–1685.

[35] P. MacAlpine, P. Stone, UT Austin Villa robocup 3D simulation base code release, in: S. Behnke, D. D. Lee, S. Sariel, R. Sheh (Eds.), RoboCup 2016: Robot Soccer World Cup XX, Lecture Notes in Artificial Intelligence, Springer Verlag, Berlin, 2016.

[36] C. Graf, A. Härtl, T. Röfer, T. Laue, A robust closed-loop gait for the standard platform league humanoid, in: Proc. of the 4th Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS Int. Conf. on Humanoid Robots, 2009, pp. 30 – 37.

A. Learned Behavior Layers

For all layers of learned behaviors the CMA-ES [14] algorithm is used as the ML algorithm (M). All input feature vector ($\vec{F}$) for learned behavior layers include the position of each of the robots' 22 joints, as well as the robots' three dimensional (x, y, z) accelerometer and gyroscope measurements. Any behavior in which a kick is learned also takes in as input the x and y position of the ball relative to the robots. The output (O) for all behavior layers are current target positions for each of the robots' joints. A diagram of how all the layers connect with each other can be seen in Figure 3 within Section 5.

$L_{1,1}$ **: Getup_Front_Primitive:** The robot learns a behavior to stand up when starting from lying on its front.

$T_{1,1}$**:** The robot is forced to fall on its front from a standing position and then attempts to get up. The robot is then given a negative return value equal to the time it remains in a fallen over state as measured over 4 seconds. The objective function used during optimization is the following where time is in seconds:

$$f_{\text{getup\_front\_primitive}} = -\text{timeNotStanding}$$

See [17] for more details about the training task.

$M_{1,1}$**:** Learning was performed across 200 generations of CMA-ES with a population size of 150.

$H_{1,1}$**:** The learned policy consists of 9 parameters specifying a fixed series of poses for a getup motion defined by our skill description language. Information about the skill description language is provided in Appendix D. Initial parameter values were seeded with those from a hand-coded policy.

$L_{1,2}$ **: Getup_Back_Primitive:** The robot learns a behavior to stand up when starting from lying on its back.

$T_{1,2}$**:** The robot is forced to fall on its back from a standing position and then attempts to get up. The robot is then given a negative return value equal to the time it remains in a fallen over state as measured over 4 seconds. The objective function used during optimization is the following where time is in seconds:

$$f_{\text{getup\_back\_primitive}} = -\text{timeNotStanding}$$

See [17] for more details about the training task.

$M_{1,2}$**:** Learning was performed across 200 generations of CMA-ES with a population size of 150.

$H_{1,2}$**:** The learned policy consists of 26 parameters specifying a fixed series of poses for a getup motion defined by our skill description language. Information about the skill description language is provided in Appendix D. Initial parameter values were seeded with those from a hand-coded policy.

$L_{1,3}$ **: KickOff_Touch_Primitive:** Single robot behavior where the robot learns to lightly touch the ball resulting in little ball motion.

$T_{1,3}$**:** The robot attempts to lightly touch the ball and is given lower return values the more the ball moves. If the robot falls over, fails to touch the ball, or touches the ball more than once it is given a negative return value. The objective function used during optimization is the following where distance is in meters:

$$f_{\text{touch}} = \begin{cases} -1 & : \text{Failure} \\ 10 - \text{distBallTraveled} & : \text{Otherwise} \end{cases}$$

See [18] for full details of the training task.

$M_{1,3}$**:** Learning was performed across 100 generations of CMA-ES with a population size of 150.

$H_{1,3}$**:** The learned policy consists of 12 parameters specifying a fixed series of poses for a ball touch motion defined by our skill description language. Information about the skill description language is provided in Appendix D. Parameters for the x, y, and θ offset standing position of the robot from the ball are also learned. Initial parameter values were seeded with those from a hand-coded policy.

***$L_{1,4}$* : KickOff_Kick_Primitive:** Single robot kick behavior where the robot learns a kick that scores on a kickoff from a motionless ball.

***$T_{1,4}$*:** The robot attempts to kick the ball on a kickoff directly into the opponent's goal. The robot is given higher return values for both kicks that travel closer to the opponent's goal, and for kicks that travel farther distances while remaining above the height of opponent robots. If the robot fails to kick the ball it is given a negative return value, and if it kicks the ball out of bounds—misses the goal—it receives a return value of 0. The objective function used during optimization is the following where all distances are in meters:

$$f_{\text{kick\_kickoff}} = \begin{cases} -1 & : \text{Failure} \\ 0 & : \text{Missed goal} \\ 100 + \text{distBallForward} + 2 * \text{distBallInAir} & : \text{Otherwise} \end{cases}$$

See [18] for full details of the training task.

***$M_{1,4}$*:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

***$H_{1,4}$*:** The learned policy consists of 59 parameters specifying a fixed series of poses for a kick motion defined by our skill description language. Information about the skill description language is provided in Appendix D. Parameters for the x, y, and θ offset standing position of the robot from the ball are also learned. Initial parameter values were seeded with those sampled from an observed kick as described in [18].

***$L_{1,5}$* Kick_Fast_Primitive:** The robot learns a short but fast to execute kick starting from a standing position behind the ball. The robot is also expected to be stable and still standing after the kick.

***$T_{1,5}$*:** The robot attempts to quickly kick the ball as far as possible from a standing position behind the ball. The robot is given higher return values for longer kicks, is penalized if the kick takes too long to execute (greater than 0.25 seconds), and is given a negative return value if it fails to kick the ball or falls over while doing so. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_fast\_primitive}} = \begin{cases} -1 & : \text{Failure} \\ -1 & : \text{Robot fell over} \\ \text{distBallForward} - \max(\text{kickTime} - 0.25, 0) & : \text{Otherwise} \end{cases}$$

***$M_{1,5}$*:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

***$H_{1,5}$*:** The learned policy consists of 33 parameters specifying a fixed series of poses for a fast kick motion defined by our skill description language. Information about the skill description language is provided in Appendix D. Parameters for the x and y offset standing position of the robot from the ball are also learned. Initial parameter values were seeded with those from a hand-coded policy.

***$L_{2,1}$* : Walk_GoToTarget:** The robot learns a walk for moving to general target positions on the field.

***$T_{2,1}$*:** The training task consists of an obstacle course in which the robot tries to navigate to a variety of target positions on the field. Each target is active, one at a time for a fixed period of time, which varies from one target to the next, and the robot is rewarded based on its distance traveled toward the active target. To promote stability, the robot is given a penalty if it falls over. After falling the robot executes one of the getup behaviors learned in $L_{1,1}$ and $L_{1,2}$—*Getup_Front_Primitive* or *Getup_Back_Primitive* respectively—so that it can stand up and continue the walk training task. Full details of the training task, known as the goToTarget task, are provided in Appendix C.1.

***$M_{2,1}$*:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

***$H_{2,1}$*:** The learned policy consists of 27 parameters for a walk engine described in Appendix B. The maxStep_x and maxStep_y parameters in Table B.6 are learned as a single value, however, as we found this advantageous in ensuring that the learned policy would not overly favor walking in the x direction over the y direction. Initial walk engine parameter values were seeded with those from a hand-coded policy used on physical robots as described in [1].

$L_{2,2}$ **: KickOff_Kick_Behavior:** A two robot behavior is learned for scoring on a kickoff with one robot lightly touching ball before the other robot kicks the ball in the goal.

$T_{2,2}$**:** One robot attempts to lightly touch the ball at the beginning of a kickoff, after which the second robot attempts to kick the ball in the goal. During the task the touch and kick robots use the previously learned *KickOff_Touch_Primitive* and *KickOff_Kick_Primitive* respectively. The task is evaluated with the same objective function as is used in the training task $T_{1,4}$, except that a negative return value is also given if the first robot fails to touch the ball before the second robot kicks the ball. The objective function used during optimization is the following where all distances are in meters:

$$f_{\text{kickoff}} = \begin{cases} -1 & : \text{ First robot failed touch} \\ -1 & : \text{ Second robot failed kick} \\ 0 & : \text{ Missed goal} \\ 100 + \text{distBallForward} + 2 * \text{distBallInAir} & : \text{ Otherwise} \end{cases}$$

See [18] for full details of the training task.

$M_{2,2}$**:** Learning was performed across 100 generations of CMA-ES with a population size of 150.

$H_{2,2}$**:** The learned policy consists of the x, y, and θ offset standing position parameters of both the robots from the ball that are part of $H_{1,3}$ and $H_{1,4}$. This learning is an example of CILB as $H_{1,3} \in L_{2,2}$, $H_{1,4} \in L_{2,2}$, and $\{H'_{1,3} \cup H'_{1,4}\} \subset H_{2,2}$ where $H'_{1,3}$ and $H'_{1,4}$ are the subsets of standing position parameters of $H_{1,3}$ and $H_{1,4}$ respectively. An additional synchronized timing parameter for how long the first robot should wait before touching the ball is also learned so that the second robot does not accidentally try and kick the ball before the first robot has touched it. This new added parameter makes this learned behavior also an example of PCLL as $\{H'_{1,3} \cup H'_{1,4}\} \subset H_{2,2}$ instead of just $\{H'_{1,3} \cup H'_{1,4}\} = H_{2,2}$.

$L_{2,3}$ **: Kick_Long_Primitive:** The robot learns a kicking motion to kick the ball a long distance starting from a standing position behind the ball.

$T_{2,3}$**:** The robot attempts to kick the ball as far as possible from a standing position behind the ball. The robot is given higher return values for longer kicks, is penalized if the kick takes too long to execute (greater than 2 seconds), and is given a negative return value if it fails to kick the ball. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_long\_primitive}} = \begin{cases} -1 & : \text{ Failure} \\ \text{distBallForward} - \max(\text{kickTime} - 2, 0) & : \text{ Otherwise} \end{cases}$$

$M_{2,3}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$H_{2,3}$**:** The learned policy consists of 70 parameters specifying a fixed series of poses for a long kick motion defined by our skill description language. Information about the skill description language is provided in Appendix D. Parameters learned in $h_{1,4}$ for the *KickOff_Kick_Primitive* are used as seed values for an initial policy ($h_{1,4} \dashrightarrow H_{2,3}$), and added parameters for speeding up the kick are also learned. Parameters for the x, y, and θ offset standing position of the robot from the ball are learned as well.

$L_{3,1}$ **: Walk_Sprint:** The robot learns a walk for quickly walking in the forward direction.

$T_{3,1}$**:** The same goToTarget training task is performed as in $L_{2,1}$, however the robot only uses the walk it is learning when its orientation is within 15° of its target. During the remainder of the training task the robot is using the learned behavior from $L_{2,1}$ thus ensuring that the walk being learned can transition from/to the previously learned *Walk_GoToTarget* behavior's walk. Full details of the training task are provided in Appendix C.2.

$M_{3,1}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$H_{3,1}$**:** The learned policy consists of 27 parameters for a walk engine described in Appendix B. Initial values for the policy are seeded with those from $h_{2,1}$ ($h_{2,1} \dashrightarrow H_{3,1}$) that were optimized for walking to general target positions on the field.

$L_{3,2}$ **: Kick_High_Primitive:** The robot learns a kicking motion to kick the ball over opponents starting from a standing position behind the ball.

$T_{3,2}$**:** The robot attempts to kick the ball as far as possible in the air from a standing position behind the ball. The robot is given higher return values for kicks that travel longer distances at a height above that of opponents, is penalized if the kick takes too long to execute (greater than 2 seconds), and is given a negative return value if it fails to kick the ball. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_high\_primitive}} = \begin{cases} -1 & : \text{ Failure} \\ \text{distBallInAir} - \max(\text{kickTime} - 2, 0) & : \text{ Otherwise} \end{cases}$$

$M_{3,2}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$H_{3,2}$**:** The learned policy consists of 70 parameters specifying a fixed series of poses for a high kick motion defined by our skill description language. Information about the skill description language is provided in Appendix D. Parameters for the x, y, and θ offset standing position of the robot from the ball are also learned. Parameters learned in $h_{2,3}$ for the *Kick_Long_Primitive* are used as seed values for an initial policy ($h_{2,3} \dashrightarrow H_{3,2}$).

$L_{3,3}$ **: Kick_Low_Primitive:** The robot learns a kicking motion to kick the ball such that it stays below the height of the goal when starting from a standing position behind the ball.

$T_{3,3}$**:** The robot attempts to kick the ball as far as possible on the ground from a standing position behind the ball. The robot is given higher return values for kicks that travel longer distances, is penalized if the kick takes too long to execute (greater than 2 seconds), and is given a negative return value if it fails to kick the ball or the ball travels above the height of the goal. The objective function used during optimization is the following where all distance is in meters and time is in seconds:

$$f_{\text{kick\_low\_primitive}} = \begin{cases} -1 & : \text{ Failure} \\ -1 & : \text{ Kick above goal height} \\ \text{distBallForward} - \max(\text{kickTime} - 2, 0) & : \text{ Otherwise} \end{cases}$$

$M_{3,3}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$H_{3,3}$**:** The learned policy consists of 70 parameters specifying a fixed series of poses for a low kick motion defined by our skill description language. Information about the skill description language is provided in Appendix D. Parameters for the x, y, and θ offset standing position of the robot from the ball are also learned. Parameters learned in $h_{2,3}$ for the *Kick_Long_Primitive* are used as seed values for an initial policy ($h_{2,3} \dashrightarrow H_{3,3}$).

$L_{4,1}$ **: Getup_Front_Behavior:** The robot learns to stand up when starting from lying on its front and then walks around.

$T_{4,1}$**:** The training task is the same as $T_{1,1}$ except that after standing up the robot is asked to walk in different directions, and its return value is penalized if it falls over while trying to do so. The objective function used during optimization is the following where time is in seconds:

$$f_{\text{getup\_front}} = \begin{cases} -\text{timeNotStanding} - 5 & : \text{ Fell over after standing} \\ -\text{timeNotStanding} & : \text{ Otherwise} \end{cases}$$

$M_{4,1}$**:** Learning was performed across 200 generations of CMA-ES with a population size of 150.

$H_{4,1}$**:** The learned policy consists of the same 9 *Getup_Front_Primitive* parameters in $H_{1,1}$ which are now unfrozen and relearned. As parameters are unfrozen and relearned this learned behavior is an example of PLLR.

$\boldsymbol{L_{4,2}}$ **: Getup_Back_Behavior:** The robot learns to stand up when starting from lying on its back and then walks around.

$\boldsymbol{T_{4,2}}$**:** The training task is the same as $T_{1,2}$ except that after standing up the robot is asked to walk in different directions, and its return value is penalized if it falls over while trying to do so. The objective function used during optimization is the following where time is in seconds:

$$f_{\text{getup\_back}} = \begin{cases} -\text{timeNotStanding} - 5 & : \text{Fell over after standing} \\ -\text{timeNotStanding} & : \text{Otherwise} \end{cases}$$

$\boldsymbol{M_{4,2}}$**:** Learning was performed across 200 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{4,2}}$**:** The learned policy consists of the same 26 *Getup_Back_Primitive* parameters in $H_{1,2}$ which are now unfrozen and relearned. As parameters are unfrozen and relearned this learned behavior is an example of PLLR.

$\boldsymbol{L_{4,3}}$ **: Walk_PositionToDribble:** The robot learns a walk for dribbling the ball.

$\boldsymbol{T_{4,3}}$**:** The robot starts from different positions relative to the ball and is asked to dribble the ball toward the opponent's goal for 15 seconds. The robot only uses the walk it is learning when positioning around the ball to dribble it. During the remainder of the training task the robot is using the learned behaviors from $L_{2,1}$ and $L_{3,1}$ thus ensuring that the walk being learned can transition from/to the previously learned walk behaviors for walking and sprinting. The robot is given a return value equal to the distance it is able to dribble the ball toward the opponent's goal while being penalized if it falls over. Full details of the training task, known as the driveBallToGoal2 task, are given in Appendix C.3.

$\boldsymbol{M_{4,3}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{4,3}}$**:** The learned policy consists of 27 parameters for a walk engine described in Appendix B. Initial values for the policy are seeded with those from $h_{2,1}$ ($h_{2,1} \dashrightarrow H_{4,3}$) that were optimized for walking to general target positions on the field.

$\boldsymbol{L_{5,1}}$ **: Walk_ApproachToKick:** The robot learns a walk for stopping at a precise position behind the ball in preparation to kick the ball.

$\boldsymbol{T_{5,1}}$**:** The robot is asked to walk to a target position near the ball from which a kick might be executed. The robot is rewarded for stopping at this target position in as little time as possible while being penalized if it runs into the ball or falls over. The robot only uses the walk it is learning when close to the ball. During the remainder of the training task the robot is using the learned behaviors from $L_{2,1}$ and $L_{3,1}$ thus ensuring that the walk being learned can transition from/to the previously learned walk behaviors for walking and sprinting. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{walk\_approach\_to\_kick}} = \begin{array}{l} -\text{timeTaken} \\ +\text{fFellOver ? } -1 : 0 \\ +\text{timeTaken} > 12 \text{ ? } -0.7 : 0 \\ +\text{fRanIntoBall ? } -0.5 : 0 \\ +\text{velocityWhenInPositionToKick} > 0.005 \text{ ? } -0.5 : 0 \end{array}$$

See [3] for full details of the training task.

$\boldsymbol{M_{5,1}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{5,1}}$**:** The learned policy consists of 27 parameters for a walk engine described in Appendix B. Initial values for the policy are seeded with those from $h_{4,3}$ ($h_{4,3} \dashrightarrow H_{5,1}$) that were optimized for positioning around the ball when dribbling. Additional parameters described in [3] for determining how quickly the robot should accelerate and decelerate to reach its target without running into the ball are also learned.

$\boldsymbol{L_{6,1}}$ **: Kick_Fast_Behavior:** The robot learns to walk to the ball and perform a short but fast to execute kick. The robot is also expected to be stable and still standing after the kick.

$\boldsymbol{T_{6,1}}$**:** The same $T_{1,5}$ optimization task for kicking a ball quickly is used, except instead of starting from a standing position behind the ball the robot walks up to the ball and attempts to kick it from different starting positions. Walk parameter sets optimized in previously learned layers are used when approaching to kick the ball thus ensuring the robot can smoothly transition between walking and kicking. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_fast}} = \begin{cases} -1 & : \text{ Failure} \\ -1 & : \text{ Robot fell} \\ \text{distBallForward} - \max(\text{approachAndKickTime} * 2 - 10, 0) & : \text{ Otherwise} \end{cases}$$

$\boldsymbol{M_{6,1}}$**:** Learning was performed across 100 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{6,1}}$**:** The learned policy consists of the same two *Kick_Fast_Primitive* x and y offset position parameters from the ball in $H_{1,5}$—the target position for the walk to reach for the kick to be executed—which are now re-optimized. This learned behavior is an example of CILB as the two independently learned behaviors *Kick_Fast_Behavior* and *Walk_ApproachToKick* ($L_{1,5}$ and $L_{5,1}$) are combined ($H_{1,5} \in L_{6,1}$, $H_{5,1} \in L_{6,1}$) by re-learning a subset of their parameters ($H_{6,1} \subset \{H_{1,5} \cup H_{5,1}\}$).

$\boldsymbol{L_{6,2}}$ **: Kick_High_Behavior:** The robot learns to walk to the ball and perform a high kick to kick the ball over opponents.

$\boldsymbol{T_{6,2}}$**:** The same $T_{3,2}$ optimization task for kicking a ball high is used, except instead of starting from a standing position behind the ball the robot walks up to the ball and attempts to kick it from different starting positions. Walk parameter sets optimized in previously learned layers are used when approaching to kick the ball thus ensuring the robot can smoothly transition between walking and kicking. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_high}} = \begin{cases} -1 & : \text{ Failure} \\ \text{distBallInAir} - \max(\text{approachAndKickTime} * 2 - 10, 0) & : \text{ Otherwise} \end{cases}$$

$\boldsymbol{M_{6,2}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

$\boldsymbol{H_{6,2}}$**:** The learned policy consists of the same 73 *Kick_High_Primitive* parameters in $H_{3,2}$ which are now re-optimized. This learned behavior is an example of CILB as the two independently learned behaviors *Kick_High_Behavior* and *Walk_ApproachToKick* ($L_{3,2}$ and $L_{5,1}$) are combined ($H_{3,2} \in L_{6,2}$, $H_{5,1} \in L_{6,2}$) by re-learning a subset of their parameters ($H_{6,2} \subset \{H_{3,2} \cup H_{5,1}\}$).

$\boldsymbol{L_{6,3}}$ **: Kick_Low_Behavior:** The robot learns to walk to the ball and perform a low kick that stays below the height of the goal.

$\boldsymbol{T_{6,3}}$**:** The same $T_{3,3}$ optimization task for kicking a ball low is used, except instead of starting from a standing position behind the ball the robot walks up to the ball and attempts to kick it from different starting positions. Walk parameter sets optimized in previously learned layers are used when approaching to kick the ball thus ensuring the robot can smoothly transition between walking and kicking. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_low}} = \begin{cases} -1 & : \text{ Failure} \\ -1 & : \text{ Kick above goal height} \\ \text{distBallForward} - \max(\text{approachAndKickTime} * 2 - 10, 0) & : \text{ Otherwise} \end{cases}$$

$\boldsymbol{M_{6,3}}$**:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

***$H_{6,3}$*:** The learned policy consists of the same 73 *Kick_Low_Primitive* parameters in $h_{3,3}$ which are now re-optimized. This learned behavior is an example of CILB as the two independently learned behaviors *Kick_Low_Behavior* and *Walk_ApproachToKick* ($L_{3,3}$ and $L_{5,1}$) are combined ($H_{3,3} \in L_{6,3}$, $H_{5,1} \in L_{6,3}$) by re-learning a subset of their parameters ($H_{6,3} \subset \{H_{3,3} \cup H_{5,1}\}$).

$L_{6,4}$: Kick_Long_Behavior: The robot learns to walk to the ball and perform a long kick.

***$T_{6,4}$*:** The same $T_{2,3}$ optimization task for kicking a ball a long distance is used, except instead of starting from a standing position behind the ball the robot walks up to the ball and attempts to kick it from different starting positions. Walk parameter sets optimized in previously learned layers are used when approaching to kick the ball thus ensuring the robot can smoothly transition between walking and kicking. The objective function used during optimization is the following where distance is in meters and time is in seconds:

$$f_{\text{kick\_long}} = \begin{cases} -1 & : \text{ Failure} \\ \text{distBallForward} - \max(\text{approachAndKickTime} * 2 - 10, 0) & : \text{ Otherwise} \end{cases}$$

***$M_{6,4}$*:** Learning was performed across 300 generations of CMA-ES with a population size of 150.

***$H_{6,4}$*:** The learned policy consists of the same 73 *Kick_Long_Primitive* parameters in $H_{2,3}$ which are now re-optimized.This learned behavior is an example of CILB as the two independently learned behaviors *Kick_Long_Behavior* and *Walk_ApproachToKick* ($L_{2,3}$ and $L_{5,1}$) are combined ($H_{2,3} \in L_{6,4}$, $H_{5,1} \in L_{6,4}$) by re-learning a subset of their parameters ($H_{6,4} \subset \{H_{2,3} \cup H_{5,1}\}$).

B. Walk Engine

The UT Austin Villa team uses an omnidirectional walk engine based on one that was originally designed for the real Nao robot [36]. The omnidirectional walk is crucial for allowing the robot to request continuous velocities in the forward, side, and turn directions, permitting it to approach continually changing destinations (often the ball) more smoothly and quickly than the team's previous set of unidirectional walks [15].

The walk engine, though based closely on that of Graf et al. [36], differs in some of the details. Specifically, unlike Graf et al., the walk engine uses a sigmoid function for the forward component and uses proportional control to adjust the desired step sizes. The walk engine uses a simple set of sinusoidal functions to create the motions of the limbs with limited feedback control. The work flow of how joint commands are generated for the walk is shown in Figure B.5. The walk engine processes desired walk velocities chosen by the behavior, chooses destinations for the feet and torso, and then uses inverse kinematics to determine the joint positions required. Finally, PID controllers for each joint convert these positions into torque commands that are sent to the simulator.

The walk engine selects a trajectory for the torso to follow, and then determines where the feet should be with respect to the torso location. The walk uses x as the forwards dimension, y as the sideways dimension, z as the vertical dimension, and θ as rotating about the z axis. The trajectory is chosen using a double linear inverted pendulum, where the center of mass is swinging over the stance foot. In addition, as in Graf et al.'s work [36], the walk engine uses the simplifying assumption that there is no double support phase, so that the velocities and positions of the center of mass must match when switching between the inverted pendulums formed by the respective stance feet.

We now describe the mathematical formulas that calculate the positions of the feet with respect to the torso. More than 40 walk engine parameters were used, but only the ones we optimize are listed in Table B.6.

To smooth changes in the velocities, the walk engine uses a simple proportional controller to filter the requested velocities coming from the behavior module. Specifically, the walk engine calculates $\text{step}_{i,t+1} = \text{step}_{i,t} + \delta_{\text{step}}(\text{desired}_{i,t+1} - \text{step}_{i,t}) \forall i \in \{x, y, \theta\}$. In addition, the value is cropped within the maximum step sizes so that $-\text{maxStep}_i \leq \text{step}_{i,t+1} \leq \text{maxStep}_i$.

The phase is given by $\phi_{\text{start}} \leq \phi \leq \phi_{\text{end}}$, and $t = \dfrac{\phi - \phi_{\text{start}}}{\phi_{\text{end}} - \phi_{\text{start}}}$ is the current fraction through the phase. At each time step, ϕ is incremented by $\Delta\text{seconds}/\phi_{\text{length}}$, until $\phi \geq \phi_{\text{end}}$. At this point, the stance and swing feet change and ϕ is reset to ϕ_{start}. Initially, $\phi_{\text{start}} = -0.5$ and $\phi_{\text{end}} = 0.5$. However, the start and end times will change to match the previous pendulum, as given by the equations

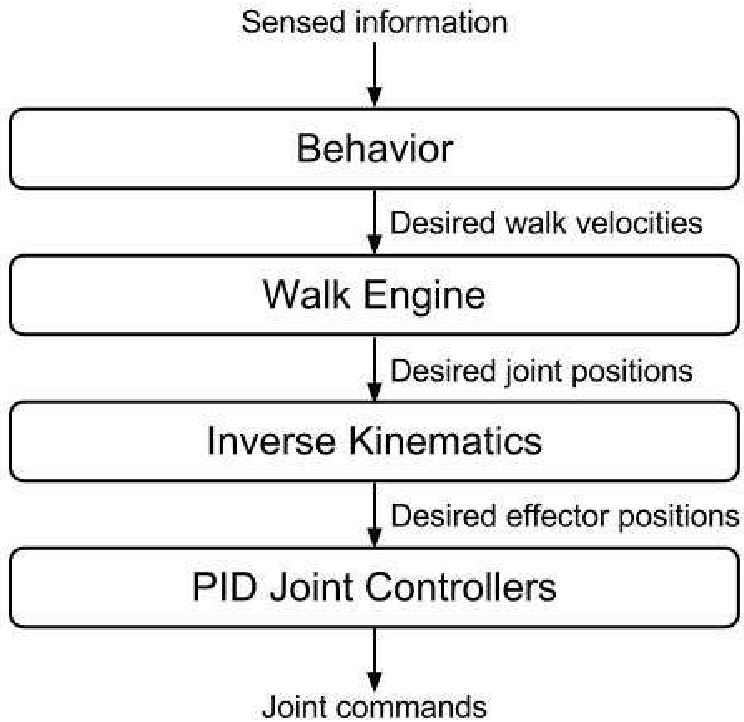

Figure B.5. Workflow for generating joint commands from the walk engine.

Notation	Description
$\text{maxStep}_{\{x,y,\theta\}}$	Maximum step sizes allowed for x, y, and θ
y_{shift}	Side to side shift amount with no side velocity
z_{torso}	Height of the torso from the ground
z_{step}	Maximum height of the foot from the ground
f_{g}	Fraction of a phase that the swing foot spends on the ground before lifting
f_{a}	Fraction that the swing foot spends in the air
f_{s}	Fraction before the swing foot starts moving
f_{m}	Fraction that the swing foot spends moving
ϕ_{length}	Duration of a single step
$\delta_{\text{step}\{x,y,\theta\}}$	Factor of how fast the step sizes change for x, y, and θ
x_{offset}	Constant offset between the torso and feet
x_{factor}	Factor of the step size applied to the forwards position of the torso
$\delta_{\text{target}\{\text{tilt},\text{roll}\}}$	Factors of how fast tilt and roll adjusts occur for balance control
$\text{ankle}_{\text{offset}}$	Angle offset of the swing leg foot to prevent landing on toe
err_{norm}	Maximum COM error before the steps are slowed
err_{max}	Maximum COM error before all velocity reach 0
$\text{COM}_{\text{offset}}$	Default COM forward offset
$\delta_{\text{COM}\{x,y,\theta\}}$	Factors of how fast COM changes x, y, and θ values for balance control
$\delta_{\text{arm}\{x,y\}}$	Factors of how fast the arm x and y offsets change for balance control

Table B.6. Optimized parameters of the walk engine.

$$
\begin{aligned}
k &= \sqrt{9806.65/z_{\text{torso}}} \\
\alpha &= 6 - \cosh(k - 0.5\phi) \\
\phi_{\text{start}} &= \begin{cases} \dfrac{\cosh^{-1}(\alpha)}{0.5k} & \text{if } \alpha \geq 1.0 \\ -0.5 & \text{otherwise} \end{cases} \\
\phi_{\text{end}} &= 0.5(\phi_{\text{end}} - \phi_{\text{start}})
\end{aligned}
$$

The stance foot remains fixed on the ground, and the swing foot is smoothly lifted and placed down, based on a cosine function. The current distance of the feet from the torso is given by

$$
\begin{aligned}
z_{\text{frac}} &= \begin{cases} 0.5(1-\cos(2\pi \frac{t-f_{\text{g}}}{f_{\text{a}}})) & \text{if } f_{\text{g}} \le t \le f_{\text{a}} \\ 0 & \text{otherwise} \end{cases} \\
z_{\text{stance}} &= z_{\text{torso}} \\
z_{\text{swing}} &= z_{\text{torso}} - z_{\text{step}} * z_{\text{frac}}
\end{aligned}
$$

It is desirable for the robot's center of mass to steadily shift side to side, allowing it to stably lift its feet. The side to side component when no side velocity is requested is given by

$$
\begin{aligned}
y_{\text{stance}} &= 0.5 y_{\text{sep}} + y_{\text{shift}}(-1.5 + 0.5\cosh(0.5k\phi)) \\
y_{\text{swing}} &= y_{\text{sep}} - y_{\text{stance}}
\end{aligned}
$$

where y_{sep} is the distance between the feet. If a side velocity is requested, y_{stance} is augmented by

$$
\begin{aligned}
y_{\text{frac}} &= \begin{cases} 0 & \text{if } t < f_{\text{s}} \\ 0.5(1+\cos(\pi \frac{t-f_{\text{s}}}{f_{\text{m}}})) & \text{if } f_{\text{s}} \le t < f_{\text{s}} + f_{\text{m}} \\ 1 & \text{otherwise} \end{cases} \\
\Delta y_{\text{stance}} &= \text{step}_y * y_{\text{frac}}
\end{aligned}
$$

These equations allow the y component of the feet to smoothly incorporate the desired sideways velocity while still shifting enough to remain dynamically stable over the stance foot.

Next, the forwards component is given by

$$
\begin{aligned}
s &= \text{sigmoid}(10(-0.5 + \frac{t-f_{\text{s}}}{f_{\text{m}}})) \\
x_{\text{frac}} &= \begin{cases} (-0.5 - t + f_{\text{s}}) & \text{if } t < f_{\text{s}} \\ (-0.5 + s) & \text{if } f_{\text{s}} \le t < f_{\text{s}} + f_{\text{m}} \\ (0.5 - t + f_{\text{s}} + f_{\text{m}}) & \text{otherwise} \end{cases} \\
x_{\text{stance}} &= 0.5 - t + f_{\text{s}} \\
x_{\text{swing}} &= \text{step}_x * x_{\text{frac}}
\end{aligned}
$$

These functions are designed to keep the robot's center of mass moving forwards steadily, while the feet quickly, but smoothly approach their destinations. Furthermore, to keep the robot's center of mass centered between the feet, there is an additional offset to the forward component of both the stance and swing feet, given by

$$\Delta x = x_{\text{offset}} - \text{step}_x x_{\text{factor}}$$

After these calculations, all of the x and y targets are corrected for the current position of the center of mass. Finally, the requested rotation is handled by opening and closing the groin joints of the robot, rotating the foot targets. The desired angle of the groin joint is calculated by

$$
\text{groin} = \begin{cases} 0 & \text{if } t < f_{\text{s}} \\ \frac{1}{2}\text{step}_\theta(1-\cos(\pi \frac{t-f_{\text{s}}}{f_{\text{m}}})) & \text{if } f_{\text{s}} \le t < f_{\text{s}} + f_{\text{m}} \\ \text{step}_\theta & \text{otherwise} \end{cases}
$$

After these targets are calculated for both the swing and stance feet with respect to the robot's torso, the inverse kinematics module calculates the joint angles necessary to place the feet at these targets. Further description of the inverse kinematic calculations is given in [36].

To improve the stability of the walk, the walk engine tracks the desired center of mass as calculated from the expected commands. Then, the walk engine compares this value to the sensed center of mass after handling the delay between sending commands and sensing center of mass changes of approximately 20ms. If this error is too large, it is expected that the robot is unstable, and action must be taken to prevent falling. As the robot is more stable when walking in place, the walk engine immediately reduces the step sizes by a factor of the error. In the extreme case, the robot will attempt to walk in place until it is stable. The exact calculations are given by

$$\begin{aligned}\text{err} &= \max_i(\text{abs}(\text{com}_{\text{expected},i} - \text{com}_{\text{sensed},i}))\\ \text{stepFactor} &= \max(0, \min(1, \frac{\text{err} - \text{err}_{\text{norm}}}{\text{err}_{\max} - \text{err}_{\text{norm}}}))\\ \text{step}_i &= \text{stepFactor} * \text{step}_i \ \forall i \in \{x, y, \theta\}\end{aligned}$$

This solution is less than ideal, but performs effectively enough to stabilize the robot in many situations.

C. Optimization Process and Training Tasks for Learning Walk Parameter Sets

The following subsections detail the optimization process and training tasks used during the layered learning approach to develop three walk parameter sets needed for general walking, sprinting, and dribbling the ball. The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) algorithm [14] was used to learn the walk engine parameters listed in Table B.6 for each set of walk parameters. Further details about the development of the optimization process—first implemented for the 2011 team—are available in [1], however here we only present the details most relevant to the 2014 team.

C.1. Walk_GoToTarget Parameter Set Optimization

To learn walk parameters for moving to general target positions on the field we created a training task—called the goToTarget optimization task—consisting of an obstacle course in which the robot tries to navigate to a variety of target positions on the field. Each target is active, one at a time for a fixed period of time, which varies from one target to the next, and the robot is rewarded based on its distance traveled toward the active target. If the robot reaches an active target, the robot receives an extra reward based on extrapolating the distance it could have traveled given the remaining time on the target. In addition to the target positions, the robot has stop targets, where it is penalized for any distance it travels. To promote stability, the robot is given a penalty if it falls over during the optimization run.

In the following equations specifying the agent's rewards for targets, *Fall* is 5 if the robot fell and 0 otherwise, d_{target} is the distance traveled toward the target, and d_{moved} is the total distance moved. Let t_{total} be the full duration a target is active and t_{taken} be the time taken to reach the target or t_{total} if the target is not reached.

$$\begin{aligned}\text{reward}_{\text{target}} &= d_{\text{target}}\frac{t_{\text{total}}}{t_{\text{taken}}} - \text{Fall}\\ \text{reward}_{\text{stop}} &= -d_{\text{moved}} - \text{Fall}\end{aligned}$$

The goToTarget optimization includes quick changes of target/direction for focusing on the reaction speed of the agent, as well as targets with longer durations to improve the straight line speed of the agent. The stop targets ensure that the agent is able to stop quickly, while remaining stable. The trajectories that the agent follows during the optimization are described in Figure C.6.

C.2. Walk_Sprint Parameter Set Optimization

To further improve the forward speed of the agent, we optimized a parameter set for walking straight forwards for ten seconds starting from a complete stop. Unfortunately, when the robot tried to switch between the forward walk and *Walk_GoToTarget* parameter sets it was unstable and usually fell over. This instability is due to the parameter sets being learned in isolation, resulting in them being incompatible.

To overcome this incompatibility, we ran the goToTarget subtask optimization again, but this time we fixed the *Walk_GoToTarget* parameter set and learned a new parameter set. We call these parameters the *Walk_Sprint* parameter set, and the agent uses them when its orientation is within 15° of its target. The *Walk_Sprint* parameter set was seeded with the values from the *Walk_GoToTarget* parameter set. This approach to optimization is an example of sequential layered learning as the output of one learned subtask (the *Walk_GoToTarget* parameter set) is fed in as input to the learning of the next subtask (the learning of the *Walk_Sprint* parameter set). By learning the *Walk_Sprint* parameter set in conjunction with the *Walk_GoToTarget* parameter set, the robot was stable switching between the two parameter sets.

- Long walks forward/backwards/left/right
- Walk in a curve
- Quick direction changes
- Stop and go forward/backwards/left/right
- Switch between moving left-to-right and right-to-left
- Quick changes of target to simulate a noisy target
- Weave back and forth at 45 degree angles
- Extreme changes of direction to check for stability
- Quick movements combined with stopping
- Quick alternating between walking left and right
- Spiral walk both clockwise and counter-clockwise

Figure C.6. GoToTarget Optimization walk trajectories

C.3. Walk_PositionToDribble Parameter Set Optimization

Although adding the *Walk_GoToTarget* and *Walk_Sprint* walk engine parameter sets improved the stability, speed, and game performance of the agent, the agent was still a little slow when positioning to dribble the ball. This slowness is explained by the fact that the goToTarget subtask optimization emphasizes quick turns and forward walking speed while positioning around the ball involves more side-stepping to circle the ball. To account for this discrepancy, the agent learned a third parameter set which we call the *Walk_PositionToDribble* parameter set. To learn this parameter set, we created a new driveBallToGoal2[8] optimization in which the agent is evaluated on how far it is able to dribble the ball over 15 seconds when starting from a variety of positions and orientations from the ball. The *Walk_PositionToDribble* parameter set is used when the agent is .8 meters from the ball and is seeded with the values from the *Walk_GoToTarget* parameter set. Both the *Walk_GoToTarget* and *Walk_Sprint* parameter sets are fixed and the optimization naturally includes transitions between all three parameter sets, which constrained them to be compatible with each other. As learning of the *Walk_PositionToDribble* parameter set takes the two previously learned parameter sets as input, it is a third layer of sequential layered learning.

D. Skill Description Language

The UT Austin Villa agent has skills for getting up after falling and kicking, each of which is implemented as a periodic state machine with multiple *key frames*, where a key frame is a static pose of fixed joint positions. Key frames are separated by a waiting time that lets the joints reach their target angles. To provide flexibility in designing and parameterizing skills, we designed an intuitive skill description language that facilitates the specification of key frames and the waiting times between them. Below is an illustrative example describing a kick skill.

```
SKILL KICK_LEFT_LEG

KEYFRAME 1
setTarget JOINT1 $jointvalue1 JOINT2 $jointvalue2 ...
setTarget JOINT3 4.3 JOINT4 52.5
wait 0.08
```

[8]The '2' at the end of the name driveBallToGoal2 is used to differentiate it from the driveBallToGoal optimization that was used in [1].

```
KEYFRAME 2
increaseTarget JOINT1 -2 JOINT2 7 ...
setTarget JOINT3 $jointvalue3 JOINT4 (2 * $jointvalue3)
wait 0.08
.
.
.
```

As seen above, joint angle values can either be numbers or be parameterized as `$<varname>`, where `<varname>` is a variable value that can be loaded after being learned. Values for skills and other configurable variables are read in and loaded at runtime from parameter files.

E. Competition Results

In winning the 2014 RoboCup competition UT Austin Villa finished with an undefeated record of 13 wins and 2 ties [3].[9] During the competition the team scored 52 goals without conceding any. Despite finishing with an undefeated record, the relatively few number of games played at the competition, coupled with the complex and stochastic environment of the RoboCup 3D simulator, make it difficult to determine UT Austin Villa being better than other teams by a statistically significant margin. At the end of the competition, however, all teams were required to release their binaries used during the competition. Results of UT Austin Villa playing 1000 games against each of the other 11 teams' released binaries from the competition are shown in Table E.7.

Table E.7. UT Austin Villa's 2014 released binary's performance when playing 1000 games against the released binaries of all other teams at RoboCup 2014. This includes place (the rank a team achieved at the competition), average goal difference (values in parentheses are the standard error), win-loss-tie record, goals for/against, and the percentage of own kickoffs which the team scored from.

Opponent	Place	Avg. Goal Diff.	Record (W-L-T)	Goals (F/A)	KO Score %
BahiaRT	5-8	2.075 (0.030)	990-0-10	2092/17	96.2
FCPortugal	4	2.642 (0.034)	986-0-14	2748/106	83.4
magmaOffenburg	3	2.855 (0.035)	990-0-10	2864/9	88.3
RoboCanes	2	3.081 (0.046)	974-0-26	3155/74	69.4
FUT-K	5-8	3.236 (0.039)	998-0-2	3240/4	96.3
SEU_Jolly	5-8	4.031 (0.062)	995-0-5	4034/3	87.6
KarachiKoalas	9-12	5.681 (0.046)	1000-0-0	5682/1	87.5
ODENS	9-12	7.933 (0.041)	1000-0-0	7933/0	92.1
HfutEngine	5-8	8.510 (0.050)	1000-0-0	8510/0	94.7
Mithras3D	9-12	8.897 (0.041)	1000-0-0	8897/0	90.4
L3M-SIM	9-12	9.304 (0.043)	1000-0-0	9304/0	93.7

UT Austin Villa finished with at least an average goal difference greater than two goals against every opponent. Additionally UT Austin Villa did not lose a single game out of the 11,000 that were played in Table E.7. These game results show that UT Austin Villa winning the 2014 competition was far from a chance occurrence.

The overlapping layered learning system employed by UT Austin Villa was also vital in the team winning the subsequent competitions in 2015 [20], 2016 [21], and 2017. Table E.8 shows post competition analysis of the 2015 competition in which UT Austin Villa was again able to beat all opponents by an average goal difference of over two goals. The team was also able to beat all opponents at the 2016 competition by at least an average goal difference of close to two goals as shown by the post 2016 competition analysis in Table E.9. Post 2017 competition analysis, provided in Table E.10, shows that UT Austin Villa was able to beat all opponents at the 2017 competition by an average goal difference of almost four goals.

[9] Full tournament results can be found at
http://wiki.robocup.org/wiki/Soccer_Simulation_League/RoboCup2014#3D

Table E.8. UT Austin Villa's 2015 released binary's performance when playing 1000 games against the released binaries of all other teams at RoboCup 2015. This includes place (the rank a team achieved at the competition), average goal difference (values in parentheses are the standard error), win-loss-tie record, and goals for/against.

Opponent	Place	Avg. Goal Diff.	Record (W-L-T)	Goals (F/A)
FUT-K	2	2.082 (0.036)	927-2-71	2178/96
FCPortugal	3	2.399 (0.040)	945-4-51	2624/225
BahiaRT	4	2.496 (0.044)	944-1-55	2501/5
Apollo3D	5	3.803 (0.046)	995-0-5	3805/2
magmaOffenburg	6	4.167 (0.051)	999-0-1	4171/4
RoboCanes	7	4.187 (0.049)	998-0-2	4235/48
Nexus3D	8	5.571 (0.044)	1000-0-0	5573/2
CIT3D	9	6.321 (0.050)	1000-0-0	6321/0
ITAndroids	11	10.125 (0.041)	1000-0-0	10125/0
Miracle3D	12	10.521 (0.056)	1000-0-0	10521/0
HfutEngine3D	10	11.897 (0.068)	1000-0-0	11897/0

Table E.9. UT Austin Villa's 2016 released binary's performance when playing 1000 games against the released binaries of all other teams at RoboCup 2016. This includes place (the rank a team achieved at the 2016 competition), average goal difference (values in parentheses are the standard error), win-loss-tie record, and goals for/against.

Opponent	Place	Avg. Goal Diff.	Record (W-L-T)	Goals (F/A)
FUT-K	2	1.809 (0.036)	888-3-109	1872/63
FCPortugal	3	2.431 (0.040)	954-1-45	2452/21
BahiaRT	4	3.123 (0.040)	985-0-15	3123/0
magmaOffenburg	5	3.921 (0.049)	996-0-4	3926/5
KgpKubs	8	7.728 (0.046)	1000-0-0	7729/1
ITAndroids	6	9.022 (0.053)	1000-0-0	9024/2
HfutEngine3D	9	10.192 (0.056)	1000-0-0	10192/0
Miracle3D	7	11.126 (0.059)	1000-0-0	11126/0

Table E.10. UT Austin Villa's 2017 released binary's performance when playing 1000 games against the released binaries of all other teams at RoboCup 2017. This includes place (the rank a team achieved at the 2017 competition), average goal difference (values in parentheses are the standard error), win-loss-tie record, and goals for/against.

Opponent	Place	Avg. Goal Diff.	Record (W-L-T)	Goals (F/A)
magmaOffenburg	2	3.756 (0.057)	983-0-17	3778/22
FUT-K	3	4.793 (0.056)	995-0-5	4823/30
AIUT3D	5	5.946 (0.054)	1000-0-0	5981/35
BahiaRT	6	6.677 (0.055)	1000-0-0	6677/0
FCPortugal	7	6.753 (0.062)	1000-0-0	6818/65
Nexus3D	11	7.486 (0.035)	1000-0-0	7486/0
KgpKubs	8	7.510 (0.057)	1000-0-0	7510/0
RoboCanes	4	7.801 (0.066)	1000-0-0	7806/5
HfutEngine3D	12	7.952 (0.049)	1000-0-0	7957/5
Miracle3D	10	8.404 (0.056)	1000-0-0	8404/0
ITAndroids	9	11.169 (0.057)	1000-0-0	11169/0
RIC-AASTMT	13	11.466 (0.051)	1000-0-0	11466/0