Worcester Polytechnic Institute
Digital WPI

Major Qualifying Projects (All Years) Major Qualifying Projects

May 2014

Robot Learning through Crowd-Based Games

Andrew Peter Wolff
Worcester Polytechnic Institute

Scott P. Cornman
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation
Wolff, A. P., & Cornman, S. P. (2014). *Robot Learning through Crowd-Based Games*. Retrieved from https://digitalcommons.wpi.edu/mqp-all/2044

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

Robot Learning through Crowd-Based Games

A Major Qualifying Project Report
submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Scott Cornman
Andy Wolff

May 1, 2014

Approved By:

Professor Sonia Chernova, Advisor
Professor Robert Lindeman, Advisor

Abstract

The field of robot learning from demonstration focuses on algorithms that enable a robot to learn new abilities from examples provided by a human teacher. This project aims to show that in the context of learning from demonstration, data collection can be facilitated by collecting demonstrations from remote users through a browser and presenting the learning task as a game with certain motivational features. Furthermore, it explores possible improvements on learning algorithms through the use of filtering by score, the game's natural fitness function. We demonstrate our approach through a system that enables users to remotely control a KUKA youBot robot to play a game of Whack-A-Mole through a common web browser. We collect data on how users play and utilize a decision tree learning algorithm to teach the robot to play autonomously. We find that filtering data by score does not significantly improve robot performance over using all data. Using A/B testing techniques, we find that motivational game features noticeably improve the quantity and quality of collected data.

Table of Contents

Abstract ---------- ii
Table of Contents ---------- iii
List of Figures ---------- v
Executive Summary ---------- 1
1.0 Introduction ---------- 6
2.0 Background ---------- 7
2.1 Machine Learning ---------- 7
2.1.1 Decision Trees ---------- 8
2.1.2 Robot Learning from Demonstration ---------- 9
2.1.3 Robot Learning on the Web ---------- 9
2.2 Demonstrator Motivation ---------- 11
3.0 Design Choices ---------- 14
3.1 Machine Learning Algorithm: Decision Trees ---------- 14
3.2 Machine Learning Library: Waffles ---------- 14
3.3 Choosing a domain ---------- 15
3.4 Whack-a-mole design ---------- 18
3.4.1 Layout ---------- 18
3.4.2 Web Interface ---------- 21
3.4.3 Game structure ---------- 23
3.4.4 Motivational Aspects ---------- 24
4.0 Methodology ---------- 29
4.1 System Overview ---------- 29
4.2 Building the Mole Mechanisms ---------- 30
4.3 Robot Control Software ---------- 35
4.4 Server Setup ---------- 38
4.5 Basic Whack-a-mole Game ---------- 38
4.5.1 Core Game Logic ---------- 38
4.5.2 Web Control ---------- 40
4.5.3 User Account Creation ---------- 44
4.5.4 Queue ---------- 45
4.6 Learning from Demonstration ---------- 46

4.6.1 Collection of Demonstration Data ---46
4.6.2 Learning Algorithm Implementation ---47
4.6.3 Execution of Learning from the Web---49
4.6.4 Evaluation of Learning Effectiveness ---50
4.7 Motivational Game Elements ---50
4.7.1 High Scores Table ---50
4.7.2 User Statistics ---52
4.7.3 Level Progression ---53
4.7.4 A/B Testing Procedures ---58
4.8 Running the Study --60
5.0 Results and Discussion---62
5.1 Study Results --62
5.2 Robustness of the System---64
5.3 Learning Effectiveness---64
5.4 Impact of Game Features --68
6.0 Conclusions---74
Works Cited--76

List of Figures

Figure 1: The KUKA youBot (http://www.youbot-store.com/) -- 3
Figure 2: Example decision tree (Lozano-Pérez & Kaelbling, 2005)---------------------------------- 8
Figure 3 - Traditional 3x3 mole layout--19
Figure 4 - Long strip of moles layout --20
Figure 5 - Disconnected areas mole layout ---20
Figure 6 - Isosceles trapezoid mole layout--21
Figure 7: Block diagram of the complete system --29
Figure 8: Full Whack-A-Mole setup---30
Figure 9: Illustration of mole mechanism behavior ---31
Figure 10: Initial mole mechanism prototype ---32
Figure 11: A finished mole mechanism --33
Figure 12: CAD models of select final prototype components --33
Figure 13: Comparison of mole prototypes--34
Figure 14: Diagram of robot position control with laser scanner-------------------------------------36
Figure 15: Robot position and mole whacking buttons ---42
Figure 16: Simulated robot visualization --42
Figure 17: Sample decision tree printout ---48
Figure 18: High Scores page--51
Figure 19: User Stats page ---52
Figure 20: Sprite-sheet of worm in air --55
Figure 21: Sprite-sheet of worm in ground --55
Figure 22: Sprite-sheet of mini-mole --56
Figure 23: Laser mechanisms and beam ---57
Figure 24: Screenshot of level four --58
Figure 25: Total games played by user category--63
Figure 26: Comparison of human-controlled and autonomous robot performance filtered by score --66
Figure 27: Score vs training data size (linear scale)---67
Figure 28: Score vs. training data size (log scale) ---67
Figure 29: Average games played per user with standard deviation ---------------------------------68
Figure 30: Percent of users who played more than 3 games --69
Figure 31: Average score achieved with standard deviations--70
Figure 32: Average state-action pairs collected with standard deviation-----------------------------71
Figure 33: Average approximate game duration --72
Figure 34: Percentage of games which lasted less than a minute -------------------------------------73

Executive Summary

Artificial intelligence allows computer programs to autonomously perform tasks independently of human control. In many cases, the algorithms that determine the behavior of these programs are manually written by human developers, but in the field of machine learning, behaviors are learned programmatically. This is especially useful for robotics applications, as it is difficult to directly program a robot to handle all of the diverse cases it may encounter in real-world situations.

Robot Learning from Demonstration (LfD) is a growing branch of machine learning in which human teachers generate training data, typically in the form of state-action pairs, by demonstrating the target task. A robot then learns to complete the task autonomously by processing this data. Most robots used for LfD studies are confined to their laboratories, and researchers must bring demonstrators into these labs to obtain training data. This requires significant extra time and effort for both the demonstrators and the researchers. Continuing a recent trend, this project dealt with these issues by bringing Learning from Demonstration to the web. By allowing users to control a robot from the browser, this project was able to more easily obtain training data from a wide variety of users.

Another difficulty encountered in many LfD studies deals with motivating users to provide useful data for the robot. Because many robotics tasks can be boring and repetitive, some researchers are forced to pay participants for providing demonstration data. This can be expensive and does little to guarantee the quality of the data. This project used a game for the learning task and focused on attracting users by making the game enjoyable. A number of motivational game features were implemented to increase both the quantity and quality of demonstration data provided.

Decision Trees were selected as the primary machine learning algorithm for this project, as they are easily human-readable, can handle a mixture of discrete and continuous data, and can be generated quickly as more training data is added. The Waffles[1] machine learning library was chosen for an implementation of this algorithm, as it was open-source, easy to use, and written in C++, the language used for this project's robot control code.

[1] http://waffles.sourceforge.net/

For the project's central learning task, the game of whack-a-mole was chosen, because it was simple for users to learn and fun to play, could be represented with a reasonably-sized state space, was easily resettable, and was not overly susceptible to lag. The layout of the physical whack-a-mole game consisted of seven mole holes, arranged in an isosceles trapezoid, such that the robot would have to drive to different positions to whack all of the moles. User input for the game was gathered through mouse controls on a web interface. A fixed-length game format was chosen, in which the goal was to obtain the greatest number of points by whacking as many evil moles as possible while not whacking the good moles.

The system architecture implemented for this project consisted of the following four components: the moles, the server, the web interface, and the robot. The first component was an eight feet long by two feet wide whack-a-mole setup consisting of seven individual mole mechanisms. Each mechanism was responsible for controlling which of two possible moles would appear in a single mole hole. This mechanism used a servo and a belt drive to rotate a pair of knitted moles about an axle. Custom pulleys were 3D printed for these mechanisms, and custom servo mounting plates were laser-cut from acrylic. Each mechanism was housed in a self-contained wooden frame. Several iterations of the pulleys, mounting plates, and moles were designed, prototyped, and tested before deciding on a final design. An Arduino RedBoard microcontroller with a servo shield was used to control the position of each mechanism's servo.

The second component of the system, the server, handled all core game logic for whack-a-mole and was responsible for communicating with all of the other components. The server kept track of the state of each pair of moles and sent commands to the Arduino accordingly. The server code was primarily written in C++, using the Robot Operating System framework (ROS). The libserial library was used for communication between the server and the Arduino.

The web interface was responsible for interacting directly with users and relaying information between the users and the server. The interface was used to display a video feed of the game, along with additional game information, to users over the internet. When users clicked buttons on the interface to control the robot, commands were relayed to the robot through the server. Additionally, the web interface handled all user account management and other administrative tasks. The interface's back end was primarily written in PHP, while the client side used HTML, CSS, and JavaScript. A MySQL database was used for storing game and user information.

The robot used for the study was a KUKA youBot. It was controlled by a program, written in ROS that responded to commands sent from the server. These commands allowed the robot to navigate to discrete positions, using a laser scanner for navigation, and to move its arm to whack nearby moles.

Figure 1: The KUKA youBot (http://www.youbot-store.com/)

To allow the robot to learn to play whack-a-mole, the interface collected state-action pairs from users' games and stored them in the database. When a user logged into the system, he could control the robot to play a game of whack-a-mole by clicking on mole and robot position buttons corresponding to actions for the robot to take. Each time a user clicked one of these buttons, the interface logged the corresponding action with the current state of the system as a state-action pair. The instantaneous state of the game was mainly represented by seven mole times, each corresponding to which type of mole was occupying a particular hole, if any, and how long it had been up. The state also contained information on the position of the robot and the orientation of its arms. The action space consisted of six possible actions: driving to one of three robot positions or whacking one of three nearby moles.

In lieu of playing a game directly, each user was given the chance to watch the robot play the game autonomously. To do this, the robot collected the state-action pairs stored in the database and used them to generate a decision tree. This tree was then used to select each action for the robot to take given the current game state. When starting the game, the user could select

whether to train the robot with only the state-action pairs he provided or with the training data provided by all users.

To improve user engagement, and thereby increase the quantity and quality of demonstration data provided, a number of motivational game features were added to the game. To foster healthy competition between users, a high scores table was implemented, allowing users to compare their best scores with those of others. Game statistics were also tracked for each user and were made available for all users to see. To keep users coming back, a progression of five levels was added to the game. Each level introduced new game mechanics and came with additional backstory. Later levels included several virtual game elements, such as worms for feeding moles to keep them up longer, mini moles that slowed down the robot unless dragged off, and lasers that could be fired at entire rows of moles at once.

Once the system was fully implemented, the game was opened to the public, and a study was run for two weeks. Over the course of this study, 523 games were played by 191 unique users, producing a total of 9568 state-action pairs. The data collected from the study was used for a few different types of analysis.

From the state-action pairs collected during the study, the robot was able to successfully learn to play whack-a-mole autonomously. When trained from all of this training data, the robot was able to outperform the average human player. Further analysis was conducted to determine the impact of both quantity and quality of demonstration data on the learned behavior of the robot. It was found that increasing the size of the training data set generally resulted in higher average scores for the autonomous robot. Filtering the data by game score revealed that the robot performed significantly better when trained with data from high-scoring users than when trained with data from low-scoring users.

In order to assess the impact of each game feature on the demonstration data provided by users, each user account was randomly assigned a study category upon creation. This category determined which game features that user had access to. Analysis of data quantity showed that more players returned for four or more games if they had access to motivational features. Access to multiple levels contributed to this slightly more than access to the high scores table, but a combination of both resulted in the highest numbers. In terms of quality, users without access to either game feature earned far lower scores on average than those with either high scores or levels. Users with access to the high score table but only the basic level performed the best on

average. This is likely because a desire to achieve the high score kept them playing this level again and again with continual improvement in skill.

1.0 Introduction

Artificial intelligence is a large and quickly-growing field of computer science. Programmers usually manually construct behaviors for specific tasks, but for robots operating in complex real-world environments, there are too many different and unforeseeable tasks a robot may need to perform for this approach to be viable. It is possible to generate behaviors autonomously using machine learning algorithms, but these can be expensive and slow.

An increasingly popular branch of machine learning is Learning from Demonstration (LfD), which involves using a human's demonstration of the task to lead the robot's learning in some way. This can generate behaviors much faster than learning from nothing. No special technical knowledge is required on the part of the demonstrator, making this much more accessible than programming behaviors using conventional means. However, some tasks are very dull to demonstrate, especially to demonstrate repeatedly. The demonstrator may become bored and begin demonstrating sloppy behavior, causing the robot to learn to behave sloppily, or may choose not to participate in further demonstration. To counteract this, some researchers pay participants for their time. However, this is expensive, and motivation research has shown that monetary incentives promote quantity of work without improving quality. Another problem is that many LfD studies require participants to be physically present. This severely limits the number and variety of participants.

To address these drawbacks, this project implemented a web interface through which users could control a robot in playing the game of whack-a-mole. The web interface was used, because it allowed more users to participate. A game was chosen for the task so as to improve engagement, in turn improving quality and quantity of demonstrations. The demonstration data provided by users was used to train the robot to play the game autonomously. This data was also used to assess the impact of filtering training data by game score on the quality of learned behavior. Additionally, the project studied the effects of various motivational game features on the demonstrators and their performance.

2.0 Background

This project builds upon prior work in a number of areas. This section gives an overview of machine learning and Learning from Demonstration (LfD), with a focus on previous work toward bringing LfD to the web. It also discusses the topic of human engagement, which the project uses to motivate participants in an LfD study.

2.1 Machine Learning

Machine learning is a technique by which computers learn a model, often in order to complete a task or classify data, based on past experience and relevant examples (Alpaydin, 2004). In machine learning, software solutions to specific problems need not be programmed explicitly. Rather, the computer program uses data analysis and pattern matching to learn solutions based on known examples. Machine learning is particularly useful for solving problems that humans are either unable to solve or unable to explain their ability to solve, such as speech-to-text conversion (Alpaydin, 2004). In such a domain, there may be wide ranges of input and output values that are related by a series of difficult-to-quantify rules. Using traditional algorithms, the programmer would be responsible for ascertaining and implementing all of these rules, a daunting or even near-impossible task for some problems. With machine learning, the program analyzes a set of example input-output pairs, known as state-action pairs for some applications, and constructs a set of rules, known as a policy, to fit them. Using these rules, it can then predict output values for previously-unseen inputs. Machine learning is also particularly suitable for solving problems that vary based on environment, such as a user interface that changes its behavior to suit the needs and habits of each user (Alpaydin, 2004). For these types of problems, machine learning allows the programmer to create a general application that can adapt to different circumstances, rather than requiring an entirely new implementation for each scenario.

The field of machine learning contains a wide variety of algorithms for learning policies for mapping input values to output values, each with its own advantages and disadvantages. For example, the k-nearest neighbor algorithm represents example state-action pairs as labeled points in n-dimensional space, with each coordinate corresponding to a single input attribute. When a new state is encountered, the algorithm chooses an action based on the most common label of some number of points nearest to this state. Gaussian mixture models behave similarly, but save memory by fitting the cloud of points to a number of ellipsoids that can be derived from

equations. Decision trees represent the process of selecting an output as a tree of choices based on input attributes. Reinforcement learning involves determining correct behavior based on rewards or punishments given for successfully or unsuccessfully completing a task. For more information on these and many other types of learning algorithms, consult Chapters 18 through 21 of Russell and Norvig's text on artificial intelligence (Russell, 2009).

2.1.1 Decision Trees

The primary learning algorithm used in this project was decision tree learning. Russell defines a decision tree as representing "A function that takes as input a vector of attribute values and returns a 'decision' – a single output value" (Russell, 2009). A decision tree determines the appropriate output value by conducting a series of tests on the input values. Each node in the tree represents a test of a single input feature, where each possible value or range of values that the feature can have corresponds to a branch. The leaves of the tree correspond to the output values. An example of a simple decision tree can be seen in Figure 2.

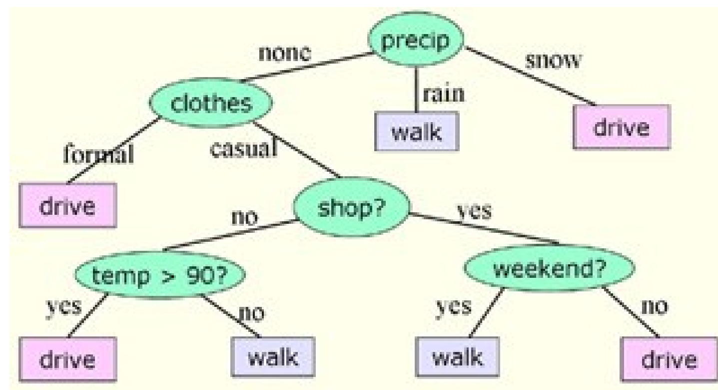


Figure 2: Example decision tree (Lozano-Pérez & Kaelbling, 2005)

In machine learning, decision trees can be induced from a set of example input-output mappings. The ideal objective when constructing a decision tree based on examples is to produce the smallest possible tree that is consistent with all of the examples. One method of finding a relatively small tree is a greedy divide-and-conquer approach, in which tests of the individual input attributes that have the largest impact on the output value are placed nearest to the root of the tree (Russell, 2009). This increases the likelihood that output values can be determined from a small number of tests, in which case the tree must be very shallow.

2.1.2 Robot Learning from Demonstration

One area in which machine learning can have a particularly large impact is the field of robotics. As robotic systems must sense and interact with the physical world, autonomous robotics tasks often involve determination of behavior based on complex state representations. Decision making in such large and imperfect domains lends itself well to machine learning, though it comes with a set of additional challenges. In robot learning, the main focus is on learning a policy that maps the state of the robot's environment to the actions that the robot will perform. Due to the complexity of the state and action spaces that many robots must deal with, it can be very difficult to program these policies using mathematical algorithms, and thus machine learning algorithms can be used to allow the robots to learn policies themselves.

Learning from Demonstration (LfD) is a method by which a robot learns a policy from examples provided by a teacher (Argall, Chernova, Veloso, & Browning, 2009). To create these examples, the teacher demonstrates the desired task, and this demonstration is broken up into a sequence of state-action pairs mapping the world state at each point in the demonstration with the action taken by the demonstrator. Using these state-action pairs as training data, the robot can then attempt to perform the task autonomously, applying machine learning techniques to determine the appropriate actions to perform throughout its attempt. A number of studies have been conducted in the field of LfD, using a variety of machine learning techniques and to accomplish a variety of tasks. For a comprehensive survey of the many approaches to LfD, consult Argall, et al.'s *A survey of robot learning from demonstration (2009).*

2.1.3 Robot Learning on the Web

In many areas of machine learning, recent efforts have focused on solving problems through the use of larger data sets, usually acquired from the web. In the field of natural language processing, Banko and Brill found that increasing the size of the training data set continued to observably improve learning results even as the size expanded to several orders of magnitude larger than the standard size (2001). Further evidence has shown that obtaining a larger data set has often been a more successful use of effort than improving the learning algorithms used to process the data. Halevy, Norvig, and Pereira found that a carefully filtered and annotated corpus of one million words was less useful for learning natural language processing than was a trillion-word corpus compiled programmatically from unmodified web pages (2009).

Applying this principle to robot learning from demonstration, obtaining training data from many users over the internet should result in greater success. Nevertheless, due to the complications involved with non-expert, off-site robot control and training, significantly less research has been done in this area thus far. Over the last two decades, several experiments have been conducted into web-based robot control. The first web-based robot control system was the Mercury Project, which allowed users to control a robot searching a sandbox for buried artifacts (Goldberg, Gentner, Sutter, & Wiegley, 2000). The project focused on the creation of a robust system that would encourage repeated user visits, and it was accessed by more than five thousand machines during its continuous 6-month run from 1994 to 1995. In 2000, Saucy and Mondada built a system through which users could drive a robot around a maze environment from an open web page, controlling the position of the robot and the orientation of an overhead camera (2000). One year of usage data from the system was collected and analyzed for user behavior patterns. Brady and Tarn conducted research into methods of dealing with the varying latency that characterizes internet robot control (2002).

More recently, researchers have begun using web-based robot control systems to aid in machine learning. Due to the importance of large sets of data in machine learning, much effort has been applied toward increasing the size of LfD training datasets by increasing the number of users participating in user studies. For a typical robot learning study, this can be difficult, as the robot, for cost and safety reasons, is usually confined to a specific laboratory. To conduct user studies, researchers typically must bring nearby subjects into the lab one at a time, a process that requires significant time for both the subjects and the researchers and limits the total number of subjects that can participate. Through the use of a web interface for robot control, human subjects from around the world can log on at their convenience and control the robot. This would allow for a greater number of teachers to contribute to the training data in the same amount of time, likely increasing the comprehensiveness of the data. Furthermore, if the robot is capable of operating and repeatedly completing the task without researcher supervision, training of the robot can occur at any hour, without taking up the time of the researchers. If enough users can be found to participate in the study, such a system presents the opportunity for much more extensive data to be captured and thus far greater LfD performance to be achieved.

Crowdsourcing of robot learning data collection has been attempted in several studies in recent years, and has been approached from multiple angles. A study conducted by Crick, et al.

involved more than one hundred users training a robot to navigate a maze via a web-based interface. The study, which used decision tree learning, focused on improving LfD by more closely matching the perception of the world that humans use to make decisions in demonstrations with the sensory perception that the robot is limited to (Crick, Osentoski, Jay, & Jenkins, 2011). The Robot Management System, a general framework for integrating crowdsourcing with human-robot interaction, provides a web-based robot control interface, as well as features to support user study tests and the ability to interface with either physical robots or simulation (Toris, Kent, & Chernova, to appear).

The primary benefit to crowdsourcing the collection of training data is that, when a sufficient number of users are involved, large sets of data can be collected without an excessive or impossible time commitment on the part of each user. One of the chief difficulties of this approach can be finding enough users to participate in a study. A solution to this that has been explored in many recent crowdsourcing ventures is the paid micro-task market. These markets allow individuals to complete very small tasks, generally taking no more than a few minutes to complete, in return for small monetary rewards, typically only a few cents to a few dollars. The most well-known paid micro-task market framework is Amazon's Mechanical Turk, which has been used for a wide variety of applications. One previous application of this framework in robot learning dealt with grasping objects of unknown shapes. To accomplish this, the robot requested information from the crowd at key steps in the process, using Mechanical Turk to crowdsource such subtasks as image labeling, object clustering, and model selection (Sorokin, Berenson, Srinivasa, & Hebert, 2010). While monetary incentive can be an effective means of obtaining users for an LfD study, it can be costly, and it does not necessarily motivate users to train the robot as effectively as they could. An alternative is to make the task of training the robot inherently enjoyable, thus motivating users to complete the task often and work to provide the best training data that they can.

2.2 Demonstrator Motivation

Learning from demonstration studies usually focus on learning algorithms, the structure and content of the data collected, or the method of collecting data. There are very few papers which focus on the engagement of the human participants. However, human engagement is an area which has been studied broadly in psychology and in the field of video games. A popular

theory in psychological studies is Self-Determination Theory (SDT). This theory classifies motivation into two distinct groups: intrinsic motivation and extrinsic motivation. Intrinsic motivation stems from the desire to do a task because it satisfies the basic psychological needs for competence, autonomy, and relatedness, whereas extrinsic motivations come from outside of the task itself, such as monetary rewards, punishments, or self-esteem pressure. According to Przybylski et al., intrinsic motivation has many benefits over extrinsic motivation, including more effective and creative completion of the task (Przybylski, Rigby, & Ryan, 2010).

Cognitive Evaluation Theory (CET) is a subset of SDT which looks at the effects of extrinsic rewards on intrinsic motivations, focusing mainly on how such rewards satisfy the needs for competence and autonomy. These two needs can come into conflict when rewards are given. For rewards to affect intrinsic motivation, they must occur during or as a result of engagement with the task they are meant to be motivating. Intrinsic motivation is hindered when a reward is seen as a controlling behavior because it works against satisfying the autonomy need. It does not matter whether the reward is given as a result of participation or of success. This applies to most extrinsic motivations. However, if the reward is given only when the user has displayed some feat of skill, the need for competence is satisfied and intrinsic motivation is enhanced. For maximum intrinsic motivation, skill must be recognized in a way which does not hinder the participant's feeling of autonomy. A verbal example of this from Deci is the difference between saying "I haven't been able to use most of the data I have gotten so far, but you are doing really well, and if you keep it up I'll be able to use yours," which was seen as controlling, and "Compared to most of my subjects, you are doing really well," which was seen as informational (Deci, Koestner, & Ryan, 1999). The controlling reward boosted intrinsic motivation less than the informational one. Sometimes giving a reward can lower intrinsic motivation after the task is complete because the participant reflects on his actions as being controlled by his desire for the reward. Offering a reward for a task can cause disinterest because participants are accustomed to being bribed to do uninteresting tasks.

Applied to video games, these theories provide some insight into why good video games are so effective at engaging players. Fun is a strong intrinsic motivation, and many game elements are good at satisfying the psychological needs on which intrinsic motivations rely. Balance between boredom and frustration provides ludic tension (Denis & Jouvelot, 2005). However, for non-game applications, adding game-like elements cannot effectively enhance an

application which does not already have engaging underlying interactions (Liu, Alexandrova, & Nakajima, 2011). It is important to avoid amotivating pitfalls such as a lack of interactivity.

3.0 Design Choices

Before building and implementing the system, the project team spent significant time designing and planning. During this period, a number of key decisions were made to shape the course of the project. Several matters of design were considered, and many infeasible or suboptimal ideas were discarded. This section details several of these choices, discussing ideas that the team considered and the reasoning behind the final decisions that were made.

3.1 Machine Learning Algorithm: Decision Trees

As machine learning was a central focus of this project, one important early consideration was the choice of machine learning algorithm to use in training the robot. In making this decision, several techniques were considered for handling mappings of world states to robot actions. The algorithms considered include k-nearest neighbor, Gaussian mixture models, decision trees, neural networks, hierarchical task networks, and reinforcement learning. In the end, decision trees were chosen as the primary learning algorithm, due to several key properties that they possess. Firstly, decision trees, more than the other algorithms that we considered, could easily represent their policies in a human-readable format. For any action that the robot took from its learned behavior, researchers could follow a visual representation of the tree from the root to a leaf to understand the logic behind choosing the particular action. Another important advantage of decision trees was their ability to natively handle mixtures of discrete and continuous state data, which are common in many robotics tasks. Finally, decision trees could be computed from training data relatively quickly. This was desirable, as it allowed the robot to continue to improve its learned behavior as more data continued to arrive via the web-based system. It also allowed the robot to quickly change its training data set on demand, allowing for comparisons between behaviors learned from a single user and those learned from the synthesis of all users' data. Taken together, these three properties made decision trees especially suitable for our project.

3.2 Machine Learning Library: Waffles

The next important decision to make regarding machine learning was the library to use as the starting point for this project's machine learning code. As so many general-purpose implementations of common learning algorithms were widely available, it would have been a

waste of effort to write an algorithm from scratch as a part of this project. On the other hand, it was important to select an open-source library, in case the algorithm needed to be modified to include functionality for combining data from multiple users and filtering this data based on the success with which users completed the task. To meet these needs, the following five open source machine learning libraries were investigated and considered: WEKA[2], Waffles[3], dlib ml[4], MLC++[5], and Shark[6]. Ultimately, the Waffles library was chosen, as it was the only one of the five that possessed all three of the ideal characteristics examined in making the decision. Firstly, it was written entirely in the C++ language, and was easy to access directly from other C++ code. This was useful, as C++ was also the language used for the software that controlled the basic functionality of the robot, simplifying the process of interfacing the two components. Additionally, Waffles was found to be very simple to set up and use immediately, remaining relatively lightweight and user-friendly. Finally, it supported a wide range of learning algorithms, including decision trees, the technique decided upon for the project. To ensure that Waffles would be capable of interfacing with the robot and performing the desired functionality, a test program was written to exercise some of Waffles machine learning demonstration code from a ROS node. This was accomplished very easily, and the experience indicated that Waffles would be quite easy to use with the project's existing software.

3.3 Choosing a domain

Whack-a-mole was chosen as the task for the robot to learn, but several other domains were also considered. Several ideal properties were weighed:

- The task must be easily autonomously resettable. Manually setting up each test would be very time-consuming and would make unsupervised studies impossible.
- As with any project, cost and complexity were limited.
- The task should be intrinsically fun and engaging for humans. Although it would be interesting to attempt to make an inherently boring task more engaging, that is not the primary focus of this project.

[2] http://www.cs.waikato.ac.nz/ml/weka/
[3] http://waffles.sourceforge.net/
[4] http://dlib.net/ml.html
[5] http://www.sgi.com/tech/mlc/
[6] http://image.diku.dk/shark/sphinx_pages/build/html/index.html

- The state space must be large enough that a single person cannot exhaust all of it quickly, but not so large as to be intractable. A feature space of three to ten mixed continuous and discrete features would be sufficient.
- Learning behaviors for the task should not take so long as to outlast the user's patience. Showing the user the behavior which the robot learned from his demonstrations could be engaging if it takes no more than a few seconds.
- The task must not be too susceptible to lag. Because the studies were to be done online, lag was unavoidable. An extremely time-sensitive task would suffer from this.
- The state of the task needed to be easily determinable. A domain with an uncertain state would have more noise in its state-action pairs, potentially interfering with learning. This may be interesting to study, but is outside the scope of this project.
- The theme and content of the task had to be family friendly. An overly violent or obscene task, aside from being unprofessional, would send the wrong message about robots and limit potential participants.
- Ideally, the task would have some useful application. Teaching the robot a task with real-world value would be better than having it learn some task with no use outside of a learning study.
- The lab area reserved for the robot was eight feet wide by eleven feet long, so whatever system was built needed to fit within that space.
- The robot's gripper had a very limited range of two inches, so the task needed to limit reliance upon the gripper.
- The robot's arm had five degrees of freedom, and its software did not have a perfect solution to the inverse kinematics of this arm.
- The task had to be safe for the robot, bystanders, and lab equipment.
- Allowing users to have full control over the robot could lead to unwanted damages, especially in the case of a malicious user, so the task had to allow for restricted or filtered control of the robot so that it would always behave safely.
- Finally, a task which had some benefit to using the physical robot over a simulation would take advantage of the availability of the robot.

Considering subsets of these constraints, several candidate tasks were proposed. More traditional machine learning domains such as chess and the 24-game were dismissed early because they were not interesting enough for the user and too difficult for the robot to manipulate reliably. Maze navigation was considered, but size constraints would prevent the construction of a maze large enough to be interesting for the robot to learn to navigate. A system by which different mazes could easily be configured was considered, but such a system would be more suitable for transfer learning studies than for learning from crowd-sourced demonstration studies. Furthermore, manual placement of the robot at the beginning of each session would be necessary in a maze navigation task if the same start and end positions needed to be studied multiple times, unless some clever solution were implemented. Carnival game-like tasks were then proposed. The three most promising were an offshoot of pachinko or pinball, a shooting gallery, and whack-a-mole.

In pachinko, players bet on where a ball will land after it has been launched onto an inclined plane with numerous pegs. The pachinko offshoot would have involved turning knobs and levers to adjust obstacles on the inclined play board with the goal of guiding a ball to a certain target position. When the obstacles were ready, the ball would be launched onto the top of the board. The state of the objects would affect the likelihood of paths the ball could take. There would be different goal positions that could activate at different times. The robot would have to learn optimal obstacle configurations for each goal position and timing considerations for manipulating the levers and knobs. Though this could work from a learning perspective, it would be difficult to build reliably, difficult to simulate with fidelity, and it may be difficult for the robot to manipulate the knobs. The main advantages of this task were that it would not require any manual reset, it would take up a fairly small space, and it would not be affected by lag.

The shooting gallery task would have involved one or multiple tracks lined with hinged targets. The robot would be able aim and to fire ping pong balls at the targets to knock them down. The tracks would rotate, causing the targets to move. The targets could be actuated in some way so that a controller could choose when each target should be up. The track speeds could be modulated. This would be difficult to build robustly, it might be difficult to get the robot to aim successfully, it would suffer greatly from lag, it could be difficult to ascertain the state, and it would have a lot of continuous features in the state space, making learning

potentially intractable. Furthermore, it would be putting a gun-like object under the control of a robot, which could make many people quite uncomfortable.

Whack-a-mole would place the robot in front of an array of holes, out of which fake moles would occasionally pop up. The robot would be equipped with a hammer, which it would use to whack the moles. A whacked mole would immediately retreat back into its hole. A mole not whacked would stay up for a fixed amount of time before retreating back into its hole. Whack-a-mole as a domain fit the project's constraints better than any proposed alternative. It was fairly simple, so building it was feasible. Its set of possible state spaces and sets of actions were very manageable while still being flexible enough to be interesting. It was simple and familiar enough for users that they would not have much difficulty learning to play it, but complex and nuanced enough that they would have fun playing it. It would be a little bit susceptible to lag, but less so than the shooting gallery. It was slightly violent, but it was more like cartoon violence than actual violence. There were many ways to make it more interesting if the need should arise.

3.4 Whack-a-mole design

Once whack-a-mole was chosen for the domain, the team still needed to make a number of decisions regarding the shape of the game. This section details considerations for the game's layout, web interface, game structure, and motivational features.

3.4.1 Layout

In a traditional whack-a-mole game, mole holes are laid out in a three by three square grid. However, this was not necessarily the best layout for the purposes of this project. The ideal layout would provide an interesting set of actions for the robot to perform while being easy for users to understand and interact with. The layout also had to satisfy some physical constraints. The robot's reach and the area the game could fill were limited, and the number of mole holes had to be few enough that it would be feasible to build.

For a given layout, there are two distinct types of action sets which could be used. The actions could focus more on the moles or they could focus more on the specific motions of the robot. If the focus fell on the moles, there would be one action for each mole, and the specific motions of the robot would be pre-programmed into the action, such as driving to a position within reach of the mole, moving the arm to a position above the mole, and whacking the mole.

This type of action set would be simpler for users to input, but they may not be able to easily predict how the robot will move to hit the moles they select. The other style of action set, focusing on specific robot movements, would contain actions such as driving to different positions, moving the arm to different positions, and whacking the currently targeted mole. While it is possible to have actions as granular as turning the arm by degrees, anything less discrete than a few preset positions would interfere with the learning and perplex users. This style of action set is slightly more complicated, but users would know exactly what the robot would do when they selected each action. There are various sets of actions that lie between these two styles.

Some layouts considered were the traditional three by three layout, a single long strip of moles, multiple disconnected areas with several moles each, and two rows in an isosceles trapezoid.

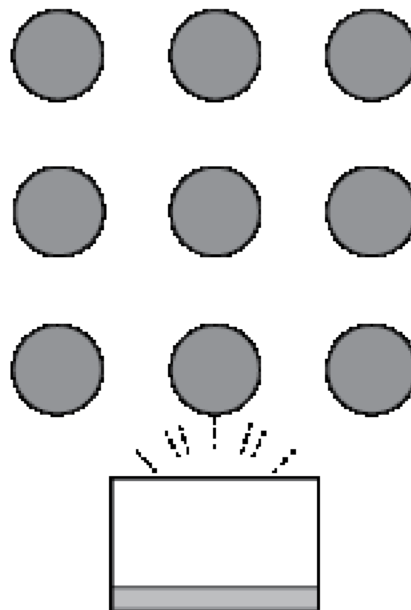


Figure 3 - Traditional 3x3 mole layout

The traditional layout, depicted in **Error! Not a valid bookmark self-reference.**, was most suited for a stationary robot whacking each mole by moving only its arm. This would allow users to click on any mole through the web interface and cause the robot to hit it without any ambiguity in the actions it would take, but the constraints of the robot's arm would have made this difficult.

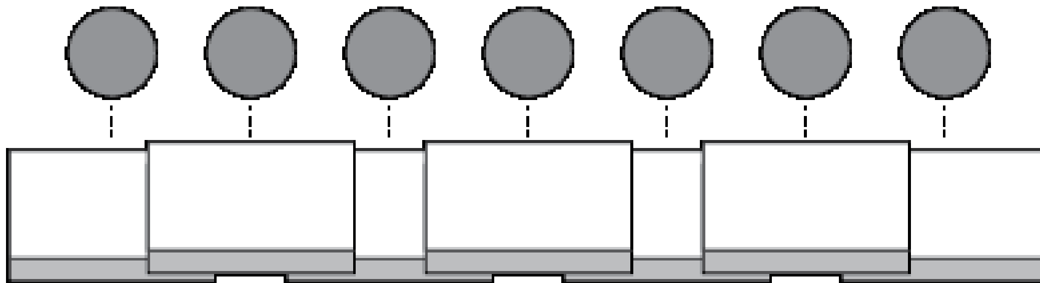


Figure 4 - Long strip of moles layout

The long strip of moles, as depicted in Figure 4, would have allowed the same simple and unambiguous mole selection, and moved the work from the robot's arm to its base. However, to achieve a large enough state space, there would need to be several moles. As the number of moles increased, the single strip of moles would quickly become larger than the physical constraints allowed.

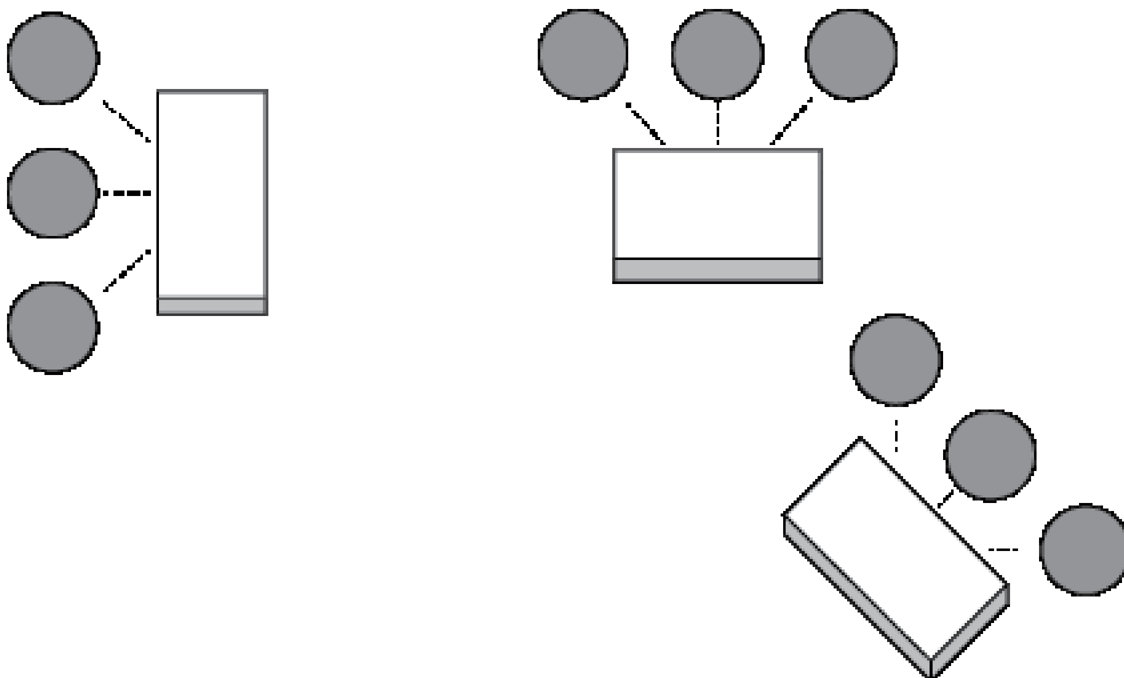


Figure 5 - Disconnected areas mole layout

Having multiple disconnected areas of moles, as seen in Figure 5, could be especially interesting if moles outside the closest area could not be seen, but this might have made the state space too complex. This may also have introduced additional difficulties in navigation.

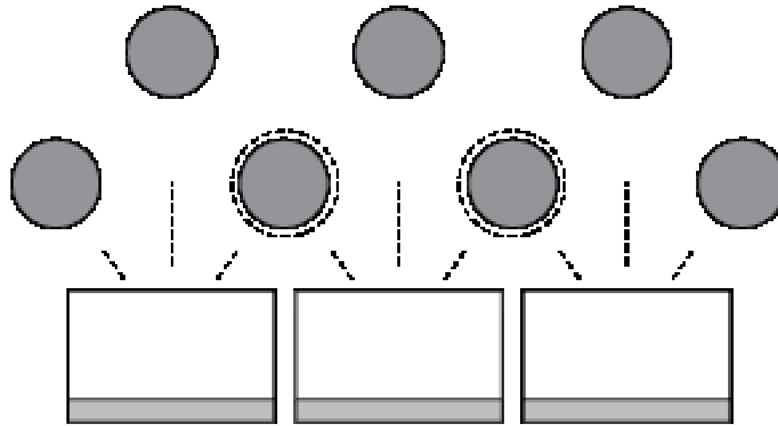


Figure 6 - Isosceles trapezoid mole layout

The layout which was chosen had seven moles in two rows, forming an isosceles trapezoid, as seen in Figure 6. There were four moles in the close row and three in the far row. This layout was especially interesting because the middle two close moles, outlined with dotted circles, could each be reached from two different robot and arm positions, from the left and from the right. For actions, a set in between the two styles was chosen. The robot could whack any of the three moles within reach or move to one of the three discrete positions.

3.4.2 Web Interface

The technology upon which the web interface would be built was decided upon from the very start. RMS was used because it had already been implemented with the robot before the start of this project. There were a number of decisions to make for the web interface, including how to display the robot and the state of the game, how users would control the robot, and how to handle multiple users in the case that more than one person wanted to control the robot at a time.

Making the interface visually appealing was left to the details and polish of the implementation, but the content of the view was considered at this time. The available options were to have one or several webcams show the robot and the moles, to display a virtual representation of the robot and moles, or both. For a game, completeness of information is important for prevention of unwanted player confusion, unless incompleteness of information

enhances or is specifically required by the mechanics of the game. This ruled out overhead cameras because the state of the moles would be difficult for players to determine. It also dismissed a view from behind the robot because the robot would occlude the view of the moles. A camera position looking down on the robot from the side was chosen because it minimized the occlusion of moles from the robot without being too high to clearly see when moles popped up. It was decided that an additional virtual display of the state of the game would be added if the webcam view proved insufficient.

The options available for the controls of the interface were to allow users to directly click on moles, to click on virtual buttons corresponding to robot actions, or to map key-presses to robot actions. Clicking on moles was simpler and more direct than clicking on isolated virtual buttons, but it proposed the added difficulty of overlaying buttons or other visuals with the proper affordances onto the video stream so users would know where they could click. This was deemed to be feasible, however, and isolated buttons were judged as too confusing, especially for new users. While keyboard controls could enable expert players to play slightly more quickly, they would be much less intuitive than mouse controls. For this reason, the input system chosen for the game consisted of using a mouse to click buttons overlaid on the moles.

This project aimed to gather data from many people, and it was possible that several people would want to participate in an online study at the same time. Because there was only one robot, time had to be divided between the users in some way. The typical way of dividing time has users sign up for a time on a schedule of available time slots. This is a clean and simple way to do it, but if someone cancels or otherwise does not attend their scheduled time, that time is lost. It would be possible to ask the user scheduled for the next time slot if he wanted to take the earlier slot, but there is no guarantee that the lost slot could be filled. An alternative would be to have a queuing system much like that of a restaurant. Users would have access to the robot in a first-come, first-served fashion. More formally, this would be a first-in-first-out queue. They could be told how many users were ahead of them in the line, or given an estimate of how much time they had left. If a person closer to the front of the line canceled, the rest of the line would simply move up without the system having lost a time slot. A third option would be to have a competition between users playing in simulation and give the next slot to the one with the highest score. For this project, the second option was chosen, and a queue was created to allow users to play on a first-come-first-served basis.

Another question related to what should be done with users who were waiting for a turn on the robot. Several options were available: the waiting users could be ignored until their time came to control the robot, at which point they would be notified; they could look at their own play statistics or the statistics of others, including a high score table; they could watch the user currently in control of the robot; they could play a simulation of whack-a-mole; or they could watch a simulated robot play whack-a-mole using behavior learned from user demonstrations. Creating the simulation of whack-a-mole was deemed outside the scope of the project, and it was decided that the best way to understand how the game works was to watch others play. For these reasons, each user waiting for a turn was able to watch the current game, with overlays on the video feed showing the estimated time until his turn. Users were later able to check their play statistics and the high score table, but not while waiting in the queue.

3.4.3 Game structure

For the structure of the game, the two main options considered were endless sessions during which the difficulty steadily increased and fixed-time sessions. Endless sessions might have been more interesting and would have facilitated a larger range of high scores, but knowing exactly how long each user will spend in each session made scheduling fixed-time sessions easier.

Traditional whack-a-mole has only one type of mole that pops up. This works acceptably for traditional whack-a-mole because it can work on very small time scales, pushing the limits of human reaction time. Because of lag and the speed of the robot, these time scales are far too small for a remote-controlled robot web interface. To keep the game interesting for users at these larger time scales, two mole types were used: good moles and evil moles. Hitting an evil mole would be rewarded and hitting a good mole would be punished. The reward would be given as points. Two options for the punishment were considered: losing points and accumulating strikes. After accumulating too many strikes, the game would end in early failure. Losing points that have been earned can be somewhat demotivating, and has a weaker effect than an early failure, but an early failure could interfere with fixed-time gameplay sessions. For this reason, losing points was chosen as the punishment for hitting a good mole. To give the punishment more weight, the number of points lost when hitting a good mole was set to be several times the number gained from hitting an evil mole. Missing an evil mole was not explicitly punished because there are many possible states in which it is impossible for the robot to hit every mole.

To further increase the fun of the game, different ways of modulating the difficulty were considered. Typically, whack-a-mole games increase difficulty by making the moles pop up and down faster as the game progresses, and making more moles pop up at a time. Adjusting the mole speeds could interfere with the robot's learned behavior, so it was ruled out. Modifying the configurations of moles presented, however, was promising. In addition to increasing the number of simultaneously-appearing moles, moles which were far apart could pop up more often, and the frequency of good mole appearances could increase. Mole configurations which had fewer recorded data could be more likely to appear, speeding the coverage of the state-space. Mole point-values could also change, but this would have been difficult to communicate to the user effectively and would have required more information in the learning's feature space. The mole configuration could go through preset sequences to convey different ideas. For example, the game could be briefly limited to bring up only one mole, but it could quickly move between different holes as a sort of boss mole. Another idea was inspired by Space Invaders, where evil moles would start appearing from the top and move their way to the side and down. Yet another idea involved moles quickly moving from one side of the mechanism to the other to indicate that they were passing by. Finally, a wave sequence was considered as a reward for players who achieved the highest score of the day or otherwise deserved something nice. These preset sequences, however, were largely ruled out because the return on time investment was estimated to be much smaller than modifying the methods of randomly popping moles up.

Increasing the difficulty at all would only serve to frustrate and confuse users if the fixed-time sessions were very short. The robot should be able to whack several moles at each perceived level of difficulty before the next appears. This provides a lower bound of at least a few minutes per fixed-time session, depending on the speed of the robot, if difficulty is to be ramped increased over the course of each session. Alternately, sessions could be shorter and players could choose difficulty at the start. This would allow them to unlock higher difficulties over time, if the range of possible difficulties allowed it. This difficulty unlocking would increase the chance for users to return to play again after stopping, if they had not yet unlocked all existing difficulty settings.

3.4.4 Motivational Aspects

There were various considerations for how to tailor the system to maximize the amount of data received. To make it as easy and enticing as possible for users to start playing the game

for the first time, account creation needed to be as easy as possible. Ideally, users would be able to play without creating an account. However, this would have made analyzing the motivational features impossible. The smallest set of required data was a username, for identifying the user, and a password, for protection. Any additional data could be set after registration.

An ideal to strive toward was for the game to be fun or interesting enough that players suggest it to their friends. If this were achieved, the number of players would grow naturally. Users could also be gathered in the usual ways: posting links to the game on various web pages and forums; sending emails to students, game writers, and friends; more traditional paid advertisement; and setting up Mechanical Turk HITs.

The system would ideally be interesting enough that users would want to play again after their first time. One step toward achieving this would be to have some longer-term arc of progression which could not be completed in a single play. This could take the form of typical extrinsic motivators such as achievements or other unlockables, a high score table, a storyline, or various information displayed to the user, such as state-space coverage or number of evil or good moles hit. Because motivation can be so difficult to predict, it was decided that several of these options would be tried and compared against each other. The largest and most time-consuming of these was to be a progression of different levels. Each new level would introduce a new game mechanic and expose more of a story involving moles and the robot. Because the constraints of a physical system were very limiting, it was decided that many of these new mechanics would be virtual.

A progression of ten levels with a corresponding pun-filled storyline was originally proposed:

Level 1: The Mole Menace
Mechanics: Standard Whack-A-Mole game
Story: None written.

Level 2: Siege of Moleville
Mechanics: Any mole-holes which would have normally been empty are filled instead with good moles.
Story: The evil moles have mounted an attack on a peaceful mole village. Stop them, but make sure not to hit the civilians!

Level 3: The Early Mole Catches the Worm
Mechanics: Virtual worms appear in the ground which can be clicked and dragged onto moles. Feeding a worm to a mole causes the mole to stay up a few seconds longer.

Story: The mole menace continues across the land, but a new discovery has been made. Moles love to feed on worms! Feeding worms to moles will keep them from popping back into the ground for longer periods of time, giving you more time to whack them.

Level 4: Molercycle Chase (moles fly past from right to left)
Mechanics: Instead of the moles appearing randomly as they normally do, they appear starting on the right side and quickly switching between holes toward left.
Story: The brilliant scientist Dr. Mole von Mole has reportedly discovered a secret weapon to aid in the fight against the evil moles. Unfortunately, an evil molercycle gang is racing toward his lab. You have to stop them and get there first!

Level 5: The Mole Doctor's Secret Weapon
Mechanics: A virtual laser mechanism can be charged up with the mouse and fired at any mole hole, instantly whacking it.
Story: Dr. von Mole has developed the latest in mole-whacking technology: The Whack-o-LASER. Pick up batteries to charge the LASER. When fully charged, it can be fired to instantaneously whack moles from long range!

Level 6: One Small Step for Mole
Mechanics: Both worms and lasers
Story: The good Doctor has been hard at work developing a cure to the spreading disease that has turned moles to evil around the world. Now he needs only to spread this antidote. He planned to launch the cure into space in his rocket ship, and scatter it down around the world. Unfortunately, there was a mole in his organization: the plans were leaked, and the evil moles are bearing down on the launch site. There is no time to launch the rocket, but you must hold off the attacking moles long enough for the doctor to escape with the cure.

Level 7: Slow Mole-tion
Mechanics: Virtual mini-moles tackle the robot. While a mini-mole is clinging to the robot, it moves more slowly. These must be shaken off with the mouse. Also, evil moles do not pop up near the robot.
Story: Many of the evil moles have been getting increasingly clever. You are tasked with taking out a particularly dangerous group of moles that have learned to avoid the robot. To make matters worse, the moles have begun to use their superior numbers to slow you down. Drag the mini moles off of the robot, or they'll drastically slow you down.

Level 8: Invader Moles from Space
Mechanics: Instead of the normal random mole configurations, moles appear in the top-left hole and move to the right, then move down and switch directions, like Space Invaders. Every mole must be whacked before it reaches the bottom-left hole. Worms and the laser are also present.
Story: Despite your best whacking efforts, the evil mole disease has spread over nearly the entire world. Dr. Mole van Mole and the remaining good mole survivors have fled to the last remaining safe place: Mole-agascar. Unfortunately, the evil moles have found a way to reach the island: in the rocket ship they stole. Stop these invader moles from space, as they parachute down to earth. Stop them from reaching the bottom left for at least 2 minutes, or the good moles are done for.

Level 9: Fall of Mole City:
Mechanics: Empty holes are filled with good moles, evil moles do not pop up near the robot, and worms, lasers, and mini-moles are all present.

Story: The attacking moles have reached the gates of the last major mole settlement: Mole City. Use every weapon in your arsenal to hold them off as long as you can. Dr. von Mole only needs a little more time to deploy the cure.

Level 10: Moles' Last Stand

Mechanics: Moles flip between good and evil at a fixed rate. The laser and mini-moles are also present.

Story: The end is near! The evil moles have pushed through the city and have surrounded Dr. Mole von Mole's new laboratory. You can't stop them all alone. But it's not over! Your heroic actions have inspired the good moles to fight back! This is where robot and mole make their last stand! As the good and evil moles are locked in combat, you must tip the scales and save the mole world!

It was determined that ten levels would be too many. If each level were two minutes long, that would be asking players to spend more than twenty minutes on the game. Depending on how long the queue waits were, that could be as long as an hour to beat the game. Even that assumes the player beats each level on their first try. This was unreasonable, especially for a game on the web, where attention spans are very short. Furthermore, this progression of levels had a somewhat sparse introduction of new mechanics, which risks losing players' interest. To shorten the levels and condense the introduction of new mechanics, several levels were merged together, and a few were cut.

Levels four and eight relied on special sequences of moles rather than random configurations. This did not seem to fit well with the rest of the game, and would have been more difficult to implement than the changes in random mole configurations, so those two levels were cut.

It was noticed that the mechanics of levels two and three were very good for each other. That is, the increased frequency of good moles would encourage players to use the worms more strategically. Therefore, those two levels were combined.

Level seven seemed to be fairly good as it was because its two mechanics complemented each other in the same way that those of levels two and three had. However, there was little point in introducing worms in one level just to take them away the next level, so worms were added to levels seven and ten. This was expected to make those levels more strategic and give players more options for play style. For the same reason, mini-moles were added to level six. Level five was cut because introducing the laser without any other mechanics did not seem very interesting.

The moles avoiding the robot on level nine was deemed excessive. Too many other mechanics were present in that level already, so the mole behavior was returned to normal in that level.

The mechanic of the laser would have been too powerful and would have required too little thought from the player. A few options were considered, such as limiting the number of uses or only allowing the laser as a rare power-up, but these did little to fix the problem. Instead, the behavior of the laser when fired was changed to actually increase its power, but also to increase its situationality. Instead of being able to fire at any individual mole, the laser would fire at an entire row of moles. Because friendly moles were worth negative three points, this meant that firing a laser at a row would only benefit the player if that row had no friendly moles in it. This restriction severely limited the laser, but a strategic use of worms could circumvent that restriction by keeping evil moles up until all friendly moles went down, allowing for very high scores.

4.0 Methodology

This interdisciplinary project involved work in several different areas. This section describes each of the major tasks that were completed as part of the project. Topics include hardware design and construction, robot control software, web development, machine learning implementation, and game design.

4.1 System Overview

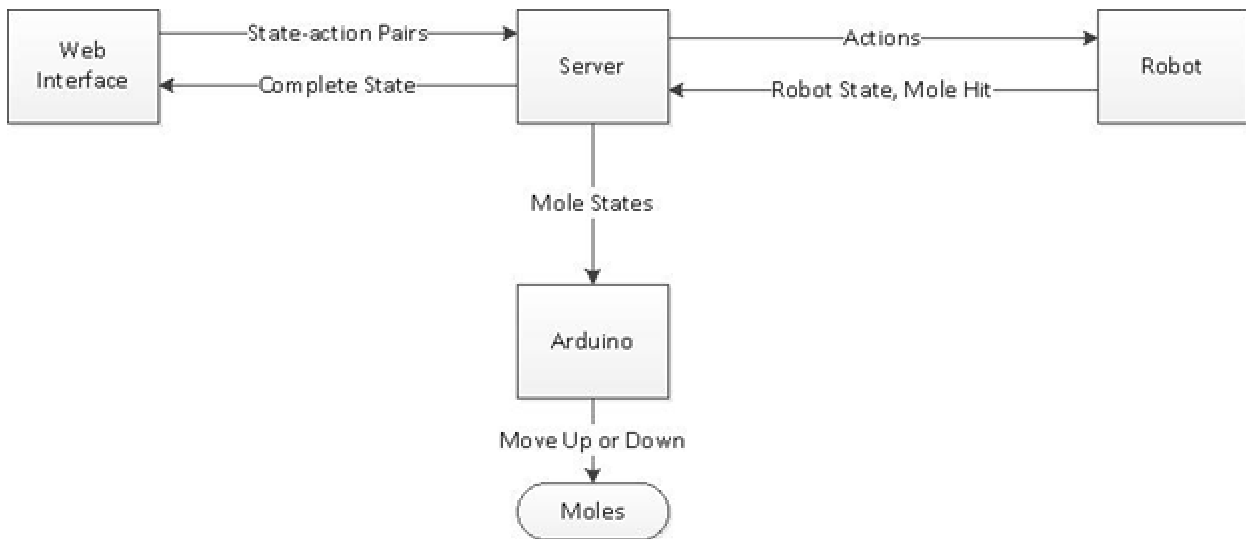


Figure 7: Block diagram of the complete system

Figure 7, above, shows a block diagram of the overall system created for this project, which consists of four primary components. The first component, the web interface, is responsible for communicating with users. Through the web interface, users may control the robot in playing the game of whack-a-mole, generating action commands when various buttons on the interface are pressed. The web interface is also responsible for displaying a camera feed of the robot and the whack-a-mole setup to show users the state of the game as they play, and for displaying game statistics and other information. The second component of the system, the server, is the piece that ties the entire system together and controls the main flow of logic through the system. The server is responsible for passing along commands to and from the web interface, robot, and moles. Additionally, the server contains all of the logic that runs a game of whack-a-mole, and it also handles all of the learning algorithms and management of state-action pairs. The third component, the robot, is responsible for moving back and forth in front of the whack-a-mole setup, rotating the arm to the appropriate orientations, and whacking the moles.

The robot selects which actions to execute by querying the server, which determines the action to perform from either the web interface or the learning algorithm, depending on whether the robot is playing autonomously or not. The final portion of the system, the moles, consists of seven mole mechanisms, each with a friendly mole, an evil mole, and a servo. The seven servos are controlled by an Arduino microcontroller, which receives commands from the server indicating when to raise and lower each mole.

4.2 Building the Mole Mechanisms

The physical whack-a-mole setup created for this project is made up of seven self-contained mole mechanisms laid out in the shape of an isosceles trapezoid. Each mechanism is associated with a particular mole hole and is responsible for controlling the evil mole and the friendly mole that pop up in that hole. Several iterations of design, prototyping, and testing were involved in the creation of the final mole mechanisms.

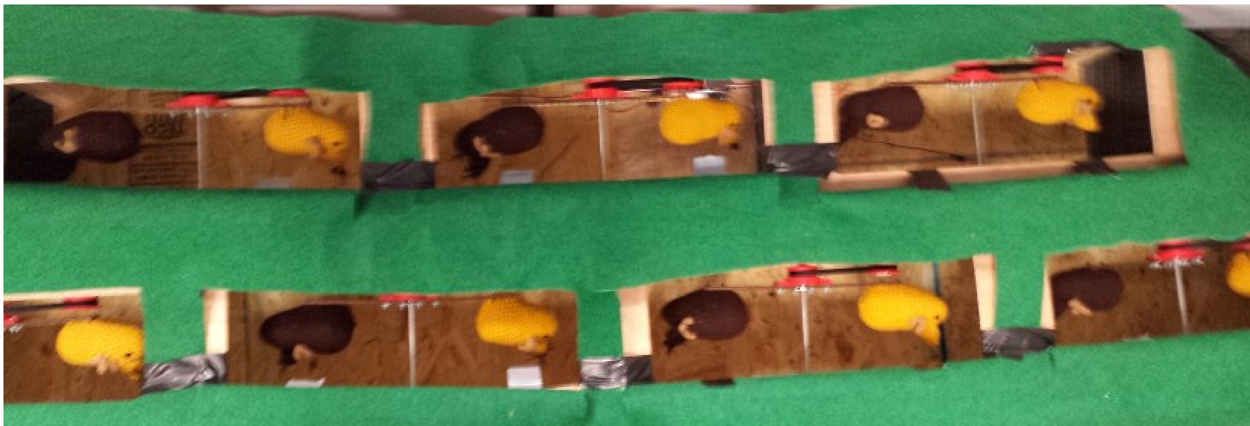


Figure 8: Full Whack-A-Mole setup

The first step in designing the mole mechanisms was to determine the method by which the moles would be actuated. In traditional whack-a-mole type games, moles typically move linearly up and down in circular holes. Unfortunately, linear actuators such as solenoids were found to provide less range for much greater costs than rotational actuators such as servomotors. More importantly, linear actuation systems do not lend themselves as well to this project's version of whack-a-mole, due to the need for two different moles to occupy each hole. Though design ideas were proposed that would allow for this behavior, they were all deemed too large, too costly, and less robust. Instead, it was determined that a motor should be used to actuate each pair of moles. This allows each mechanism to easily spin up either type of mole by rotating in the correct direction. Figure 9 shows an early sketch illustrating this concept.

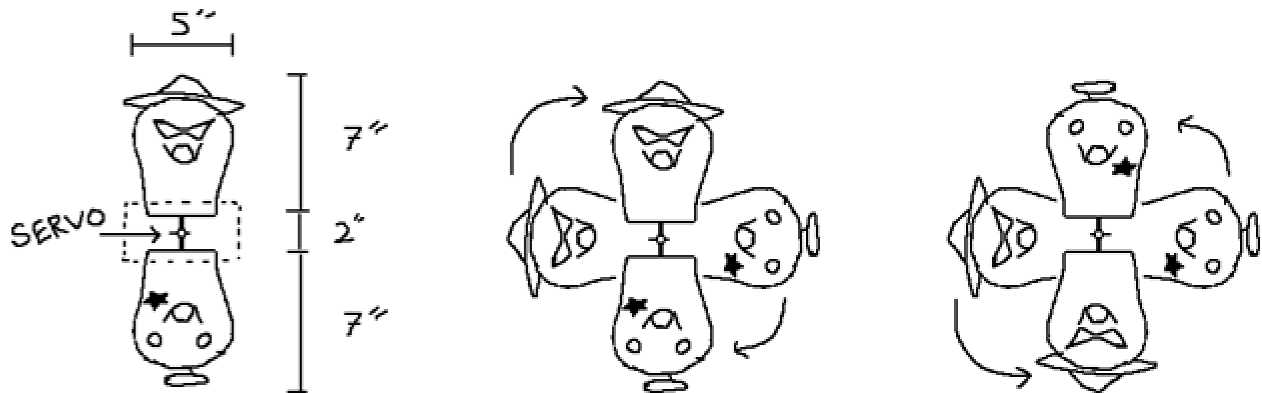


Figure 9: Illustration of mole mechanism behavior

Once the general concept for a mole mechanism was decided upon, a proof-of-concept prototype was put together to show that rotating moles into position was both feasible on a reasonable budget and capable of creating the necessary range of motion in limited space. After this test proved successful, a more detailed prototype was created. The moles were mounted on wooden mounting bars rotating about an axle. Though the plan was for the robot to only gently whack the moles, there was a concern that it could whack too forcefully if out of alignment. To mitigate risk of damage to the servo in this situation, the servo was mounted to the side of the axle and connected to it with a pair of pulleys and a belt. This initial prototype, shown in Figure 10, was designed to allow for either a rubber band or a string to be used for the belt. The entire mechanism was mounted in a wooden frame consisting of a plywood base and two 2x4s to hold the axle. The pulleys were created using a 3d printer and contained holes for mounting directly to the mole bars and servo.

Figure 10: Initial mole mechanism prototype

Once completed, this prototype was tested for robustness to determine whether it could run successfully for the long periods of time that the whack-a-mole game required. This test was conducted by programming an Arduino microcontroller to spin the servo back and forth repeatedly. The mechanism was left running for an entire night in front of a webcam, which automatically took a picture of the mechanism in the same position every minute. These pictures were used to measure the change in angle of the popped-up moles over time to determine whether they had become misaligned. When testing this first prototype, it was found that most of the mechanism worked reliably, but that neither the rubber band nor the string could function as an effective belt for long periods of time. The rubber band would lose traction on the pulleys over time, allowing the moles to slip. The string worked well at first, because it remained tied to the pulleys, but it would come loose over time and was tedious to fix.

The issues with the prototype that this test demonstrated were used in the creation of further prototypes. Due to the limitations of the rubber band and the string, the pulleys were redesigned to use a timing belt. The team went through several iterations of these pulleys to get all of their alignments with the other components correct.

The final prototype used acrylic servo mounting plates and wooden mole mounting bars, both cut with a laser cutter, in addition to the 3d-printed pulleys. To increase stability, 4

additional wooden legs were added to each mechanism's frame, which contained a thick plywood base and a thinner wooden cover with a slot cut out for the moles to pass through. A photograph of this mechanism can be seen in Figure 11, and the CAD models used to create some of its components are shown in Figure 12.

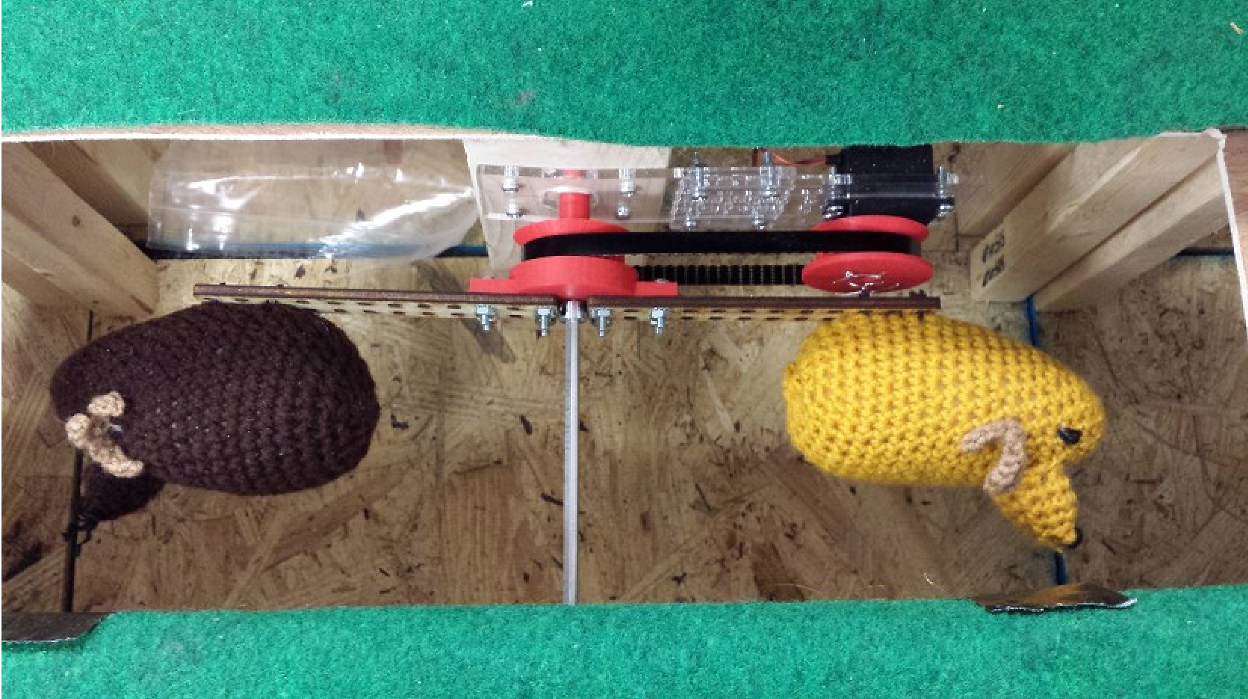


Figure 11: A finished mole mechanism

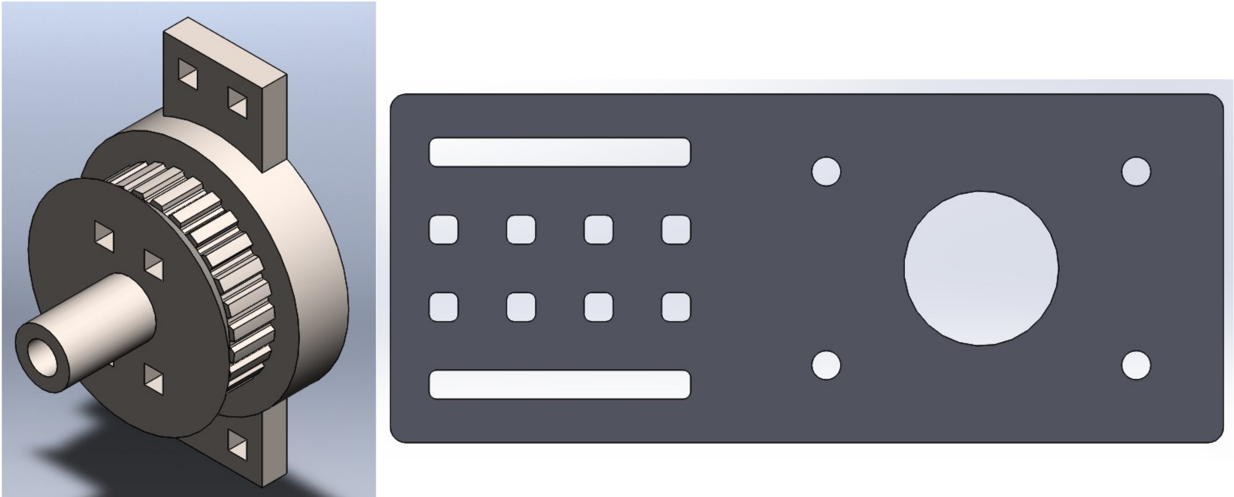


Figure 12: CAD models of select final prototype components

The new prototype underwent the same testing as its predecessor, but was found to work much more robustly. The timing belt eliminated almost all of the slop in the system, and was also

very easy to adjust. Once the prototype had passed the testing, six more identical mechanisms were created. These seven mechanisms were placed adjacent to each other to form the final mole setup. A green carpet, with holes cut in it, was later draped over the mechanisms to cover up their inner workings and to simulate a grassy field for the moles.

Another aspect of the system that needed to be decided upon at this stage was the moles themselves. Figure 13, below, shows comparison photos of each of the mole prototypes that were tested. The first prototype mole was created out of cardboard, and covered with felt. This mole worked reasonably well, but took a long time to create and appeared rather unfinished. The second prototype consisted of commercially available stuffed moles. Unfortunately, the only stuffed moles that could be obtained in sufficient quantities were too large and heavy to be easily lifted by the servos, and arguably too cute to whack. They also would have required significant modification to distinguish friendly moles from evil moles. For the third iteration, another student offered to crochet custom moles. She made these moles in two different colors, allowing the good and evil moles to be easily distinguished. These moles also had the benefit of being light and easy to affix to the mounting bars. Due to these advantages, these moles were selected for the final design.

Figure 13: Comparison of mole prototypes

The servos from all of the completed mole mechanisms were connected to a single Arduino RedBoard microcontroller through a servo shield. This shield, with its external power

supply, allowed the Arduino to drive all seven servos simultaneously. The Arduino was then programmed to control these motors based on inputs sent to it via USB serial communication. A ROS node was written to run on the server and handle communication with the Arduino to update the states of the moles as determined by the whack-a-mole game logic.

4.3 Robot Control Software

With the mole mechanisms complete, it was time to program the robot's interactions with the moles. The robot needed to be able to drive forward and back to different positions and to move the arm in such a way that it appeared to whack the moles. This behavior needed to be built upon some version of the ROS drivers for the youBot, as those were provided the most convenient and easily available method of controlling the robot. The drivers version selected was the ROS Groovy version which the RAIL lab had developed, as that was the most recent version at the time this project was started. Also, a member of the project team had helped to develop it, so it was more familiar than other versions.

To drive back and forth, a ROS node was written which sent command velocities to the robot's base. Although the youBot had odometry which could have been used to dead-reckon its position, using it would have been too unstable over long periods of time. Instead, a Hokuyo laser range-finder was used to provide feedback and close the control loop for position,as seen in Figure 14. The laser had a range of several meters over 180 degrees, and was mounted to the front of the robot's base. A wooden board was placed at the far side of the mole mechanisms, allowing the laser to see an obstacle with a known location. By finding the distance between the robot and this wall, the robot was able to know its horizontal position relative to the mole mechanisms, allowing it to drive to the desired positions accurately and repeatably.

However, the robot did not always stay at the same vertical distance from the mole mechanisms, and it occasionally introduced some rotational error. To fix both of these, a wall was constructed in the front of the mole mechanisms to allow the laser to see the nearest border of the mechanisms. The first attempt to create this wall involved draping a white cloth sheet over the mole mechanisms. However, the sheet was not opaque enough to consistently reflect the laser. Next, several long sheets of wood were taped together. This had the advantage of being foldable and fully obstructed the laser, but the black duct tape used to hold the wood together at the joints was too shiny and caused problems with the laser. To fix this, a less reflective masking tape applied on top of the duct tape. Having this wall allowed the robot to find its vertical

position relative to the mole mechanisms. The laser simply found the distance to the wall. To correct angular error, all of the visible measurements along the wall were used to calculate the angle of the wall with respect to the robot. This worked, but had the problem that as the robot came closer to the far wall, fewer points along the other wall would be visible for calculating the angle. This made it so that the rightmost position occasionally took more time to correct the angle than the other positions. The far wall could have been used to calculate angle when there was not enough visible space on the other wall, but it was not wide enough to be useful for calculating angle. Because of the omni-wheels of the youBot, the robot was able to turn in place or while driving. This allowed constant correction of the robot's complete position and rotation, removing any chance of error accumulating over time.

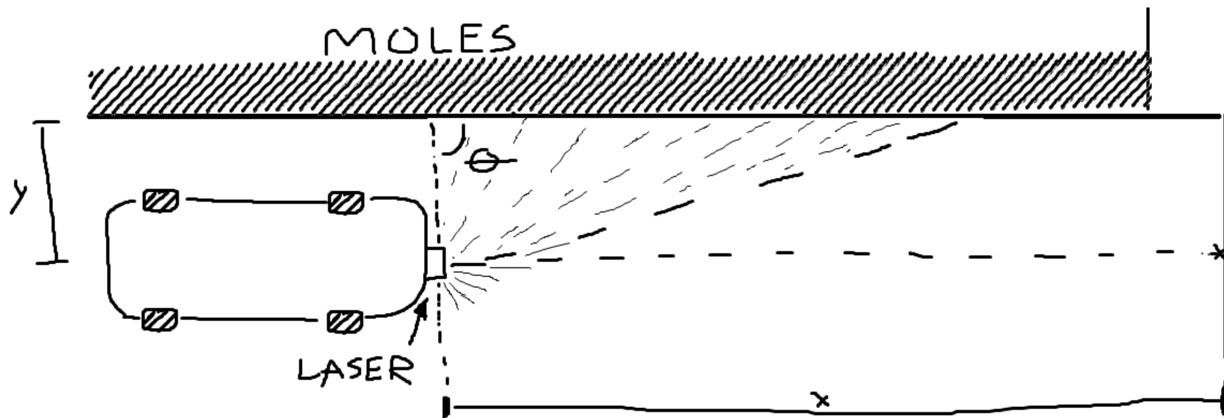


Figure 14: Diagram of robot position control with laser scanner

To move the arm, the drivers required a message describing the desired position for each joint. The drivers ran PID control on each joint separately, eventually causing the arm to reach the desired configuration. No self-collision avoidance was done by the drivers, so care had to be taken when choosing desired arm positions. Specifically, the arm had a home position which it moved to whenever the robot started. Positions had to be chosen such that the arm would not collide with the base at any point along the path from the home position to each other position. To make the arm appear to whack the moles, it was estimated that two positions for each mole was enough. To achieve a whack motion, the arm was held up in a ready-to-whack position, then it moved to a position holding the hammer just above the mole, then it returned to the ready-to-whack position. Ideally, the speed of the motion would increase until stopping abruptly just above the mole to achieve a more convincing whack motion, but this would have been unsafe, and stressful for the arm motors. To find reasonable positions, a keyboard teleoperation node

from the drivers was used which allowed jogging of individual joints. The robot was placed in position relative to the moles and the arm was moved manually to each necessary position. The positions were judged based on how they appeared in the camera feed. Each arm configuration was recorded and entered into the code for a new node which handled the whack-a-mole arm positions. This arm contained a small finite state machine which, upon receiving a whack command, moved the arm to the corresponding ready-to-whack position first, then down to the whack position, then back up to the ready position. By moving to the ready position first, it prevented any strange motion which could have occurred if the arm tried to whack a mole with which it was not already properly aligned.

To make these two position nodes handle command messages, they were first combined into one robot control node, then callbacks for robot position and mole whack command messages were added. These callbacks simply changed the desired position of the robot or the state of the arm. This robot control node had a loop which determined whether the current state of the robot was close enough to the desired positions, using thresholds that were tuned to provide a reasonable balance between response time and stability. When an action finished, a message saying so was published. For safety, a boundary was set so that if the robot strayed too far from the acceptable positions, all motion stopped. This happened if the robot had been moved while it was off, if it ran over an unknown obstacle and was knocked off course, or if an obstacle obstructed the laser in such a way as to irreparably confuse the robot's knowledge of its position.

At one point about midway through the project, a hardware problem with the robot arm arose. It was beyond the team's abilities to fix, so the arm was sent back to KUKA for repair. It was determined that a very basic simulation of the robot would allow development to continue while the arm was absent. The positions node was duplicated and all references to the robot were stripped out. Every place in the code which waited for the robot to reach its desired position was replaced with a timer with a fixed delay. The delays used were estimates until the robot returned to full functionality, at which point each delay was set to match the actual robot's action durations as closely as possible. This simulation sped up development significantly even after the robot returned to full functionality, and provided much ease of mind to the team when the system was publicized.

4.4 Server Setup

The server needed to allow ROS to communicate with clients remotely, handling connections and user accounts. To accomplish this, we used the Robot Management System. Installation involved setting the server up as a LAMP (Linux, Apache, MySQL, PHP) server as described by the RMS installation tutorial. RMS allowed communication between clients and any ROS nodes running on the server by using rosbridge to relay ROS topics and services to and from each client through websockets. ROS nodes ran normally on the server and were able to treat any messages from the client as if they had been published by another ROS node running on the server. Clients connected to the server using a standard web browser. Pages were written in PHP and HTML. Pages which required communication with ROS achieved it using a JavaScript library called roslibjs, which had equivalents for ROS topics, messages, and services.

A vital tool during development and maintenance was PHPMyAdmin, which allowed easy communication with the database. This allowed for experimentation with MySQL queries and easy visualization of database structure and contents.

4.5 Basic Whack-a-mole Game

This section details the implementation of a playable whack-a-mole game. It covers the central logic controlling the game, the web-based robot control, and key website features. Robot learning and additional game features are discussed in later sections.

4.5.1 Core Game Logic

All of the core game logic for the whack-a-mole game was handled by a single ROS node, the whack-a-mole game node. This node was responsible for determining the states of the moles throughout the game and for keeping track of game information such as score and time remaining. It also communicated directly with the web interface and the robot control software.

The primary task of the game node was to manage the state of each game of whack-a-mole. When it received a start game message from the web interface, it first ensured that the robot was initialized to its starting position. Once the game node received confirmation of this from the robot, it would begin the game and start the time remaining countdown. The time remaining in the game was published every second, so that it could be displayed to the user on the interface, and when it reached zero, the game would end. The game node also tracked and published the player’s score, so that this too could be displayed.

While a game was in progress, the game node kept track of the state of each of the seven mole holes. These states represented both the time for which each mole had been up and whether the mole was good or evil. This information was all captured in an array of seven values. The sign of each value kept track of whether a mole was good or evil, while the magnitude kept track of the length of time for which the mole had been up. A zero for a particular mole hole indicated that the hole was empty, because neither mole was up. The game node published these state values at regular intervals, as they were used for generating state-action pairs for learning. When a mole popped up or went back down, it also published the states for the Arduino control node, which updated the position of whichever mole mechanism had changed.

When creating the whack-a-mole game, it was decided that the moles should come up in a random fashion, leading to a different game each time. This made for a much better robot learning problem, as there was no set solution to the game, and it increased the replay value for human players. During a game, the game node is responsible for randomly determining whether each mole pops up in a given iteration of the game loop. Each mole that is not already up has a set probability of popping up, with a chance of appearing as either a good mole or an evil mole. These probabilities are easily modified to allow tuning of the flow of the game. If not whacked by the robot, each mole will then stay up for a fixed amount of time before going back down.

One problem was occasionally encountered with this system, in which a mole would appear not to go down, or would immediately switch types of mole, because the mole would come back up immediately after going down. This issue was fixed by adding a minimum cool-down time for which a mole needs to remain down. Another issue that came with randomly selecting moles to bring up was the possibility that no evil moles could come up for a significant length of time. This would leave the player with nothing to whack, and could lead to boredom and frustration. This problem was rectified by adding functionality to ensure that at least one evil mole is up at any given time. If there are no evil moles up at any point in the game, the game node randomly selects one eligible mole to bring up as an evil mole.

The game node was also responsible for updating the mole states in response to whacks from the robot. The robot control node never directly accessed the states of the moles, but it did keep track of its own position. When given a command to whack a particular mole hole, it published a whack impact message upon reaching the low point in its whacking arc, regardless of whether the hole that it whacked actually contained a visible mole. The game node listened for

this message and checked its record of the mole states to determine whether a mole was hit. If so, it updated the mole states accordingly, causing the Arduino control node to bring down the mole. This gave the appearance that the moles duck back under ground when hit by the robot's hammer. Depending on the type of mole that was hit, the game node either increased or decreased the player's score. Whacking an evil mole granted a player one point, while hitting a good mole lost him three points. These values, along with those which controlled when and how frequently moles popped up, were function parameters that could be easily modified if necessary.

4.5.2 Web Control

For users to be able to play the game, they needed to understand the current game state. This was accomplished through a video feed froma USB webcam which was connected to the server. The webcam was connected to ROS using the usb_cam_node, which handled communication with the camera's drivers and did some minimal calibration. The video was streamed to users with the MJPEG stream package which came with RMS. In effect, this served individual JPEG images to clients in sequence. Unfortunately, for unknown reasons, this only worked correctly in the Chrome browser. Streaming services such as Twitch, Ustream, and Livestream were considered, but Linux support was weak for these services and integrating them with ROS would have been a major project. Because the mjpeg stream only worked in Chrome, a check was added to all PHP pages which redirected to a page informing users not in Chrome that Chrome was the only working browser for the system. Being limited to Chrome did, however, make writing the pages easier, because compatibility with other browsers was not an issue.

While the system was first being set up, it was necessary to test how well it handled lag. To measure the latency of the ROS communication, a service was created through which the client occasionally requested the current time from the server. The client recorded the current client-side time when it sent the request and again when it received the response. The difference between these times was the round-trip time. A relatively consistent round-trip time of 4-16ms was common for those on the wired network at WPI. The WPI wireless was a bit worse and less consistent at 10-50ms. A person from the U.K. reported a round-trip time of between 200 and 400ms. These delays were just for raw data, however, and not for the video stream. Because this method of measuring latency actually caused a fair amount of overhead with ROS, it was disabled before the system was publicized.

The user interface for the game was written mostly in JavaScript with some HTML and CSS3 for structure and formatting. The robot position and mole whacking buttons needed the following images: Normal; Hover; Disabled; Pressed; and Active. While the buttons were enabled, the image would switch between Normal, Hover, and Pressed depending on the mouse position and button state. When the buttons were disabled, they would normally show the Disabled image, but the button corresponding to the robot's current action would show the Active image. Whenever a button was pressed, the corresponding command was sent to the robot, and all command buttons were disabled until the robot sent back a message claiming the action had been completed. To accomplish these image switches, the various mouse events were used. OnMouseEnter and OnMouseExit switched between Normal and Hover. OnMouseMove was used as a global update function so that buttons which had been disabled would update their appearance. To determine whether a button was pressed, the mouse button state needed to be obtained. However, there was no way to poll for the mouse button state, so global OnMouseDown and OnMouseUp functions kept track of a the global mouse button state. Once these image switches were implemented, it was quickly noticed that images took time to load when they were switched to for the first time. To fix this, each image was preloaded when the page finished loading. Separate images were drawn for the robot position and mole whacking buttons. Figure 15, below, shows the robot position buttons on the left and the mole whacking buttons on the right. In each group, the buttons appear in the following order: Normal, Active, Disabled, Hover, and Pressed. Because the buttons would be overlaid atop the video feed, they needed to be either transparent or very small to prevent obscuring the game state. Very small buttons would have been difficult to click, and having hit-boxes larger than the buttons would have been confusing to users, so large, transparent buttons were drawn.

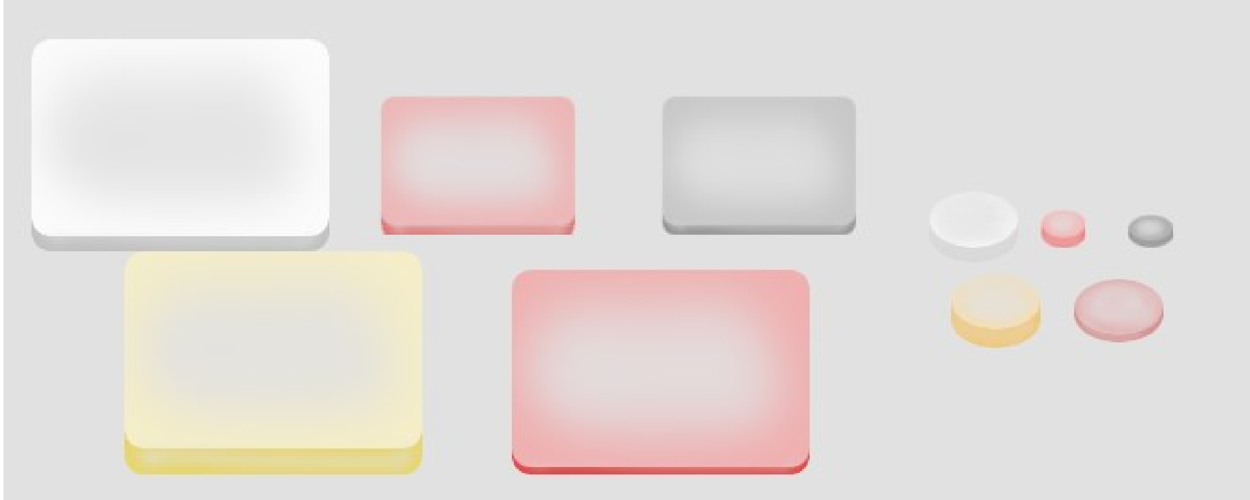


Figure 15: Robot position and mole whacking buttons

When the simulated robot was running, it did not appear in the video feed. To visualize it, a box and line representation of the robot, as seen in Figure 16, below, was drawn onto a JavaScript canvas in the interface, directly on top of the video feed.

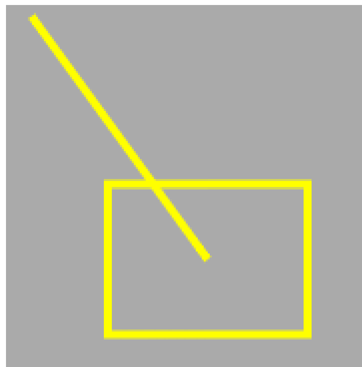


Figure 16: Simulated robot visualization

To get the state of the simulated robot, the fake robot positions node published the current arm and base positions. However, this looked very choppy when there was any sort of lag, so client-side interpolation was used. The fake robot positions node published the time since each action started and the remaining time until the action finished, as well as the current position. Time differences were used instead of exact times so that the interface would not be confused if it were in a different time zone or if lag had caused significant desynchronization. When the interface received an update, it set the robot visualization to the exact position in the update and stored the two time differences. For each frame until the next update, the interface updated the robot position using linear interpolation between the start and end position, based on the time that had passed since the update. This made the motion appear much smoother in laggy situations

than it did without interpolation, but the linear interpolation did not match the acceleration curves of the real robot. To make it look a bit closer, the smoothstep function was applied to the interpolation. Altogether, the interpolation worked well in most situations, but it had occasional hiccups where the robot visualization would jump when it received an update that did not match its expectations.

In addition to the video feed and buttons, a representation of the main variables of the game state was necessary. Divs were added to the top of the video feed with simple text for the score and remaining time. When the score changed, the game node published a message with the new score. When the interface received it, a callback function was called. This function updated the text of the score div. A similar function was called for game time updates. However, changing the text was not an obvious enough form of user feedback. To enhance it, the score div was made to change color when the score changed. When the score went up, it flashed yellow, the same color as the evil moles. When the score went down, it flashed red, to show that it was bad. This was achieved using CSS color transitions. The default behavior of a CSS transition is to apply the transition duration equally to any change, but it was desired that the color would change instantly and then slowly return to its original value. A function which accomplished this was found in Mozilla's X-Tag library and used.

The basic progression of a game was as follows: The user would enter the game interface from elsewhere and see a start game button. The user would press start game, causing the interface buttons to appear and a message to be sent to the server indicating that the game should start. The user would play the game by pressing the buttons until the time ran out, at which point all the buttons would disappear and the user would be redirected to an end-game page. This was accomplished with a state machine with three states: waiting-to-start, in-game, and game over. The game would start in the waiting-to-start state, but if the interface received a message which indicated that a game was currently in session, the interface would switch to the in-game state. This allowed users to refresh the page mid-game without worrying about interface problems. The waiting-to-start state would also transition to the in-game state if the user pressed the start game button. The in-game state would switch to the end-game state if the game time hit zero. When in the end-game state, the interface set some POST variables describing the game and score, and redirected the user to an end-game page. The end-game page told the user what they just scored.

4.5.3 User Account Creation

The RMS framework that was used for this project was initially intended for controlled, pre-scheduled user studies, in which potential users would sign up to participate in experiments at set times. System administrators would create accounts for these users, filling in all of the applicable user information. They would also manually schedule experiments for each user, setting the times in which the user account could have access to the robot interfaces. One of the goals of this project was to make LfD studies more accessible to potential demonstrators without requiring as much advanced planning by either the demonstrators or the conductors of the studies. To accomplish this goal, functionality was needed that would allow users to dynamically create accounts for themselves at any time.

The first step in developing this feature entailed the creation of a new page on the site that could create a new user account for the current user. The initial version of this page was rather simple, and did not include any user interaction. Rather, it implemented the necessary RMS system calls to create a new user with randomly assigned user information. It then logged in the user with that account before redirecting to the main page.

While this page allowed users access to the site, the randomly-assigned username and password meant that each account was one-use-only, and a user would have to create a new account each time he returned to the site. This was not desirable, because it does not allow for tracking of statistics for individual users who may visit the site multiple times. A better solution would allow each user to supply a chosen username and password for her account, allowing her to login again later with the same information. Thus, a signup page was created, with fields for username and password that match those on the login page. RMS supports storing of first name, last name, and email address for each user account, but this information was unnecessary for this study, so these fields were omitted from the form to reduce the hassle of account creation and allow for greater anonymity. It was hoped that this would encourage more users to take the time to create accounts. As the LfD study that was conducted could be described as an experiment with human subjects, the research study was cleared with the WPI Institutional Review Board, and a link to an informed consent document was included on the signup sheet. This document outlined the purpose of and procedures used in the study, as well as the lack of risk to the personal safety of study participants. Users were required to check a box indicating their consent before they could create their new accounts.

To facilitate robot learning, a second account was created for each user corresponding to the autonomous robot that would be trained with that user's data. This account was given the same username, but with the word "robot_" appended onto the front. The user was not given access to this account's password and thus could not directly log in as the robot. When the robot was run autonomously using data from only this user, though, the user ID for the game would be recorded as that of this robot user, so that its scores could be compared with those of its human teacher.

4.5.4 Queue

With multiple users and only one robot, control over the robot had to somehow be assigned to users fairly. As had been decided earlier, a first-in-first-out queue system would be used. This was implemented as a ROS node which stored and published a double-ended queue of user IDs. Before users were allowed to control the robot, they would be shown the queue page, which contained a video feed of the robot and a button with the text "Play Soon". When a user hit the play soon button, they were added to the queue. An important aspect of the queue was that it allowed those waiting in the queue to watch people play before they did, giving them a chance to learn how to play. When a user reached the front of the queue, they were redirected to the game interface as it has been described above. If a user failed to hit the start game button within thirty seconds of reaching the main game interface, they forfeited that game and were sent back to the end of the queue. This prevented users from maintaining an endless monopoly on the robot. When the user ended their game, they were taken out of the queue, allowing the next person to play. Because games were two minutes long and the maximum wait between the end of one game and the start of the next was thirty seconds, the time until reaching the front of the queue could be estimated from the number of users ahead of a user in the queue. Because the entire queue was published to each user, each user knew their location in the queue, so an estimated time was shown to those in the queue. The thirty second wait time was averaged down to 15 seconds, because in practice very few users waited more than a few seconds to start a game.

To avoid rewriting the code for the HUD elements, the queue functionality was added to the same JavaScript file that was used for the game interface. Three new states were added: not-in-the-queue, waiting-in-the-queue, and at-the-front-of-the-queue. Upon loading the queue page, the

state was set to not-in-the-queue. If the interface received a queue message stating that the current user was in the queue, it would switch to the waiting-in-the-queue state. This could happen if the user had just pressed the play soon button or if the user had reloaded the queue page after entering the queue. As soon as the user's id was the first in the queue, the state was set to at-the-front-of-the-queue. In the state, the user was immediately redirected to the game interface.

4.6 Learning from Demonstration

One of the primary goals of this project was to develop an engaging Robot Learning from Demonstration study on the web. Creation of this study required a number of steps to support the application of machine learning to the chosen domain. Demonstration data had to be collected and recorded as the system was used. A machine learning algorithm implementation was needed to allow the robot to learn a behavior from the data. Functionality was required to allow users to remotely begin the learning process with specific subsets of this data. Finally, the effectiveness of the learning process needed to be evaluated at the conclusion of the study.

4.6.1 Collection of Demonstration Data

In Robot Learning from Demonstration, a robot learns how to autonomously complete a task based on training data provided by human demonstrators. Thus the first step in creating an LfD study entails developing a means of recording this training data. For this project, the task consisted of playing a game of whack-a-mole, and the training data was made up of state-action pairs corresponding to each move or whack of the robot. Each time a user clicked a button to either move the robot to a new position or to whack one of the moles, a state-action pair was generated. The action recorded in each pair was a numerical value corresponding to which of the six possible whack or move actions was selected by the user. The state for each pair included the seven mole states at the time the action was chosen, as well as the position of the robot and the orientation of its arm.

This information was recorded in a database table with each row corresponding to a single state-action pair. Every time a user clicked on a button on the interface, a JavaScript function was called to log the corresponding state-action information. The state data used in this function was the most recent data obtained from the game node. Due to security concerns, the client-side JavaScript could not have direct access to the database. Instead, the client-side

logging function called an AJAX function, which was responsible for passing the state and action information to the PHP running on the server. In the PHP, the appropriate queries were made to insert the new state-action pair rows into the database.

4.6.2 Learning Algorithm Implementation

Once it was possible to collect training data from users, the next step was to implement an algorithm with which the robot could learn from this data. This was accomplished through the creation of the Whack-A-Mole learning node, a ROS node responsible for training the robot and for executing its learned behavior. For training the robot to play whack-a-mole, this node implemented the decision tree algorithm provided by the Waffles machine learning library. This algorithm took in a specified set of state-action pairs and used it to produce a decision tree, which the robot could follow to select each action based on the current game state. The algorithm also printed this tree to a text file, allowing the tree to be examined and analyzed. An example of part of one of these printed trees can be seen in Figure 17, below. The learning node was set up to accept messages specifying the parameters by which to filter the set of state-action pairs in the database. These parameters included user, game version, and game mode. Functionality was also added to support filtering of the training data by score, to analyze whether the robot performed better when trained only with data from high-scoring games.

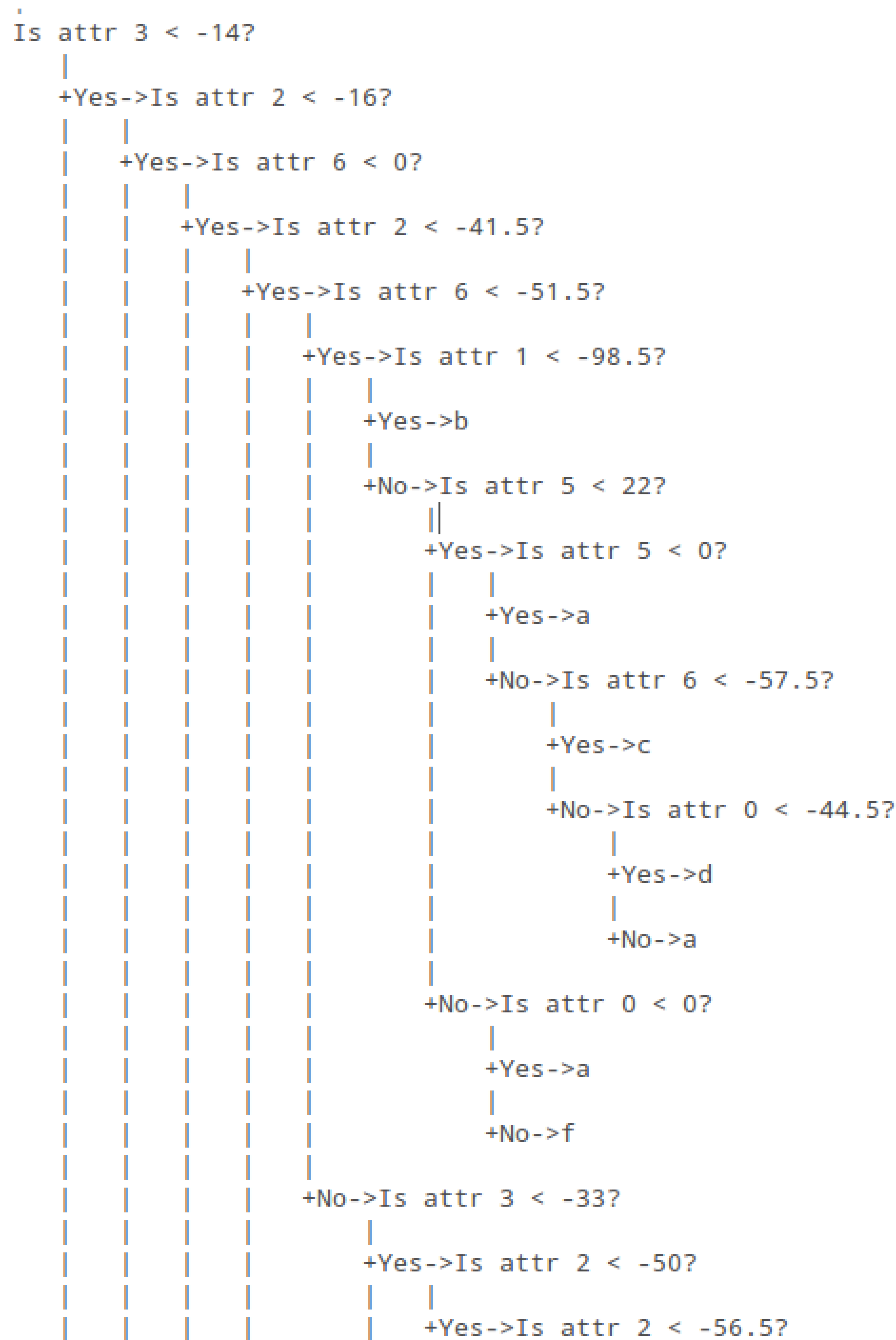


Figure 17: Sample decision tree printout

One issue found in early tests of the learning algorithm was that the decision trees that were produced often downplayed the importance of the robot position state variable. Because this was a discrete variable with only three possible values, while each mole state was represented by a continuous variable, the learning algorithm was able to find points in the mole

states on which to split with higher information gain near the root of the tree. Meanwhile, the robot position variable, which defines what moles correspond to the left, middle, and right whack actions, was often split on only near the leaves of the decision trees. This meant that the tree was often over-fitting, as the small differences in the time that a certain mole has been up were playing a much larger role in action selection than they should have. This hurt the algorithm's ability to fit data from new games of whack-a-mole and significantly reduced the performance of the autonomous robot. This issue was mitigated by manually forcing the decision tree to always split first on the robot position. With this split in place, the algorithm could easily match up the three whack actions with the moles that corresponded to them in the given robot position.

4.6.3 Execution of Learning from the Web

Once the learning algorithm was in place, it was important to give users the ability to execute robot learning from the web. By giving users the opportunity to see the effects of their training data in action, this project aimed to increase user engagement with the study as a whole. To make this possible, the start game screen was modified to give a user the option of watching the robot play autonomously, rather than playing himself. When a user reached the front of the queue and was redirected to the game interface, he would be presented with three radio buttons for selecting the type of game. The first button allowed the user to control the robot and thereby provide it with additional training data. The second button trained the robot using only that user's training data and allowed him to observe the robot playing autonomously. If the user had not yet provided any training data by playing, this option was disabled. The third button trained the robot using all of the training data provided by all users and allowed the current user to observe this community-trained robot's autonomous performance.

When a user selected either the second or third button and started the game, the interface sent the necessary decision tree creation parameters to the learning node. The learning node then selected the corresponding data and constructed the tree. In this autonomous case, the interface also included the robot's user ID in the start game message to the game node. This set the game node to listen for action commands from the learning node, rather than the web interface. During the game, the learning node constantly listened for the current mole states, and it maintained the robot and arm positions. Each time the robot completed an action, the learning node inputted the current mole, robot, and arm states into the decision tree and determined the appropriate action to take. This action was then sent to the robot for execution.

4.6.4 Evaluation of Learning Effectiveness

One important component of the study that was conducted involved evaluating the effectiveness of the robot's learning to play whack-a-mole. This evaluation was done primarily by running a series of autonomous whack-a-mole games under different conditions and comparing their results. First, the robot was trained with the data provided from every user, and its distribution of game scores was compared directly to the distribution of scores achieved by the human players. Next, the training data was filtered by various metrics, and the resulting learned behaviors were compared. This analysis sought to determine the impact of both the quantity and the quality of provided demonstration data on the autonomous robot performance. These relationships were explored by filtering the training data based on number of state-action pairs and game score, respectively.

4.7 Motivational Game Elements

To explore the effects of motivating study participants, several additional game elements were added, each designed to improve user engagement. These included a high scores table, user statistics, and a level progression. A/B testing was used to assess the effectiveness of each of these features.

4.7.1 High Scores Table

The first motivational game feature implemented in the game was the high scores table, a table comparing the best scores achieved by each player. This was intended to motivate users to achieve better scores, and therefore provide better training data, by facilitating competition between users.

Before the high scores table could be implemented, it was necessary to record the user ID, final score, and timestamp for each game. To do this, a game data table was created in the database for storing this information. The game node was then modified to record the information in the database at the end of each game. Every game would provide one row of information to the table, including a unique game identifier. Additional columns were added to this table later to support other game features such as user statistics and levels.

Once this infrastructure was in place, a high scores page was created where a user could view how his scores compared to those of other players. This page dynamically constructed the table upon page load with the most recent score information, pulled from the database in the

PHP. The scores themselves were presented as a series of tables, each in its own tab. Tabs were presented for displaying the high scores for all time or for only the current day. Within each of these, additional tabs were later added for filtering high scores by level, once additional levels were added as another motivational game feature.

All-Time High Scores | Today's High Scores

ALL-TIME HIGH SCORES

Total Scores | Level 1 | Level 2 | Level 3 | Level 4 | Level 5

PLACE	PLAYER	TOTAL SCORE	LEVEL 1 SCORE	LEVEL 2 SCORE	LEVEL 3 SCORE	LEVEL 4 SCORE	LEVEL 5 SCORE
1	davidkent	100	19	18	18	31	14
2	scornman	93	21	21	18	19	14
3	andy	78	18	19	13	25	3
4	rob	76	17	19	16	22	2
4	Not-Sonia...Seriously	76	18	25	14	19	0
6	foshiro	72	20	18	15	14	5
7	morteza	65	19	18	15	13	
8	ccunningham	55	17	17	10	11	
9	pansteak	52	17	20	13	2	
10	TrainMan5135	46	14	12	10	10	
11	tstabor	45	13	21	10	1	
12	forkicks	40	18	14	8		

Figure 18: High Scores page

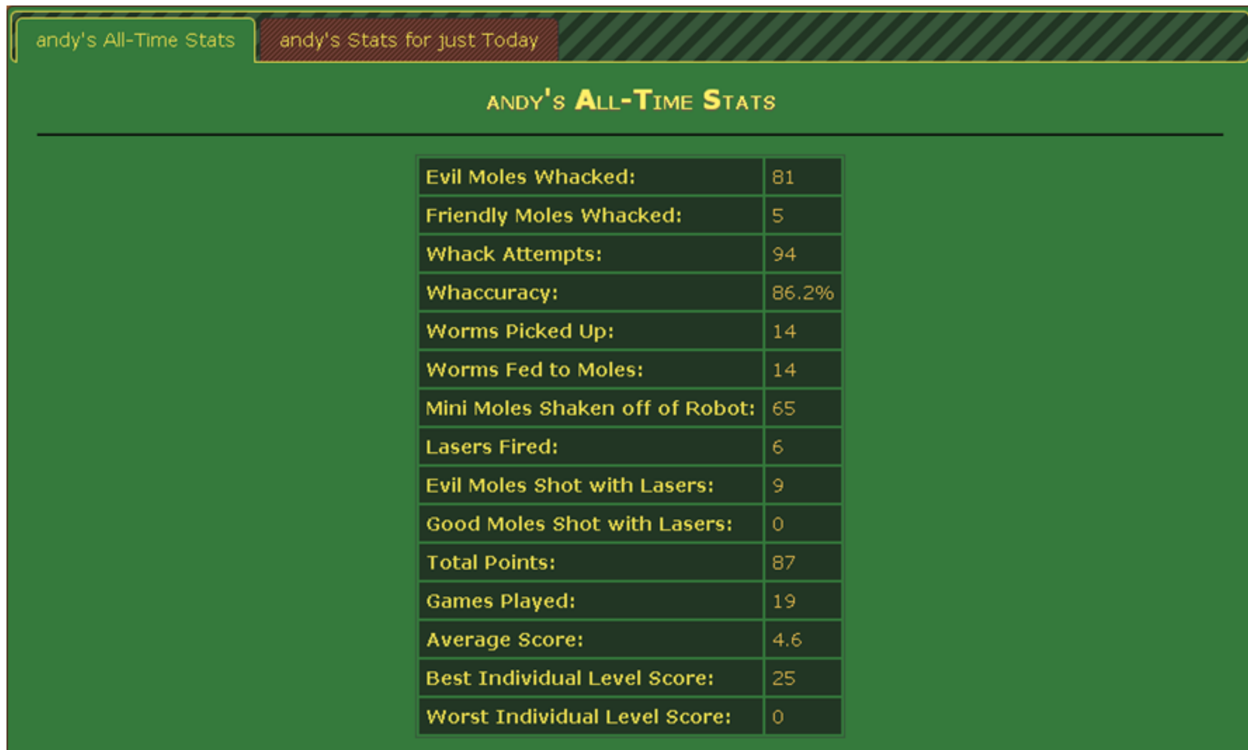
Each high score table determined the scores to display by selecting all of the rows from the game data table that matched the filter of the given tab. These rows were then sorted in descending order by score and iterated through to display the high scores. A separate list of user IDs in the table was kept to ensure that only the best score from each user was included. This was done to avoid cluttering the table with many scores from a single user, which could make it difficult for another user to find his best score in the table. To assist users in locating their own scores, the logged-in player's username and high score were colored red in the table. This player's most recent score was also displayed at the top of the page, along with the most recent scores achieved by the robot trained from his data, and by the robot trained from all players' data. The high scores achieved by the autonomous robot with each player's data and with all data were also included in the table, and these entries were colored blue to allow for easy comparison between the performances of robot and human players. These features can all be seen in Figure 18, a screenshot of the high scores page. The high scores page was made easily accessible to

users through a button in the menu bar at the top of every page. A link to the high scores table was also included in the results page that displayed to the user at the end of each game.

4.7.2 User Statistics

The second major motivational feature added to the game was the ability to view user statistics. By giving users access to numerical statistics about their games, as well as the ability to compare these statistics with those of other players, this feature aimed to motivate users to improve.

Implementation of this feature required significant support from the Whack-A-Mole game node. To store the information necessary to generate the statistics, several columns were added to the game data table. These new columns recorded the number of moles whacked of each type, as well as the total number of whack attempts. Once new mechanics were added with additional levels, they also included information on the number of worms picked up and fed to moles, the number of mini moles shaken off the robot, the number of each type of mole hit with lasers, and the number of laser firings.



Evil Moles Whacked:	81
Friendly Moles Whacked:	5
Whack Attempts:	94
Whaccuracy:	86.2%
Worms Picked Up:	14
Worms Fed to Moles:	14
Mini Moles Shaken off of Robot:	65
Lasers Fired:	6
Evil Moles Shot with Lasers:	9
Good Moles Shot with Lasers:	0
Total Points:	87
Games Played:	19
Average Score:	4.6
Best Individual Level Score:	25
Worst Individual Level Score:	0

Figure 19: User Stats page

To display these statistics to the users, a stats page was created. Similarly to the high scores page, this page contained two tabs, allowing the user to view either all-time statistics, or

statistics for the current day only. Each tab contained a two-column table detailing the value of each tracked statistic for the selected player. Some of these statistics, such as Evil Moles Whacked and Whack Attempts, were taken directly from summed columns in the game data table, while others, such as Average Score and Whaccuracy, were computed from these values indirectly. A screenshot of one player's user stats page can be seen in Figure 19, above. To allow users to compare their statistics with those of other players, the stats page was tied into the high scores page. When viewing the high scores table, a user can click on any player's username to view their statistics page, or use the menu bar to access their own statistics page.

4.7.3 Level Progression

Another major feature added to the game was a series of five different levels. The basic game as has been described, without any changes, was the first level. The other four levels modified the game logic to change the way the moles came up and added various virtual elements to the user interface. A screenshot of the fourth level which displays all virtual features can be seen in Figure 24. Players started out on the first level, and would progress to the next if they scored enough points to beat the level. The required number of points was different for each level, and was balanced in such a way as to be more difficult to achieve in later levels. Because the first level was the only one without virtual elements, it was the only level for which the collected data was usable for teaching the robot. To maximize the data collected on the first level, users who did not score enough points on a level to progress to the next were sent back to the first level. A page was added with descriptions for each level which introduced the new mechanics present in that level and exposed the story. The end-game page was modified to tell players whether they had scored enough to progress, and to link them to the page describing the next level if they had.

In order to implement the virtual features, a small object-oriented game engine was written in JavaScript. Although JavaScript does not have natural support for object-oriented classes in the same way that languages like C++ or Java have, class methods and inheritance can be achieved by careful manipulation of function prototypes. The core of the game engine was the sprite, a game object which could be regularly updated and which could draw images to the canvas. Support for animated sprite sheets, updating position from velocity, and checking the mouse against the sprite's bounds was implemented. The main game interface JavaScript file was modified to implement this game engine. The divs which had been used for the robot

position and whacking buttons were replaced with identical sprites, making the code much cleaner and more manageable. All of the onMouse events were modified to call corresponding sprite methods on all existing sprites. A window interval was set to call a game update function at a fixed rate. This update function called each existing sprite's update function.

This JavaScript game engine also used the WebAudio API for playing sounds. The WebAudio API allows for loading sounds at runtime and modifying them during playback. This was used to slightly randomize the pitch and volume of sounds for added variation. Although it was intended to make sounds for each game element, time constraints only allowed for sounds of moles coming up and down. Several recordings of the actual mole mechanisms were chosen from randomly and played whenever a mole state changed.

To support variation in mole behavior between levels, the game node was restructured with increased reusability in mind. The main loop of the game node managed the state of the game at a very high level, and a hierarchy of parameterized helper functions was used to encapsulate each basic task. Variations in game mechanics were handled by a set of flags indicating the state of each mechanic in the current game. A function was created for changing the game mode of the game, and it determined the correct new values of these flags. This system allowed the developers to easily add or remove existing features in each game mode. Though it was not used for the chosen level set, this system also allowed the game mode to be changed mid-game. This was intended to add support for dynamically-triggered bonus rounds and similar mechanics. Lookup tables linked each level with its starting game mode, the score necessary to beat it, and a goal message to display to the user.

The second level introduced virtual worms that would appear at random locations overlaid on the video feed. These virtual worms extended the sprite class from the JavaScript game engine. The area in which they could appear was chosen such that it would not interfere with the existing buttons. The user could click on a worm and drag it around, then release the mouse to drop it. The worms had an animation for appearing and disappearing in the ground, one for idling while in the ground, and one for being dragged around in the air. The frames of these animations can be seen in Figure 21 and Figure 20, below.

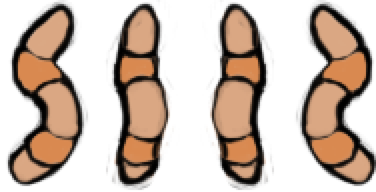


Figure 20: Sprite-sheet of worm in air

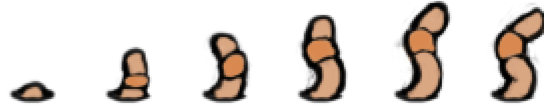


Figure 21: Sprite-sheet of worm in ground

At its core, each worm was a state machine with states corresponding to its animation, with one additional state for being invisible. If the player dropped a worm on a mole, that mole would eat the worm. When a mole ate a worm, that mole would stay up for an additional few seconds. This allowed players to strategically choose which moles to feed. To encourage players to use the worms and to maximize their usefulness, the mole behavior was also changed. Any mole hole which would have been empty was now filled with a friendly mole. This increased the danger of missing evil moles because if a player missed badly enough, they would hit a friendly mole and lose three points. This change in mole behavior was implemented in the game node through the inclusion of the noDownAllowed flag, which corresponded to three modifications in the game logic. Firstly, this parameter changed the behavior of the function responsible for bringing down a single mole, such that a good mole was brought up instead. Secondly, it modified the function that checked whether a mole was eligible to be brought up, allowing existing good moles to be switched to evil moles at any time. Finally, it modified the mole spawning probabilities to keep the rate of evil moles approximately the same, while eliminating the chance of good moles randomly popping up when already up by default.

The third level introduced virtual miniature moles which would tackle the robot and hang onto it. Each mini-mole gripping the robot slowed its speed while driving. If all five mini-moles were on the robot, it could not move. The arm speed was unaffected by mini-moles. To prevent the robot from slowing down, players could click and drag the mini-moles off the robot. This would stun the mini-moles for a time before they would tackle the robot again. Like the worms, the mini-moles were implemented as a child of the sprite class, and had a state machine governing their behavior. The mini-moles had states for being stunned, preparing to tackle the robot, and gripping the robot. The mini-moles had a stunned image and a normal image, which can be seen in Figure 22, below.

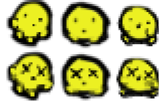


Figure 22: Sprite-sheet of mini-mole

To encourage players to pay attention to the mini-moles, the mole behavior was set so that no evil moles would appear within whack-able range of the robot. They would appear only in further holes which could not be reached without driving the robot. The game node implemented this by tracking the position of the robot based on updates from the robot control node. A function was written for determining the set of valid evil mole locations based on a given robot position. The function that checked for whether a mole could come up as an evil mole was modified to disallow moles at these locations. To compensate for the fewer valid locations, the overall mole spawn rates were increased. In all other aspects, the mole behavior was the same as in level 1. Worms also appeared in this level, so there was some interplay between the mechanics of the mini-moles and the worms. With both of these mechanics appearing at once, players had more choices. They could choose to use worms strategically to give them more time to shake off the mini-moles, or they could focus on shaking off the mini-moles as quickly as possible.

The fourth level introduced a laser. Players charged it up by shaking a knob back and forth on a virtual laser mechanism. When the charge passed a certain threshold, two buttons would glow red. Pressing the top button would fire the laser at the entire top row of moles, and pressing the bottom button would fire it at the entire bottom row of moles. This was chosen over targeting individual moles because being able to target individual moles gave players too much freedom. Targeting entire rows was both more powerful because players could hit multiple evil moles if they timed it well, and more limited because players would have to manage the risk of hitting friendly moles that were up in the targeted row. To accentuate this balance between power and risk, the overall mole spawn rate was increased in the game node, but the ratio of friendly moles to evil moles was increased as well. Improving power and situationality in this way can encourage more strategic play. The laser was implemented as a hierarchy of multiple sprite subclasses. At the top was the laser mechanism, which slid onto the screen from the sides and managed the sub-components of the laser. The sub-components were the battery, which handled the charge and determined when the laser could be fired; the knob, which could be dragged left and right to increase the battery's charge; the buttons, which activated when the

charge was sufficiently full and caused the corresponding laser to fire when clicked; and the laser itself, which provided a visual effect for the laser firing. The laser effect drew a rectangle to the canvas with a horizontal centered blue-white gradient which started out thin, grew tall, then grew thin again, in a squared sine curve, before disappearing. The entire laser mechanism can be seen in Figure 23, below. On the server side, the game node was responsible for maintaining and publishing the state of the battery, as well as processing laser firing messages. In the game node, firing the laser at a particular row was treated as equivalent to simultaneously whacking all moles in that row.

Figure 23: Laser mechanisms and beam

The fourth level also included the previous two levels' virtual mechanisms: the worms and the mini-moles. Because the evil moles could spawn near the robot on this level, players had even more choices than before. An effective strategy was to ignore the mini-moles, choosing instead to charge the laser. The combination of the worms and the laser allowed for interesting strategies for using the worms to line up entire rows of evil moles for the laser to hit. If a player were fast enough to manage all three virtual game elements, they could achieve a very high score on this level.

Figure 24: Screenshot of level four

4.7.4 A/B Testing Procedures

In order to achieve the goals of the project, it was necessary not only to implement game features in a robot learning from demonstration study, but also to evaluate the effectiveness with which they increased the quality and quantity of demonstration data. This analysis was key to showing whether similar motivational elements could be successfully applied to future LfD studies.

To evaluate the impact of the motivational features, a control group was needed to allow for comparisons between users with and without such features. To this end, each user was randomly assigned a user category upon creating an account. This category determined which game features would be accessible to that user. For the purposes of this study, the user base was split into four categories of approximately equal size.

The first category was the Minimal Features category, and it acted as the control group. Users in this category could play the basic whack-a-mole game and watch the robot play

autonomously, but they had no access to high scores, user statistics, or extra levels. While playing the game, these users were given no indication that more levels existed past Level 1. When a user in this category completed a game, the end game screen would contain no link to the high scores table, so the user would have no way to compare his score to those of others. The menu bar for this user type also excluded buttons for high scores, user stats, and levels.

The second user category was called Scores Only. Users in this category were given access to the high scores table and the user statistics page, but could also only play Level 1. When a user in this category completed a game, the end game screen would contain a link to the high scores table. Buttons for the high scores page and user stats page were present in the menu bar for these users. The high scores table that was accessible to these users did not contain tabs for showing scores from different levels. Rather, the only scores displayed in the table were those from Level 1. Splitting up the high scores table and user statistics pages into separate categories was considered. This idea was ultimately rejected due to the links between these features and the desire to limit the total number of categories, so that each category would have enough users for effective analysis.

The third user category was called Levels Only. Users in this category were given access to all of the game's levels, but not to the high scores table or user statistics. The game interface for these players displayed not only the score of the current game, but also the total score summed from this score and the scores of all previous levels in the streak. The end game page for a user in this category displayed the user's score along with a message about whether or not the player had beaten the level. If so, it told the user which new level he had reached, and included a link to the description of this level. Users in the Levels only category could access the level descriptions from the menu bar, but could not access the high scores or user stats pages.

The final user category was the All Features category, which included all of the implemented game elements. Users in this category were given access to the high scores table, the user stats page, and all game levels. The high scores page visible to these users included a tab for the best total scores from a series of consecutive levels, as well as a tab for each individual level. At the end of a game, a user in the All Features category would be shown her score and told whether she beat the level. If so, a link would be displayed to the description page of the new level she had reached, and if not, a link to the high scores page would be shown. Users in

this category had easy access to high scores tables, user statistics, and level descriptions from the menu bar.

To analyze the effectiveness with which each game element motivated users, game results were compared across the four categories with respect to several metrics. The analysis sought to determine the impact that the presence or absence of high scores tables and levels had on both the quantity and quality of demonstration data collected. The quantity of collected data was assessed by comparing metrics such as average number of games played per user, number of state-action pairs collected, and percentage of games ended early, across categories. The quality of data was assessed using metrics such as average score and top score per user.

4.8 Running the Study

Before making the system public, a failsafe was put in. A system status table was added to the database. If there were some malfunction or a change needed to be made, a flag in this table could be set to bring the system offline for everyone except the admins. If a normal user attempted to access any of the pages that interacted with either the robot or the database while the system was offline, they would be redirected to a page saying that the system was offline for maintenance.

For similar reasons, functionality was added for displaying admin messages to users. To accomplish this, a new column was added to the system status table for holding the current admin message string. On the web interface, a message box was added at the top of the screen that would be visible to users at from all pages on the site. On load, each page would read in the current admin message from the table and update the box accordingly. Because players in the queue would not be reloading the page while they wait, a ROS listener was added to the queue and game interfaces to listen for a new admin message published to the admin message topic. A bash script was created to automate these two methods of displaying an admin message. The script allowed admins to update the message both in the database and on the interface with a single execution.

Once these final features were in place, it was time to begin obtaining users for the study. As there was only one robot, only one player could play the game at a time. Since extremely long wait queues could hinder the chances of users playing multiple games, the game was advertised separately to different groups in several waves. Before official release, the game was play-tested by a few friends, who provided feedback that was instrumental in making final modifications to

the game. Once released, the game was first advertised to friends of the developers over social media sites such as Facebook and Twitter. Next, it was advertised via email to WPI students with relevant majors. Emails were sent separately to groups of a few majors at a time to avoid excessive queues. Finally, the game was advertised on a few online game developer forums. The advertising periods were all separated by at least one day, which was sufficient time for most users to complete most of their playing, making space in the queue for players from the next advertising period.

5.0 Results and Discussion

Upon completion of the system, the project team opened the game to the public and conducted a user study. As users played the game, data was collected about their play. The team analyzed this data to assess the robustness of the system, the robot's ability to learn the game, and the effects of motivational game features on the quantity and quality of demonstration data.

5.1 Study Results

The system was publicized over a period of two weeks. In that time, a total of 523 games were played by 191 unique users, making a total of 9568 state-action pairs. These games accounted for approximately 52,000 seconds of play time, which means that players were in games for about 4.3% of the two weeks. Furthermore, these account for over a third of the time that the robot was up, which was roughly an average of three hours per day. 11% of those games ended early. There were an additional 88 users accounts which were created but which never played a game. This means that 70% of users who created an account played at least one game with it. Stats on the number of people who visited the site but did not create an account were not collected because it would have been impossible to distinguish people who were visiting the site for the first time from those who had created an account already and were not logged in.

The advertisement of the system was fairly successful. Each round of advertisement brought in a somewhat gradual stream of new users for a few hours, which was the desired effect. On one day, however, an email was sent to several mailing lists at once, and too many people signed up and tried to play at the same time. The queue was filled with 10 people for several minutes, but they quickly left, presumably because of the large wait time. Afterwards, the stream of new users was slightly slower than the other days. The number of users over time can be seen in Figure 25, below

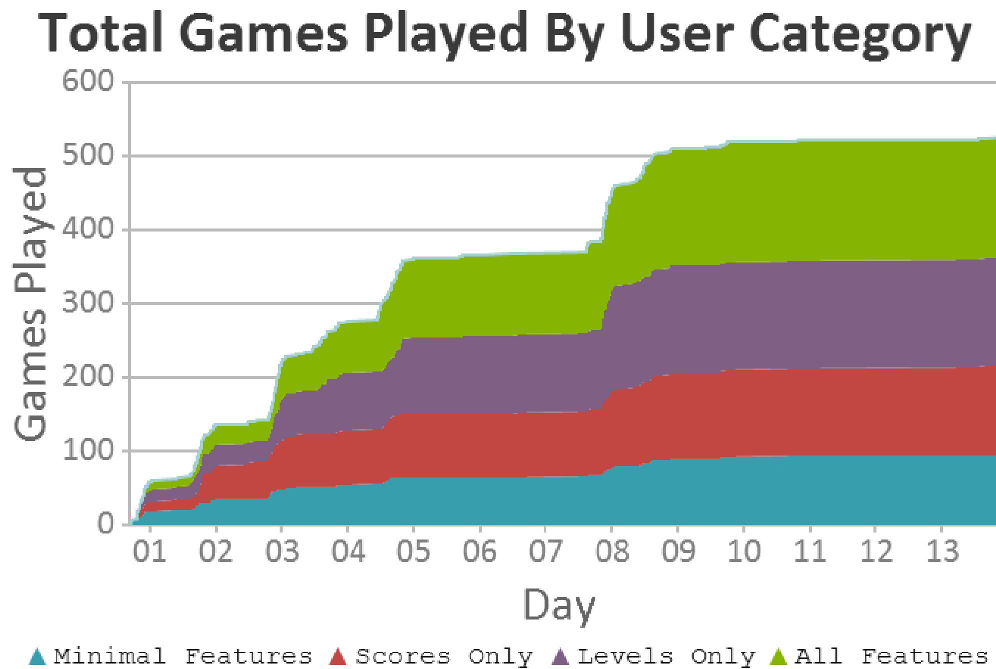


Figure 25: Total games played by user category

The game was very popular amongst the other students in the lab. Of the top five players with the most games played, three were from this group. Several of the top names on the high score table were students in this group. Every student in the lab who played, played several games. Why was being in the lab more engaging than playing from far away? It is possible that there was some difference between the students in the lab and the other players that made the game more appealing to the students in the lab. For instance, the students in the lab had watched the mole mechanisms be built over the course of the preceding months, and some of them were familiar with the youBot. Some of them had run or participated in similar remote robot control studies. It is also possible that the experience of playing in the lab was better than playing from elsewhere. The lag was lowest in the lab, the mole and robot sounds could be heard instantly, and a player could see the physical robot rather than a video feed of it. There was also a more personal competition between students in the lab than a player only looking at the high score table would get. However, because there were other users who were just as engaged as the students in the lab, these advantages were not necessary for full enjoyment of the game.

5.2 Robustness of the System

One of the lesser goals for this project was to make a system robust enough to survive being left on for days at a time without supervision. This goal was met with the mole mechanisms. They were left unattended every night during the two weeks the study was active. There was one incident where a mole mechanism's pulley fell apart, but this was easy to repair and was most likely caused by improper construction. None of the other mole mechanisms had any problems throughout the entire study. The Arduino was left powered on for the whole study, as well, and also did not have any problems.

The largest mechanical problem encountered was the rare incident in which the robot ran over its own power and Ethernet cables. This happened only twice during the study, but both times it happened it disabled the robot until it was manually untangled and reset. Hanging the wires from the ceiling was considered, but this had the danger of getting caught by the arm, which would have probably pulled the wires out of the ceiling and caused more damage than running over the wires did.

Another hardware problem which occurred before publicizing the system was the malfunction of the youBot's arm. The frequency with which it would fail to start increased slowly over a few weeks until it would not start at all. With the help of the youBot support hotline, the problem was diagnosed as a hardware problem, probably with the control circuitry, and the arm was sent to KUKA for repair. Because KUKA did not return the arm in a timely manner, an identical arm was borrowed from MIT. This arm worked properly for the whole span of its use.

The most serious software problem was a race condition which occurred between a user reaching the front of the queue and being redirected to the game interface. As soon as the system kicked out the previous user, it brought the next user to the front of the queue and created an experiment for that user, which is what allowed that user to see the game interface page. If the user were redirected to the game interface page before the experiment was created, the user would not be allowed to see the game interface page. This happened at least twice.

5.3 Learning Effectiveness

Analysis of the robot's ability to learn to play whack-a-mole from demonstration yielded a number of interesting results. The first major result to become apparent was the dramatic improvement in autonomous robot performance following the addition of the robot position split.

Before this change was made, the robot performed somewhat decently when trained with data from a small number of reasonably skilled users, but started playing atrociously once the number of training data points got too high.

It is believed that this was caused by over-fitting in the decision tree due in part to it often ignoring the robot position value. From the perspective of a human playing the game, in most situations the position of the robot is the single most important factor in deciding which direction to whack or move. Often this is the primary difference in state between two state-action pairs with similar mole states but different actions. Yet, because this attribute was represented as a discrete variable, the algorithm would often attempt to split repeatedly on mole state variables instead. With a huge amount of state-action pairs available, the decision tree would often attempt to fit all of these data points based solely on slight differences in mole times. This would result in the robot constantly taking actions that would only make sense if it were in a different robot position.

Once the tree was manually split into three trees based on robot position, results improved greatly. This change had the largest impact when the robot was trained from fairly large sets of data. Players who played many games with high scores were able to train the robot to achieve very high scores with this new algorithm, and the robot trained from all user data became significantly better as well.

After completion of the study, the robot played several games autonomously using behavior learned from various subsets of the collected training data. The scores achieved in these games were recorded, compared, and examined for trends. The first notable result from these games was that on average the robot, when trained from all collected user data, was able to outperform the average human. As seen in Figure 26, below, the average score achieved by the autonomous robot during these trials was 16 points, while human players averaged only 12 points per game over the course of the study. This was a clear indicator that the robot was able to successfully learn the task. Because the robot indiscriminately used all training data for these trials, one might assume that it would perform about as well as the average human, but there are a couple of factors that help explain its higher scores. Firstly, many of the humans that scored poorly in their games did not fail due to poorly-chosen actions, but instead either left long delays between actions or ended their games early. Secondly, depending on user location, some amount of network latency would always be added to users' response time, which contributed to these

types of delays even among skilled players. When running autonomously, the robot did not have these problems. Because the state space defined for this project did not include a wait action, state-action pairs were only recorded when users commanded the robot to drive or whack a mole. In autonomous mode, the robot always selected an action after completing the previous action, so it did not learn these delays. In these trials, the autonomous game was also never ended early. As Figure 26 shows, the robot was not quite able to match the average performance of top-scoring players when trained with only the top-scoring quartile of the training data, but when trained with the bottom-scoring quartile of the data, it scored far better than its demonstrators, many of whom likely suffered from the aforementioned issues.

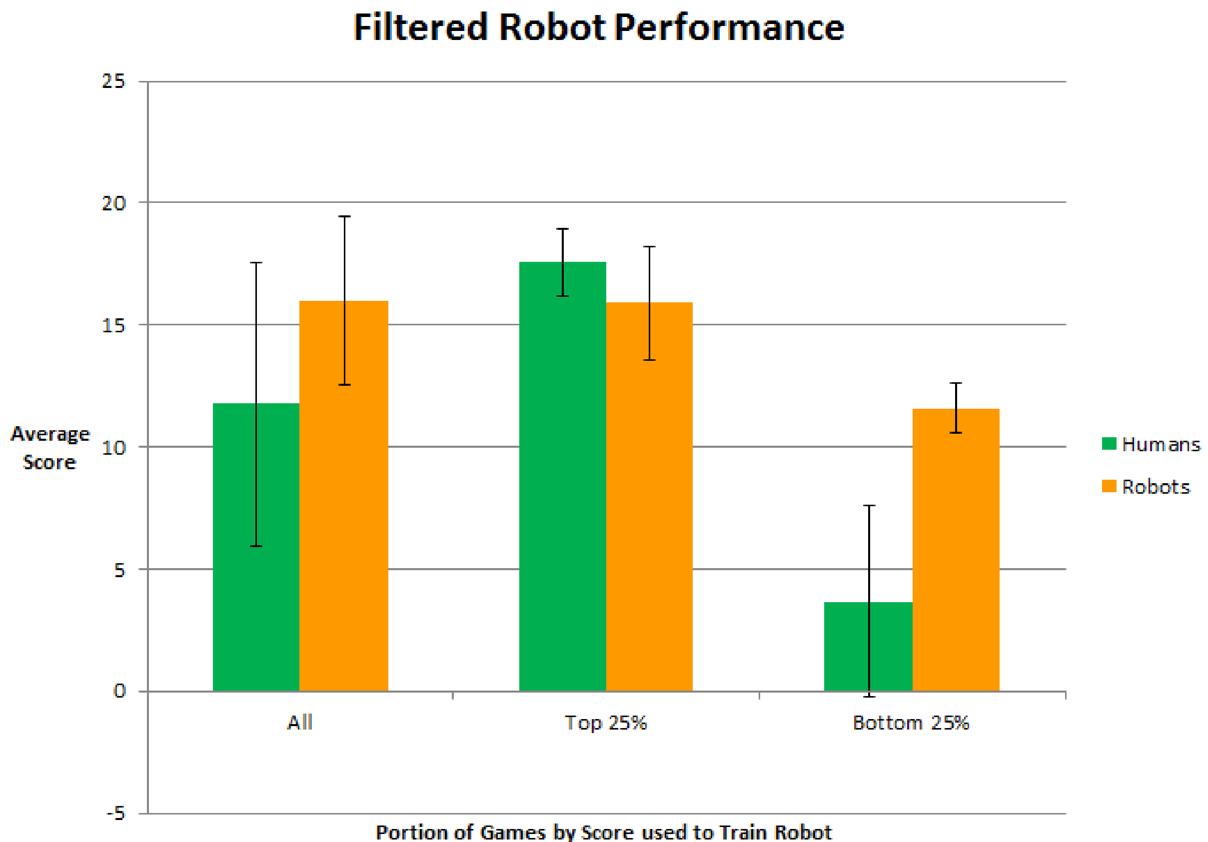


Figure 26: Comparison of human-controlled and autonomous robot performance filtered by score

The next bit of analysis performed with these autonomous robot trials was to examine the effects of training data quantity on learned behavior. To do this, the robot was trained several times with sets of randomly-selected state-action pairs. The number of state-action pairs in the training set was varied, and correlated with the average score achieved by the autonomous robot trained with that data. Figure 28 shows the general relationship between average score and

dataset size on a logarithmic scale. This shows that increasing the number of state-action pairs clearly increased the performance of the robot up to a certain point. Figure 27 is a similar plot, but the dataset size is plotted on a linear scale, showing data points at or above 1000 state-action pairs. This view makes it clear that there is less of a clear correlation once the training data size exceeds 1000 pairs. At this size, the robot seemed to fairly consistently make intelligent decisions, but a few key negative actions each game, caused by bad state-action pairs, limited the average score it could achieve.

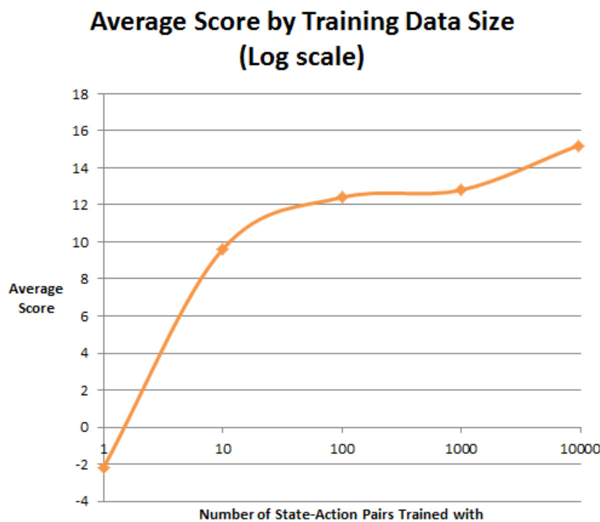


Figure 28: Score vs. training data size (log scale)

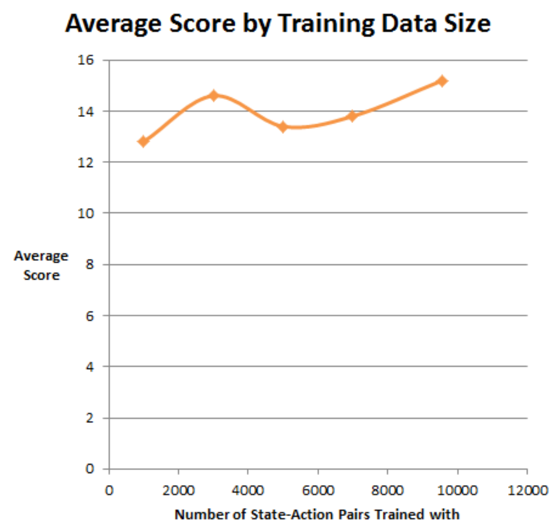


Figure 27: Score vs training data size (linear scale)

To deal with these harmful data points, the final portion of the study's learning analysis dealt with filtering state-action pairs by game score. This analysis sought to ascertain the correlation between the scores users achieved and the scores attained by the robot when trained with their data. To ensure that data quality would be the only variable in this evaluation, the quantity of state-action data was kept constant for each trial. Figure 26 compares the performance of the robot when trained from the top-scoring 25 percent of all state-action pairs with its performance when trained with the bottom-scoring 25 percent of all data. The chart clearly shows that the robot trained with better data performed significantly better, though the difference was less pronounced than the difference in performance between the humans who provided the data.

5.4 Impact of Game Features

One of the goals of this project was to analyze the effect that adding game features to a robot learning from demonstration task could have on the quality and quantity of the data collected. As was described above, users were split into four categories, each with a different subset of game features. It was expected that the high score table would improve quality of data because it added a level of feedback and competition to achieving higher scores, and the levels would improve quantity of data because they would give players more interesting challenges to overcome than the basic game. These trends did appear in the data, but they were somewhat weak.

To analyze data quantity, the number of games played by each user was considered first. The most obvious metric would be the average number of games played per user, which can be seen in Figure 29, below. The group with all features had a slightly higher average, but the standard deviations were so great that the difference was statistically meaningless.

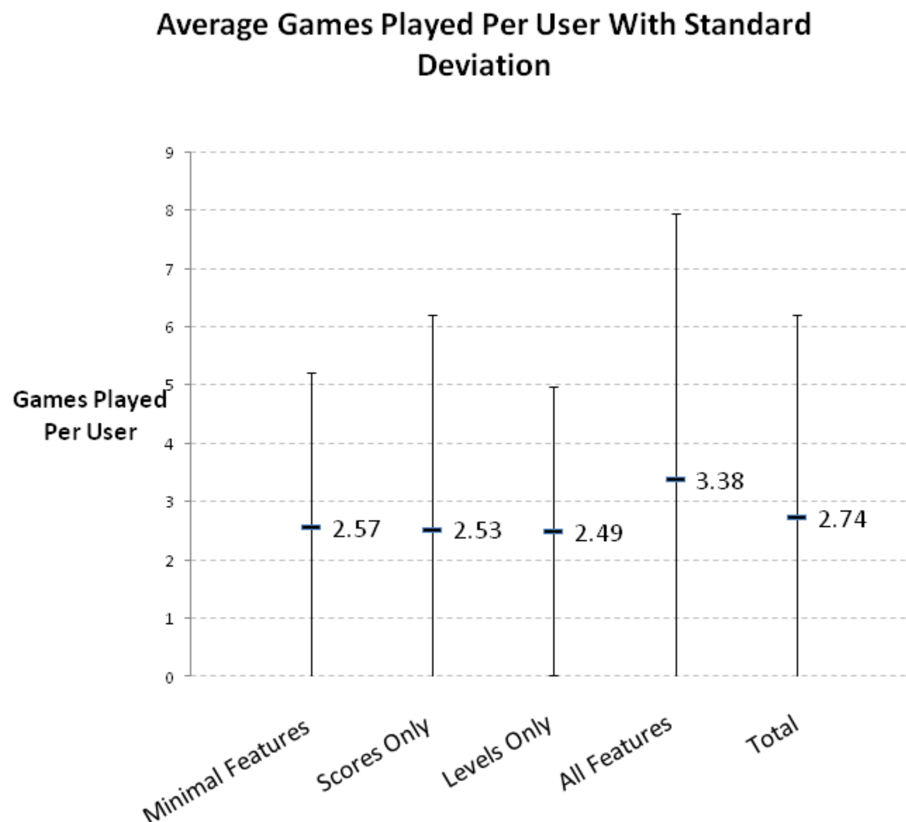


Figure 29: Average games played per user with standard deviation

To get a better idea of which categories were more likely to bring in more data, the percentage of users who played more than a certain number of games was considered. The

clearest trend was visible when looking at the percentage of users who had played more than three games, which can be seen in Figure 30, below.

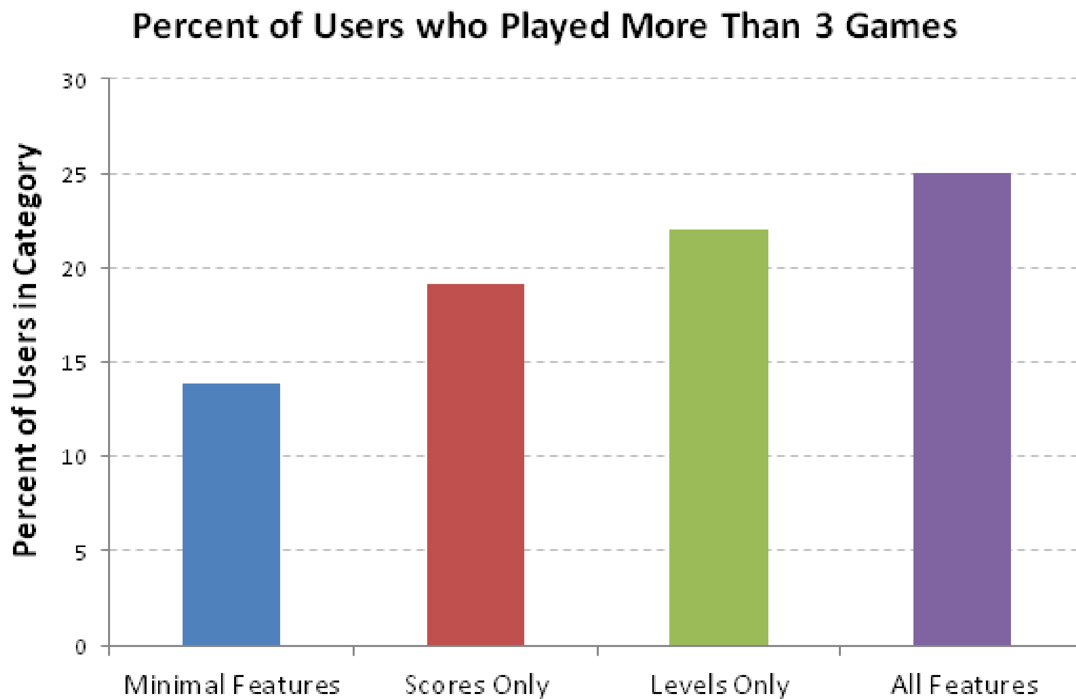


Figure 30: Percent of users who played more than 3 games

The scores category's percentage was higher than the minimal features category's percentage by about 4%, and the levels category's percentage was higher by about 8%. Although no expectation was held about the effect of a high scores table on data quantity, it is not surprising that there was some improvement, and that levels had more improvement than scores. The all features category's percentage was higher by almost 12%, which is the sum of the gain from the scores and levels. This suggests, as far as quantity of data goes, that the gains from scores and levels were orthogonal. They each provided separate sources of motivation which could be added together without losing anything in their combination. However, it should be noted that the high scores table was slightly more featureful when levels were present because, in addition to the individual score for each level, it also counted the total score of a player's run across multiple levels. Although it could not be measured, this could have affected the motivating effect of the high scores table, which may have allowed the lossless addition of scores and levels. If this feature were not present, it is possible that the all features category would not have been as far above the other categories.

To analyze data quality, the main metric used was score. A higher score meant a game with better data. However, only the scores from the first level were considered because that was the only level which was present across all categories. The average score per game achieved by each category was considered. These can be seen below, in Figure 31. The minimal features category had the lowest average score, and it was low enough to actually be statistically significantly lower than the other groups. Interestingly, the scores only category had the highest average score. It was expected that the scores category would have higher data quality than the levels category, but not that it would be greater than the all features category. This may be due to the fact that the users in the all features category spent at least some portion of their time playing on the later levels, giving them less time on the first level to achieve higher scores. Also interesting was that the levels only category had a higher average score than the minimal features category. This could be due to the fact that the levels category required users to reach a certain score on level one in order to proceed to the next level.

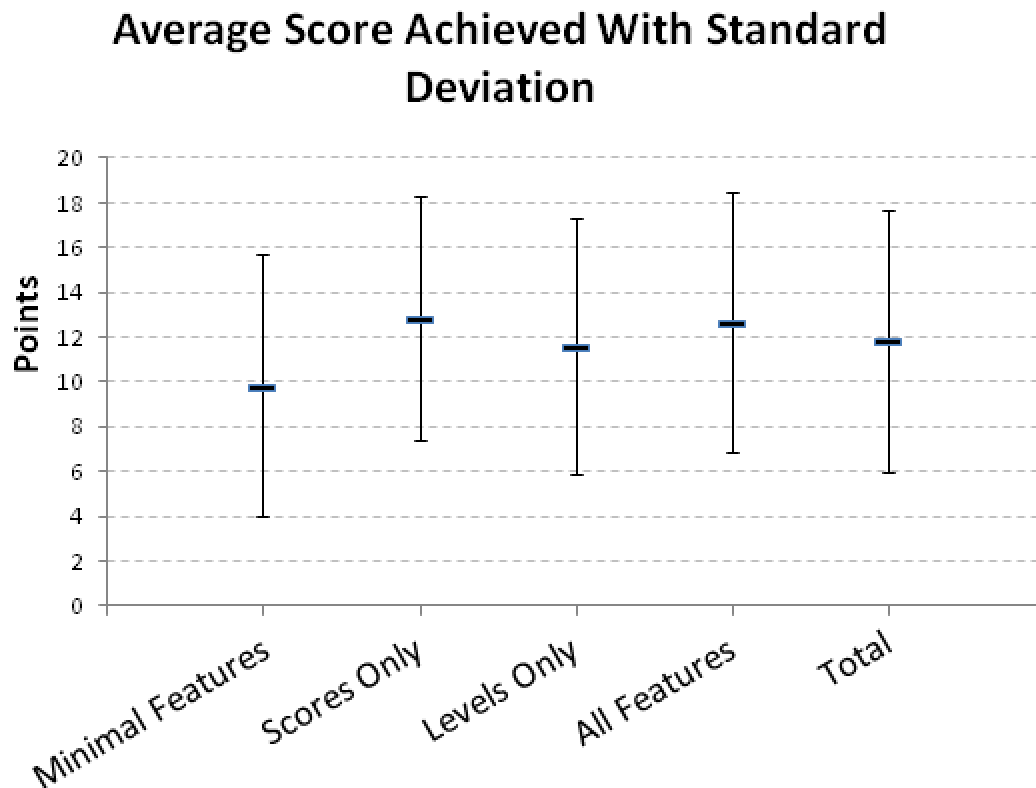


Figure 31: Average score achieved with standard deviations

Another metric for data quality considered was the number of state-action pairs collected per game. The averages turned out to be very nearly the same between all categories except the minimal features category, which was significantly lower than the other three, as can be seen in Figure 32, below.

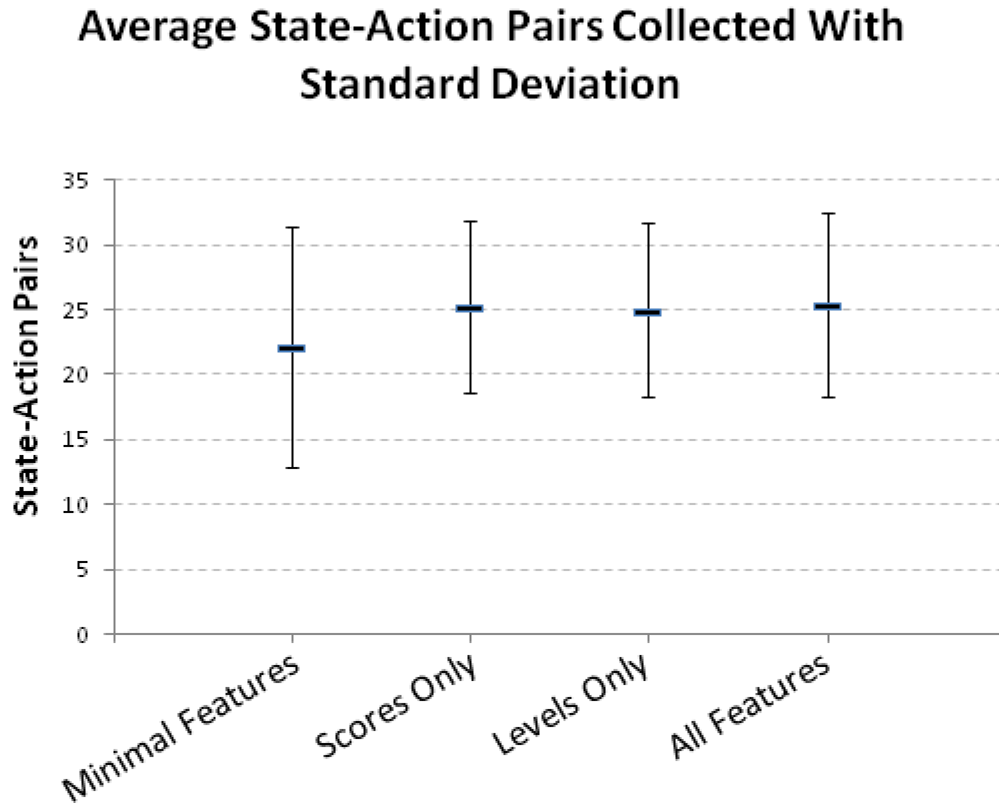


Figure 32: Average state-action pairs collected with standard deviation

To determine why the minimal features category was so deficient, game durations were analyzed. Because game duration was not actually recorded in the database, it had to be calculated from the difference between the earliest and latest timestamps of the state action pairs in each game. The average game duration graph, seen in Figure 33, below, shared the deficiency in the minimal features category with the average state-action pair graph. However, the levels only category had a slightly lower average game duration than the other two featureful categories.

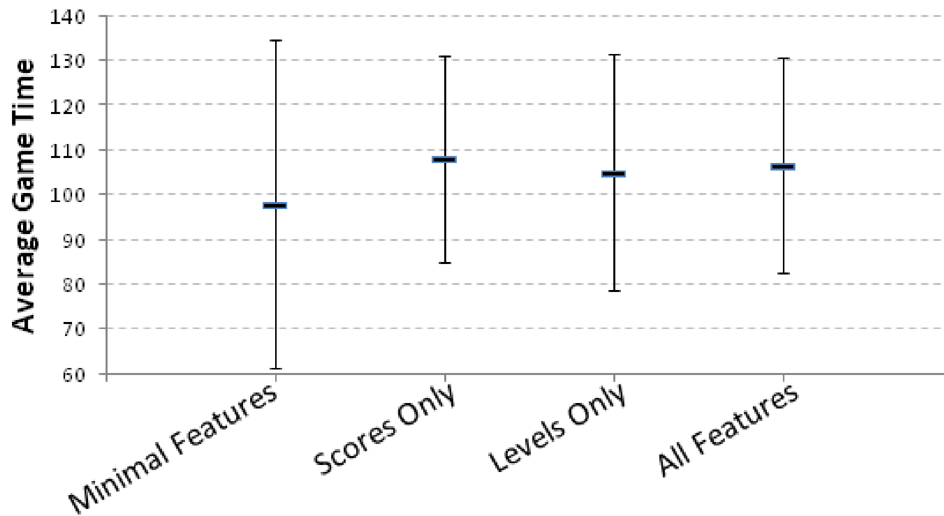


Figure 33: Average approximate game duration

It became clear that the deficiency in these two categories was related to players ending their games early. For a game to end early, the player had to either stop entering actions or navigate away from the page after starting a game. If a game lasted less than one minute, which was half of the length of a full game, it was considered as ending early. The percentage of games which lasted less than a minute in the minimal features category was significantly higher than it was in the other categories, and it was slightly higher in the levels only category than in the other two featureful categories. This can be seen in Figure 34, below. The increase in the levels only category may have been due to players giving up on the more difficult levels. The all features category also had a slightly higher drop-out rate than the scores only category, possibly for the same reason, but it was much less pronounced than the levels only category.

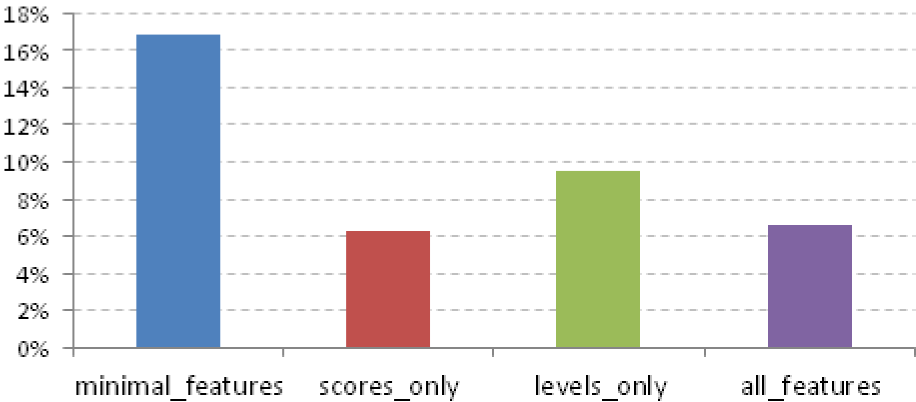


Figure 34: Percentage of games which lasted less than a minute

6.0 Conclusions

Despite the lack of statistical significance in the results, the data showed clear trends which matched expectations and supported the goals of the project. The building of the web-based learning from demonstration system was a success, and it was robust enough to serve its purpose well. Although no control study was done offline, putting the system on the web definitely enabled more users to provide demonstrations than an offline system would have had. Many of the users who played Whack-A-Mole were from out of state, and a few were even from overseas. These users would not have been able to provide demonstrations if they were required to do so from within the lab.

Presenting the task as a game was also a success. Aside from showing excitement about controlling a real robot, many users reported that the game itself was fun. The added motivational game features, the high scores table and the levels, improved both quantity and quality of the data collected. It was interesting to note that high scores improved quality more, and levels improved quantity more, but both features had an effect of both aspects of the data. Although those players who gave us feedback claimed to enjoy the levels, very few players made it all the way to the end of the game.

Our chosen task, Whack-A-Mole, does not have an immediately obvious useful non-game equivalent. It would be interesting to see a study similar to this one which compares a non-game task with a near-equivalent game task, especially if the non-game task solved a real problem.

We speculate that lag was the greatest enemy of player engagement. The game became more difficult and unstable as lag increased. Though still playable to a point, the difficulty level with more than a certain amount of lag was too high for most players. This may have been responsible for the somewhat high percentage of users who played only one game. Each game's round-trip time was not recorded because of scalability considerations, but these could probably be solved. Recording or otherwise minimizing lag should be a priority for any groups considering undertaking a remote robot control study.

The other goal of this project, analyzing the machine learning, was also a success, but less clearly so. The machine learning itself was a success as the robot was able to learn to not only play Whack-A-Mole, but to play it better than the average human. However, it never got better than the best humans. This may be a limitation of decision trees, or of the definitions for

states and actions that were used. Testing different algorithms and multiple different definitions for states and actions was beyond the scope of the project, but doing so could provide some insight about why the robot couldn't beat the best humans. Future groups could also look into ways to prevent over-fitting. Although the Waffles library had support for tuning tree parameters to reduce over-fitting, it was weak and too slow to use here. The goal related to machine learning was to analyze filtering by score and show that it could improve performance. This improvement was not seen. Rather, the average score without filtering was the same as the score achieved after filtering out all but the best quarter of the data. This may have been related to the strange effect seen when filtering out all but the worst quarter of the data. With the bad data, the robot was still able to score reasonably well. This suggests that the decision trees being created were surprisingly robust to bad data, which could explain why the trees made from all data were just as good as the ones made from the best data.

Overall, this project showed that bringing robot learning from demonstration studies to the web as games has many real and potential advantages over the traditional methods.

Works Cited

Alpaydin, E. (2004). *Introduction to machine learning*: MIT press.

Argall, B. D., Chernova, S., Veloso, M., & Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and Autonomous Systems, 57*(5), 469-483.

Banko, M., & Brill, E. (2001). *Mitigating the paucity-of-data problem: Exploring the effect of training corpus size on classifier performance for natural language processing.* Paper presented at the Proceedings of the first international conference on Human language technology research.

Brady, K., & Tarn, T.-J. (2002). Handling Latency in Internet-Based Teleoperation. In K. Goldberg & R. Siegwart (Eds.), *Beyond Webcams: An Introduction to Online Robots*: MIT Press.

Crick, C., Osentoski, S., Jay, G., & Jenkins, O. C. (2011). *Human and robot perception in large-scale learning from demonstration.* Paper presented at the Proceedings of the 6th international conference on Human-robot interaction.

Deci, E. L., Koestner, R., & Ryan, R. M. (1999). A meta-analytic review of experiments examining the effects of extrinsic rewards on intrinsic motivation. *Psychological bulletin, 125*(6), 627.

Denis, G., & Jouvelot, P. (2005). *Motivation-driven educational game design: applying best practices to music education.* Paper presented at the Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology.

Goldberg, K., Gentner, S., Sutter, C., & Wiegley, J. (2000). The mercury project: A feasibility study for internet robots. *Robotics & Automation Magazine, IEEE, 7*(1), 35-40.

Halevy, A., Norvig, P., & Pereira, F. (2009). The unreasonable effectiveness of data. *Intelligent Systems, IEEE, 24*(2), 8-12.

Liu, Y., Alexandrova, T., & Nakajima, T. (2011). *Gamifying intelligent environments.* Paper presented at the Proceedings of the 2011 international ACM workshop on Ubiquitous meta user interfaces.

Lozano-Pérez, T., & Kaelbling, L. (2005). An example of a decision tree: MIT OpenCourseware.

Przybylski, A. K., Rigby, C. S., & Ryan, R. M. (2010). A motivational model of video game engagement. *Review of General Psychology, 14*(2), 154.

Russell, S. (2009). Artificial Intelligence: A Modern Approach Author: Stuart Russell, Peter Norvig, Publisher: Prentice Hall Pa.

Saucy, P., & Mondada, F. (2000). KhepOnTheWeb: open access to a mobile robot on the Internet. *Robotics & Automation Magazine, IEEE, 7*(1), 41-47.

Sorokin, A., Berenson, D., Srinivasa, S. S., & Hebert, M. (2010). *People helping robots helping people: Crowdsourcing for grasping novel objects.* Paper presented at the Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on.

Toris, R., Kent, D., & Chernova, S. (to appear). The Robot Management System: A Framework for Conducting Human-Robot Interaction Studies Through Crowdsourcing. *Journal on Human-Robot Interaction*.