DEVELOPING INTELLIGENT AGENTS FOR TRAINING SYSTEMS THAT LEARN THEIR STRATEGIES FROM EXPERT PLAYERS

A Thesis

by

JONATHAN HUNT WHETZEL

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2005

Major Subject: Computer Science

DEVELOPING INTELLIGENT AGENTS FOR TRAINING SYSTEMS THAT LEARN THEIR STRATEGIES FROM EXPERT PLAYERS

A Thesis

by

JONATHAN HUNT WHETZEL

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Richard A. Volz
Committee Members,	Thomas R. Ioerger
	Winfred Arthur, Jr.
Head of Department,	Valerie Taylor

August 2005

Major Subject: Computer Science

ABSTRACT

Developing Intelligent Agents for Training Systems That Learn Their Strategies from Expert Players. (August 2005).

Jonathan Hunt Whetzel, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Richard A. Volz

Computer-based training systems have become a mainstay in military and private institutions for training people how to perform certain complex tasks. As these tasks expand in difficulty, intelligent agents will appear as virtual teammates or tutors assisting a trainee in performing and learning the task. For developing these agents, we must obtain the strategies from expert players and emulate their behavior within the agent. Past researchers have shown the challenges in acquiring this information from expert human players and translating it into the agent. A solution for this problem involves using computer systems that assist in the human expert knowledge elicitation process. In this thesis, we present an approach for developing an agent for the game *Revised Space Fortress*, a game representative of the complex tasks found in training systems. Using machine learning techniques, the agent learns the strategy for the game by observing how a human expert plays. We highlight the challenges encountered while designing and training the agent in this real-time game environment, and our solutions toward handling these problems. Afterward, we discuss our experiment that examines whether trainees experience a difference in performance when training with a human or virtual partner, and how expert agents that express distinctive behaviors affect the learning of a human trainee. We show from our results that a partner agent that learns its strategy from an expert player serves the same benefit as a training partner compared to a programmed expert-level agent and a human partner of equal intelligence to the trainee.

For my Mom & Dad.

Without them, you would not be enjoying this excellent thesis.

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Dr. Richard Volz, whose guidance helped me throughout my graduate studies. He has been my teacher, my boss, but most importantly he's been a role model whose advice and example I'll try to follow for the rest of my career.

Several people played a role in the development of this project that I should acknowledge. Funding for this research was provided by the DoD MURI grant (#F49620-00-1-0326) and a National Physical Science Consortium fellowship. Thanks to my committee member, Dr. Winfred Arthur, Jr., and his students, Anton Villado and Alok Bhupatkar, for their knowledge and assistance with the cognitive psychology aspects of this research. I thank Barbara Braby for her help in managing the human subject experiment. Also, thanks to the following people whose insight helped form the project for this thesis: Raymond Harrigan, Thomas Ioerger, Travis Bauer, Ann Speed, Robert Abbott, and Kevin Dixon. As well, thanks to the following people for assisting in the creation of the game environment used in this thesis: Sen Cao, Maitreyi Nanjanath, J. Colby Johnson, Linli He, and Joseph Sims.

Special thanks to the following people I have met over the past few years who have been invaluable mentors to me during my graduate studies: Richard Volz, Thomas Ioerger, Luke McClannahan, Diana Eichert, Winfred Arthur, Jr., and Raymond Harrigan.

Lastly, I would like to thank my family (Mom, Dad, and Christie) and my friends for their continuing love and support.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In the recent past, we have witnessed an explosion in the popularity of video games serving as training tools. These games can provide a trainee with a meaningful experience at a fraction of the cost of trying to provide the same experience in the real world. As these games model more tasks that arise in real life, we question how can computer systems better assist in the development of a trainee's skills for these complex tasks. A complex task refers to a task that challenges a person's cognitive abilities (i.e., reasoning, memory, attention) and often includes their physical abilities.

In particular, researchers have studied how intelligent agents could assist trainees by serving as virtual teammates or tutors. Researchers have altered accepted team training protocols by replacing the human partners with artificial ones. For example, The AIM (Active Interlocked Modeling) protocol [1] is a known method for helping trainees learn a game's complex task by decomposing the game into a set of subtasks, and having trainee teams work together with each trainee performing a specific subtask. Periodically, the trainees will trade subtasks, eventually exposing every trainee to all the subtasks. At the conclusion of training, the trainees working under the AIM protocol had achieved a higher performance than trainees who learned the game by themselves. In a paper by Ioerger *et al*. [2], the protocol was taken a step further by replacing some human teammates with intelligent agents. It was shown that human trainees working with intelligent agents in this modified protocol also attained better performance than trainees who learned the game alone.

Studies have shown that for maximizing the effectiveness of training with a partner, virtual or human, the partner must: have great acumen for the task at hand, and not perform at a level that the trainee could never reach [3]. Satisfying the first requirement means that the agent's strategy must reflect that of a human expert. Hence, we need some methodology for attaining an expert player's strategy and translating that knowledge into the agent. Typically, we rely on cognitive psychologists for eliciting an

This thesis follows the style and format of *IEEE Transactions on Knowledge and Data Engineering.*

expert player’s strategy through a cognitive task analysis; a series of interviews and observing the expert player’s behavior within the game [4]. Yet these techniques have inherent flaws due to their reliance on subjective information from both the player and the cognitive psychologist. For instance, a player may forget what motivated them to take certain actions within the game [5], or they may learn the skills needed for the game implicitly, without gaining any understanding about the game’s structure [6]. In either case, one would obtain information from an expert that does not detail how changes in the game environment correlate to the player’s actions. A developer would have to interpret this incomplete, and potentially inaccurate, information into the strategy used by the agent; increasing the likelihood that the agent’s behavior will not resemble the human expert player.

The second requirement states that the abilities of a teammate should not greatly exceed those attainable by the trainee. In Ioerger *et al*. [2], they developed a partner agent for the game *Space Fortress* by using documented information about an expert player’s strategy. The first iteration of the agent achieved total scores of 8000, considerably above a typical human expert player’s total score of 5000 – 6000. During these initial experiments, trainees teamed with this agent complained that the behavior of the agent posed more of a distraction than an asset. In the AIM protocol, researchers explain that while a trainee handles their subtask, they will internally model the behavior of their teammates. Once the trainee switches subtasks with their teammates, the trainee will use their model to help them better understand and perform the new subtask [7]. This suggests that as the expertise of the teammate increases, they should provide better strategies for the trainee to imitate and learn. In this case, the teammate handled the subtask “too perfectly”, producing a strategy that the trainee could not comprehend, and thus preventing the trainee from developing a mental model that would aid them in learning the teammate’s subtask.

As a solution, random delays were inserted into the agent’s activation and release of the game controls, hoping that incorporating these delays into the agent would emulate the physical reaction times of a human expert. Once in place, the delays were adjusted until the agent’s total scores matched the range of a human expert. Trainees responded positively to this modified agent, and their performance exceeded that of trainees who learned the game on their own. On the other hand, the random delays altered the behavior of the agent such that it no longer resembled a human expert’s strategy. An agent whose scores not only equal that of a human expert, but also utilize the same strategies as an expert player may improve the training experience for human trainees.

This thesis looks into how computing systems can assist in developing agents that better reflect the behaviors of expert human players. Instead of using the conventional methods that require obtaining expert player knowledge and using it to program the agent’s strategy, we used techniques that allow an agent to “learn” an expert player’s strategy. Using computing systems, we captured a player’s strategy by observing empirical data that describes the player’s behavior in the game. This thesis documents the design and production of an agent that learns an expert’s strategy for the game *Revised Space Fortress*, a game that represents a complex task found in real-life training systems. We compared and highlight the behavioral differences between our agent and an agent developed using the traditional methods of expert knowledge acquisition. Through experimentation, we checked if the discrepancies in behavior impact the performance of trainees working with either agent. We hypothesized that an agent that learns its strategy from an expert player will provide an equal, if not greater, benefit to a human trainee as a partner during training compared to a partner agent that follows an explicit, programmed strategy. This thesis describes the human subject experiment we established for testing this hypothesis and results from it.

2 ARTIFICIAL INTELLIGENCE IN GAME ENVIRONMENTS

For several years, researchers have studied the application of artificial intelligence principles into agents for game environments. Yet, as the tasks and environments that the games modeled grew in complexity, so did the difficulty in utilizing traditional AI techniques for effective agents. Traditional AI techniques refer to a developer programming the algorithms or rules the agent will follow in order to accomplish the goals within the task environment [8]. One may envision including the extensive research conducted over discrete game environments, such as advanced search algorithms for board games (i.e., IBM's Deep Blue) [9]. Yet, our focus only exists on agents developed for continuous, real-time game environments. If the developer has complete knowledge about the environment and the appropriate strategies for the task, this approach should generate an effective agent. However, we noted in the introduction that agent developers for complex task environments do not receive comprehensive information about an expert player's strategy. Thus, relying solely on the developer's knowledge about the expert player may not necessarily be in the best approach for creating an agent for a complex task environment.

As a solution, researchers have looked toward other fields for assisting in agent development such as machine learning, classification algorithms that automatically improve in accuracy over time [10], and the field of data mining, building algorithms that take large quantities of data and find useful, interesting information within them [11]. These fields can provide tools for analyzing a player's behavior and discover how they develop their strategies for completing a task. In this section, we will explain the pitfalls of traditional AI for complex task environments, and give more detail about using machine learning and data mining techniques for developing agents. This section includes discussing two different approaches for training an agent for a complex game environment, supervised learning and reinforcement learning, and explains why we chose supervised learning for our agent. Also, this section covers a couple of algorithms one can use for implementing a supervised learning approach for learning a human player's behavior (i.e., neural networks and rule-based learners). Furthermore, we discuss some related work involving these machine-learning techniques for learning a human expert's behavior.

2.1 Difficulties in Traditional AI Techniques

By definition, a strategy is a plan for solving a problem or performing a task [12]. This statement gives the impression that we could develop an agent that successfully completes a specific task by having the agent simply follow a series of rules. Russell and Norvig [13] classify this as a knowledge-based approach, where the agent possesses a rule set that will dictate its actions based upon the agent's current knowledge about the environment. For building an expert agent, a developer crafts the rule set such that the agent will take the most rational action for any situation it faces. While immersed in the environment, the agent will perceive data that describes the world and update its knowledge base. Based upon the changes to the agent's knowledge base and goals, a subset of the rules will trigger and guide the agent toward attaining its goal. Using this approach, researchers have developed agents for complex game environments, such as the *QuakeBot* produced by Laird [14] and his associates for the first-person-shooter *Quake*. Consulting with expert players in the *Quake* tournaments, Laird generated a rule set that allowed the agent to mimic their behaviors in a variety of scenarios within the game.

For example, in the game *Grand Theft Auto III: Vice City* a player must reach certain locations in a virtual urban environment within a fixed time deadline. A player has a number of methods for accomplishing this goal, yet the most popular (and disturbingly entertaining) choice for experts is carjacking a nearby vehicle and driving to that location. As shown in Figure 1, we could easily use the knowledge-based approach for instructing an agent to follow this same behavior.

If CarNearby(PlayerPosition) AND
(OverFiveBlocks (Distance(PlayerPosition, GoalState))) $\rightarrow$ *Take Car*

Figure 1: Example of Using a Traditional Rule Set Within a Complex Task Environment.

In this example, an agent will take a car if it observes one nearby and the distance between itself and the goal state exceeds five city blocks. Although this instance sounds simple, how would we define the rules that describe the agent's behavior for the rest of the trip? This requires taking into account a multitude of potential variables: the terrain of the city, traffic flow, location of police officers, etc. As the number of possible scenarios increases, so does the task of generating the rules that will handle every scenario, which makes the knowledge-based approach less attractive for developing an agent in complex task environments.

2.2 Supervised Learning

Instead of attempting to describe how a player operates in every possible scenario, we need a system that helps us generalize the behavior of the player. Through supervised learning [8], we can design an agent that develops a strategy for handling a task by observing a human player. Supervised learning begins by collecting training examples, data that describes scenarios that the agent will see while performing the task. Every training example the agent views will have a classification attached to it. As the agent views each training example, the agent will make a prediction about the example's classification. For each training example the agent classifies incorrectly, the agent makes some adjustment to its learning structure. The agent continues viewing the training examples, adjusting its learning structure when necessary, until its accuracy in classifying the training examples can no longer improve. Once the agent enters the task environment, the agent should be able to recognize the scenarios from the training examples and make the proper classifications. For an unknown scenario, one not included in the training examples, the agent will classify it based upon which known scenario is the most analogous to the unknown one. Hence, the agent can make educated, and often correct, classifications about every scenario without exposure to all possible scenarios beforehand.

For learning human player behavior, the agent will learn to classify what actions a player takes at different moments in the game. We start by recording data of the player's performance in the game environment. This data describes the game environment at certain times along with the actions that a player took. From this data, we will select key features and generate training examples from them. These features will include information that describes the environment as seen on the screen (i.e., distance between objects, objects present, etc.) as well as traits that characterize the player's behavior (i.e., reaction time, the last time an action occurred, etc.). Each training example produced will be classified by what action the player took at that point in time. During training, the agent will observe the training examples and learn what action the player takes. Once placed into the game environment, the agent will interpret the input it receives and determine the scenario that best fits its current state. The agent will return with a classification for the scenario, the action that the agent should do, and execute it within the game. If trained correctly, the agent should mimic the actions that a player takes in the known scenarios. As well, the agent should have reasonable predictions for unknown scenarios, and take actions similar to what the human player would do at those times.

2.3 Reinforcement Learning

In another training approach, an agent can learn its strategy on its own through a process known as reinforcement learning [15]. In supervised learning, an agent trains by observing recorded data of a human player's behavior, and learns from the player the moments when it should take certain actions within the game. Reinforcement learning, though, begins by immediately placing the agent into the game environment, letting the agent experiment with different actions. Using a reward function, specified by the developer, the agent determines what actions during the game provide the largest reward. Through experience, the agent learns the correct actions that produce an effective strategy for the game environment. However, as the game environment grows in complexity, so does the difficulty for selecting a proper reward function [16]. As well, the reward function becomes further complicated if the agent must reflect the behaviors of a human player. With these challenges present, we decided upon using the supervised learning approach for our agent. Section 5 shows how we used the supervised learning approach for training our agent for a complex task environment.

2.4 Rule Based Learners

Using a supervised learning approach, one can develop an agent that generates a rule set based upon the examples and their classifications within the training data [17]. These rules represent the hypotheses about the observed training examples. Based upon the attributes' values within an example, the agent will use the rule set for predicting the example's classification. Given a set of training examples and a classification for each one (i.e., a player performing an action), rule based learners utilize the process of sequential covering for building the entire set. Sequential covering begins by determining the rule that most accurately describes the examples that contain the target classification. Hence, most of the positive examples, ones that contain the target classification, will be covered by this new rule. The algorithm repeats by greedily selecting another rule for the positive examples not included within the previously generated rule(s). This process repeats until the algorithm builds a set of rules that describes every positive example.

Although rule based learners can assist developers in creating rule sets for an agent, these learners require that the examples contain only discrete attributes, attributes with only a small finite number of values. Complex task environments, though, possess several real-number attributes, which can have an infinite range of values. Developers using rule based learners for these complex environments face the arduous task of converting these real-number attributes into discrete questions that the learner can use (ex.. "Is the value of attribute *x* greater than zero?"). If the training examples consist of several real-number attributes and the developer does not have detailed knowledge about the player's behavior, rule based learners may not provide the best solution for extracting a player's behavior in complex task environments.

2.5 Neural Networks

Another learning structure uses the analogy for biological learning systems for finding the classification hypotheses within a training data set [10]. Neural networks use a series of interconnected units, called neurons, with each unit taking a number of real-valued inputs and outputting a single real-valued number. Each neuron contains a set of weight values, with a weight assigned to a particular input. After combining these neurons together into a network, the network begins receiving the training example's attributes as input into the network. As the network observes each example, the neuron weights tune themselves such that the network output maps to the proper classification. After several iterations through the data, the weights within the network represent the classification hypotheses for the training data. The network represents a function that given any example should produce the correct corresponding classification as output.

Neural networks lend themselves to complex task environments due to their ability to process real-number attribute values. For any discrete attributes, a developer can easily convert its value into a set of real-number classifications that the network can accept. For this thesis research, we selected using a neural network structures for learning a player's behavior in a complex task environment. Yet, unlike rule based learners, neural networks do not give their classification hypotheses into a format easily understandable by a human. Developers have the challenge of interpreting the resulting network weights for determining what attributes had a greater bearing on classifying an example than others. This downside actually became a major problem in developing our learning agent for a complex task environment. Section 4 and Section 5.2.2 discuss how we alleviated this problem by using simplified neural networks for our agent.

2.6 Related Work in Agents Learning Human Behavior

Researchers have also recognized the drawbacks of traditional AI techniques for mimicking human behavior in game environments. This led to the development of several projects which were directed at determining how computing systems could assist in the knowledge elicitation process. As well, these researchers built agents that learned their strategy for a particular game by observing the behavior of a human expert. However, the game environments or the tasks carried out by agent do not match the complexity of the training systems that we will target. Some projects used agents for turn-based games, allowing agents to make decisions without adhering to a strict time limit [18]. In our focused training systems, the agent engages in dynamic environments where the game state continually changes, forcing the agent to make quick decisions at the appropriate times. For modeling human behavior, the agent must learn the moments when the player perceives the need for an action and the player's reaction time to execute the action.

Even when the game environments studied possessed the characteristics and complex tasks found within the training systems in which we are interested, the agents constructed only learned how to accomplish a simple sub-task. For example, Geisler's paper [19] investigated various machine learning algorithms for learning human behavior in a first-person shooting game. Typically, these games have a player handle a multitude of tasks such as engaging and defeating enemies, protecting themselves from attackers, navigating through hazardous, life-like terrains, and potentially seeking out a goal item. Yet, the agent in Geisler's paper concentrated on only learning how to negotiate through the environment, tracking down a goal item while handling potential threats (i.e., should the agent move toward or away from the enemy). Thus, the agent learned just a fraction of the skills needed for playing the entire game. Complex task training environments demand that an agent handle multiple tasks at the same time. Not only must the agent learn the skills needed for each task, but also the agent must have some framework for deciding when it should switch from working on one task to another.

Researchers are now taking steps in solving the challenges of eliciting expert player knowledge for highly complicated tasks through computing systems. At the University of Michigan, a system has been developed that lets a player document their strategy by drawing a series of diagrams [20]. The system, named *Redux* (short for "Rapid Knowledge Acquisition"), begins with an expert player drawing a diagram of a given game scenario and describing the end goal. The player then shows the actions he/she took in this situation along with their beliefs about the environment (i.e., location of unseen enemies, beneficial items, etc.). After recording this information, the system produces a new diagram reflecting the results of the player's actions, and prompts the player to input his/her actions and beliefs for this new situation. As a player progresses through the diagrams, the player can highlight new goals that arise, and show how activities were balanced to accomplish all of the known goals. This process repeats until the expert player completes every situation needed for accomplishing the task. Afterwards, the system will analyze the player's actions, beliefs, and goals attained in all situations, generating a set of rules that explains the player's strategy for that scenario. During validation of these rule sets, researchers observed that agents following these rules would exhibit different behavior than the expert player. They realized that expert players would highlight the wrong features when documenting the decision-making process in the diagrams, either neglecting critical changes in the environment or over-emphasizing trivial ones. With human players interpreting their own strategies, this system runs into the same problems as the traditional techniques for knowledge elicitation. By having computer systems focus more on empirical data about a player's behavior, we might improve the ability of these systems in capturing human expert player strategy.

3 REVISED SPACE FORTRESS

This section describes the game environment used for our research, *Revised Space Fortress (RSF)*. Developed at Texas A&M University [21], this game serves as research tool for testing training protocols. Although the game does not train an individual for any real-life scenarios, the game does represent the complex tasks that occur within military situations. This game is an adaptation of *Space Fortress*, an older game widely used by cognitive psychologists for testing experimental training methods [22]. *Revised Space Fortress* improves upon the original game by incorporating features such as the capability of running on multiple platforms and extensible game definitions. However, *Revised Space Fortress* does replicate the game environment found within *Space Fortress*, with tests showing that *Revised Space Fortress* offers a similar experience to a trainee [23]. In this section, we will explain the rules of a standard trial in *Revised Space Fortress* and explain how we evaluate a player's performance.

3.1 Rules of the Game

During a game, a player controls a ship on the computer screen (see Figure 2). The player may rotate the ship and move in the direction the ship faces by using a joystick. The player must navigate the ship and destroy the fortress located in the center of the screen. The fortress cannot move, but rotates to target and fire shells at the ship. The player must destroy the fortress as many times as possible without being hit by a fortress shell. A player must fire 10 missiles at the fortress followed by a double shot, two missiles that hit the fortress in less than 250 msec. All shots prior to the double shot must hit at intervals greater than 250 msec. The game lasts approximately three minutes, with the player told to destroy the fortress as many times as they can within that time frame. More precisely, a game consists of 3600 cycles; the timing can vary slightly due to a variable number of reset delays for certain events such as when a player destroys the fortress.

In addition, the player must handle mines that appear and pursue the ship. Mines may be classified as either friendly or foe, identified by an ASCII character that shows up at the bottom of the screen whenever a new mine appears. Prior to the start of the game, the player is given the three ASCII characters that signify a foe mine. If a player sees one of these characters when a mine emerges, the player must double-click the right mouse button before shooting at the mine (with an interval of 250-400 msec. between each click). If the ASCII character shown does not match any of the three for a foe mine, then the mine is considered friendly and the player may simply fire at it. If a player does not correctly identify the mine, the mine will collide with the ship or disappear after a period of time.

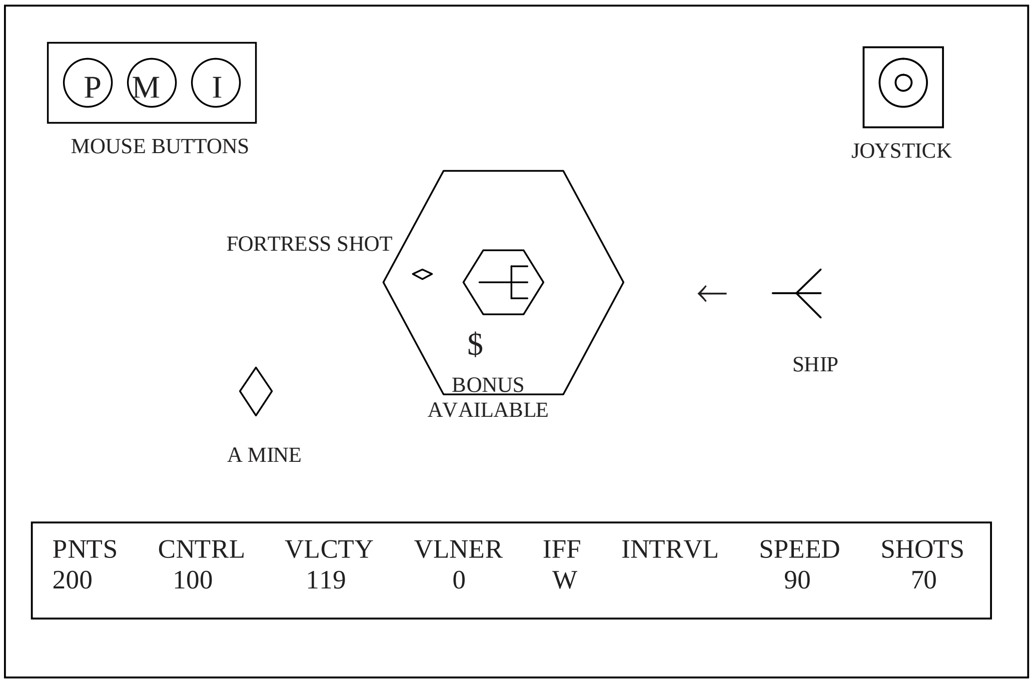


Figure 2: Screen from *Space Fortress.*

A player also has an opportunity to receive bonuses that appear within the game. A random character will periodically appear in lower left portion of the screen. If the player sees a ‘$’ character, then this indicates a player will have a chance at a bonus opportunity. Following this character, another ‘$’ character will appear, and the player may collect a bonus before the second ‘$’ character disappears. The player may increase their score or retrieve more missiles by pressing either the left or center mouse button respectively.

3.2 Evaluating Player Performance

For evaluating a player’s performance, *Revised Space Fortress* uses a scoring system based upon four sub-scores. For each game, the player is told to maximize their Total score, the sum of these four sub-scores. We have listed below a description of each sub-score and how they increase and decrease during the game.

- Points – This score increases whenever the mines or fortress are damaged or destroyed, and decreases if the ship takes damage. If a player chooses more points for a bonus opportunity, then this score will increase. Also, the player will lose points if their missile supply count goes negative (this count is raised by acquiring the missile bonuses).
- Velocity – This score increases if the player keeps the ship’s speed below a certain threshold, and decreases if the ship goes faster than the threshold.
- Control – This score increases if the ship stays between the two hexagons. It decreases if the ship wraps around the screen, goes outside the large hexagon, or runs into the inner hexagon.
- Speed – Player receives points based upon how quickly they identify and destroy a mine.

By analyzing these scores, we can attain a sense of the player’s behavior within the game. For instance, an expert player will generate high scores in each one of the above categories. Their high Velocity and Control scores indicate that the player moved the ship slowly in a clockwise pattern while keeping the ship within the hexagon boundaries. A high Points score not only reflects the accuracy of the player's shots, but how well they can accomplish the primary task while attending to secondary tasks that appear throughout the game. Novices usually receive total scores ranging from -3000 – -2000 with expert scores in the range of 5000 – 6000. Typically, experiments using this game have a trainee play 100 "3-minute" trials, allowing them ample practice to learn the skills for each task, and the attention management needed for deciding which task to perform.

4 AGENT DESIGN

As a preliminary step in designing our agent for *Revised Space Fortress*, we had to determine what types of machine learning techniques our agent would utilize. For learning an expert player's patterns of action, we initially decided upon using a series of neural networks, with each network responsible for controlling a specific action. Inside each network contains several weights that comprise the function the network represents. During the training process, we used the BACKPROPOGATION algorithm [10] for altering each of the weights. Typically, as the complexity of the task increases, so does the complexity of network used for learning it. However, we kept the sizes of the networks relatively small, with all networks containing either no or one hidden layer of neurons. With fewer weights, we could easily calculate by hand the network's output for different inputs, enabling us to see which inputs the agent considered most significant after training. This served as a great debugging tool that allowed us to explain what produced the agent's behavior within the game environment, and guided us in making the proper corrections to the input set for improving the agent's performance.

This section will discuss challenges in using this type of design for handling a complex task, and how one can alter this design to accommodate for these difficulties. This includes discussing how we decomposed the game into several simpler tasks for the agents to learn, and how we reassembled the skills the agent learned from the smaller tasks so it could perform the entire game.

4.1 Complex Task Decomposition

Initially, we envisioned that a single neural net (agent) for each game control would learn how to use the control by observing several games recorded from an expert player. However, we discovered that the problem the network(s) must solve was more complicated than we predicted. During the game, a player has several subtasks that they must manage, causing a player to periodically switch their focus from one subtask to another. Essentially, the agent must learn not only the skills needed for each subtask, but also learn how to recognize what subtask to execute at any given time. With each network needing to know how to recognize each situation and how to use their assigned control within each situation, the problem escalated beyond what could be accomplished with a single neural net.

Instead of abandoning the design to find another method, we adapted the design into a new approach. Guan *et al.* [24] show that as the complexity of a problem grows, so does the challenge in training a neural network that will solve it. Thus, we took the approach of decomposing the problem into a set of simpler ones, and constructing a network for each smaller problem. After training all the networks, we combine them for handling the original, complex problem.

Following this approach, we decomposed the game based upon the skills that one would need in order to play successfully. We determined the skills through a cognitive task analysis performed earlier for *Space Fortress* [25]. The skills that we selected were as follows:

- Correctly identifying and destroying a mine.
- Circumnavigating the fortress while remaining within the hexagon boundaries.
- Firing missiles at the fortress (normal shots and double shots).
- Recognizing and choosing the bonuses that appear within the game.

The agent will have a set of neural networks per skill, with each network within the set handling a control used by a player for that skill. For example, correctly identifying and destroying a mine would require a network for each of these controls: turning the ship left, turning the ship right, firing a missile, and pressing the right mouse button. The agent will learn the skills from an expert player, and will incorporate the player's task recognition behavior when we reassemble these skills together for playing the entire game.

The idea of relying on a cognitive task analysis (CTA) for the design of the agent seems contrary to our claims earlier in this thesis about the flaws inherent with CTAs. Yet, the dialogue between experts and developers in our approach focuses on helping developers understand the problem at hand within the complex task. This leads to a high-level explanation of what skills a player needs in order to play the game successfully. In the traditional agent design method, developers used expert knowledge for programming exactly how the agent executes each skill within the game environment. Instead, our approach uses machine learning techniques for learning the player's behavior for the skills. We still depend on expert players for understanding the complex task, but use an analysis of empirical data that describes the player's behavior, instead of continued conversations, for determining how the player carries out the task.

For learning each skill, we decided on building what we call "mini" game environments. Each of the "mini" games, which are reduced versions of *Revised Space Fortress*, focus solely on one of the skills in the above list. An expert player would play each of these "mini" games while we captured data about the player's behavior in the game. Thus, the agent would learn a specific skill by observing the data recorded from the according "mini" game.

Alternatively, one might draw an analogy to human learning which has been shown to be more effective with a Multiple Emphasis on Components (MEC) approach [26]. Considering this approach, we would use data from an expert player performing a series of normal games of *RSF* with different emphases. The data from these MEC games would show how an expert player handles all of the subtasks within the game, removing the need for these "mini" game environments. However, the analogy between human learning and agent learning breaks down at this point. Nothing within the data easily indicates what subtask the player handles at any given time. For each skill, we would have to parse through the data and extract only the information needed for learning how the player accomplishes the subtask associated with the skill. This would become further complicated since we cannot know precisely when a player switches from one subtask to another. For instance, a player may recognize and switch subtasks, but choose to take no action. Since the data reflects only visually observable behaviors of a player, we cannot correctly divide the data into a player's behavior for each subtask. By using the "mini" game environments, we obtain data that represents the player's behavior for each subtask, free of any noise produced by the player handling other subtasks.

4.2 Reassembling the Skill Sets

Once the agent learns each of the separate skills for the game, they must be combined so that the agent can play the entire game. Remember that each skill handles a different subtask within the game. Assembling the skills is then a matter of determining what subtask the agent should handle at any given time during the game. Ideally, we would want a framework that models the player's recognition patterns. In fact, researchers have started developing models that represent the context recognition process by humans in some task environments [27]. Instead we opted for rule-based approach using expert player knowledge about when a player decides which subtask they will handle at a particular time. Although simplistic compared to the context recognition modeling research, this rule-based approach provided a sufficient framework for our agent. We observed that some of the game controls appeared in more than one skill learned by the agent. For instance, the agent uses the ship's turning capability differently when circumnavigating the fortress and handling a mine. This meant that skills where the controls overlapped needed a priority structure. After consulting with expert players, we developed a decision-tree structure that would assign priorities to the various subtasks within the game. Once the agent determined which subtask it would handle, it would use the appropriate skill it had learned for handling that subtask. Figure 3 gives the decision-tree we created for the agent.

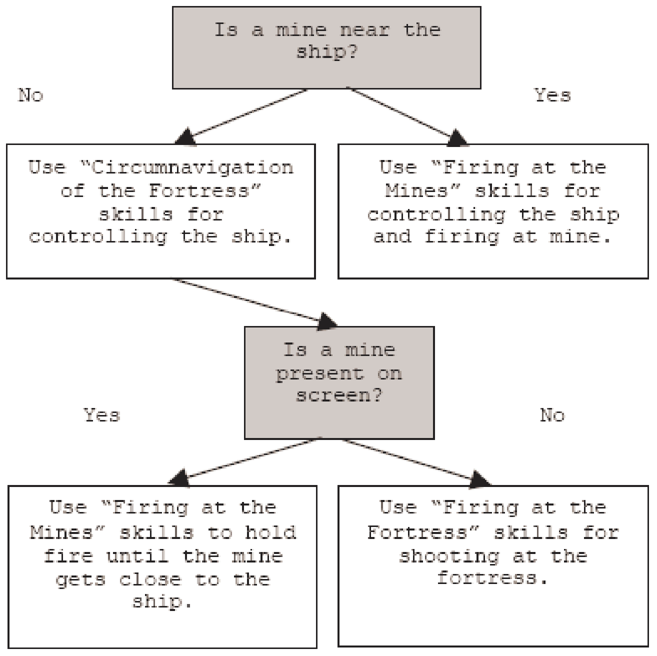

Is a mine near the ship?
No
Yes
Use "Circumnavigation of the Fortress" skills for controlling the ship.
Use "Firing at the Mines" skills for controlling the ship and firing at mine.
Is a mine present on screen?
Yes
No
Use "Firing at the Mines" skills to hold fire until the mine gets close to the ship.
Use "Firing at the Fortress" skills for shooting at the fortress.

Figure 3: Decision Tree for Selecting a Skill Set.

Not all of the game controls were handled within the decision-tree shown. Some controls only appear in only one of the skills learned by the agent (i.e., the left and center mouse buttons for bonus selections, the right mouse button for IFF). Since these controls were never in conflict with another skill, we did not need to include them within the decision tree structure. We let the neural networks handling these non-conflicting controls run in parallel to the decision tree structure.

Whiteson and Stone [28] present similar ideas about agents learning strategies for a game environment. Their process, concurrent layered learning, also decomposes the game environment into simpler tasks, and has the agent learn the skills needed for each of these tasks. Furthermore, reassembling the skills for the entire game relies on using a decision-tree structure that tells the agent which task it should handle at any given time. On the other hand, Whiteson and Stone decompose the game based upon a hierarchical design for the skills necessary for the game. Once the agent learns a skill, it will use that knowledge for learning the next skill. Hence, the complexity of the agent's behavior should increase as more skills are learned and included in the next phase of learning. However, the hierarchical skills require that an order exists for training each of the skills. For example, an agent must learn how to intercept, or run toward the ball, before it can learn how to pass the ball. In *RSF*, we deal with a series of *independent* tasks as opposed to *interdependent* tasks. With *independent* tasks, an agent may learn the skills in any order the developer chooses. Figure 4 gives a diagram of the skill hierarchy and decision-tree structure for their test environment *Keepaway*, a game of where three agents pass a ball between each other while an agent between all three tries to take the ball away. The above diagram gives the hierarchical design of the skills for the *Keepaway* task. Each box represents a skill and the arrows show the dependencies between the skills. The diagram below it shows the decision tree structure that determines which skill set will control the agent at any given time.

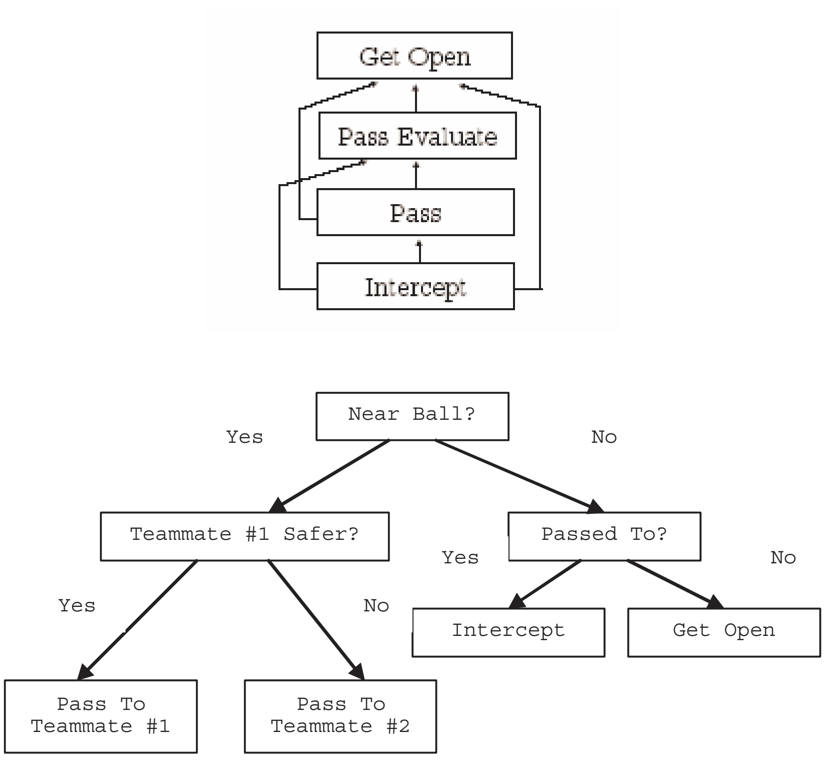


Figure 4: Concurrent Layered Learning Example.

The noticeable difference between the research performed with concurrent layered learning and our approach does not concern the design, but rather the implementation of it. Whiteson and Stone also used neural networks for learning the game controls, yet they did not focus on the need for replicating human behavior. Instead, their neural networks used a reinforcement learning algorithm, allowing the agent to choose its own actions during training and determine for itself the optimal strategy for playing the game. Since our interest involves keeping the agent's behavior reflective of a human expert, we needed a supervised learning approach that trains the agent how to replicate a human player's decisions. The next section will discuss the issues with the supervised learning approach in a real-time game environment, and our solutions to those issues.

5 ISSUES AND TECHNIQUES FOR AGENT TRAINING

After one decomposes the game's complex task into a set of subtasks, the next step is training the agent the skills needed for each subtask. We train an agent by allowing it to observe data that describes the game environment and the actions the player takes at any given time. As stated in Section 4, we use neural networks for learning how to handle each of the game's controls. Each network contains a series of weights that create the function the network represents. By adjusting these weights, one can generate a function that models the changes in the environment to the action that a player must execute. For manipulating these weights, we used the BACKPROPOGATION algorithm for each training example the agent observes. Usually, the BACKPROPOGATION algorithm alters each weight in a network through the following equation.

$$\Delta w_{ji} = \eta \delta_j x_{ji} \quad (1)$$

In the equation, the change to weight w_{ji} equals the product of the neuron's output error, δ_j, the input associated with the weight, x_{ji}, and a fixed constant, η, known as the learning rate. The learning rate limits the amount that the network weights will change after observing a particular instance. Typically, the learning rate is kept at a small number (i.e., 0.01 for our agent), allowing for only small weight alterations. Thus, the agent must view the training data several times before converging on a set of net work weights that produce an accurate modeling function.

For our agent, the number of training iterations needed for learning a particular skill was based upon the complexity of the networks used within that skill set. For the *Firing at the Fortress* and *Bonus Selection* skills, we determined that simple networks, which contained no hidden layers and few inputs, could sufficiently learn the human expert's behavior. In this case, we had the agent iterate through the training data a fixed number of times (i.e., 20 iterations for both skills). For the C*ircumnavigation of the Fortress* and the *Firing at the Mines* skill sets, we needed larger, more complex networks for learning the human expert's behavior. In these instances, we used a cross-validation scheme that would check the accuracy of the networks against the training data. After a training iteration, the agent would go back through the entire data set and see how well the networks' classifications matched compared to the actual actions taken by the player. If a network increased its accuracy by 5% from the last cross-validation check, the agent would store the weights and mark them as the most accurate weights for that network. If a network dropped in accuracy by 55% from the best weight set cross-validation check, the agent would halt the training of that particular network. As well, an agent would stop training a network if its accuracy did not improve by 5% after 20 training iterations. Once the agent halted the training on all networks for the skill, the agent would recover each network's saved weights, the weights deemed most accurate during the cross-validation checks. See Appendix A, B, C, and D for a diagram of the networks used for learning each skill, along with a description of the input sets selected for each network.

Figure 5 gives an example of the data recorded during the game, and used by the agent for training its networks. The data is separated into increments of time, which we call simulation cycles. Each simulation cycle serves as a snapshot of the game at a given moment. These cycles contain data describing palpable traits of each object on the screen along with the action carried out by the player. The lines prefixed with `"Object:"` describe the physical characteristics of an object on the screen, while the lines beginning with the `"#"` symbol describe any action taken by the player at that time. Each line ends with an index that tells which simulation cycle during the game the information was recorded. Utilizing this data, the agent will learn what situations require activating a particular action, and likewise when the agent should not take any action.

```
Object: Ship1, Position: (114.0, 149.249), Velocity: (0.0, -1.3),
Damage: 1, <16>
Object: Fortress1, Position: (227.0, 159.0), Velocity: (0.0, 0.0),
Damage: 0, <16>
#Ship1:TurnCW<16>

Object: Ship1, Position: (114.0, 149.979), Velocity: (0.0, -1.3),
Damage: 1, <17>
Object: Fortress1, Position: (227.0, 159.0), Velocity: (0.0, 0.0),
Damage: 0, <17>
#Ship1:TurnCW<17>

Object: Ship1, Position: (114.0, 146.649), Velocity: (0.0, -1.3),
Damage: 1, <18>
Object: Fortress1, Position: (227.0, 159.0), Velocity: (0.0, 0.0),
Damage: 0, <18>
```

Figure 5: Example of Training Data Recorded for *Revised Space Fortress*.

This section will describe the environments used and difficulties we encountered training the agent for the game. We will explain the "mini" game environments, mentioned in the previous section, and how they concentrate on a particular skill needed for the game. Also, we will discuss problems during training that we believe would exist if we applied the agent to other popular gaming environments today, and then disclose the techniques we created for solving these problems.

5.1 Skill Set Training

In the previous section, we listed what a player would need to learn in order to play the game at an expert level. These skills include:

- Correctly identifying and destroying a mine.
- Circumnavigating the fortress while remaining within the hexagon boundaries.
- Firing missiles at the fortress (normal shots and double shots).
- Recognizing and choosing the bonuses that appear within the game.

We will now detail how an expert player performs these skills, and how we designed each "mini" game environment for capturing their behavior.

5.1.1 Correctly Identifying and Destroying a Mine

This skill defines how an expert player handles a mine that appears while the game is being played. Typically, an expert player will stop firing at the fortress when a mine appears, and allow the mine to approach the ship. Once the player steers toward the mine and identifies it, the player will fire at the mine. If a foe mine appears a player must double-click the right mouse button, with the interval between the two clicks lasting from 250 – 400 msec. If a friendly mine appears, a player can destroy the mine by simply firing at it. For this skill, we want to capture the following:

- How well an expert player lines up the nose of the ship with an incoming mine?
- How quickly does a player identify a mine? In this case, how quickly do they react to foe mines and press the right mouse button.
- How close does a mine get to a player before they fire?
- How long does the delay between right mouse button clicks typically last?

In this “mini” game, we place the ship at the center of the screen and have mines appear at a set rate. Each mine appears in a random location, and will advance toward the ship. The player is instructed to identify and destroy each mine as quickly as possible, preventing the mines from hitting their ship. A player may steer the ship in any direction, but cannot move from the center of the screen. They must also use the right mouse button for identifying a foe mine that appears on the screen. The game lasts for two minutes (2,400 cycles[1]) with the agent training data coming from 10 trials of this game.

5.1.2 Circumnavigating the Fortress

This skill describes how an expert player flies around the screen without being shot by the fortress. An expert player takes a clockwise pattern around the fortress, keeping the nose of the ship pointed toward the fortress. Expert players also have the ability to keep the ship’s speed fairly constant: fast enough to avoid any shells fired by the fortress yet below the speed threshold monitored by the Velocity score. Expert players achieve

[1] A simulation cycle in *Revised Space Fortress* equals 46 msec.

this behavior by keeping their joystick movements at a minimum. Hence, we should see long delays between turning and thrusting inputs. For capturing this skill, we look for the following:

- What instances cause a player to turn or thrust?
- How long does a player keep the turning or thrusting input active before releasing it?

The "mini" game environment looks like the normal game with the absence of mines and bonus characters. The fortress may shoot at the ship, but the player cannot fire back at the fortress. The object of this "mini" game involves maximizing Velocity and Control scores while keeping the ship's damage to a minimum. In order to accomplish this goal, the player must fly the ship in a clockwise pattern within the hexagon boundaries, while keeping the ship's speed at an optimal rate (below the Velocity threshold but fast enough to avoid incoming shells).

During the creation of this game, we designed two versions for recording data. In the first, the player starts at the same position as in the normal game and flies the ship for two minutes (2,400 cycles), maintaining a clockwise flight pattern without being hit. In the second, we start the ship at a random position, orientation, and speed. The player must correct the ship and return it to the clockwise flight pattern within 20 sec. (400 cycles).

The reasoning for the second version deals with the problems that arise in a game environment when a player switches between tasks. In a normal game of *RSF*, a player must periodically divert their attention to handling a mine that appears on the screen. As the player switches focus to the mine, they no longer concentrate on flying the ship. Once the mine disappears, the player may find their ship drastically off its normal flight path (i.e., drifting outside the hexagon boundary, heading directly toward the fortress, etc.). The player must make adjustments, radically different than their usual behaviors, in order to return the ship onto its normal flight path. If we try training the agent how to fly using only data about a normal game, the irregularities of the player's flight path produce enough noise to prevent learning this skill properly. By including the second version, we facilitate the agent in learning how to correct the ship when it ends up outside the flight path. In total, we used 10 games from the first version and 100 games of the second version for the agent training data. While training the agent, we had the best results by interleaving the data from the first and second versions. We made several attempts at separating the data, and training the agent with data from one version followed by the other. However, that approach caused the agent to only learn the behavior from the last set it observed, making the agent ineffective for playing a normal game.

5.1.3 Firing Missiles at the Fortress

In the game, a player destroys the fortress by first hitting it with missiles 10 times, with the shots hitting the fortress at a rate greater than 250 msec. Once completed, the player must fire a double shot, two missiles that strike the fortress in less than 250 msec. Since an expert player has the ability to destroy the fortress numerous times while following these constraints, we want to capture the following behavior:

- What is the expected firing rate between normal shots (the 10 shots needed before the double shot)?
- As well, how quickly does the player fire both shots for the double shot?
- How well does the player align the nose of the ship with the fortress before they start firing? In other words, does the player try to line up the ship perfectly before firing, or do they start shooting whenever they believe a missile will hit?

In this game environment, we place a stationary fortress, unable to shoot, at the center of the screen. The ship will appear at a random location and orientation from the fortress. The player must turn the ship toward the fortress and begin firing at it. Once the player hits the fortress 10 times, they must fire a double shot to kill the fortress. After destroying the fortress, the ship reappears in its starting position and orientation, and the player must do the task again. The player is instructed to destroy the fortress as many times as possible during the game. Each trial lasts about 25 sec. (500 cycles) with an expert player completing 100 of these trials for the agent training data.

5.1.4 Recognizing and Choosing the Bonuses

During the game, a player has the opportunity to collect bonuses that will either raise their Points score or give them more missiles in their supply. Usually, an expert player takes advantage of every bonus opportunity that appears within the game. Therefore, we do not need to worry about learning how many bonuses a player receives during a game. Expert players generally choose which bonus they take by looking at the number of missiles they have in their supply. If the player thinks they have too few missiles, they will opt for the missile bonus. Otherwise, they will take the points bonus. In that case, the agent simply needs to learn how many missiles are in the player's supply whenever they choose the missile bonus. Instead of designing a new "mini" game for capturing this skill, we used data from an expert playing the normal game. Using only the information that tells when the player selected a bonus, the agent learns how to mimic a player's bonus selection habits. Our agent used data from 10 normal games, each three minutes in length (3,600 cycles), for acquiring this behavior.

5.2 Difficulties in Learning Human Behaviors

The purpose of the "mini" game environments was to aid the agent in learning an expert player's behavior by reducing the complexity of the game environment into smaller, learnable problems. Yet, even in these environments the agent still had difficulty capturing the player's pattern of behavior. During the development process, we discovered features within the "mini" game environments that made the player's behavior difficult for the agent to learn. We have categorized these features, which we believe exist in other real-time game environments, and give our solutions for handling them with our neural-network-based agent. Listed below are the features discussed within this thesis:

- Sparseness in the number of player actions.

- Challenges with attributes that describe a real-time game environment.
- Multiple contexts for using a game control.
- Dependent behaviors exhibited by a player.

5.2.1 Sparseness in the Number of Player Actions

The first issue concerns the rarity that a player takes an action during the game. We trained a neural network by checking each simulation cycle in a dataset for the action that the network controls. The network gets trained so that it outputs a high value if the action occurs and, likewise, outputs a low value if no action takes place. Looking at the recorded data, we observed that only 10-15% of the simulation cycles had an action tied to them. This means that an expert player spent a majority of the game observing the screen, waiting for the right moment to perform an action. Since few simulation cycles had actions connected to them, the network seldom trained to output a high value. Thus, the networks would end up always outputting a low value, having the agent never take an action while playing the game.

This problem fits the description of a known issue in the machine learning community called weak learnability, where the target classification type only has a few instances within the training data. In our case, we would consider the occurrence of an action as the rare classification type, and the state of environment when the action occurred as an instance. A common method for curing weak learnability involves boosting the learning algorithm, repeatedly training the learning structure on the examples it classifies incorrectly [29]. Usually this is accomplished by copying the instances of the rare classification type until its equal to the number of instances for other classification types.

Instead of expanding the amount of training data, we simply adjusted the learning rate of the BACKPROPOGATION algorithm whenever an action occurred in the training data (see Equation 1). With every possible action that appeared in the training data, we compared the number of simulation cycles when an action occurred, α, against the number of cycles when the action did not take place, β.

$$ratio = {(\beta - \alpha)}/{\alpha} \quad (2)$$

When training the network, we multiplied this inverse ratio to the learning rate whenever we observed an action occurrence, transforming the learning rate from a small constant to a larger value. This effectively created a drastic change in the network weights, Δw_{ji}, and forced the network to output a high value in this situation. By keeping the learning rate a small value when the action did not occur, the network weights became balanced over several training iterations. This process produced a network that would trigger an action at the appropriate times.

5.2.2 Challenges with Attributes That Describe a Real-Time Game Environment

The second issue we encountered dealt with a variety of problems stemming from the attributes used for describing the game environment. These problems ranged from the actual process of selecting the correct information for the neural networks to changing attributes due to our simplistic network structures. As well, attributes that grew too large in value ended up disrupting the algorithm that tuned the network weights, thus preventing the networks from learning the correct behavior. This subsection gives descriptions of all these problems, and our resolution for them within our agent.

One problem we faced involved selecting the proper information the networks needed to accomplish their tasks. Even in a 2D environment such as *Revised Space Fortress*, someone could collect a wealth of information for describing the environment to the agent. However, the inclusion of unneeded inputs increases the difficulty of the training process. By enlarging the number of inputs given to a neural network, we increase the number of possible weight combinations within the network. Not only does this mean the agent needs more time for training, but the likelihood that the agent will find a set of network weights whose function models the human player’s behavior drops considerably. For solving this problem, we would search for an optimal input set for the agent, a set that contains the least amount of information but effectively describes the game environment. Using expert knowledge of the game, we carefully selected the information necessary for each network to carry out its specific action. After training and testing the agent in the game environment, we would analyze what prevented the agent from correctly mimicking the human player's behavior. By keeping the neural networks fairly small (using zero or one hidden layer(s)), we could easily calculate how the inputs and their respective weights influenced the network output. Once we understood what inputs caused the unwanted behavior, we would make the proper input changes and repeat the process.

Another problem we discovered was that choosing the proper attributes was also influenced by the inability of our simplified network structures to learn certain behaviors.. Specifically, these behaviors pertain to learning the thresholds that characterize a human player's reaction time for executing specific actions. We define a threshold as the amount of time that elapses between a triggering event and when the player performs the necessary action. For example, the agent must learn the threshold associated with the player doing foe mine identification. The player identifies a foe mine by double-clicking the right mouse button with the interval between the clicks lasting for 250 – 400 msec. Analyzing the recorded data from the *Firing at the Mines* "mini" game, we observe that a vast majority of double-clicks have this interval; providing the agent with minimal noise within the training data. Using our simplified network approach, the network that handles learning this threshold will be a single neuron with three inputs. One input serves a Boolean value telling whether or not the triggering event has occurred (i.e., the first click in the foe mine identification process). The second input counts the number of cycles that have elapsed since the triggering event appeared. The third input is a constant of 1.0 whose associated weight serves as the firing threshold for the neuron; the combination of the other two inputs and their weights must exceed the firing threshold for the action to occur.

$$w_1 x_1 + w_2 x_2 \geq w_3 \qquad (3)$$

During training, the agent should tune the neuron's weights such that the neuron outputs a high value when: the Boolean input value is set to true (0.99), and the counting input reaches a mean estimate of the double-click interval distribution [30]. Using our training data, the player performs the second click of the foe mine identification at an average of 6.4 cycles (s.d. 1.1) after the first click. Figure 6 presents the interval distribution from the 191 observed foe mine identifications by a player during the *Firing at the Mines* "mini" game. When the neuron outputs a high value, the agent will execute the second click needed for completing the foe mine identification. Since the two clicks will occur within the desired threshold, the agent should mimic the expert player and successfully perform the foe mine identification.

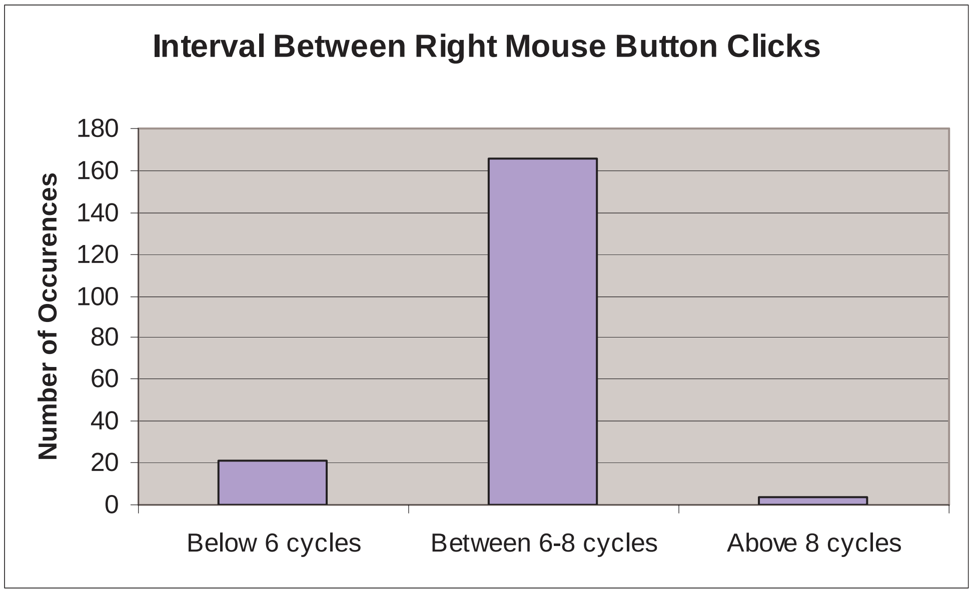


Figure 6: Histogram of the Double-Click Intervals Executed by an Expert Player.

Yet after training, the neuron did not learn the threshold needed between the two right mouse button clicks. Instead, the neuron learned to output a high value immediately after the occurrence of the triggering event (i.e., the first mouse button click). This problem was further complicated since the agent learns a threshold similar in length for another task. For the *Firing on the Fortress* “mini” game, the agent must learn the cadence required for firing the normal shots, the 10 shots that hit the fortress prior to the double shot. Each normal shot must hit the fortress at a rate greater than 250 msec; a normal shot that arrives earlier penalizes the player by removing all damage from the fortress. Analyzing the recorded data, the expert player has a mean interval of 7.8 cycles (s.d. 4.8) between each normal shot. Figure 7 presents the shot interval distribution from the 5,241 observed shots fired by the player during the *Firing at the Fortress* “mini” game. Although we developed a slightly different network for the learning the normal shot threshold (see Appendix B), the network contains inputs similar to those for the neuron learning the foe mine identification threshold. The network relies on a counter that measures the number of cycles since the triggering event (i.e., the previous shot fired), and learns to output a high value when the counter reaches the mean estimate of the player’s normal shot delay. Using the training data from the *Firing on the Fortress* “mini” game, the network learned a threshold above 250 msec, allowing the agent to fire normal shots without incurring the penalty.

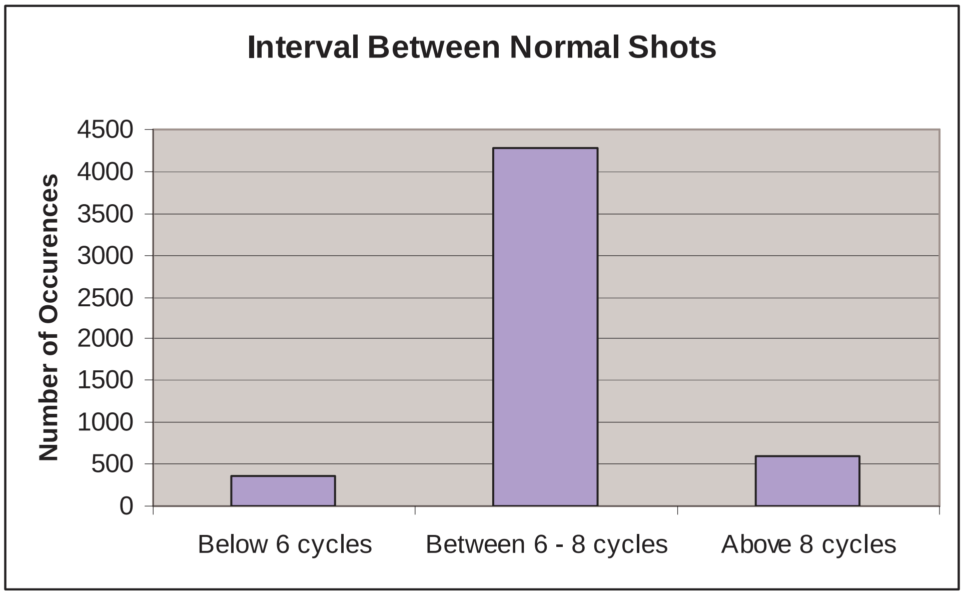


Figure 7: Histogram of the Normal Shot Intervals by an Expert Player.

Unfortunately, the cause of this problem with the foe mine identification threshold stems from our choice of simplified networks and how we handle learning actions that sparsely occur within the recorded data. Traditionally, one follows a couple of guidelines when using the BACKPROPOGATION algorithm for training a neural network: the patterns within the data are presented in a repeated, cyclic order, and the algorithm maintains a sufficiently small learning rate, η. The patterns mentioned refer to the training examples which contain the network input (i.e., game state information) and the desired output (i.e., action/no-action). Using these guidelines, one would assume that the network weights would eventually converge to a successful solution [31].

In our case, the learning rate does not remain constant throughout the training session. Whenever the agent observes an action, we boost the learning rate by multiplying it by the inverse ratio of action occurrences (see Equation 2), producing a drastic change in the network weights. Comparing the inverse ratios, the firing action in the *Firing at the Fortress* “mini” game had an inverse ratio of 35, but the second mouse click action for the *Firing at the Mines* “mini” game had a ratio of 133, nearly four times larger than the firing action. This means that the neuron learning the foe mine identification threshold underwent far more dramatic alterations during training than the normal shot threshold network. These radical changes could not give us the precision needed for detecting the exact interval threshold, leaving the neuron with weights that under-fit the problem. For the foe mine identification threshold, the under-fit neuron generated a high value whenever the triggering event was detected, a trait common among all of the training examples where the second click occurs.

Another issue is that a single neuron only has the ability of solving linearly separable problems. This means a linear function must exist that can separate the training examples into their two respective categories (i.e., action and no-action). For the foe mine identification, we would like a demarcation that separates the training examples when the elapsed cycle counter reaches six cycles. All examples that have elapsed count greater than or equal to six will be classified as “action”, and those below six cycles marked as “no-action”. Viewing Figure 6, we notice that some of the observed intervals were less than six cycles, moving the actual demarcation line to a time earlier than six cycles. These earlier intervals can serve as noise within the training data, preventing the network from learning the desired threshold.

Known solutions exist for curing the problematic networks, such as the neuron learning the foe mine identification threshold, yet these solutions require major revisions to our agent design. First, we could increase the complexity of the neural network handling the threshold by including more neurons and connections between them. Adding more weights could help distribute the large weight shift that occurs when the agent detects an action, producing a network that can learn the threshold. Also, using more than a single neuron removes the need for having a linearly separable problem, opening the network for handling linear non-separable problems, such as mean estimation. Yet, as we enlarge the network we will need more time for training, and increase the difficulty in debugging the network. Second, we could follow the traditional boosting technique discussed in Schapire [29]. We would make several copies of the training examples associated with the action occurrence, and include these copies into the training data. Thus, we would reduce the ratio of examples that contain an action and examples where no action takes place, removing our need for changing the learning rate. Keeping a small learning rate throughout training gives us the precision we need for the network weights to converge to a solution that learns the threshold. However, this requires expanding the size of the training data, subsequently lengthening the amount of time needed for training the network.

Our solution for learning the foe mine identification threshold involved changing the focus of the problem into a task that the agent could learn using our design. Since expert players regularly execute the mine identification correctly, we determined that the agent did not necessarily need to learn the amount of time between clicks, but rather how to follow the rule for correctly identifying a foe mine. For learning the interval threshold, we instituted pseudo-Boolean input values that effectively incorporated this rule of the game. Instead of using the elapsed time as a network input, we used a value that classified the current elapsed time into one of three categories: below 250 msec, above

400 msec, or between 250 – 400 msec. (0.33, 0.66, or 0.99 respectively). The network should now output a high value whenever the triggering event has occurred and the interval classification equals 0.99. Previously, we tried having the network learn the mean estimate of the intervals observed within the training data, meaning the network would observe several values where an action should take place. In this case, the network will always see the same value for whenever an action occurs, eliminating the need for learning the mean estimate. Without the need for precisely tuning the weights for learning the mean estimate, the network can learn this activation pattern without changing the network structure or our policy on boosting the learning rate. As a drawback, the agent consistently activates the second mouse button click at the same rate. Whenever the elapsed cycle counter reaches six cycles, the classification for the interval becomes 0.99, forcing the network to execute the second click after six cycles elapse. Human players, though, have variation within their behavior that creates shorter or longer intervals. The Discussion section will address how we can redesign the network learning structure such that we can better replicate this variance in a human player's reaction times.

In some instances, we cannot always rely on having a rule present that helps facilitate learning the proper threshold. Another threshold that we wanted the agent to learn was the expert player's recognition of a foe mine appearance. This threshold equates to the amount of time that elapses from when a foe mine appears on screen to when the player makes the first of the two right mouse button clicks needed for identifying the foe mine. For controlling the first right mouse button click, we again used a single neuron with three inputs: a Boolean value that indicated if a foe mine has just appeared (triggering event), a count that gives the number of cycles that had elapsed since the appearance of the foe mine, and a constant of 1.0. During training, the agent would observe a foe mine appearance in the data, and begin counting the cycles until the agent detected a right mouse button action. The difference between when the button click occurred and the mine's appearance served as the desired delay for the agent to learn. Figure 8 gives a histogram that charts the interval thresholds observed for this case, which produces a pattern similar to one for the foe mine identification threshold (i.e., the interval between right mouse button clicks). This histogram was also generated from the 191 observed foe mine identifications by a player during the *Firing at the Mines* “mini” game. From our analysis, a human player begins the foe mine identification process 25-34 cycles after the appearance of a foe mine.

.

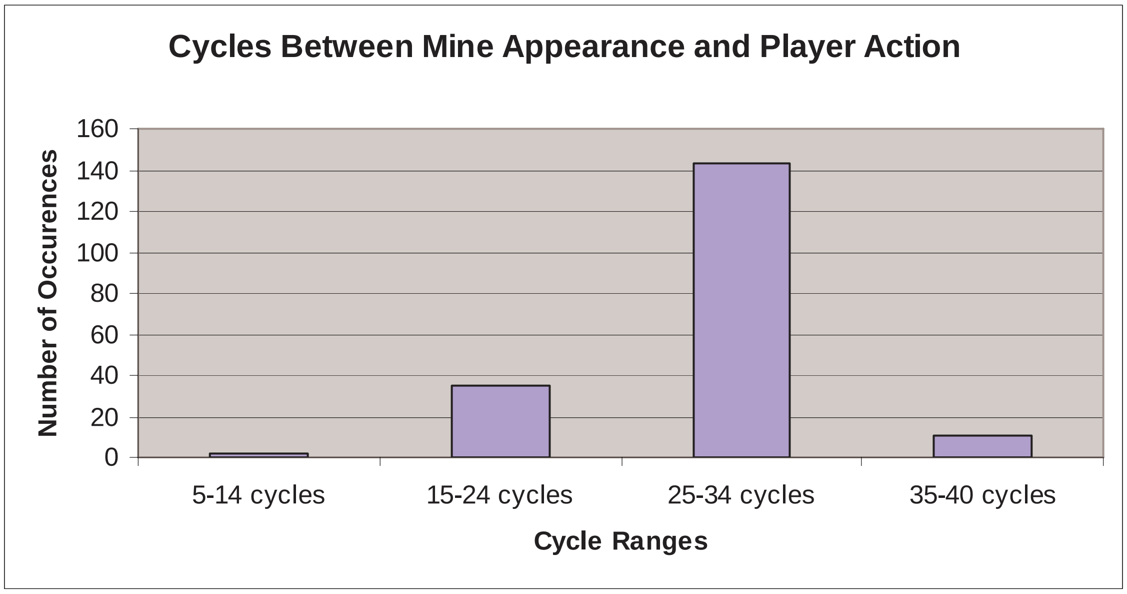


Figure 8: Histogram of the Mine Recognition Delays for an Expert Player.

After training, we observed that the agent learned a threshold of approximately 8 cycles, significantly lower than the typical 25-34 cycle threshold exhibited by human players. Again, the problem stems from having a simplified network and a large inverse ratio (133) that drastically alters the weights whenever the agent observes an action during training. Since no game rule exists that dictates an expert player’s behavior in this situation, we cannot use our solution from the previous case. Instead, we left the agent’s learned threshold of 8 cycles and decided to compare the agent’s performance against the human expert. We had the agent play several trials of the *Firing at the Mines* “mini” game, and observed that the agent’s scores did not differ considerably from the human expert's scores for this "mini" game, even though the agent executed the foe mine identification earlier than the human expert. Since the earlier foe mine identification did not make a significant difference in performance between the agent and human player, we decided to leave the threshold alone. Yet, for other games, not learning the desired threshold could make a noticeable contrast in human and agent performance. The Discussion section will mention ways in which the agent could better learn these thresholds and more closely match a human player's behavior.

The last problem associated with attributes for real-time game environments concerns the range of value an attribute can possess. In the BACKPROPAGTION algorithm, the value of input (attribute) affects the magnitude of a change in the weight associated with that input. Equation (1) shows that as an input gets larger, so does the change on the weight associated with that input. If any large input value can drastically alter the network weights, we could face a problem of the algorithm loosing the precision needed for converging on a proper set of weights. For preventing these drastic changes for any large input value, one can normalize the input values given to a network [32]. For our agent, we normalized inputs that counted a number of elapsed cycles since the occurrence of a particular event.

This normalization occurred by either dividing the value by a constant integer, or applying the input to a sigmoid function. For normalization through division, we chose a constant we believed the input should never exceed, but close to the observed largest value in the data. Equation (4) lists the sigmoid function, the other means we had for normalization. We used the current elapsed cycle count as the value for *x*, and the resulting value for *f*(*x*) became the input into the network.

$$f(x) = \frac{1}{1 + e^{\frac{-x}{\varepsilon}}} \qquad (4)$$

In the equation, the cycle counter *x* is divided by a positive constant *ε*, which can vary for between the different attributes where we apply the sigmoid function. By setting *ε* =

1, the output of the sigmoid quickly approaches one as the value of x increases (ex., $f(1) = 0.73$, $f(2) = 0.88$, $f(3) = 0.95$). Once $x > 3$, the sigmoid function produces similar output values, removing any distinct differences between larger values of x. As ε increases in size, we slow the growth of $f(x)$, meaning that larger values of x are required for the function to attain values near one.

The constant ε helps us generate a sigmoid function that better models the trends that occur within the training data. For example, the network for learning the normal shot threshold should learn that 7 to 8 cycles should elapse between each normal shot. Using the sigmoid function for the elapse cycle input and setting $\varepsilon = 1$, the network will observe extremely small differences for the input when the value is within firing threshold (6 – 8 cycles), and near the threshold (3 – 5 cycles). Altering the weights to reflect these differences, however, could not be accomplished through our implementation of the `BACKPROPAGATION` algorithm. Hence, the network learned to fire at a rate less than the normal shot threshold. By using a value of $\varepsilon > 1$ (i.e., $\varepsilon = 2$), the network observed noticeable differences for the input when its value was below threshold and within it. Using a larger value for ε helped generate a sigmoid function that better representation for the network learning the player's behavior.

For determining how we normalized a certain attribute, either through division or a sigmoid function, we simply tested the agent using both methods. We trained the networks using one normalization technique for an attribute, and observed the agent's behavior within the game. Then, we changed the normalization technique and analyzed if the agent's behavior had improved or not. See Appendix A, B, C, and D for a description of the inputs used within every network, and how we normalized certain inputs given to these nets.

5.2.3 Multiple Contexts for Using a Game Control

The third issue concerns handling the same control for different circumstances in the game. For instance, consider the mine identification process in the *Firing at the Mines* "mini" game. An expert player will only press the right mouse button under two circumstances: once when the player recognizes a foe mine has appeared on the screen, and again after 250 – 400 msec. has elapsed since the previous click. Although the button presses are related, the information needed by a player for determining when to do either click is dissimilar. Originally, we built a single neural network for learning how the player handles the right-mouse button control, requiring that the network would have to handle both cases. During training, our agent would learn when to perform the first click, but never execute the second click that completes the foe mine identification.

We determined that this problem had a connection with the topic earlier about limiting the inputs to a neural network. A single network would require inputs that describe how a player uses the control in every possible situation. Not only must the network learn when to recognize the present situation, but also handle the control in the same manner as a human player. Our simplified network, however, could not learn a problem of this complexity. As a result, we decomposed the network into two parts, with each network learning how the player handles each circumstance (i.e., one network for the first mouse click and another for the second click). With multiple networks for the same control, we now needed a mechanism that would combine the outputs of the networks into a single value that would indicate whether or not to activate the control. Figure 9 gives a diagram of the entire network structure constructed for learning the player's behavior. Also, see Appendix A for a detailed description of the design for the first and second click networks.

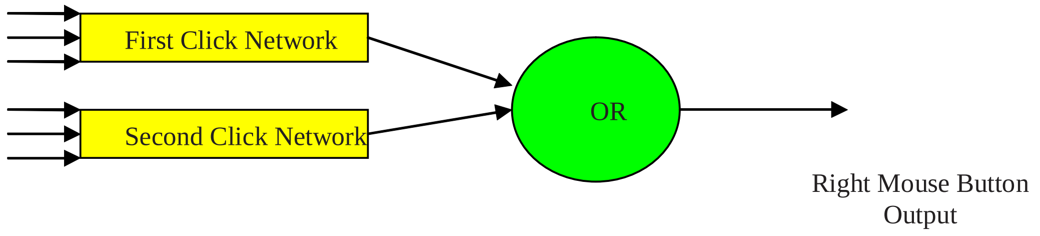


Figure 9: Diagram of a Network Structure for Handling Multiple Contexts.

If a control gets used in multiple circumstances, a player will activate the control whenever they recognize one of the conditions as present. In that case, the mechanism that combines the multiple networks should resemble the function of an OR gate. When one of the networks outputs a high value, indicating the activation of the control, then the OR gate will let a high value pass through as output. Instead of building an OR gate, we had the networks feed into a single neuron. Figure 10 displays how a single neuron can behave as a two-input OR gate. We predicted that if we trained the entire structure together, the neuron would train itself to output a high value anytime one of the incoming networks produced a high output, reflecting this OR gate behavior. After training the structure in parallel, the agent could model how the expert player used the right-mouse button. Each network correctly learned how to recognize each circumstance, and the binding neuron produced the OR gate behavior we desired.

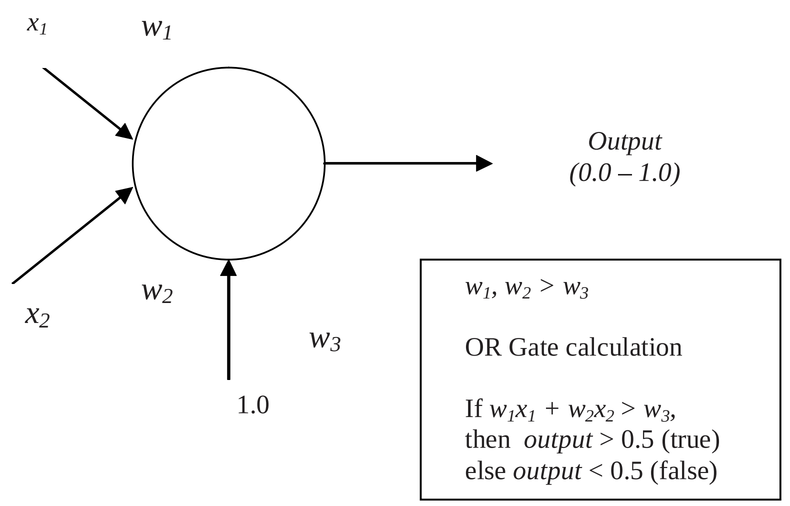


Figure 10: Diagram of a Neuron as an OR Gate.

5.2.4 Dependent Behaviors Exhibited by a Player

The last issue we will discuss is based on dependent behaviors, a behavior that occurs only after the completion of another. For example, a player will only shoot at a foe mine after the player properly double-clicks the right mouse button. As well, a player will only shoot at the fortress once they have the ship's nose aligned with the fortress. In the previous case, a player would activate a control when one of a number of possible situations was present. However, several situations must now exist before the player will activate the control. Again, we initially had a single network for each control, with the networks checking for both the completion of an earlier behavior and the conditions needed presently for the action. After training the networks, we observed that the networks did not learn to wait for independent behaviors to finish. This caused the agent to take actions prematurely (i.e., shooting at foe mines before finishing the identification, firing at the fortress before the ship faces it), producing an agent that did not correctly resemble the player's behavior.

To solve this problem, we decomposed it into the dependent and independent behaviors required for solving the problem. One network would learn the player's dependent behaviors (i.e., ship nose aligned with the incoming mine) while another network would handle the independent behavior (i.e., player successfully executed the foe mine identification). Figure 11 shows how we combined these networks for defining how the player handles the entire task.

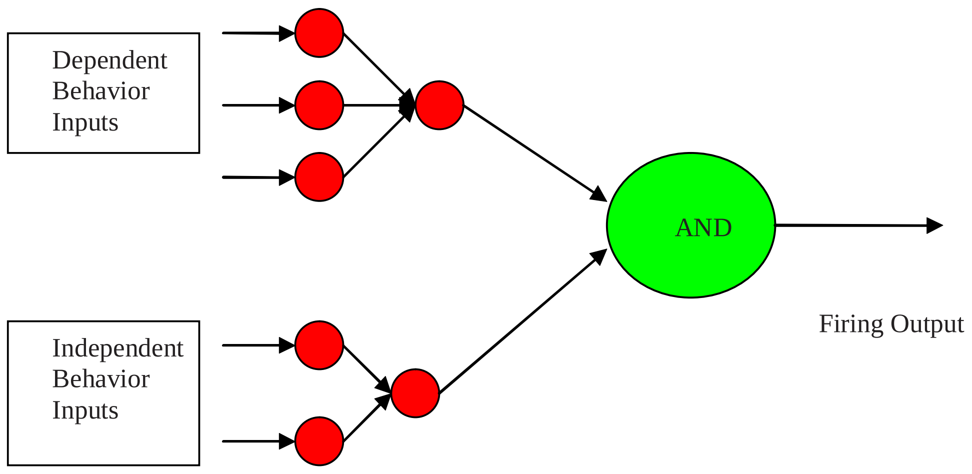


Figure 11: Diagram of a Network Structure for Handling Dependent Behaviors.

Feeding the network outputs into a single neuron again combined both networks. However, the neuron in this case should only output a high value when both networks have a high output, meaning that present conditions for the action exist and necessary preceding behaviors have occurred. Thus, the neuron's behavior should reflect that of an AND gate. Figure 12 diagrams how a single neuron can behave like an AND gate. Once more, we trained the entire network structure in parallel, and produced an agent that correctly mimicked the player's actions.

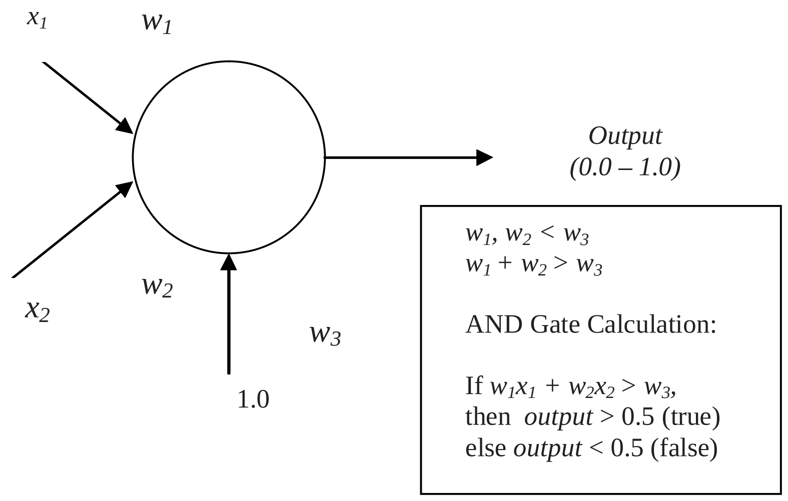


Figure 12: Diagram of a Neuron Acting as an AND Gate.

5.3 Evaluating the Accuracy of Our Trained Agent

With the agent completed, we compared its performance against a human expert and a known expert agent that follows an explicit strategy. The programmed agent was a version of the agent described in [2], an agent which utilizes random delays for matching human expert behavior. Table 1 shows the results of our performance comparison between a human expert and the two agent types. In this test, each player (human expert and two agents) performed ten 3-minutes games of *Revised Space Fortress*. The table gives the average for each sub-score across a series of ten games, with the standard deviation listed in parenthesis.

Table 1: Comparison Between Human Expert and Two Agent Types.

	Total	**Points**	**Velocity**	**Control**	**Speed**
Human Expert	5485 (518)	2219 (404)	1196 (67)	1200 (25)	871 (134)
Agent w/ Programmed Strategy	5308 (326)	2743 (248)	1019 (125)	556 (77)	990 (90)
Trained Agent	6089 (248)	2663 (258)	1242 (19)	1248 (7)	936 (99)

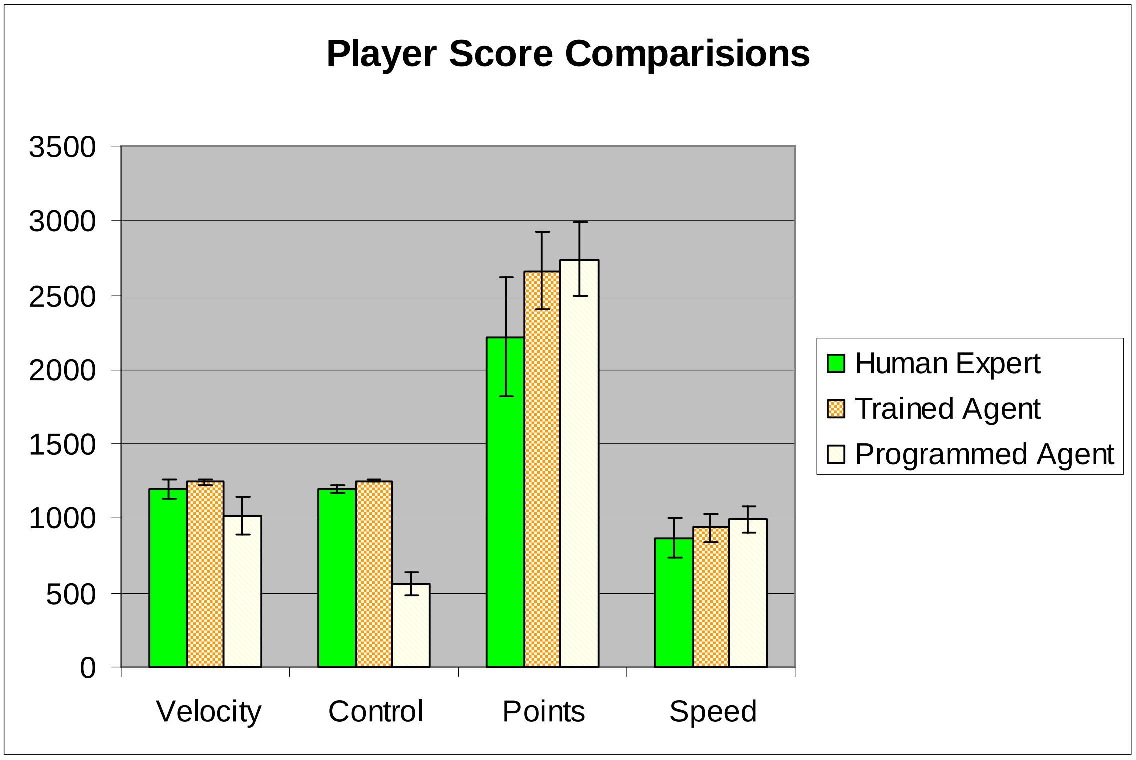


Figure 13: Graphical Representation of the Results Presented in Table 1.

Although the Total scores would indicate that programmed agent did a better job at mimicking the human expert, the sub-scores show obvious flaws in the programmed agent. Yet from viewing Figure 13, the results reveal that the trained agent better matched the performance of the human expert than the programmed expert. The extremely lower Control score by the programmed agent indicates that the agent had difficulty remaining within the hexagon boundaries, taking a different flight path than the human expert. On the screen, the trained agent follows the human expert's flight path with the same smoothness and consistency, resulting in similar Velocity and Control scores, an enormous difference compared to the programmed agent. Also, the trained agent obtained a Speed score closer to the human expert, indicating that our approach does better at modeling the reactions of the human expert in handling mines as opposed to the random delays used in the programmed agent.

Another noticeable difference, not reflected within the sub-scores, also appeared between the two agents. As noted in Section 5.1.2, an expert player flies the ship in a clockwise path around the fortress, keeping the nose of ship pointed toward the fortress. While programmed agent flies the ship, the programmed agent makes minor changes in the ship's orientation frequently. By doing so, the agent keeps the angular difference between the ship nose and the fortress at a minimum throughout the game. Performing these adjustments requires quick activations and release of the turning controls. However, due to the sensitivity of the joystick, a human player cannot mimic this same behavior. Instead, a human player will make fewer, but larger adjustments in the ship's orientation to keep the ship nose pointed toward the fortress. While observing the trained agent playing the game, we saw the agent handling the ship in a similar manner. The trained agent would let the angular difference between the ship's nose and fortress grow considerably larger than what the programmed agent allowed. As the ship moved closer to the outer hexagon boundary, the agent would make a large correction to point the ship at the fortress, and then thrust to keep the ship within the hexagon borders. Although we cannot quantify this difference within the game's sub-scores, we believe these movements by the trained agent show a better reflection of human expert behavior than those demonstrated by the programmed agent.

Both agents show significantly larger Points scores than the human player, with the programmed agent showing an even larger difference than the trained agent. This occurs since the agents can switch between subtasks quicker than a human. The agents do not suffer from fatigue or other mental factors that hinder a human's recognition and reaction to a subtask change. Instead, the agents will constantly observe the game environment and immediately act when required to do so. In particular, agents can resume attacking the fortress after handling another task faster than a human player. Using an advanced framework that better models a human player's recognition abilities, such as including the effects of fatigue or memory loss, may provide some assistance in having our agent better resemble a human player's performance. Also, the ability for the agent to always take the same actions in the same situations explains why the agent keeps a lower variance in its sub-scores than the human expert.

6 HUMAN SUBJECT EXPERIMENT

After designing and developing an agent that learns its game strategy by observing an expert player, we assembled the test for our hypothesis. Our hypothesis is that:

> *An agent that learns its strategy from an expert player will provide an equal, if not greater, benefit to a human trainee as a partner during training, compared to a partner agent that follows an explicit, programmed strategy.*

In the AIM training protocol, a trainee will focus on a specific task while observing their teammates handling the other tasks, giving the trainee a mental model about how they should perform the other tasks that they will handle later. Looking at Table 1, we notice that our trained agent better resembles the human expert's behavior as opposed to the programmed agent. We hypothesized that a trainee would learn better from a partner like the trained agent than another agent following a strategy considered unattainable by human experts. At the very least, we will prove that trainees working with either agent achieve the same performance since both agents are considered expert-level players. This section will detail the creation and results of the six-week human subject experiment for these hypotheses.

6.1 Game Design and Conditions for the Experiment

For this experiment, we will focus on complex task training through a teamwork setting. Using the AIM protocol [1], we took the complex task represented in *Revised Space Fortress* and decomposed it into two parts, the joystick and mouse controls. In teams of two, known as dyads, one player would control the joystick, flying the ship around the fortress while firing missiles at both the fortress and mines. The other player would handle the mouse controls, identifying foe mines and seizing bonus opportunities whenever they appeared. After each game, the two players would swap roles and play another game. So, a player would serve as the pilot (joystick) in half of the games played and as the co-pilot (mouse) in the other half. Periodically, we would test the trainees by reassembling the complex task and having the trainee play the entire game individually, handling both the joystick and mouse controls while playing the game.

We based each condition for the experiment upon what partner each trainee would have while playing the game. Each trainee could have one of three possible partners:

- Another human trainee.
- An agent that learned its strategy from an expert human player.
- An agent that plays at an expert level, but follows a programmed strategy.

The first condition would pair a trainee with another person learning the game along with the trainee. This condition serves as the control group, replicating the research conducted over the AIM protocol using human teams. Although we had prior research about the performance of trainees working with human trainees and partner agents, we had no research that directly compared the two types. By using this condition, we could see if any benefits or hindrances existed for a trainee paired with a human partner instead of a virtual one. The remaining two conditions would test the hypothesis we developed for our experiment.

Yet, if we refer to the results from our test in Table 1, we have a major discrepancy in performance between the trained and programmed agent. We have already noted that a partner's ability-level will influence a trainee's resulting performance [3]. Thus, we felt that both agents should perform at the same level to ensure that any differences observed between the two conditions come from the agent's behavior, not their ability-level. The programmed agent, an adaptation of the agent used in [2], uses a series of random delays for the activation and release of game controls by the agent. By altering these delays, we can change how well the agent plays the game. We adjusted the programmed agent such that its ability-level, measured by the Total score in *RSF*, approximately equaled that of the trained agent. Nevertheless, the programmed agent still attained this score in a different manner than the trained agent. Table 2 shows the results of both agents playing 40 games of *RSF* with each game lasting three minutes in length. Again, we display the averages of the sub-scores along with the standard deviations written in parenthesis.

Table 2: Comparison Between Trained and Modified Programmed Agent.

	Total	Points	Velocity	Control	Speed
Human Expert	5485 (518)	2219 (404)	1196 (67)	1200 (25)	871 (134)
Trained Agent	6016 (345)	2517 (247)	1236 (40)	1239 (16)	1024 (105)
Programmed Agent w/ new random delays	6083 (296)	2604 (259)	1206 (55)	1070 (21)	1203 (74)

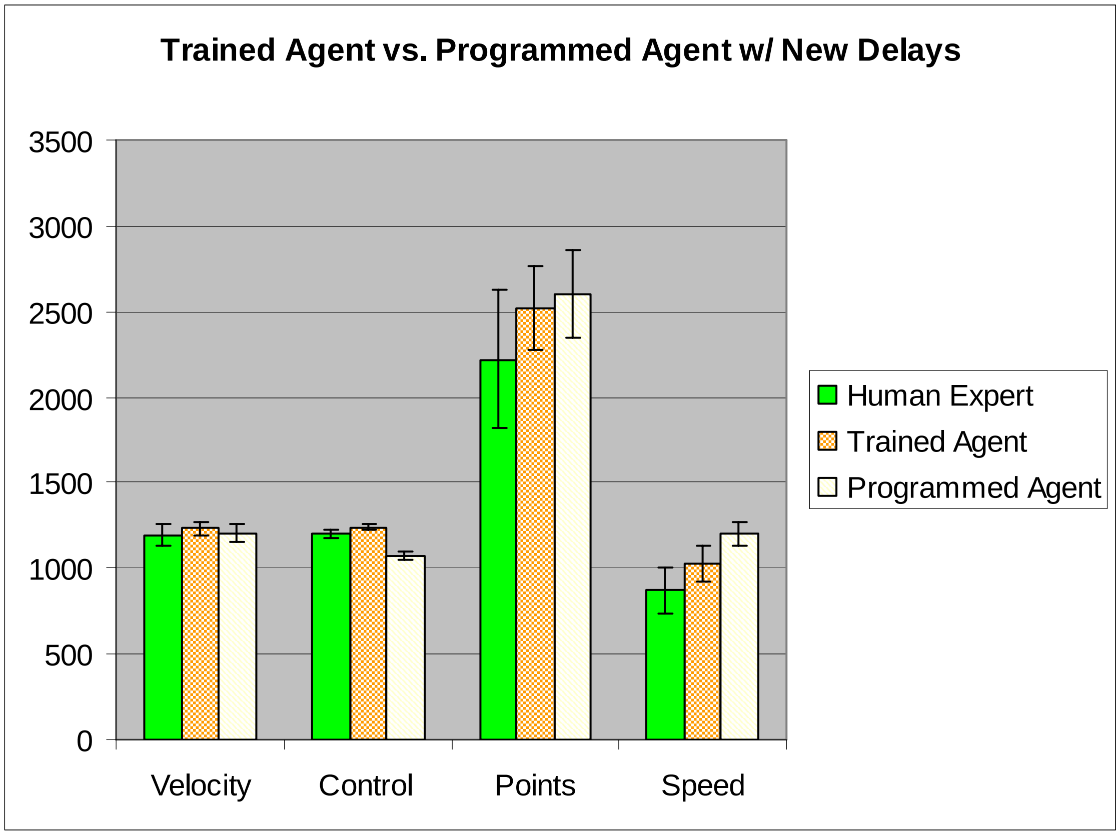


Figure 14: Graphical Representation of the Results Presented in Table 2.

From these results, we can observe the trained agent and programmed agent significantly differ in the Control and Speed scores (the *p*-value between both the Control and Speed score is less than 0.01). The programmed agent does not stay within the hexagon boundaries as well as the trained agent. Instead, the programmed agent quickly identifies and destroys all mines that appear on the screen, boosting its Total score to the same range as our trained agent. Conversely, the trained agent scores higher in the Control score, but worse in Speed. Still, the Control and Speed scores for the trained agent better match that of a human expert, giving us an agent that more closely equals the sub-scores recorded for a human expert. We have copied the results of the human expert from the earlier experiment (documented in Table 1) to show how the agents' performance now compares to a human player. Both agents now perform at the same level, yet the agents differ in the actions they take to attain these scores. With the ability-level of the two agents equalized, we can now test to see if their contrasting behaviors have any impact on trainees working with these agents.

6.2 Raven's Advanced Placement Matrices (APM) Test

Past research demonstrates that a person's ability to acquire the skills for *Space Fortress* correlates with their scores on a standard intelligence test [33]. People with higher intelligence generally outperform people of lower intelligence in the game environment. Without knowing the intelligence level of all trainees prior to the experiment, we could end up having an imbalance in the number of higher intelligence and lower intelligence participants between the conditions. Our results would show that the condition that had a majority of the higher intelligence participants outperformed participants in other conditions. In that case, we would not know if these participants excelled because of the training condition or their natural ability. Controlling for this threat requires balancing the conditions so each has an equal number of higher intelligence and lower intelligence participants. For accomplishing this, we administered the Raven's APM test to all participants in this experiment. The Raven's APM test [34] is a general intelligence test with 36 questions ascending in difficulty, and scoring based upon the number of questions a participant gets correct. The 177 participants for our experiment had a mean score of 26.7 (s.d. 4.7) and a median score of 27. All participants that score at or above the median were classified as high-ability, while those whose scores fell below the median were considered low-ability. In total, because of several ties, we had 101 high-ability and 76 low-ability participants.

6.3 Partner Questionnaires

Although the game scores would tell us which partner type attains the best performance from a trainee, we would not have much information telling us why. This led us to creating questionnaires that would get the reactions from the participants after training with their respective partner. The 10 question, paper-and-pencil measure was created to give the participants a series of statements that fell under two categories. Half of the questions asked the trainee about their affect towards their partner: how much they enjoyed working with their particular partner. The other half asked about the utility of their partner: how much did their partner contribute to their learning of the game. The participants answered each statement on a scale ranging from 1 – 5, with 1 meaning to "strongly disagree" with the statement and 5 meaning to "strongly agree". We scored a participants affect or utility by averaging the values they gave for the respective statements. An overall score, called *Reaction*, served as the average of all the question responses. Higher *Reaction* scores from a participant meant they had a more positive response to their partner.

6.4 Procedure

Several weeks before the start of the experiment, we heavily advertised for our experiment on and off campus. As compensation for the experiment, we offered people $7.50 per hour, with the experiment requiring five hours of their time. As well, a cash incentive was offered, rewarding the participants with an extra $20 if their scores were among the top-quarter of participants. Since the experiment took place within a university campus setting, we mainly recruited college-aged students with a large majority of them pursuing or just completed a higher-education degree. This advertising campaign netted us 177 participants for the experiment.

We began the experiment by having all participants take the Raven's APM test, with participants given a 40-minute time limit for completing the exam. After scoring the test, we randomly assigned all participants to one of the three conditions for the experiment. However, we had three guidelines for the experiments that forced us to reassign some individuals to another condition. The guidelines were:

- Keeping an equal number of high-ability male participants and low-ability male participants in each condition.
- Keeping an equal number of high-ability female participants and low-ability female participants in each condition.
- Keeping the human dyad teams homogeneous in terms of ability and gender. That means no teams will have a high-ability trainee and a low-ability trainee, and teams will not have mixed sexes.

The next phase of the experiment involved the participants training in the *Revised Space Fortress* environment. All participants would play the game over a two-day span, spending two hours per day in our lab. They began the computer lab activities by watching a 20-minute video explaining the rules and optimal strategies for *RSF*. During that time, they received a reference packet that covered the same material presented in the video. Participants were told they could use the packet or talk to the proctor for answering any questions they had while playing the game. After watching the video, all participants played four baseline games of *RSF*. These games, played individually, allowed each participant some practice in the game environment, and gave us an indication about a person's skills for the game before starting their training. Following the baseline games, the participants watched a five-minute video giving them a summary of the game instructions and strategies. Once completed, the participants began the training regiment for *RSF*.

Seating assignments for the experiment were only required for those who had a human partner. All participants working in the human dyad condition had their partner sitting beside them at an adjacent computer workstation. Both players would see the same game screen on their monitors, but have their own joystick and mouse set for inputs. We encouraged teammates to talk to one another, developing strategies and helping each other in improving the team's Total score.

The training regiment for *Revised Space Fortress* consisted of four sessions, with each session comprised of eight practice games, and two test games. During the practice games, we would divide the controls between the trainee and their partner. One player would only handle the joystick controls, while the other handled the mouse controls. At the conclusion of each practice game, the trainees swapped roles with their partner. Once they finished the eighth practice game, the trainees would complete two test games. During the two test games, the trainee would handle both the joystick and the mouse controls. These games served as a method for checking the progress of a trainee at the end of each session; showing us how well they have learned the skills needed for handling the complex task within the game. The test games were also the ones of interest for the cash bonus. From the eight test games a trainee would play during the experiment, we took their highest Total score from these games and compared them to the highest Total score for the other participants. The participants who had a Total score in the top-quarter would get the extra $20. Hence, we strongly encouraged the trainees to do their best during the test games. After completing all the training sessions, the trainees would fill out the questionnaire asking them about their assigned partner.

At the conclusion of the experiment, we could only use the data from 143 participants, giving us an attrition rate of 19.2%. Out of the 34 participants whose data we could not use, 22 people dropped before finishing the final training session. The remainder includes people who changed partners during the experiment, mainly due to their human partners not showing up at their assigned lab times. We have listed below a breakdown of the participant totals for each condition based upon ability and sex. This list only includes the 143 participants whose data we used for our analysis.

Table 3: Participant Count for Each Condition.

Condition		Male	Female	
Programmed Agent	High ability	23	7	N = 51
	Low ability	13	8	Raven APM score = 27.14
Trained Agent	High ability	22	7	N = 52
	Low ability	15	8	Raven APM score = 26.46
Human Dyad	High ability	19	6	N = 40
	Low ability	9	6	Raven APM score = 27.90

6.5 Results

Table 4 presents statistics that describe how participants progressed as they moved through the training sessions for *RSF*. The table shows the mean Total score from the practice and test games for each session. For example, *Practice #1* refers to a team's average Total score across all practice games in the first training session. *Test #1* represents a trainee's average Total score for the two test games at the end of the first session. During all practice games, the trainee handles only half of the game controls, with their partner responsible for the other half. The trainee handles all of the game controls, the joystick and the mouse, during the test games. An increase in the average test game scores between training sessions shows progress in learning how to perform the complex task. The rate of this increase will depend on the effectiveness of the trainee's condition. The scores for the three conditions listed in the table incorporate everyone who played under that condition, regardless of gender or ability level. The high standard deviations for the mean Total scores listed result from including all participants within that condition. Also, we see that no significant difference exists between the Baseline scores for each of the conditions (*F* = 0.22, *p* = 0.80). Note in Table 3 that the average intelligence test scores were nearly equal across the conditions. Since each condition group has equivalent intelligence and ability playing the game prior to starting their training, then any differences that appear within our data should only result from the affects of the trainee's partner.

Table 4: Performance Results from the Human Subject Experiment.

	Human Dyad		**Programmed Agent**		**Trained Agent**		
Session	Mean	S. D.	Mean	S. D.	Mean	S. D.	*F*-value
Baseline	-2286	1382	-2454	1276	-2430	1152	0.22
Prac. #1	-692	1840	2506	1190	2564	1193	-
Prac. #2	448	2018	3131	1266	3230	1167	-
Prac. #3	1407	1926	3629	1247	3608	1236	-
Prac. #4	1629	1792	3781	1281	3298	1144	-
Test #1	-415	2097	-246	2242	-571	2155	0.29
Test #2	547	2005	722	2401	307	2338	0.43
Test #3	1098	2162	987	2305	724	2270	0.34
Test #4	1379	2184	1295	2313	1247	2252	0.04

Observing the scores from the practice sessions in Table 4, we see a noticeable difference in the scores between the human dyad condition and both agent conditions. In the agent conditions, a trainee works with an agent regarded as an expert player, meaning that the agent could produce a Total score from 5000 – 6000 playing the game alone. Although the agent only handles half of the controls when partnered with a human trainee, the agent still does exceptionally well performing the tasks assigned to it. In the human dyad case, we have two trainees learning the game together. Since neither of their ability levels matches that of an expert, they will attain Total scores far less than a team comprised of a novice and expert player.

For testing our hypothesis, we must look at how a trainee performed at the conclusion of their training regiment. After playing the practice games for the fourth training session, each participant has two test games where they must handle both the joystick and the mouse. These games will indicate how much their skills have developed since starting their training for the complex task represented within *RSF*. Using a student *t* test between the conditions with the largest difference, the human dyad and trained agent conditions, we observe that no significant difference exists between the final Total scores ($p = 0.78$, $t = 0.28$, $df = 90$). We conclude from this finding that a trainee will achieve the same performance regardless if their partner is a trained agent, programmed agent, or another trainee of equal intelligence. This supports our hypothesis that an agent who learns its strategy from an expert player provides the same benefit as agent that follows an explicit, programmed expert strategy. On the other hand, we made an assumption that trainees would develop their skills better working with the trained agent since its behavior better reflected that of a human expert. Even though we proved that the trained agent had a more naturalistic approach toward the game than the programmed agent, their differing behaviors did not influence how well a trainee adopted the skills needed for the game.

Figure 15 plots the performance of the participants in each condition as they complete the training sessions. The points represent the average Total score for the test games played at the end of the training session. We notice that the slopes of these lines vary between the conditions. In some conditions, participants improved quickly (i.e., programmed agent, high-ability) but then had minimal changes toward the later training sessions. Others though maintained a steady rate of improvement throughout the training sessions (i.e., trained agent, high-ability). Yet, the results from Table 2 show that no significant difference existed in the trainee’s performance at the end of each training session across all of the conditions. This plot supports our discovery that participants will receive the same benefit learning the game by training under any of the conditions used.

However, this experiment only shows results for trainees learning the game within a limited time frame. As mentioned in Section 3.2, other experiments using the *Space Fortress* environment had trainees play 100 3-minute trials (10 hours of training), while our experiment subjects a trainee to 44 trials (4 hours of training). Schneider *et al*. [35], defines that a trainee requires over 100 hours of training in order to attain expertise for a complex task. As well, Schneider *et al*. states that a trainee's initial performance does not predict their success at the conclusion of training. Relating these statements to our experiment, we did not provide the trainees enough time in order to attain expert-level ability. Hence, we cannot claim that our results would remain the same if we extended the number of training sessions for the experiment. Instead, our results show the possibility that all three conditions (a human partner, a programmed agent, or our trained agent) can equally assist a trainee in attaining expert-level performance for a complex task.

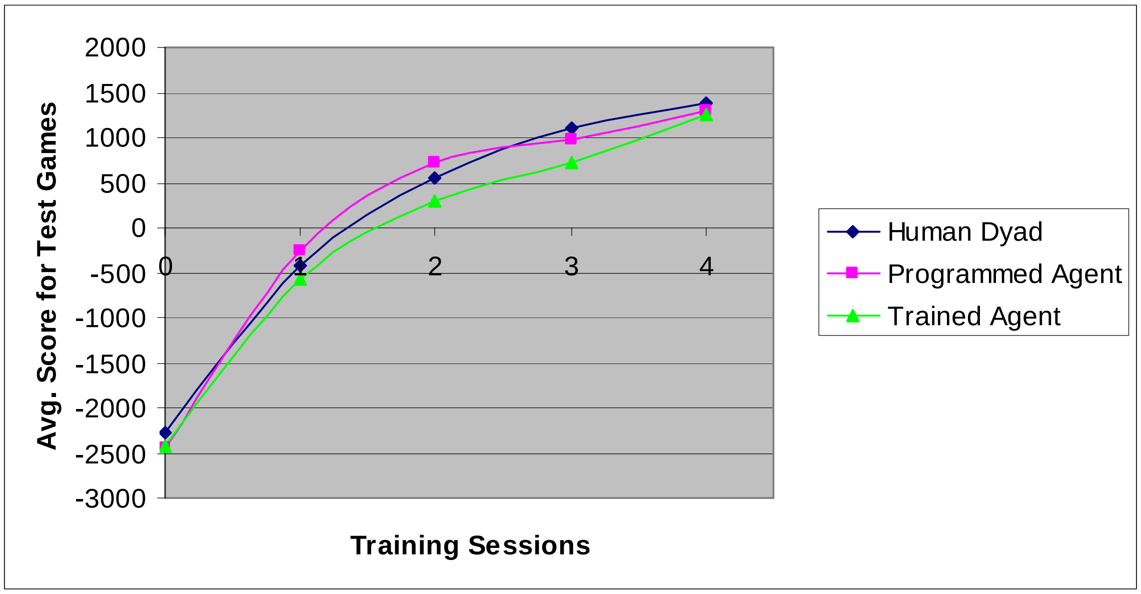


Figure 15: Plot of Each Condition Group's Performance Throughout Training.

6.6 Notable Observations

Once a participant finished all of the training sessions for *RSF*, they would report their opinions about working with a partner through our questionnaire. Table 5 displays the mean scores for the questionnaires from each of the conditions. From our results, we observe that participants playing in the human dyad condition favored their partners more than both the trained and programmed agent conditions. Comparing the *Reaction* scores between the three conditions, an analysis of variance (ANOVA) test reveals a trend toward a significant difference (F = 2.42, p = 0.09), and a statistically significant difference across the *Affect* scores (F = 3.38, p = 0.04). However, we do not see any differentiating trend when analyzing the *Utility* score between the conditions (F = 1.26, p = 0.29).

Interpreting these results, we could state that trainees prefer working with a human partner to a virtual one. However, we saw that the greatest difference in the questionnaire responses came from the *Affect* questions. These questions focus on how much a trainee enjoys working with their partner. A major difference between the human and agent partner, besides their performance in the game, deals with social interaction. Human teams working together could discuss strategy, give each other feedback on the other’s performance, and occasionally joke with one another. These interactions created a more inviting environment for participants, increasing their appreciation for their partner. Although participants in the agent conditions missed out on the social aspect, their partner could demonstrate to them immediately the proper way for playing the game. Working with an expert-level player proved useful to participants working in this condition, thus explaining the smaller discrepancy in *Utility* scores between the human dyad and agent conditions. Even though participants had a bias in which type of partner they preferred (human over virtual), both partner types still helped participants attain the same ability level for *Revised Space Fortress*.

Table 5: Results from the Partner Questionnaire.

	Human Dyad		**Programmed Agent**		**Trained Agent**		
	Mean	S. D.	Mean	S. D.	Mean	S. D.	*F*-value
Reaction	3.98	0.69	3.69	0.72	3.67	0.78	2.42*
Affect	4.01	0.70	3.63	0.79	3.65	0.83	3.38**
Utility	3.95	0.74	3.75	0.75	3.70	0.84	1.26

***: p < 0.10**

****: p < 0.05**

7 DISCUSSION

Our project gives a new approach for designing agents that aid people in complex task training environments. Yet the results from our experiment along with the struggles we faced during the creation of this agent open several avenues of further research. In this section, we will look at two topics of particular interest: improving the process of computer-aided knowledge elicitation and questions about using agents as virtual teammates in other training games.

7.1 Improving Computer-Aided Knowledge Elicitation

For our selected game environment, *Revised Space Fortress*, we built an agent that learned its strategy by observing an expert player. Although this may conjure fantasies of us sitting back while the agent figures out the player's behavior, we actually had to put forth an extensive effort understanding the player's strategy before we could train the agent. For designing an agent that models a player handling a complex task, we ended up researching several issues such as: what tasks must a player carry out in the game, when does a player switch from one task to the other, and how does a player perform each task. Through cognitive task analysis, we determined what tasks exists for a player and how they recognize when to switch between them. For learning how a player performed each task, we built "mini" games that made it easier for our agent to learn a player's behavior. If problems arose during the training iterations, the agent's simplified network structure allowed us to easily see why the agent behaved improperly, and let us confer with expert players about how we could resolve the problem.

As we move toward games that better reflect the real world, the challenge in creating agents that learn from players escalates. In *RSF*, we dealt with a 2D game environment where a player considered all present objects when making any decision. A realistic game, though, usually involves a 3D world filled with interactive, detailed environments and a multitude of tasks that must be attended to throughout the game. Besides the design challenges, a developer has the seemingly impossible tasks of filtering out the descriptive data for the environment and selecting the most useful information for the agent. Furthermore, players in these environments do not always rely on physical cues in the environment for deciding how to act. Some games require that a person use their beliefs about their surroundings, such as predicting the behavior of an opponent, in order to accomplish the game's objective. Without an immediate change occurring in the physical environment to instigate the action, the captured data will not fully contain the player's motivations for their actions, preventing the agent from accurately learning the player's behavior.

Through modifications in our process, we believe we can still build learning agents for these realistic environments. Until tools are created that can automate the entire knowledge elicitation process, communication between the developer and expert players will still exist [36]. We must rely on experts for helping us understand the tasks at hand and determine what information an agent needs for modeling a human player's decisions. As the game grows in complexity, the amount of information necessary for making a decision may also expand. As we have shown in our agent for *RSF*, the more inputs given to an agent, the more difficult it becomes to learn an appropriate model for decisions. We must continue looking into methods that will better represent the input in such a way that an agent can easily learn the patterns within it. For example, the paper by Thurau *et al*. [37] explains the challenges they had in capturing a human expert's movements through a first-person shooter game. In particular, they had trouble teaching the agent how to approach an enemy within the environment. Typically, an expert player will not approach the enemy head-on, but rather try to sneak up on their opponent for surprise attack, effectively killing their opponent while minimizing the amount of damage they would take from an attack. During training, the agent had difficulty interpreting the player's movements since the expert takes a circuitous path towards the opponent. By interpreting the captured data of a player's movements through a Neural Gas algorithm [38], the agent could easily recognize and learn the player's movement behavior. In this realistic environment, adapting the game environment input assisted an agent in detecting the patterns in the player's movement behavior, producing a more human-like agent. For these types of environments, developers should concentrate on finding inventive ways for filtering the enormous amount of information describing the environment and player's behavior into a form that better represents the player's interactions with the environment.

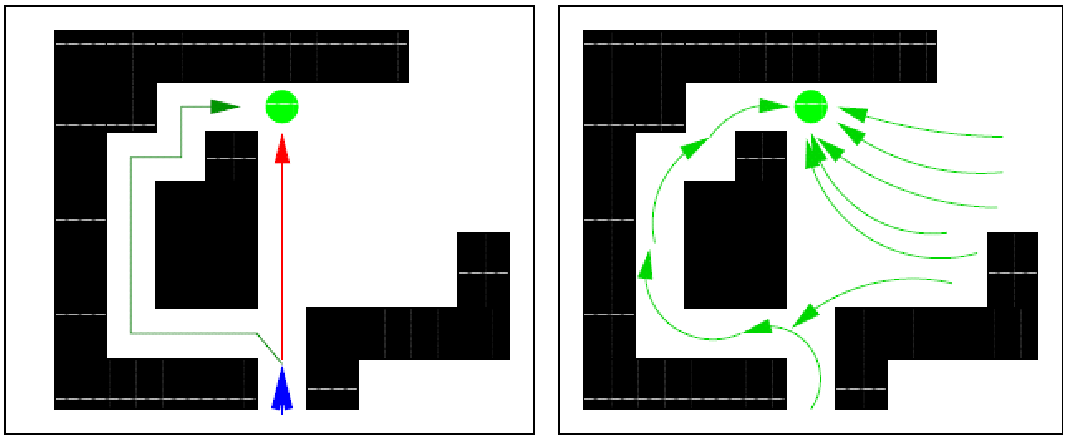


Figure 16: Example of Representing a Player's Movement in a 3D Environment.

Figure 16 shows how the Neural Gas algorithm alters the player's behavior into a format understandable to the agent. The left diagram shows the path an expert takes to approach an enemy, with the straight line serving as the shortest route to the opponent. The right diagram shows a representation of the same path after being converted through a Neural Gas algorithm. Using this representation, the agent learns not to take a head-on approach toward the enemy, but rather move so it intersects the enemy on either side.

As well, we should investigate using other machine learning techniques for modeling a human player's behavior. We chose to use a neural network architecture for our agent due to the network's ability for easily handling a number of continuous, real number inputs. Yet, for more realistic games, our approach for using simple neural networks may not be satisfactory for capturing a player's behavior. For instance, we revealed in

Section 5.2.2 that our neural network system could not mimic a human player's variations in thresholds. We developed networks that would learn the mean estimate of the thresholds observed, yet our design prevented the agent from learning the thresholds in some instances (i.e., interval between right mouse button clicks during foe mine identification). Even if the network did learn the mean estimate threshold, the network would always wait the length of the estimate before executing the action, following a predictable pattern of behavior. Rather than learning the mean estimate, a better approach may involve having the network learn the probability distribution of the observed thresholds [30]. We would again have as an input into the network the number of cycles that have elapsed since the triggering event, but the network output would now correspond to the probability that the action occurs after *x* elapsed cycles. This differs from our previous network which generated a high value whenever the elapsed cycles reach the mean estimate threshold. Using some randomization technique, the agent could automatically change the probability the network output must surpass before an action takes place. By doing so, the same number of elapsed cycles will not always trigger the action, only the number of cycles that will output the necessary value from the network. The combination of the probability distribution network and randomizing the required probability needed for an action will provide variance in the thresholds and let the agent better reflect the behaviors of a human player.

As we have mentioned earlier, solving a complex problem with a neural network can be a challenging feat, forcing us to look for other methods for handling these problems. Usually an increase in problem complexity yields an increase in the size of the networks, adding more neurons or connections yet requiring more time needed for training the network. Fortunately, other tools exist in the machine learning community that may assist us in agent development. Using a Bayesian learning framework [39], an agent learns a player's behavior by observing their behavior and determining the probability that an action will occur given an instance in the environment. A major advantage of this approach over a neural network system is that a Bayesian learner requires a fraction of the training time. Bayesian learners simply use observed data examples for calculating the probabilities of a player's action, removing the need for tuning weights of a mathematical function that models the player's behavior.

As well, our neural network design does not lend itself toward learning how a player makes decisions based upon belief. The idea of approaching this problem only equipped with a neural network seems like an impossible chore. How does someone design a network that represents beliefs? How do those beliefs get updated? How does that impact the actions taken by a player? Instead, we believe an approach such as the *Redux* system presented in [20] would work better. The *Redux* system has an expert player run through a series of situations they would face while solving a particular complex task, highlighting how they manage and pursue the goals for the task during any situation. Using a computer system, we could interact with an expert player and elicit information about how they decide what goal(s) they want to accomplish at any given time. Through machine learning, we could analyze their responses and comprise a rule set that defines what situations can exist for a player. Once we understand how players recognize their situation, then we can determine how they handle it. Again, this reflects the design of our agent for *RSF*: a cognitive framework for deciding which task must be carried out, and a structure that performs the task just like a human player.

7.2 Agents as Virtual Teammates in Other Games

From our experiment results, we showed that a trainee receives the same benefit working with an intelligent agent as they would if they had another, equally able human partner. This should excite military and private organizations, which utilize gaming environments for training purposes, since these agents would remove the financial costs and time required for assembling human teams needed for training. Yet, continued research could give insight as to how virtual partners could provide an even better training experience.

Studies in collaborative team training refer to the concept of zone of proximal development (ZPD) [40], which claims that a trainee's potential for learning a skill is not solely influenced by the trainee's ability level, but that the ability levels of their teammates also play a vital role. Within the paper by Day *et al*. [3], they state that optimal learning occurs whenever small distances exist between the ability levels of the team members. If the partner's ability is noticeably lower than the trainee, the trainee will not get any exposure to methods that will improve their performance. Oppositely, a partner with a significantly higher ability will not help since the trainee will not understand why the partner's methods will improve performance. Therefore, an ideal partner would have a slightly higher ability level than the trainee, improving in relation to the trainee's performance as they progressed through the training sessions.

These requirements would make the task of finding a suitable human partner impossible. However, with virtual teammates we can control the ability-level of trainee's partner, producing a teammate tailored specifically for the trainee. By periodically reassessing the trainee's ability, we can alter the agent such that its ability-level always remains slightly greater than the trainee. This involves answering questions about how we can determine a player's ability and dynamically change an agent to fit our requirements, both of which fall under our extended research interests involving the improvement of computer-aided knowledge elicitation. By conducting this research, we could try to confirm the idea of an optimal partner, and possibly further expand the benefits of virtual teammates in training systems.

Furthermore, another research endeavor could involve using the results from this project into other team training environments. In our project, the trainee would collaborate with a partner by separating the game's controls into two roles: the joystick and the mouse. This created interdependency between the tasks each partner handled in the game. The players must plan or recognize how their actions will impact the other's ability to carry out the tasks associated with their own role. In the *RSF* environment, the interdependency does not seem tightly coupled. The player handling the joystick depends on their teammate for correctly identifying foe mines and collecting missile bonuses when needed. If their teammate does not handle the mouse properly, the player's performance will suffer. On the other hand, the player controlling the mouse is not at all affected by their teammate's ability to control the ship. So, the player will not suffer any performance degradation in terms of their mouse clicks to handle bonuses and foe identification if their teammate does a terrible job at flying the ship or attacking enemies. Current training systems, such as battlefield scenarios, still include collaboration toward an overall goal. Yet, players in these games typically have more freedoms in their actions and a number of responsibilities that not only impact their performance, but also the performance of their teammates. This escalates the interdependency between teammates to the point where the performance of one team member can influence the outcome of the entire team completing their tasks. Trainees in these environments must not learn only *taskwork* skills, the abilities for handling each game task, but also *teamwork* skills, how to cooperate with other players in order accomplish the overall goal. Cooperative behaviors that a player must learn include: exchanging information in a proactive manner, giving guidance to other players, compensating for another player's error, and employing effective communication [41].

Not only must virtual teammates complete their tasks like human players, but also communicate and cooperate with their human partner. Cognitive psychologists, industrial engineers, and computer scientists have started laying the groundwork for producing agents that can operate in these environments. Researchers have studied how human teams share and collect information, and how they use belief reasoning for cooperating between teammates. Belief reasoning means that when a player makes a decision about what actions they will pursue, the player takes into consideration not only their own goals and capabilities, but also the goals and capabilities of teammates, along with the shared responsibilities between everyone. Agent languages such as MALLET (Multi-Agent Logic Language for Encoding Teamwork) create a framework that allows developers to model this cooperative behavior in agents [42]. Also, other researchers have developed methods that allow human trainees to communicate verbally with a virtual teammate. In particular, Traum *et al*. [43] looks into how human players can negotiate with a virtual teammate during a training simulation. Using a cooperation framework, agents can assess both their situation and their human partner's situation. Through this verbal communication link, the agent can receive requests from their human partner and decide whether to accept the request or offer another solution. As well, the agent can make requests to the human player, and react to the human's responses. By further looking into representing cooperate behavior in human/agent teams, we can develop systems that provide a similar training experience as their all-human team counterparts. Afterwards, we can exploit the advantages of agents over human partners, and alter the agents such that they provide the ideal training environment for any trainee.

From our experiment for this project, we observed that the varying behaviors between the trained agent and programmed agent did not affect the performance of participants in these conditions. This may result because the differences between behaviors were not significant. Both agents followed the same strategies reported by expert players of *RSF*, with just the trained agent achieving sub-scores slightly closer to those of an expert player. As the complexity of the game environment increases, so does the number of possibilities for accomplishing the goals of the game. In fact, we may have players that we consider expert-level, yet have entirely dissimilar strategies. Suppose multiple strategies existed in *RSF*, and we had two agents that executed different strategies. Would trainees collaborating with one agent have an advantage over trainees working with the other? Also, do trainees learn better with teammates whose strategies appear more intuitive or better suit the trainee's abilities? Answering these questions will aid in the development of future team training systems, and possibly give more incentive for using agents as virtual teammates.

8 CONCLUSION

As more institutions turn toward computer games for assisting in the training of complex tasks, researchers and game developers will further investigate how they can improve these games for providing a cost-effective, worthwhile experience for trainees. In these environments, people have turned toward using intelligent agents as virtual teammates and tutors as a method, enriching the game environment by having trainees work with and learn from these agents. Yet, as the games increase in complexity, so will the challenge of eliciting knowledge from expert players about their game strategies. Without understanding how an expert player formulates their strategy, developers cannot create agents that will assist trainees in learning the skills for the game. As a solution, this project looked into how computer systems could help in the knowledge elicitation process. Combining traditional knowledge elicitation techniques with machine learning algorithms, we designed an agent that could learn from an expert player by observing how a player handled each task within the game. We showed that this agent better reflected the behavior of a human expert than a known expert-level agent developed using traditional knowledge elicitation techniques. From our human-subject experiment, we showed that trainees working with our agent can receive the same benefit as trainees working with a classical, programmed agent or another human trainee of equal ability. These findings make us optimistic that as games grow more complex, computer systems can assist in the knowledge elicitation of experts.

From the Discussion section, we have a number of potential research possibilities that extend this work. In the immediate future, we would like to investigate the idea of an “ideal” partner, building an agent that can dynamically adjust its performance based upon the ability of the human trainee. If we can develop such an agent, we would then test to see if this approach would provide a better training experience over an agent whose ability level remains static and a human partner that has equal ability to the trainee. As well, we want to see how we can improve upon our design process for building a learning agent for a complex task environment. This includes looking at changes in our task recognition framework as well as the algorithms used for learning each of the player's tasks. Furthermore, we would like to study how our process could assist in capturing expert player strategies for environments with highly interdependent tasks. This requires understanding not only how a player recognizes and handles tasks, but also how a player cooperates with its teammate(s) for accomplishing the overall goals of the game. Using the *RSF* environment, we could bolster task interdependency by having two ships on the screen, with each player fully controlling their own ship. The partners would cooperate by collaborating on the primary task, destroying the fortress, and handling the shared responsibilities between each other (i.e., handling mines, adhering to rules about shooting the fortress, sharing bonuses, etc.). The results from all of the immediate future work projects could then be transferred and applied to more complex game environments that reflect real-life tasks.

REFERENCES

[1] W. L. Shebilske, J. W. Regian, W. Arthur Jr., and J. A. Jordan, "A Dyadic Protocol for Training Complex Skills," *Human Factors*, vol. 34, no. 3, pp. 369-374, 1992.

[2] T. R. Ioerger, J. Sims, R. A. Volz, J. Workman, and W. L. Shebilske, "On the Use of Intelligent Agents as Partners in Training Systems for Complex Tasks," *Twenty-Fifth Annual Meeting of the Cognitive Science Society, CogSci 2003*, Boston, MA, 2003.

[3] E. A. Day, W. Arthur Jr., S. T. Bell, B. D. Edwards, W. Bennett Jr., J. L. Mendoza, and T. C. Tubre, "Ability-based Pairing Strategies in a Team-Based Training of Complex Skill: Does the Intelligence of Your Partner Matter?," *Intelligence*, vol. 33, no. 1, pp. 39-65, 2005.

[4] E. P. Hall, S. P. Gott, and R. A. Pokorny, "A Procedural Guide to Cognitive Task Analysis: The PARI Methodology," Air Force Armstrong Laboratory, Human Resources Directorate, Manpower and Personnel Division, Brooks AFB, San Antonio, TX AL/HR-TR-1995-0108, 1995.

[5] R. Kluwe, "Knowledge and Performance in Complex Problem Solving," in *The Cognitive Psychology of Knowledge*. Amsterdam: North-Holland, 1993, pp. 401-423.

[6] W. Putz-Osterloh, "Strategies for Knowledge Acquisition and Transfer of Knowledge on Dynamic Tasks," in *The Cognitive Psychology of Knowledge*. Amsterdam: North-Holland, 1993, pp. 331-350.

[7] A. Bandura, *Social Foundations of Thought and Action: A Social Cognitive Theory*. Englewood Cliffs, NJ: Prentice Hall, 1986.

[8] S. Russell and P. Norvig, *Artificial Intelligence A Modern Approach*. Upper Saddle River, NJ: Prentice Hall, 1995.

[9] M. Kubat, "Should Machines Learn How To Play Games?," in *Machines That Learn To Play Games*, M. Kubat, Ed. Huntington, NY: Nova Science Publishers, Inc., 2001, pp. 1-10.

[10] T. Mitchell, "Artificial Neural Networks," in *Machine Learning*, E. M. Munson, Ed. Boston, MA: McGraw-Hill, 1997, pp. 81-105.

[11] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From Data Mining to Knowledge Discovery: An Overview," in *Advances in Knowledge Discovery and Data Mining*. Menlo Park, CA: The MIT Press, 1996, pp. 1-36.

[12] A. Gellatly, *The Skillful Mind: An Introduction to Cognitive Psychology*. Philadelphia: Open University Press, 1986.

[13] S. Russell and P. Norvig, "Agents that Reason Logically," in *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Prentice Hall, Inc., 1995, pp. 151-180.

[14] J. Laird, "Using a Computer Game to Develop Advanced AI," *Computer*, vol. 34, no. 7, pp. 70-75, 2001.

[15] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, vol. 4, no., pp. 237-285, 1996.

[16] A. Moore and C. Atkeson, "The Parti-Game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State Spaces," *Machine Learning*, vol. 21, no., pp. 199-233, 1995.

[17] T. Mitchell, "Learning Sets of Rules," in *Machine Learning*, E. M. Munson, Ed. Boston: McGraw-Hill Companies, Inc., 1997, pp. 274-304.

[18] N. Baba, T. Kita, and N. Oda, "Application of Neural Networks to Computer Gaming," *IEEE International Conference on Neural Networks*, Perth, Australia, 1995.

[19] B. Geisler, "An Empirical Study of Machine Learning Algorithms Applied to Modeling Player Behavior in a "First Person Shooter" Video Game," in *Department of Computer Science*: Madison, WI: University of Wisconsin, 2002.

[20] D. Pearson and J. Laird, "Redux: Example-Driven Diagrammatic Tools for Rapid Knowledge Acquisition," *Conference on Behavior Representation in Modeling and Simulation*, Washington D.C., 2004.

[21] S. Cao, R. A. Volz, J. Johnson, M. Nanjanath, J. Whetzel, and D. Xu, "Development of a Distributed Multi-Player Computer Game for Scientific Experimentation and Development of Computer Games," *The Electronic Library - The Int. Journal for the Applications of Technology in Information Environments*, vol. 22, no. 1, pp. 43-54, 2004.

[22] A. Mane and E. Donchin, "The Space Fortress Game," *Acta Psychologica*, vol. 71, no., pp. 17-22, 1989.

[23] W. L. Shebilske, R. A. Volz, K. M. Gildea, J. Workman, M. Nanjanath, S. Cao, and J. Whetzel, "Revised Space Fortress: A Validation Study," Technical Report TSSTI-TR-4-03, Penn State University, University Park, Texas A&M University, College Station, Wright State University, Dayton, June 2003.

[24] S.-U. Guan, S. Li, and S. K. Tan, "Neural Network Task Decomposition Based on Output Partitioning," *The Journal of the Institution of Engineers, Singapore*, vol. 44, no. 3, pp. 78-90, 2004.

[25] J. Fredericksen and B. White, "An Approach to Training Based Upon Principled Task Decomposition," *Acta Psychologica*, vol. 71, no., pp. 89-146.

[26] D. Gopher, M. Weil, and D. Siegel, "Practice Under Changing Priorities: An Approach to the Training of Complex Skills," *Acta Psychologica*, vol. 71, no., pp. 147-177, 1989.

[27] J. C. Forsythe and P. G. Xavier, "Cognitive Models to Cognitive Systems," Technical Report SAND2004-0991, Sandia National Laboratories, Albuquerque, NM, February, 2004.

[28] S. Whiteson and P. Stone, "Concurrent Layered Learning," *AAMAS 2003: Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Melbourne, Australia, 2003.

[29] R. Schapire, "The Strength of Weak Learnability," *Machine Learning*, vol. 5, no., pp. 197-227, 1990.

[30] D. Husmeier, D. Allen, and J. G. Taylor, "A Universal Approximator Network for Learning Conditional Probability Densities," in *Mathematics of Neural Networks*. Boston, MA: Kluwer Academic Publishers, 1997, pp. 198-203.

[31] S. W. Ellacott and A. Easdown, "Numerical Aspects of Machine Learning in ANN," in *Mathematics of Neural Networks*. Boston, MA: Kluwer Academic Publishing, 1997, pp. 176-180.

[32] I. Rivals and L. Personnaz, "MLPs (Mono-Layer Polynomials and Multi-Layer Perceptrons) for Nonlinear Modeling," *Journal of Machine Learning Research*, vol. 3, no., pp. 1383-1396, 2003.

[33] E. A. Day, W. Arthur Jr., and W. L. Shebilske, "Ability Determinants of Complex Skill Acquisition: Effects of Training Protocol," *Acta Psychologica*, vol. 97, no. 145-165, 1997.

[34] J. C. Raven, J. Raven, and J. H. Court, *A Manual for Raven's Progressive Matrices and Vocabulary Scales*. London: H. K. Lewis, 1998.

[35] W. Schneider, "Training High-Performance Skills: Fallacies and Guidelines," *Human Factors*, vol. 27, no. 3, pp. 285-300, 1985.

[36] D. Pyle, *Data Preparation for Data Mining*. San Diego, CA: Academic Press, 1999.

[37] C. Thurau, C. Bauckage, and G. Sagerer, "Learning Human-like Movement Behavior for Computer Games," *The Eighth International Conference on the Simulation of Adaptive Behavior*, Los Angeles, CA, 2004.

[38] T. Martinez, S. Berkovich, and K. Schulten, "A Neural Gas Network for Vector Quantization and its Application to Time-Series Prediction," *IEEE Transactions on Neural Networks*, no., pp. 558-569, 1993.

[39] M. Tipping, "Sparse Bayesian Learning and the Relevance Vector Machine," *Journal of Machine Learning*, vol. 1, no., pp. 211-244, 2001.

[40] L. S. Vygotsky, *Mind in Society: The Development of Higher Psychological Process*. Cambridge, MA: Harvard University Press, 1978.

[41] W. Zachary and J.-C. L. Mentec, "Modeling and Simulating Cooperating and Teamwork," *Military, Government, and Aerospace Simulation*, vol. 32, no., pp. 145-150, 2000.

[42] J. Yin, M. S. Miller, T. R. Ioerger, J. Yen, and R. A. Volz, "A Knowledge-Based Approach for Designing Intelligent Team Training Systems," *Fourth International Conference on Autonomous Agents*, Barcelona, Spain, 2000.

[43] D. Traum, J. Rickel, J. Gratch, and S. Marsella, "Negotiation over Tasks in Hybrid Human-Agent Teams for Simulation-Based Training," *AAMAS 2003: Proc. Second International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Melbourne, Australia, 2003.

APPENDIX A

DIAGRAMS FOR THE *IDENTIFYING AND DESTROYING MINES* SKILL SET

This section contains information about the neural networks used by our agent for playing the *Firing at the Mines* “mini” game. During training, the agent observed data over how an expert player identifies, targets, and destroys mines that appear within a normal game of *Revised Space Fortress*. The neural network topology and the inputs given to the networks assisted in the agent in capturing the expert player’s behavior for this skill. We have printed network diagrams for each of the game controls used within the game (turn left, turn right, fire button, and the right mouse button) along with explanations about each of the chosen inputs.

Right Mouse Button Network

- DoFirstClick – This pseudo-Boolean input will have only two possible values, 0.99 and 0.001. The input gets set to 0.99 whenever a foe mine appears, and switches back to 0.001 once the player performs the first click for foe mine identification. If the player does not successfully complete the foe mine identification (the second click was less than 250 msec. or greater than 400 msec. since the first one), then this input becomes 0.99 again. Otherwise, it will remain at 0.001.
- First Click Interval – This input counts the number of cycles that have elapsed since DoFirstClick has been active (i.e., how long a foe mine has been present without the player taking any action). When DoFirstClick has a value of .001, indicating that the player should not click the mouse button, this input value will remain static at 0.001. The count resets and starts again whenever DoFirstClick switches back to 0.99 (i.e., new foe mine appearance or a failed identification attempt by the player). For bounding this input between 0 and 1, we use the

count within a sigmoid function (see Equation 4, $\varepsilon = 5$), with the resulting value from the squashing function serving as the value for the input.

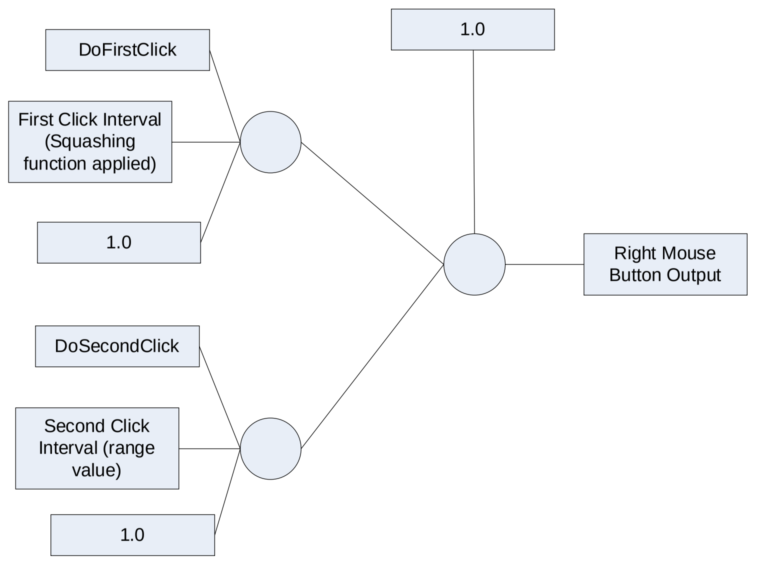
- DoSecondClick – A pseudo-Boolean input that checks if the player has started the foe mine identification process. This input receives a value of 0.99 whenever the player makes the first click in the identification process (a foe mine must be present in order for the input to be set at 0.99), and switches to 0.001 after the second click occurs. The value returns to 0.99 when another right-mouse button click occurs and a foe mine is present.
- Second Click Interval – For this input, the agent counts the number of cycles that have elapsed since the first click for foe mine identification occurred. Based upon the count, this input will have one of three possible values: 0.33 if the count is less than six cycles (46 msec.* 6 = 276 msec.), 0.66 if the count is larger than or equal to 9 cycles (46 msec. * 9 = 414 msec.), or 0.99 if the count is between six to eight cycles. Once the second click occurs, the count resets and remains at zero until the player starts another foe mine identification (by making the first click for the next identification).

Turn Left/Turn Right Networks

- Angular Difference – This input gives the angle between the tip of the ship's nose and the center of the mine. We normalize the value by dividing the angular difference by 180, producing a value range from –1.0 to 1.0. A negative input value means that the mine exists on the left side of the ship's nose, and a positive value means the mine appears on the ship's right side.
- Object Distance – This value contains the distance between the ship and the mine, normalized by dividing the value by the distance of the diagonal from the center of the game screen to the corner.

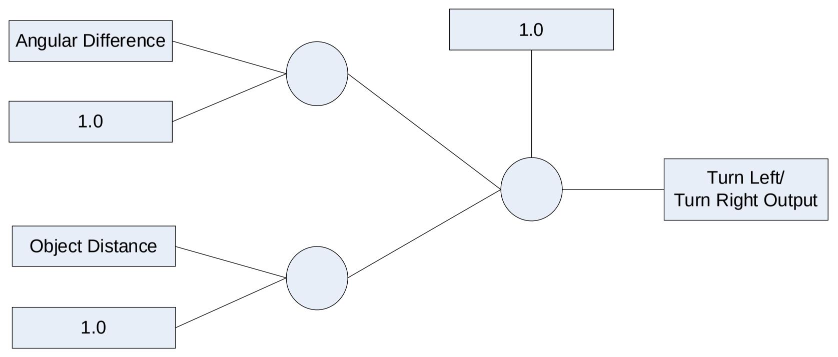
Fire Button Network

- Angular Difference – The absolute value of the “Angular Difference” input used within the turn left/turn right networks.
- Object Distance – The same value as the “Object Distance” input for the turn left/turn right networks.
- Last Shot Interval – This input counts the number of cycles that have elapsed since the last shot occurred. The count resets whenever a player shoots a missile. Again, we use a sigmoid function with the count to normalize the value between 0 and 1 (see Equation 4, $\varepsilon = 5$). Observing the other two inputs (angular difference and object distance), the ship should have a stronger likelihood to fire as the other two inputs decrease to zero. We want to continue this pattern for the shot interval, facilitating the network in learning the desired behavior. In this case, we subtract the squashing function value from one, using the difference as the input value. By doing this, the input value *decreases* as the interval between shots *increases*, as opposed to the input value increasing as the count grows.
- FoeMinePresent – A pseudo-Boolean value that indicates whether or not a foe mine is present on the screen (0.99 if a foe mine exists and 0.001 otherwise).
- FoeIDSuccess – A pseudo-Boolean that tells if the player has successfully identified a foe mine. This is accomplished by the player double-clicking the right mouse button with an interval of 250 – 400 msec. separating the clicks; giving the input a value of 0.99. If the player does not perform the double-click action with a foe mine present, or if a foe mine is not on the screen, then this input has a value of 0.001.


DoFirstClick
First Click Interval (Squashing function applied)
1.0
DoSecondClick
Second Click Interval (range value)
1.0
1.0
Right Mouse Button Output

Right Mouse Button Network

First Click and Second Click Networks
Combined By an OR Gate Neuron



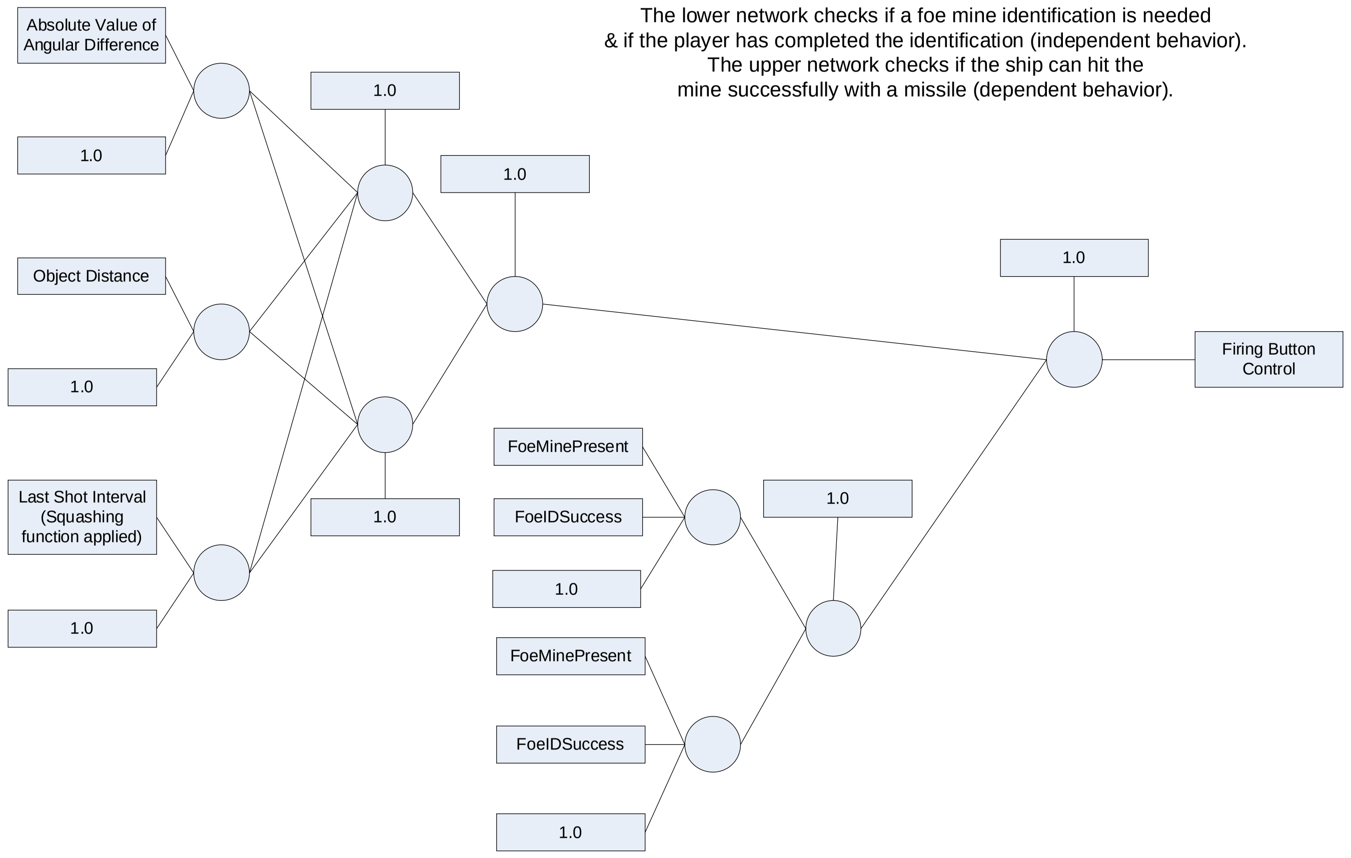
Turn Left / Turn Right Network

Two instances of this network exist
(one for controlling left turns &
the other controlling right turns)

Firing at the Mines Skill Set
Neural Network Controllers

Firing Button Network

The lower network checks if a foe mine identification is needed
& if the player has completed the identification (independent behavior).
The upper network checks if the ship can hit the
mine successfully with a missile (dependent behavior).



Firing at the Mines Skill Set
Neural Network Controllers
(continued)

APPENDIX B

DIAGRAMS FOR THE *FIRING MISSILES AT THE FORTRESS* SKILL SET

This section contains information about the neural networks used by our agent for playing the *Firing at the Fortress* "mini" game. During training, the agent observed data over how an expert player aims at the fortress, and identified the firing rates for both the normal shots (10 shots that hit the fortress at an interval greater than 250 msec.) and the double shot (two shots that collide into the fortress in less than 250 msec.). Again, the neural network topologies and inputs selected assisted the agent in learning the expert player's patterns of behavior for handling this task. We will present diagrams of the neural networks along with explanations about each of the inputs used within each one.

Normal Shot Network

- Last Shot Interval – This input counts the number of cycles that have elapsed since the last shot fired. We have normalized this value using a sigmoid function similar to the one shown in Equation 4 ($\varepsilon = 2$), using the result of the sigmoid function as the input value. The count resets whenever a player fires a missile.
- Object Distance – This value contains the distance between the ship and the fortress, normalized by dividing the value by the distance of the diagonal from the center of the game screen to the corner.

Double Shot Network

- Last Shot Interval – This input shares the same value as the "Last Shot Interval" input listed for the Normal Shot Network. The input tells the number of cycles that have elapsed since the last shot occurred. Since the Double Shot and Normal Shot Networks will trigger the fire button under different circumstances, the weight associated with the Last Shot Interval input in each network will have a unique value. For the Normal Shot Network, the weight will have a value such that the network should fire whenever the input shows that six or more cycles have passed. In the Double Shot Network, the weight will be set such that it fires whenever the Last Shot Interval is smaller than six cycles.

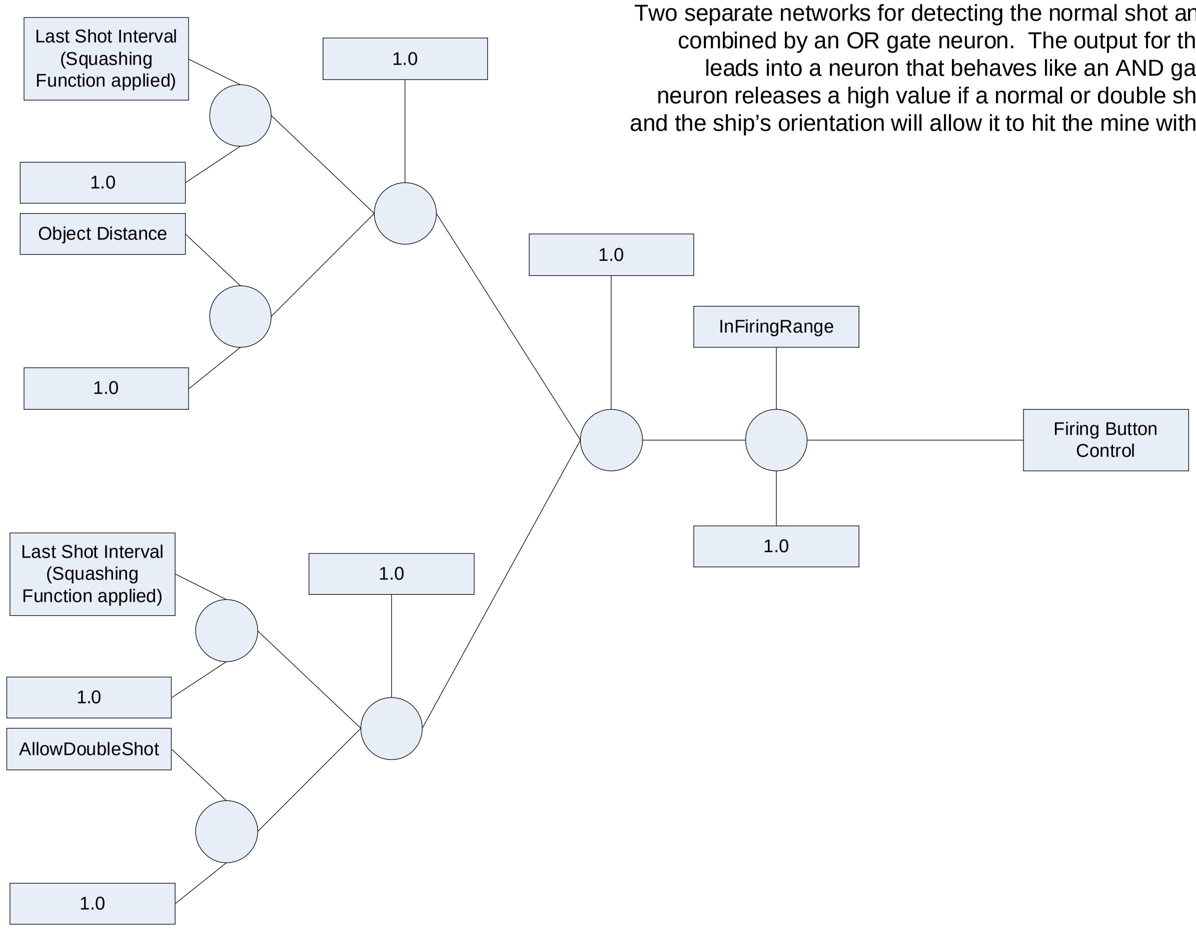
- AllowDoubleShot – A pseudo-Boolean input that tells if a player can fire a double shot to destroy the fortress. The input receives a value of 0.99 whenever the fortress' vulnerability counter reaches 10, accomplished by the player hitting the fortress 10 times without prematurely firing the double shot. Once this input has a value of 0.99, the combined firing network will continue triggering double shots until the fortress is destroyed. If the fortress' vulnerability counter is less than 10, this input will equal 0.001. During this time, all shots coming from the combined firing network will only be normal shots.

OR Gate Neuron and Angular Difference Neuron

We use a binding neuron that receives the output from both the Normal Shot Network and Double Shot Network as its inputs. Whenever either of these networks outputs a high value (meaning the player should fire), the OR Gate neuron will also output a high value, successfully combining these two networks. The output of the OR Gate neuron feeds into another neuron, which we call the Angular Difference Neuron. The purpose of this neuron is to suppress the ship from firing in case the ship's nose is not aligned with the mine. Besides the OR Gate output, the neuron has another input named InFiringRange. InFiringRange is a pseudo-Boolean input that receives a value of 0.99 whenever the angular difference between the ship's nose and the mine is less than 20 degrees, and 0.001 for an angular difference greater than 20 degrees. When both the OR gate output and InFiringRange have high values, the Angular Difference neuron also releases a high value. If InFiringRange contains a value of 0.001, the network will not trigger a firing output, even if the OR Gate output has a high value. Thus, the Angular Difference neuron behaves like a two-input AND gate, releasing a high output only when it detects high values from both its inputs. The output for the Angular Difference neuron serves as the firing button control, completing the combination network.

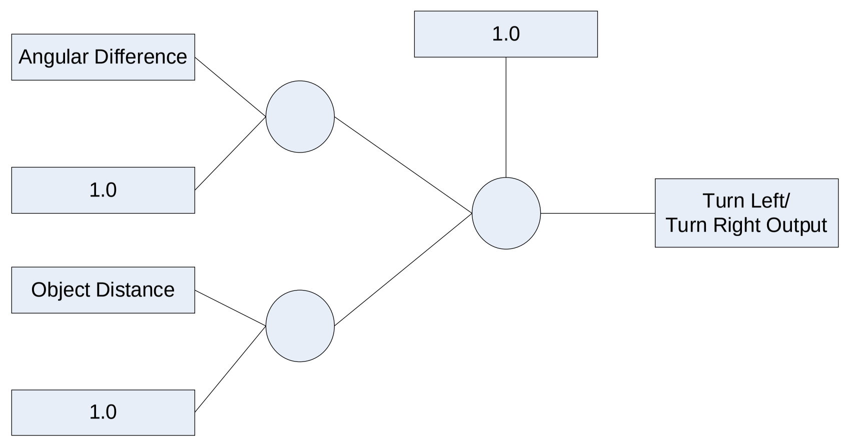
Firing Button Control

Two separate networks for detecting the normal shot and double shot,
combined by an OR gate neuron. The output for the OR gate
leads into a neuron that behaves like an AND gate. The
neuron releases a high value if a normal or double shot is needed
and the ship's orientation will allow it to hit the mine with a missile shot.



Last Shot Interval (Squashing Function applied)
1.0
Object Distance
1.0
1.0
Last Shot Interval (Squashing Function applied)
1.0
AllowDoubleShot
1.0
1.0
1.0
InFiringRange
1.0
Firing Button Control

Firing at the Fortress Skill Set
Neural Network Controllers



Turn Left / Turn Right Network

Two instances of this network exist
(one for controlling left turns &
the other controlling right turns)

Firing at the Fortress Skill Set
Neural Network Controllers
(continued)

APPENDIX C
DIAGRAMS FOR THE *CIRCUMNAVIGATING THE FORTRESS* SKILL SET

This section contains information about the neural networks used by our agent for playing the *Circumnavigating the Fortress* "mini" game. During training, the agent observed data over an expert player's flight patterns within the game. An expert player typically flies the ship in a clockwise path around the fortress, with the nose of ship facing toward the fortress at all times, while keeping the ship within the hexagon boundaries. Also, the player maintains the ship's speed below the Velocity score threshold, which penalizes the player if the ship travels too fast, yet keeps the ship traveling fast enough to avoid incoming shells from the fortress. This section will present diagrams of the neural network structures used for handling each required game control and explanations about the inputs selected for the networks.

Turn Left Network

- Angular Difference – This input gives the angle between the tip of the ship's nose and the center point of the fortress. We normalize the value by dividing the angular difference by 180, producing a value range from –1.0 to 1.0. A negative input value means that the fortress exists on the left side of the ship's nose, and a positive value means the fortress appears on the ship's right side.

The following three inputs require knowing the ship's position in relation to the hexagon boundaries. For accomplishing this, we divided the game screen into six areas, and determined which area the ship appeared during the game. Each area contained a section of the hexagon boundary, one inner wall and one outer wall. Using the ship's position, we found the parametric value, *t*, which told us the position's corresponding point along the each outer hexagon wall. The corresponding point and the ship's position formed a normal line with the outer wall. For each outer wall, we calculated each normal and its distance. We stated that the ship existed in the area that produced the normal with the shortest distance and a parametric value *t* that fell between 0 and 1 ($0 \leq t \leq 1$).

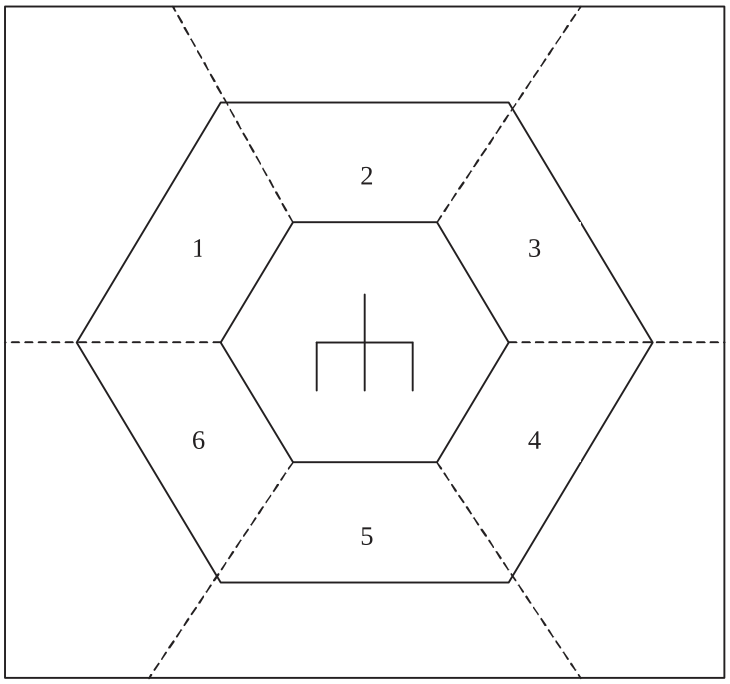


Figure C - 1: Game Screen Division Into Hexagon Boundary Areas.

- Distance to Wall – This input gives the distance between the ship's position and the outer wall within the section the ship appears. We use the distance of the normal that connects the ship's position and the corresponding point on the outer wall, normalized by the estimated radius of the outer hexagon.
- Distance to Corner – As we noted earlier, an expert player flies in a clockwise pattern around the fortress. From the diagram in Figure C-1, this means a ship will cross each area in ascending order. This input tells the distance between the ship's position and the boundary into the next section (the section right of the

current one). We take the normal between the ship's position and the boundary line, and calculate the length of this normal for the distance. We normalize the value by dividing it by a fixed distance.

- Nearest Wall ID – This input tells which area the ship currently appears. We use the ID's shown in Figure C-1 as the value for the input.
- Velocity Magnitude – This input gives the magnitude of the ship's velocity vector. If the ship is at rest, producing a magnitude less than 0.001, we will set the input at 0.001. According to equation (2), if we allowed the input to have a value of zero, we would prevent the network weights associated with this input from changing. Thus, we must keep any input value greater than zero to allow for alterations to the network weights.
- Ship Angle – This input gives the angular difference between the ship's velocity vector and the outer hexagon wall within the ship's current screen area. Ideally, the angular difference between the velocity vector and hexagon wall should be zero, meaning the ship is moving in parallel with the direction of the wall.

Turn Right Network

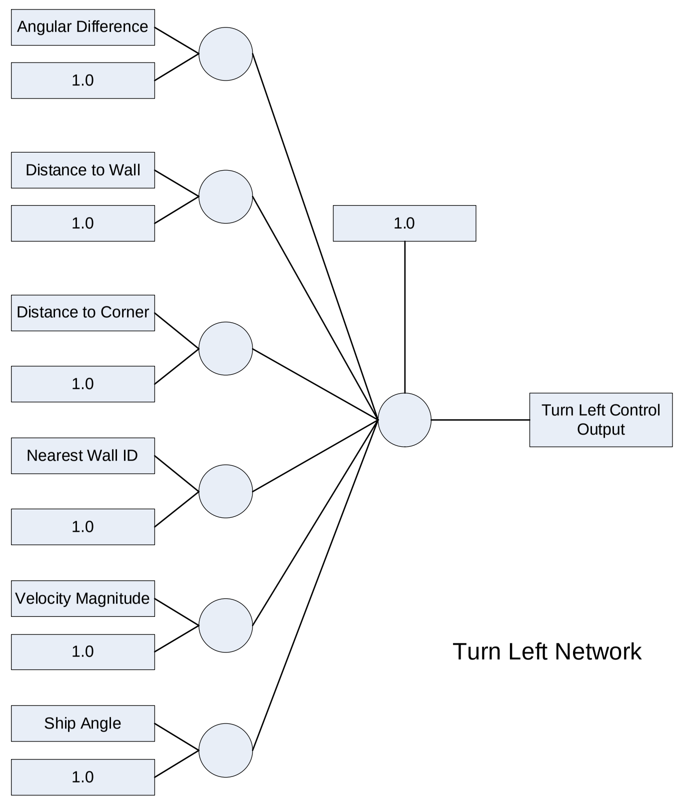
The Turn Right Network shares the same inputs as the Turn Left Network, yet it contains a couple of inputs exclusive only to the Turn Right Network. To prevent redundancy, we will only detail the new inputs for the network.

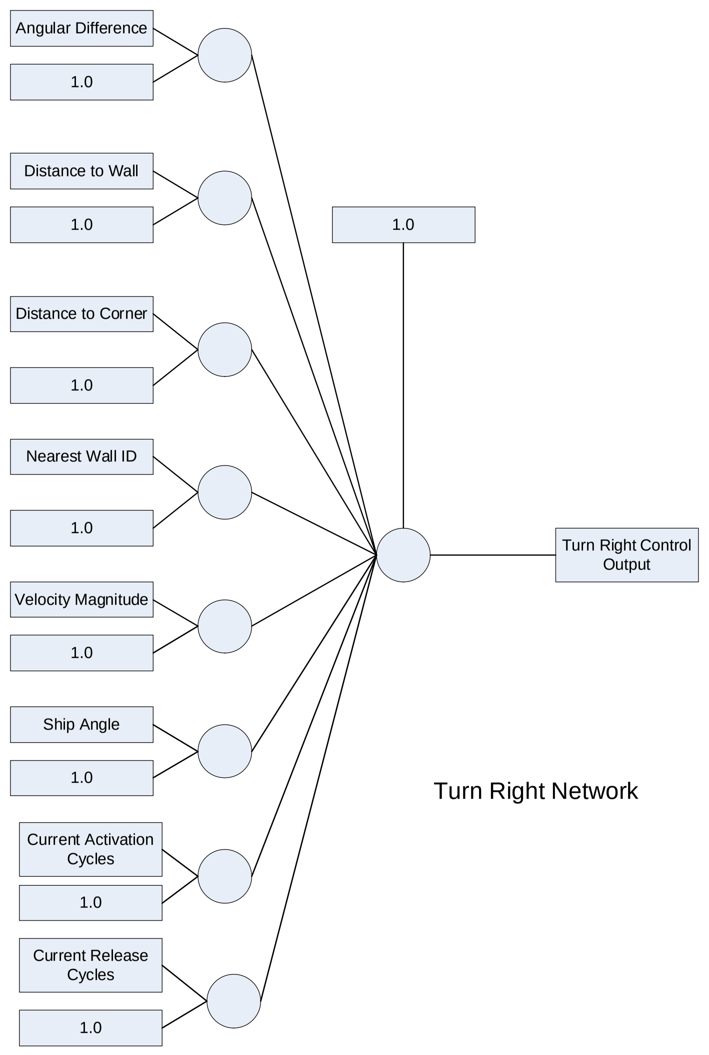
- Current Activation Cycles – This input counts the number of consecutive cycles the input has been active. A human's physical limitations prevent them from keeping the turn control active for only one simulation cycle (46 msec.). This input will help our agent capture how long an expert player holds the joystick to the right for completing a turn. Once the player releases the control, the count will reset to zero, and remain there until the player executes another right turn. We normalize this value by dividing the count by 10.
- Current Release Cycles – This input counts the number of consecutive cycles the player refrains from executing a turn. Since a player can maintain a fairly

constant velocity for the ship, the player should enter and exit each screen area at the same rate. This means that the player should wait about the same duration before turning the ship to enter into the next section, giving our agent an input that will help capture this behavior. The count resets to zero whenever the player starts making a right turn, and resumes once the player releases the control. Again, we normalize this value by dividing the count by 10.

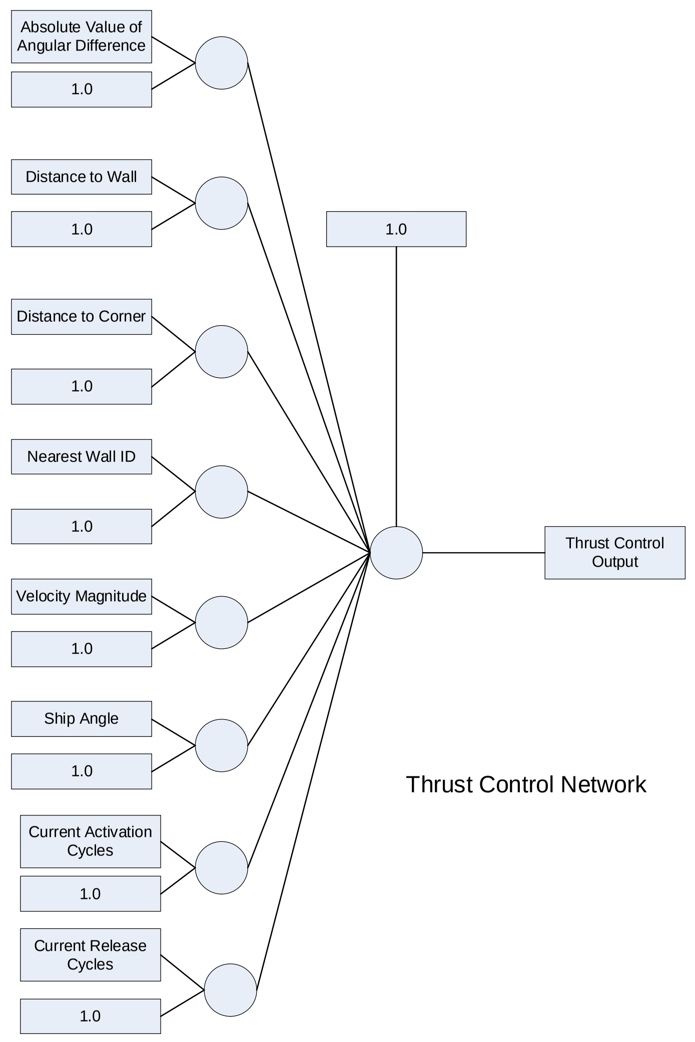
Thrust Network

The Thrust Network also uses the same inputs as the Turn Left Network, as well as inputs that count the number of consecutive cycles the player has kept the thrust control active or released. These activation and release counts work similarly to the ones presented for the Turn Right Network, helping the agent determine how long a player holds the thrust control active and how often they thrust within the game. We also normalize these values by dividing them by a constant value (10 for the activation and 20 for the release).





Circumnavigating the Fortress Skill Set
Neural Network Controllers



Thrust Control Network

Circumnavigating the Fortress Skill Set
Neural Network Controllers
(continued)

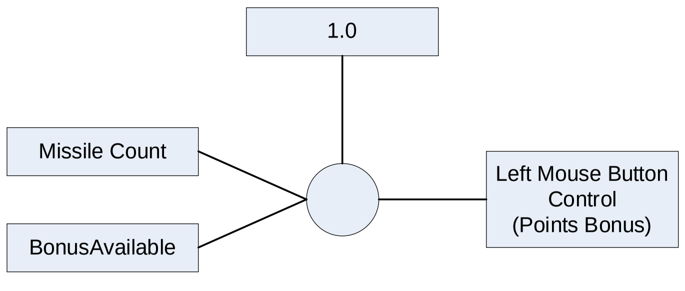
APPENDIX D

DIAGRAMS FOR THE *RETRIEVING BONUS OPPORTUNITIES* SKILL SET

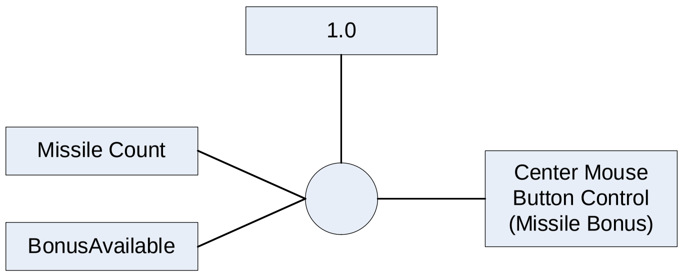
This section contains information about the neural networks used by our agent for capturing an expert player's tendencies when collecting bonus opportunities. An expert player usually capitalizes on all bonus opportunities that appear during the game. Thus, our agent had to learn when a player chooses a points bonus over a missile bonus. An expert player uses their missile supply count as their deciding factor for bonus selection. If the player believes they have too few missiles, the player will take a missile bonus; otherwise they will choose the points bonus. The agent must learn how much a player allows their missile supply to be depleted before they choose a missile bonus over the points. This section will show the diagrams for the networks that control the points and missile bonus selectors (the left and center mouse button respectively), and explain the inputs used within the networks.

Points/Missile Networks

- Missile Count – This input contains the player's current missile supply count. We normalize this value by dividing it by 100; the maximum number of missiles a player may have in their supply at any given time.
- BonusAvailable – Instead of teaching the agent to detect when a bonus opportunity occurs, we simply tell the agent when a bonus opportunity is present. This simplifies the network and provides the same functionality, assuming the agent would observe the player picking up every bonus opportunity presented. This input is a pseudo-Boolean value that receives a value of 0.99 when the second '`$`' character appears on screen (see Section 3.1 Rules of the Game for an explanation over how the game presents bonus opportunities). Otherwise, the input maintains a value of 0.001.



Points Bonus Network



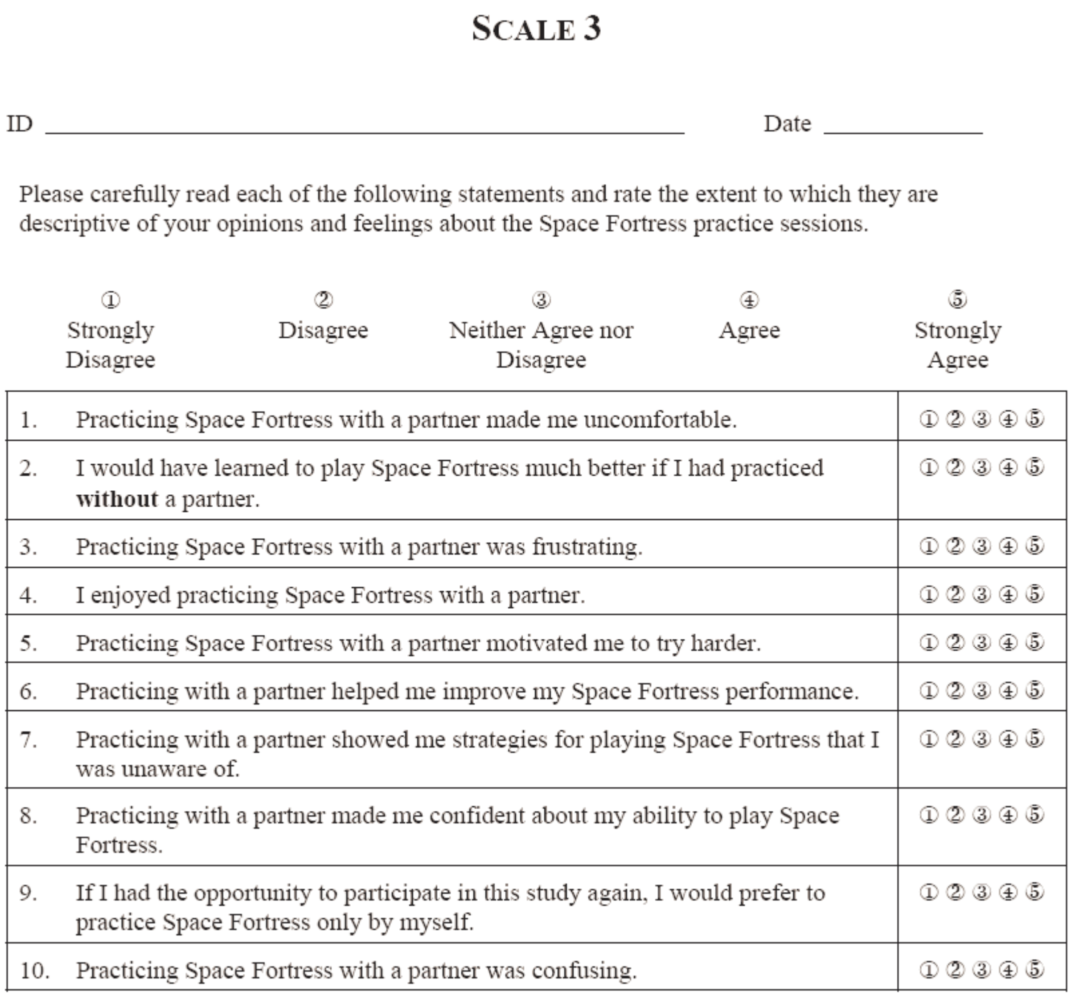
Missile Bonus Network

Bonus Selection Skill Set
Neural Network Controllers

APPENDIX E

PARTNER QUESTIONNAIRES

This section provides the questionnaires given to all the participants at the conclusion of their training regiment. These questionnaires helped us understand how much people perceived each partner type (trained agent, programmed agent, and human partner) as an asset. Questions 1, 3, 5, 7, and 9 pertain to the partner's *Affect*: how much did the trainee enjoy working with their particular partner. Questions 2, 4, 6, 8, and 10 ask about a partner's *Utility*: how much did the trainee believe that their partner helped them in acquiring the skills for the game. Each question has a score from 1 – 5, 1 meaning the trainee "strongly disagrees" with the statement and 5 meaning the trainee "strongly agrees" with it. The *Affect* and *Utility* scores are the average responses given for their respective five questions. The *Reaction* score, how much of an asset did their partner serve, is the average across all 10 questions.



SCALE 3

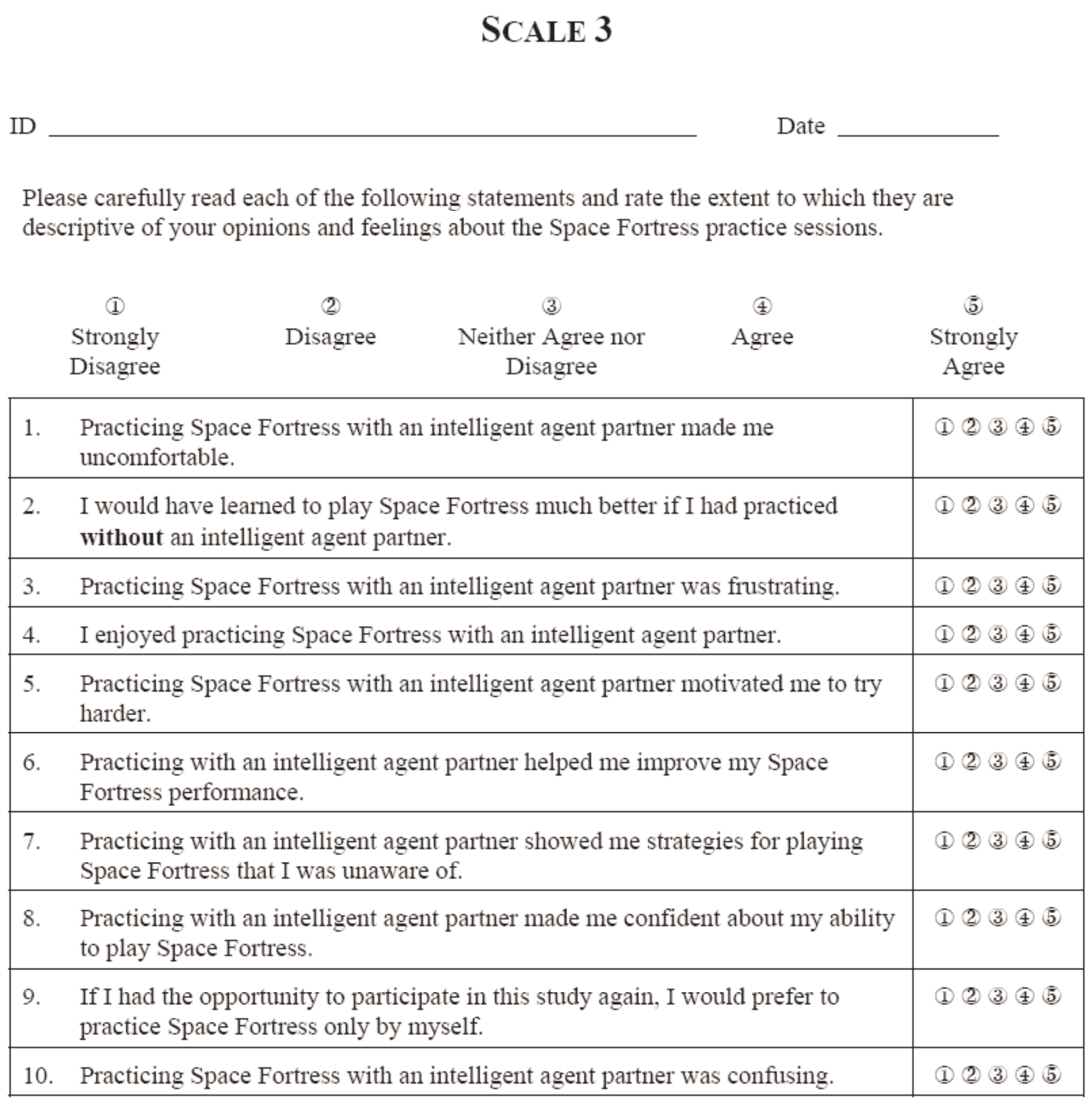
ID ______________________________ Date __________

Please carefully read each of the following statements and rate the extent to which they are descriptive of your opinions and feelings about the Space Fortress practice sessions.

①	②	③	④	⑤
Strongly Disagree	Disagree	Neither Agree nor Disagree	Agree	Strongly Agree

1.	Practicing Space Fortress with a partner made me uncomfortable.	① ② ③ ④ ⑤
2.	I would have learned to play Space Fortress much better if I had practiced **without** a partner.	① ② ③ ④ ⑤
3.	Practicing Space Fortress with a partner was frustrating.	① ② ③ ④ ⑤
4.	I enjoyed practicing Space Fortress with a partner.	① ② ③ ④ ⑤
5.	Practicing Space Fortress with a partner motivated me to try harder.	① ② ③ ④ ⑤
6.	Practicing with a partner helped me improve my Space Fortress performance.	① ② ③ ④ ⑤
7.	Practicing with a partner showed me strategies for playing Space Fortress that I was unaware of.	① ② ③ ④ ⑤
8.	Practicing with a partner made me confident about my ability to play Space Fortress.	① ② ③ ④ ⑤
9.	If I had the opportunity to participate in this study again, I would prefer to practice Space Fortress only by myself.	① ② ③ ④ ⑤
10.	Practicing Space Fortress with a partner was confusing.	① ② ③ ④ ⑤

Figure D - 1: Questionnaire for Trainees Working with a Human Partner.



SCALE 3

ID ______________________________ Date __________

Please carefully read each of the following statements and rate the extent to which they are descriptive of your opinions and feelings about the Space Fortress practice sessions.

① Strongly Disagree	② Disagree	③ Neither Agree nor Disagree	④ Agree	⑤ Strongly Agree

1.	Practicing Space Fortress with an intelligent agent partner made me uncomfortable.	① ② ③ ④ ⑤
2.	I would have learned to play Space Fortress much better if I had practiced **without** an intelligent agent partner.	① ② ③ ④ ⑤
3.	Practicing Space Fortress with an intelligent agent partner was frustrating.	① ② ③ ④ ⑤
4.	I enjoyed practicing Space Fortress with an intelligent agent partner.	① ② ③ ④ ⑤
5.	Practicing Space Fortress with an intelligent agent partner motivated me to try harder.	① ② ③ ④ ⑤
6.	Practicing with an intelligent agent partner helped me improve my Space Fortress performance.	① ② ③ ④ ⑤
7.	Practicing with an intelligent agent partner showed me strategies for playing Space Fortress that I was unaware of.	① ② ③ ④ ⑤
8.	Practicing with an intelligent agent partner made me confident about my ability to play Space Fortress.	① ② ③ ④ ⑤
9.	If I had the opportunity to participate in this study again, I would prefer to practice Space Fortress only by myself.	① ② ③ ④ ⑤
10.	Practicing Space Fortress with an intelligent agent partner was confusing.	① ② ③ ④ ⑤

Figure D - 2: Questionnaire for Trainees Working with a Virtual Partner.

VITA

Born in the arctic north of Lansing, Michigan, Jonathan has lived in several areas across the US with his family, but now calls San Antonio, Texas his hometown. He received a BS in Computer Science in 2002 from Texas A&M University. During his undergraduate studies, Jonathan held intern research positions at Princeton University and Sandia National Laboratories. He accepted a fellowship from the National Physical Science Consortium in 2002 to pursue a MS degree in Computer Science at Texas A&M University. While a graduate student at Texas A&M University, he served as a research assistant working on projects such as the development of the game *Revised Space Fortress* and collaborating with Sandia National Laboratories on computer systems that aid in eliciting game strategy knowledge from expert players. He received his MS in Computer Science from Texas A&M University in August 2005. He is now a Member of the Technical Staff at Sandia National Laboratories in Albuquerque, New Mexico.

His research interests include artificial intelligence, machine learning/data mining, computer-human interaction, and simulation.

Jonathan Hunt Whetzel
635 Stoneway Drive
San Antonio, TX 78258