Evolving a Non-playable Character Team with Layered Learning

Sean Mondesire
Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL, USA
mondesire@knights.ucf.edu

R. Paul Wiegand
Institute for Simulation and Training
University of Central Florida
Orlando, FL, USA
wiegand@ist.ucf.edu

Abstract— **Layered Learning is an iterative machine learning technique used to train agents how to perform tasks. The technique decomposes a task into simpler components and trains the agent to learn how to perform progressively more complex sub-tasks to solve the overall task. Layered Learning has been successfully used to instruct computer programs to solve Boolean-logic problems, teach robots how to walk, and train RoboCup soccer playing agents.**

The proposed work answers the question of how well does Layered Learning apply to the evolved development of a heterogeneous team of Non-playable Characters (NPCs) in a video game. The work compares the use of Layered Learning against evolving NPCs with monolithic based approaches. Experiment data show that Layered Learning can result in the successful development of NPCs and demonstrates that the approach performs well against monolithic evaluation.

Keywords: Layered Learning, Genetic Algorithm, Decision Making

I. Introduction

Layered Learning is an iterative machine learning technique that has been used to automate the process of training computer-based agents to perform tasks. The model decomposes complex tasks into simpler sub-tasks and trains the agent on one sub-task at a time. The aim with Layered Learning is that once the agent has learned to perform all of the sub-tasks, then the agent will be able to perform the overall assigned task. The approach attempts to lessen the arduous goal of learning how to perform a complex task at once by learning how to perform a series of increasingly difficult component tasks, where none is more complicated than the original task. The decomposition from a single task to a set of multiple criteria allows the agent to distribute the workload of learning across several phases instead of being overwhelmed all at once.

Layered Learning is being used to develop the decision making abilities of both simulated and physical agents. Examples of uses of Layered Learning include using the method to teach a computer-based agent to solve Boolean-logic problems [1], to train a robot how to walk [2], and teach simulated robots how to play variations of soccer [3] [4].

The work presented here seeks to determine how effectively Layered Learning can be applied to the development of an evolved heterogeneous team of Non-playable Characters (NPCs) in video games. To make this determination, the work uses Layered Learning with a Genetic Algorithm (GA) to modify the behaviors of a team of NPCs in a predator-prey scenario. The approach evolves each teammate's behaviors with the goal of giving them the ability to solve a task shared among the entire team. The proposed method decomposes the task into 3 sub-task layers: learn how to perform the basic task 1) effectively, 2) efficiently, and finally 3) effectively and efficiently at the same time.

For performance evaluation, Layered Learning is compared with several GAs using monolithic fitness criteria variations. This evaluation is sought because existing work that compares the performance of Layered Learning and monolithic GAs leaves questions of when and how Layered Learning should be used to develop teams of agents.

The results for this paper suggest that Layered Learning is successful in the production of a heterogeneous multi-agent team for a specific class of predator-prey problems, and the approach may be effective in more complex domains such as the heavily tested RoboCup and other robotic domains. In addition, favorable results mean the approach can be used as an additional method of developing the decision making ability for an evolved team of agents assigned shared complex tasks.

II. Layered Learning

Layered Learning has been used to develop the decision making ability of autonomous robotic and other computer-controlled agents. Proposed by Stone and Veloso, Layered Learning extends the robot shaping [5] idea of decomposing a task into simpler components by training the learning agent on progressively more complex sub-tasks one at a time [3]. The bottom-up approach sees the agent first learn how to perform the less difficult sub-tasks before progressing to the more complex sub-tasks. Because an agent learns one sub-task at a time, each sub-task is seen as a layer of training the agent experiences before learning to perform the overall task.

Each layer is comprised of at least one training scenario that is designed to train the agent to perform the layer's sub-task independent of other sub-tasks. Within each layer, the agent learns and adapts its decision making knowledge to perform the assigned sub-task. The agent remains in the layer until the layer's halting criteria is satisfied. The halting criterion

978-1-61284-069-7/11/$26.00 ©2011 IEEE

is a specified level of proficiency in performing the sub-task, a limit on time training in a layer, or any other condition that signals the learning model when to progress to the next layer.

The knowledge acquired in each layer is progressive because the agent's knowledge-state at the end of one layer is fed into the next layer. This forwarding of knowledge allows each layer to build on top of what was previously learned by adding new knowledge during the training of the next sub-task.

Within layers, Layered Learning relies on another machine learning technique to be employed to develop agent behaviors. Neural Networks [6] and genetic programming (GP) [7] have been used to manipulate copies of the agent and evaluate how well the agent performs the current layer's sub-task. The modification-evaluation process within a layer repeats until the layer's halting criterion is satisfied before moving to the next layer.

Because the process of Layered Learning is based on successive layers, the aim of the layered approach is to have the agent be able to perform the assigned task once all of the sub-tasks have been learned. With that said, Layered Learning is best used when a task is too complex to be learned in one phase and when the task can be decomposed into simpler sub-tasks.

Layered Learning is a natural choice for the development of decision making abilities of autonomous agents because of its break down of a complex task into simpler components. With autonomous agent design, such as the development of NPCs, Layered Learning provides an intrinsic, abstract scheme for agent training. Any agent developer can employ this natural learning model to give shape to an agent by teaching one behavior at a time or how to perform basic actions before moving to more complex ones.

The work proposed here uses Layered Learning to evolve a team of NPCs. Layered Learning is chosen because it has shown to reduce the solution space at each layer, lessen the pressure of discovering the solution to a complex task all at once, and provide a natural and simple way of decomposing an arduous task into manageable portions [8] [6].

III. Related Work

The motivation behind Layered Learning is to provide an effective model of decision making for computer-based agents. Since the model's introduction, several noteworthy works have expanded the learning technique and explored new areas where Layered Learning can be successfully deployed.

Stone and Veloso first used Layered Learning to train simulated robots to play soccer [3]. The work produced an agent team that adhered to the rules of soccer prescribed by the RoboCup Challenge. Their model splits learning into two phases. The first phase trains the agent to perform basic, individualistic actions, such as movement. The second phase trains the agent to perform more complex tasks, including team-involvement behaviors, such as passing. Their work provided the introduction of the Layered Learning model and shows how the decomposition of a complex task could train agents how to progressively learn sub-tasks.

Layered Learning has also been applied to the RoboCup sub-domain of Keep-away. In Keep-away, instead of trying to score goals, players with the role of *keepers* try to maintain ball possession by passing the ball among teammates. *Takers* are tasked with stealing ball possession from the *keepers*.

Gustafson and Hsu use Layered Learning to train a team of *keepers* [7]. Their method uses GP to produce a controller that determines which actions the *keepers* will perform. The results from their experiments show that Layered Learning can produce Keep-away teams that outperform *keeper* implementations based on a standard GP technique with an aggregated fitness function and a hand-coded approach.

Whiteson, Kohl, Miikkulainen and Stone [6] also tackled the development of *keepers*. Their work compares Layered Learning's performance against Tabula Rasa with Incremental Reuse and co-evolution. Here, Neural Networks control *keeper* decisions and are evolved in each layer. The coevolutionary approach does not decompose the task but instead trains the *keepers* on the entire task at once. These coevolved agents are homogeneous because each agent shares the same decision making knowledge. Experiments show that the coevolutionary approach produces the best team of *keepers*, opposite of Gustafson and Hsu's findings that Layered Learning produces a better *keeper* than an aggregate function. It should be noted that both works differ in the machine learning technique used within layers, aggregation function, and implementation.

Jackson and Gibbons [1] examined Layered Learning with GP to solve Boolean-logic problems. Similar to Gustafson and Hsu, their work evolved a genetic program that makes the Boolean-logic decisions to solve the problems at each layer. Their work also compared Layered Learning with GP against an approach with standard GP (an aggregate/monolithic fitness function), and an Automatically Defined Function GP based system. Results show that Layered Learning solved the Boolean-logic problems with less computational effort than the standard GP and ADF approach.

The work presented in this paper uses Layered Learning to tackle high-level tasks in a multi-agent environment. This work analyzes the effectiveness of using Layered Learning to evolve a team of heterogeneous agents against monolithic GAs.

IV. Method

The proposed work provides a method of producing a team of agents to perform a shared task. To achieve this goal, Layered Learning is used to decompose a complex task into simpler components and trains agents to perform each sub-task.

There are a number of qualities that distinguish this work from others. First, the evolved agents are heterogeneous because each team member has its own set of knowledge evolved to learn the shared task. The heterogeneity creates diversity within the team and the potential for team roles to emerge. Second, the team of NPCs is evolved as one unit to produce the best composition of roles and distinct decision making to achieve the shared goal. Third, the proposed method designs the layers in a way where the complex task is decomposed into sub-tasks that aim at improving task performance at each layer. The final layer unifies all of the sub-tasks with an aggregate function that ultimately determines the

quality of performance. Finally, this method allows the same training scenario to be used for each layer and sub-tasks. Typically, agents in Layered Learning train on different scenarios for each layer to allow the agent to fully focus on the sub-task at hand. The proposed method uses the same scenario but with different fitness criteria for each layer. This scenario reuse allows the agent developer to have more time on other development tasks.

A predator-prey scenario is used to evaluate the effectiveness of the proposed method. In the scenario, a team of agents is assigned the shared complex task of capturing its prey as effectively and efficiently as possible. To implement the scenario, a Pac-Man-inspired game, Pac-Clone, is used to provide the game environment, agent constraints and roles, and rules the agents follow for task achievement. In the implementation, the ghosts in Pac-Man represent the NPC team that is being learned. The shared complex task the team is given is to capture Pac-Man with a high probability of success and in as few time-steps as possible.

To train the team of NPCs, the ghosts' task is decomposed into three layers: 1) capture Pac-Man, 2) minimize the game's duration, and 3) an aggregate function that rewards fast Pac-Man captures. The capture Pac-Man sub-task favors ghost teams that are able to locate and capture the target (*effectiveness*). The capture sub-task does not discriminate how ghosts capture their prey but rewards their performance only if the team is able to accomplish the sub-task.

The second layer seeks to minimize game duration (*efficiency*). There are two ways a game can terminate before it has reached the time-limit: 1) a ghost has captured Pac-Man and 2) Pac-Man has achieved his own task. The first cause means the ghosts solved the sub-task from the first layer; the second means the ghosts were unsuccessful at capturing Pac-Man. Regardless of the cause of a game ending prematurely, in this sub-task, the team is only tasked with ending the game as early as possible.

The final layer aggregates both sub-tasks by rewarding teams that can capture Pac-Man the most often and the fastest (*effectiveness* + *efficiency*). The aggregation creates the need for the team to be able to perform the first two sub-tasks together and removes any reward for Pac-Man eluding capture as seen in the second layer. With the complex task decomposed into these 3 layers, the team of NPCs is evolved in progressive steps which lead to the overall task to be addressed.

The following subsections discuss the implementation of the proposed use of Layered Learning. Here, the agent environment and the evolutionary training process are described in detail.

A. *Pac-Clone*

Pac-Clone is a 2D video game that is inspired by Pac-Man [9]. It was developed to evaluate different machine learning techniques' abilities to produce agent decision making capabilities. The Pac-Man concept is useful because it is an established predator-prey problem that encourages team actions to accomplish a shared task. In addition, Pac-Man provides an interesting dynamic where the roles of predator and prey are flipped at certain points in the game. Dynamic role reversal makes the problem challenging for the agents to learn both predator and prey-like decisions.

In short, the Pac-Clone game consists of two opposing sides: the playable character Pac-Man and three NPC ghosts. Pac-Man is tasked with collecting all of the dots, power-pills, and fruit scattered through the game environment by moving to their locations. In Pac-Man, the ghost team is tasked with preventing Pac-Man from achieving his task by capturing him. Captures are achieved by a ghost sharing the same location in the environment as Pac-Man at the same time.

In Pac-Clone, when Pac-Man consumes a power-pill, he is able to capture ghosts for 15 time-steps. Power-pill consumption reverses the roles in the game by making Pac-Man the predator and the ghosts the prey. Once a ghost is captured, it is eliminated from the game environment. If another power-pill was not consumed within this 15 time-step duration, Pac-Man is vulnerable to capture by a ghost again at the end of the interval. Ghosts cannot consume any items in the environment but can block Pac-Man from obtaining an item by standing at the item's location or capturing Pac-Man.

Pac-Clone follows all of the rules of the classic Pac-Man game, as outlined in [10], with a few variations. Differences include level design, character movement, scoring, and game termination. For game termination, Pac-Man is given one life and the game has a maximum time limit. If Pac-Man is captured, collects all of the dots, power-pills, and fruit, or the time-limit expires, then the game ends. Each agent is restricted in movement by being permitted to move to at most one space per time-step and only in up, down, left, and right directions. Both Pac-Man and ghost characters use the A* route planning algorithm to generate routes more than two spaces away. Agents are not permitted to move to spaces where walls are located and two ghosts cannot occupy the same map location.

To make the game more challenging, each character has a limited view of the environment, meaning they can only detect items and agents from sensors and perceptions. All agents have access to the immediate squares one space around them from its sensor. Each character also has a heading that change in the up, down, left, and right directions. Heading is used to give the characters perceptions of the environment based on line-of-sight (LOS). LOS is interrupted if another agent or wall crosses its heading path.

Pac-Clone uses scripting to drive the actions of the characters. Each character has its own script that determines what actions to perform based on inputs from the environment and character's state. At every time-step in the game, each non-captured character is polled for an action to perform. It is up to the character's script to reply with an action; otherwise, the character will not perform a new action for that time-step. Some actions require more than one time-step to perform to completion, for example, moving to a location two or more spaces away. Actions like these produce a sequence of actions the agent will perform in subsequent time-steps unless interrupted with a new action or an invalid action from the game's engine. Each script is broken down into actions, conditionals, and IF-statements. **Table 1** lists all actions and conditionals used in Pac-Clone. Actions are the individual behaviors a ghost can execute. The "clear route to follow"

action stops the agent from continuing on a route it is currently following. “Attack” sends the ghost along a route to Pac-Man’s last known location. This action only executes if Pac-Man’s location is known by the ghost through communication, perception, or sensor detection. “Evade” sends the ghost in the opposite direction of Pac-Man. “Keep the same heading” maintains the ghost’s current heading. “Move randomly” moves the ghost to a random 4-directional one space location. “Move to staring location” sends the ghost back to its starting location on the map. “Noop” explicitly tells the game engine that there is no other action to perform for the current time-step. “Set heading on movement” changes the ghost’s heading to follow the direction it is moving to. The “set heading” actions sets the ghost’s heading to be up, down, left, or right. “Transmit Pac-Man’s location” sends a message with Pac-Man’s last detected location to all of its teammates. This action represents agent communication and the sharing of knowledge.

TABLE 1. SCRIPT ACTIONS AND CONDITIONALS IN PAC-CLONE

Actions	Conditionals
Clear Route to Follow	Am I Already Moving to the Planned Destination
Attack	Have I Generated a Route
Evade	Is Movement List Empty
Keep the Same Heading	Is Pac-Man Vulnerable
Move Randomly	Is Pac-Man Detected
Set Heading	Is Pac-Man Detected by Communication
Set Random Heading	Is Pac-Man Detected by Perception
Set Heading on Movement	Is Pac-Man Detected by Sensor
Transmit Pac-Man’s Location	Is Pac-Man within n Spaces
Move to Starting Location	
Noop	

Multiple actions can be given to the game engine at once but priority goes to the order the actions are received if there are conflicting actions, e.g. multiple heading changes, movement requests, and the presence of a noop with any other action. Although basic, this set of actions gives the ghosts the foundation of what is needed to capture their prey.

IF-statements evaluate conditionals that lead to nested IF-statements or actions to be executed based on the conditional’s outcome. Conditionals evaluate a specific aspect of a ghost’s or environment’s state. The “Am I already moving to the planned destination” conditional is true if the ghost is currently following a route to a specified location. “Have I generated a route” is true if the ghost has produced a route to follow and is travelling along it. “Is movement list empty” is true if the ghost has movement actions to follow. “Is Pac-Man detected” is true if Pac-Man is currently detected by communication transmitted, by self-perception, or by the ghost’s sensor. The communication detection conditional is based on the action to transmit Pac-Man’s location. Pac-Man is detected by perception if the ghost’s heading is pointing at Pac-Man and there is LOS. Pac-Man detection from a sensor is activated if Pac-Man is within one space of the ghost. “Is Pac-Man vulnerable” is true if Pac-Man has not consumed a power-pill within 15 time-steps. If this conditional is true, the ghost can capture Pac-Man if they share the same environment location. The range conditionals return true if Pac-Man lies within the appropriate range. For instance, the condition “is Pac-Man within 3 spaces” is true if Pac-Man’s location is 3 or less spaces from the ghost.

IF-statements can have nested conditionals with the use of AND, OR, and NOT operators. The IF-statements determine which actions to execute and when. **Fig. 1** is an example of a ghost script with 2 top-level IF-statements. Together, the 2 IF-statements guide the agent towards or away from Pac-Man, depending Pac-Man detection and vulnerability state.

This implementation of Layered Learning method outputs scripts for each NPC ghost to eliminate Pac-Man in Pac-Clone. Because scripts are used to drive the decision making of each game character, and the fact that the game provides a defined agent environment, set of constraints, and task, Pac-Clone is an excellent test bed for analyzing how the proposed method performs at solving a shared complex task for a team of agents.

```
IF(isPacManDetected AND isPacManVulnerable,
attack,moveRandomly)

IF(isPacManDetected AND NOT(isPacManVulnerable)
AND isPacManWithIn3Spaces, evade,
MoveToStartingLocation)
```

Figure 1. Pac-Clone ghost script example

B. *Layered Learning with a Genetic Algorithm*

The goal of the implementation is to produce a team of three ghosts that is able to capture Pac-Man the fastest. To accomplish this goal, the implementation evolves three scripts together that represent the decision making knowledge of the teammates of a ghost team. To start the evolution process, a collection of teams are created. Each team in the collection is generated initially with a script of random conditions and actions. At each layer, the collection of teams makes up the population that is inputted into a GA and each team is denoted as a chromosome. The GA feeds each team’s set of scripts into Pac-Clone to play a series of games against a hard-coded, static script Pac-Man, evaluates how well each team performs the current sub-task, modifies the scripts, and repeats until the halting condition has been reached. Here, the halting condition is 50 generations. In each generation, every team will play 10 games, be evaluated on a sub-task’s fitness function, and experience modification through a genetic operator or elitism.

The population of teams at the end of the GA phase is passed onto the next layer. The next layer repeats the GA process but with a different sub-task and a different fitness evaluation criterion. Once the final layer is complete, the team that produces the best aggregate results in the last batch of games is the output of the proposed method. The descriptions below further explain each GA component in detail.

C. *Chromosomes*

Each chromosome in the GA is made up of three alleles that represent the scripts that drive the decisions of the three ghosts in the game. During the creation of the population, each script

initially contains at most 20 randomly generated, top-level IF-statements; the script is allowed to grow and shrink in IF-statement size over the entire learning phase. Each top-level IF-statement has no more than 4 degrees of nested IF-statements, meaning the maximum number of IF-statement levels that can exist in the script is 5 nested IFs deep. The number of nested IF-statements decreases by one at each degree level below the top. Each IF-statement is also restricted to have at most 5 conditionals and 5 actions that correspond to true and false paths. The number of nested IF-statements, conditionals, and actions are randomly generated during each IF-statement's creation. Size limitations are in place because individual scripts can become long, redundant, and unreadable for analysis.

D. Pac-Man

Pac-Man's static script aims at collecting all of the dots, power-pills, and fruit through the current level. If Pac-Man detects a ghost while vulnerable, he will evade the ghost. If Pac-Man is invulnerable and detects a ghost, he will pursue and attack the ghost until the ghost is captured or Pac-Man becomes vulnerable. To make Pac-Man efficient, every dot, power-pill, and fruit location is stored in a memory bank until Pac-Man reaches that location. At times when Pac-Man does not perceive or sense one of these items, he checks his memory bank for an item to pursue. If Pac-Man is not evading, attacking, or in pursuit of an item, he is moving randomly, with the hope of detecting an undiscovered item. Although Pac-Man's script is static and susceptible to exploitation, it provides a formidable training foe and standardizes team evaluation.

E. Fitness Function

During each layer, one of the three discussed fitness functions is used to rank the teams. In the first layer, teams that produce the highest probability of capturing Pac-Man in a generation receive the highest fitness. The second layer rewards teams that lead to shorter game durations. Finally, the third layer gives preference to teams that excel in the aggregate evaluation of capturing Pac-Man with a high success rate and with time efficiency. Below is the equation for calculating the aggregate fitness function. Capture and game duration weights were used to give more importance to Pac-Man captures and to reduce occurrences of short game durations from ghosts allowing Pac-Man to achieve his task. $f(x)$ in **(1)** is the function used to calculate the aggregate fitness for a chromosome team after it has played a set of games, x. $W_{Capture}$ and $W_{GameDuration}$ are the fixed weights that scale the capture and game duration values; they are set at 80 and 20, respectively. n is the total number of games played by a team in a generation. *Maximum_Game_Length* is the constant limit of how long a game can be played, and is fixed at 500 time-steps. c_i is the capture result in the ith game of the sample of games passed in by x. d_i is the game duration in time-steps of the ith game in the set of games x.

$$f(x) = \frac{W_{Capture} \cdot \sum_{i=0}^{n} c_i}{n} - \frac{W_{GameDuration} \cdot \sum_{i=0}^{n} d_i}{Maximum_Game_Length} \quad (1)$$

F. Elitism

Upon the completion of a generation, the best fit 10% of the population is copied directly into the succeeding generation's population. This form of elitism guarantees that the best fit chromosomes are not lost after applying the genetic operators to make up the rest of the next generation's population.

G. Selection and the Genetic Operators

Tournament selection for reproduction in the GA works as follows: 10% of the population is randomly selected and the best fit chromosome is chosen to be a parent for crossover. The process is performed again to select a second parent. To perform crossover, a random location on a ghost's script is selected from the two parents. All of the actions and IF-statements from the beginning to the random crossover point of the script in the first parent are copied to the first offspring. All of the actions and IF-statements from the random point to the end of the script on the second parent are appended to the first offspring's script. The process is repeated for the second offspring, but this time starting by copying the beginning sequence from the second parent followed by the ending sequence from the first parent. Using these methods of selection and crossover permits only better fit chromosomes to pass their information to future generations and emphasize the idea of survival of the fittest.

Mutation occurs by going through every top-level IF-statement in an offspring's script and randomly generating a number. If the random number is less than 10% (the mutation rate) then that IF-statement is replaced with another random IF-statement. Also, every non-elite chromosome has a 1% chance that it will be replaced with a new, randomly generated set of scripts. Mutation and a chance of new chromosome production adds variability to the population and assists in fighting the localization problem, where chromosomes start fixating on a single solution and never explore other possibilities.

H. Genetic Algorithm Parameters

The normally distributed population of the GA is fixed at 50 chromosomes and evolved for 50 generations for each fitness function. At every generation, each chromosome plays 10 games, 5 games in the first fixed training scenario (game level) and 5 games in a second. The maximum time limit for a single game of Pac-Clone is 500 time-steps.

To summarize, this proposed Layered Learning method decomposes the complex task of capturing prey quickly in 3 training layers: capture the prey, minimize the hunting duration, and an aggregate sub-task to unify what was learned in the first 2 layers. To develop the decision making of a team of NPC predators, a GA is used by each layer to train sets of 3 scripts to perform the layer's sub-task. The genetic operators are used to modify the scripts of each team. In every layer, the games played are used to evaluate how well each team performs at the current sub-task. Each layer's final population is directly copied into the next layer, where only the fitness criterion for team evaluation is updated for the new sub-task. The best performing team of the last layer's final population represents the evolved team of heterogeneous NPCs.

V. EXPERIMENTS

Experiments were established to compare Layered Learning with monolithic GAs. The monolithic GA approach uses a single fitness function for evolution; it is the equivalent to evolving a population with one of Layered Learning's layers.

For these experiments, the monolithic approaches use the proposed Layered Learning method's fitness functions. The first monolithic approach trains agents to capture Pac-Man, using the same fitness function as the first layer in the proposed method. The second monolithic approach uses layer two's fitness function and seeks to produce ghosts that minimize the game duration. The third and final monolithic approach uses the aggregate function of layer three and is completely independent of the first two monolithic approaches. Reusing Layered Learning's fitness functions for the monolithic approaches allows for fairness in comparison to see if Layered Learning's progressive technique leads to more effective production of an NPC team.

Because the monolithic approaches are single layered GAs, the monolithic GA parameters are exactly the same as the proposed set-up, only deviating in the number of generations per layer. Due to the Layered Learning approach having 50 generations per layer with 3 layers, each monolithic approach is allowed to train with one layer of 150 generations. The equivalent total number of generations further promotes fairness in the comparison along with the corresponding fitness functions between the monolithic and layered approaches.

The experiments break up the process of evolving NPCs for the Layered Learning and monolithic methods into two phases: learning and evaluation. In the learning phase, 100 evolutionary runs are made that result in each team's decision making behavior to be modified. The evaluation phase takes each method's final population and runs 200 additional games, using the same parameters and game scenarios used during the learning phase (100 games per game scenario). No scripting modification takes place in the evaluation phase. The same scenarios and parameters are used in the evaluation and learning phases because the goal is to show which method performs the best on a trained scenario.

To assist in the comparisons, each method ranks its population based on its respected fitness criterion and gathers results on how the method performs with the other fitness functions, solely for analysis purposes. The unused metrics have no effect on fitness ranking and does not affect the performance of any of the evolved teams.

The experiment results show that the proposed Layered Learning with a GA approach performs well against the monolithic GAs. This conclusion is determined by the performance results of Layered Learning's progress during the learning phase and the best fit generated teams at the end of the evaluation phase of each approach.

As a result of the learning phase, the Layered Learning approach was able to evolve its population to progressively solve the Pac-Man task. This conclusion is supported by **Fig. 2**, which plots the average aggregate score of each approach's best fit ghost-team received at each of generation over the course of 100 independent trials (the shaded regions indicate the 95% confidence intervals). It should be noted that the provided confidence intervals are half the value of their respective confidence region. From this phase, several observations are noted. First, as identified in **Fig. 2 and Table 2**, monolithic aggregation outperforms the monolithic captures and game duration approaches, in terms of average aggregate fitness. Monolithic aggregation excels because its fitness function causes the GA to explicitly maximize the aggregate fitness function during learning. The other two monolithic approaches focus on different measures of fitness, which act as handicaps in this aggregate comparison. **Table 2** displays the average aggregate fitness of the best individual at the end of the final generation over 100 trials, where the monolithic aggregate approach scored 59.92 (95% confidence interval 0.88), monolithic captures scored 59.26 (95% CI 0.99), and monolithic game duration scored 53.47 (95% CI 1.27) on average. From these averages, there is an advantage monolithic aggregation has over the other two monolithic approaches. The advantage is slight when comparing the aggregation and captures approach because the average fitness for captures falls within monolithic aggregation's 95% confidence interval. The advantage is more noticeable in game duration because the average aggregate score for the game duration approach falls outside of monolithic aggregate's 95% confidence interval.

Second, the Layered Learning method produces individuals that rival the top average performing monolithic approach, monolithic aggregation. The Layered Learning approached produced an average aggregate score of 60.67 (95% CI 0.87). Layered Learning's performance at this phase is attributed to the GA training the population on all 3 sub-tasks and spending the last segment of learning on maximizing the aggregate score.

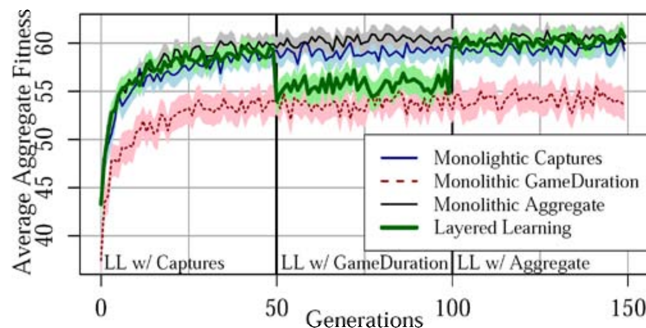

Figure 2. Average aggregate fitness of the best team per generation (95% confidence intervals are shaded)

TABLE 2. AVERAGE AND MAXIMUM FINAL GENERATION AGGREGATE FITNESS OF THE BEST TEAMS OF EACH LEARNING METHOD

Method	Average	Std Dev	95% CI	Max
Layered Learning	60.67	5.427846	0.86862	67.594
Aggregate	59.92044	5.473278	0.875891	68.11
Captures	59.26306	6.205052	0.992997	68.008
Duration	53.46854	7.916967	1.266955	68.08

Third, oscillations in average aggregate fitness are noticeable when Layered Learning transitions from layer to layer. These oscillations are caused by the layered approach changing its fitness evaluation every 50 generations. As displayed on **Table 2**, on average, the monolithic captures and

aggregate approaches outperform the game duration approach when the learning phase is complete. This performance difference explains how when the fitness evaluation is changed to an underperforming fitness function then the performance of the overall task can suffer; the opposite is true for well-performing fitness functions, as observed when the third layer begins and trains with the aggregate sub-task. The dip in performance indicates that the composition of layers may not be ideal for this problem.

Finally, the jump in aggregate fitness between the second and third layer shows Layered Learning's robustness. Although the aggregate fitness score suffers in the second layer, the layered approach is able to quickly recover early in the third layer. The robustness is caused by the fitness function transitions inherent to Layered Learning; in this composition of sub-tasks, the layered approach moves to a fitness function that performs the overall task well to third layer. The robustness is also attributed to the layered approach's ability to retain information from past layers. This ability to call on knowledge learned in previous layers permits the approach to quickly revert unnecessary changes made in the game duration layer and use helpful knowledge obtained from the first two layers during the aggregate sub-task.

The learning phase demonstrates that this approach of Layered Learning produces a team of NPCs that performs as well as two of the monolithic approaches (captures and aggregate) and outperforms the monolithic game duration approach on average during learning. Although Layered Learning produces the best aggregate scoring team at the end of the learning phase, the averages for the top performing monolithic approaches are within Layered Learning's 95% confidence interval. In addition, the learning phase results demonstrate how adaptable and robust Layered Learning is at learning a task by being able to overcome sub-optimal task decomposition and produce the best performing aggregate results on average.

As the learning phase concentrated on comparing the different approaches by examining performance averages, the evaluation portion is concerned with answering the question of which approach produces the best team of NPCs. From evaluation, the Layered Learning method's best individual produced a ghost team that was capable of capturing Pac-Man 161 out of the 200 attempts and in an average 90.59 (standard deviation 87) time-steps. With both metrics, the Layered Learning method produced an aggregated score of 50.9146. The aggregate approach produced an 86% capture rate with 172 captures (std. 0.35), and an average of 84.22 (std. 67.49) time-steps per game. The monolithic method with the capture Pac-Man sub-task produced 163 (std. 0.39) captures and an average game duration of 107.76 (std. 146.42). The monolithic method based on minimizing game duration captured Pac-Man 85% of the time (std. 0.36) in the 200 evaluation games and required 75.03 (std. 53.69) time-steps to do so on average. **Tables 3 and 4** display the evaluation phase results of the 200 games for each of the methods.

The main observation is how the Layered Learning method performed compared to the aggregate methods. To make this comparison, T- and Z-tests were made to see if there is any significant difference between the averages of the best performing teams generated between the different approaches. Although Z-tests show that there is no significant difference between any of the monolithic and layered learning approaches for the captures metric, the best performing Layered Learning team produced the least amount of captures and the third best averaged game duration. T-tests show there are no significant differences between the layered and the monolithic aggregate and captures approaches for minimizing game duration. While the comparison between Layered Learning and the monolithic game duration approach is quite close, once the critical values are adjusted via the Holm-Bonferoni correction for 3-way comparison, there's no significant difference here, as well. Lastly, there are no significant differences in means between the capture rates of Layered Learning and any of the monolithic methods. P-values from the T- and Z-tests can be found on **Table 5**. In the aim of comparing average outputs of the monolithic and the proposed layered approaches, the conclusion of which method outperforms the other cannot be definitively made.

TABLE 3. EVALUATION RESULTS OF THE BEST PERFORMING MONOLITHIC CAPTURES AND GAME DURATION GAS TEAMS

	Sub-task Captures		Sub-task Duration	
	Captures	**Duration**	**Captures**	**Duration**
Sum	163	21552	169	15006
Average	0.82	107.76	0.85	75.03
Std Dev	0.39	146.42	0.36	53.69

TABLE 4. EVALUATION RESULTS FROM THE BEST PERFORMING MONOLITHIC AGGREGATE GA AND LAYERED LEARNING TEAMS

	Sub-task Aggregate		Layered Learning	
	Captures	**Duration**	**Captures**	**Duration**
Sum	172	16843	161	18118
Average	0.86	84.22	0.81	90.59
Std Dev	0.35	67.49	0.4	87

TABLE 5. P-VALUES FROM T- AND Z-TEST RESULTS COMPARING LAYERED LEARNING TO EACH MONOLITHIC METHOD

Method Comparison	Captures (Z-Test)	Duration (T-Test)
Layered Learning- Mono. Aggregate	0.0704	0.4134
Layered Learning- Mono. Duration	0.1462	0.0321
Layered Learning- Mono. Captures	0.3994	0.1549

The evaluation phase results show that the monolithic game duration approach produced the best aggregated fit team. At first glance, this conclusion seems contradictory to what was found in the learning phase. However, the evaluation phase is concerned with examining the upper tails of the distributions of the 4 approaches and not the averages. In both phases, monolithic game duration produced the best aggregate performing team, although the average team produced with this method was the most underperforming with the highest variance among the approaches. **Table 2** helps explain how the monolithic game duration approach produced an outlier that

excelled in the aggregate evaluation by listing the aggregate score of the best produced team for each approach. As a result of the learning phase, 95% of teams produced with the game duration method hovered around the average aggregate score of 53.47.

The experiments generated results that justify the use of Layered Learning to evolve a team of NPC agents. The layered approach performs statistically as well as monolithic methods under the same conditions and with the same metrics collected. During the learning phase, the Layered Learning method outperformed the monolithic approaches by producing the highest average aggregation score. It should be noted that the top 2 monolithic approaches of captures and aggregation produced averages that fall within range of Layered Learning's aggregate 95% confidence interval. The evaluation phase deemed the monolithic game duration approach to be the top performing. Although the best team is considered an outlier, the outcome is interesting as the approach generated the worst performing average aggregate score among the approaches at the end of the learning phase. From these experiments, one can conclude that Layered Learning has demonstrated that it is an effective alternative to the standard monolithic approach of evolving decision making.

VI. CONCLUSION

The work presented here produces an effective team of heterogeneous agents who can perform a shared, complex task. To accomplish this feat, Layered Learning with a Genetic Algorithm was used to decompose a complex task into simpler components and train an agent team how to perform each sub-task. The proposed method decomposed the shared task into 3 layers: 1) perform the task effectively; 2) perform the task efficiently; 3) perform the task effectively and efficiently. The final layer aggregates the sub-tasks of the first 2 layers and represents the overall task to learn.

To evaluate the performance of the proposed layered method, a team of video game NPCs was evolved to learn how to excel in a predator-prey scenario. Pac-Clone, a Pac-Man inspired game was used to set the environment, roles, and constraints of the agents. The implementation saw a team of NPC ghosts locate and capture Pac-Man with a high probability and in a relatively small amount of time. The results are interpreted that Layered Learning performs statistically as well as the standard monolithic GA method to evolve the decision making of computer-based agents. The results are significant because conflicting literature exists differing on the performance of the layered-based approach.

This work also shows that Layered Learning has benefits to being deployed over monolithic GAs. First, the technique provides a natural way of teaching a complex task by decomposing it into simpler sub-components. Second, Layered Learning creates an organized approach of instruction through phases of simple sub-tasks to learn. Third, decomposing a task into 3 sub-tasks that progressively makes the agent team effective and efficient is proven to be a valid layering technique. Finally, Layered Learning is shown to be a robust method of learning. The robustness is demonstrated by the technique's ability to overcome underperforming layers and retain acquired knowledge from well-performing ones. Future work will examine the role of adequately decomposing the task into layers, study how certain tasks should be decomposed, and the effects varying task decompositions will have on performance. In summary, the experiments showed that the proposed Layered Learning method produces a team of heterogeneous NPCs that performs similarly effective to those evolved with a monolithic approach but the layered technique stands out as it has several key benefits of being deployed.

REFERENCES

[1] D. Jackson and A. P. Gibbons, "Layered Learning in Boolean GP Problems," in Proc. *10th European Conference on Genetic Programming*, Valencia, Spain, 2007, pp 148-159.

[2] P. Fidelman and P. Stone, " The Chin Pinch: A Case Study in Skill Learning on a Legged Robot," in Gerhard Lakemeyer, Elizabeth Sklar, Domenico Sorenti, and Tomoichi Takahashi, editors, *RoboCup-2006: Robot Soccer World Cup X*, Berlin, Germany, 2007, pp. 59-71.

[3] P. Stone and M. Veloso, "A Layered Approach to Learning Client Behaviors in the RoboCup Soccer Server," *Applied Artificial Intelligence*, vol. 12, pp. 165-188, 1998.

[4] P. Stone and M. Veloso, "Layered Learning," in Proc. 11th European Conference on Machine Learning, Barcelona, Spain, May/June 2000, pp. 369-381.

[5] F. Gomez and R. Miikkulainen. "Incremental Evolution of Complex General Behavior," Adaptive Behavior, vol. 5, 1997, pp 317-342.

[6] S. Whiteson, N. Kohl, R. Miikkulainen, and P. Stone, "Evolving Keepaway Soccer Players through Task Decomposition," *Machine Learning*, vol. 59, pp. 5-30, 2005.

[7] S. M. Gustafson and W. H. Hsu, "Layered Learning in Genetic Programming for a Cooperative Robot Soccer Problem," in Proc. *European Conference on Genetic Programming*, 2001, pp. 291-301.

[8] W. H. Hsu and S. M. Gustafson. "Genetic Programming for Layered Learning of Multi-agent Tasks". *In Late-Breaking Papers of the Genetic and Evolutionary Computation Conference*, San Francisco, USA. 2001.

[9] T. Iwatani, Pac-Man, Namco Limited, Tokyo, Japan, 1980.

[10] J. Pittman, (2009, February 23). *The Pac-Man Dossier*. [Online]. Available: http://www.gamasutra.com/view/feature/3938/the_pacman_dossier.php