Abstract

Until now, reinforcement learning has mostly been implemented using high-level programming languages such as Java and C. Our goal was to implement reinforcement learning in a simple graphical environment. We achieved this by programming algorithms in the graphical programming environment, Scratch. We built an algorithm that averaged successful distances to calculate the ideal actions needed to capture a ball in a simple single-variable game. After using the algorithm, an agent computed ideal actions, and was eventually able to play the game perfectly.

We also built a reinforcement learner for the classic computer game, Copter, in which a keyboard controlled helicopter dodges scrolling blocks. Our algorithm considered future game states to compute the value of actions in particular game states. After significant tuning of relevant variables, the Copter learner played almost perfectly. Our research shows that reinforcement learning can be simply implemented and therefore has the potential to become commonplace in everyday life.

1 Introduction

Computer science has evolved over the last 50 years from a scientific field involving simple calculations computed by vacuum tube monstrosities, to a world-wide presence that controls infrastructure, provides entertainment, and drives scientific advancement. Reinforcement learning is still in its early stages, but has the potential to change the way the world works at a rate never before seen in history.

Reinforcement learning is a subset of machine learning in which a computer learns to solve a problem efficiently by interpreting information about its environment and its experiences. The goal of our research is to implement reinforcement learning in simple environments and use algorithms to make the agent quickly learn to solve problems as efficiently as possible. The two environments we chose for this project are a Harry Potter Quidditch game, and a helicopter game. Both of these games were written in a programming language called Scratch, which was developed by MIT for its simplicity and ease of use. The open source nature of all Scratch applications was also a major factor in our decision to use Scratch.

The first game we picked was based off the Harry Potter sport of Quidditch. In Quidditch, a player known as the seeker flies on a broomstick and tries to capture a ball called the Snitch. The Harry Potter Quidditch game, created by Michael Littman, was picked due to its lack of environmental
variables (factors that change while playing the game) and the single control input that is required to play the game. This game consists of a seeker that scrolls across the screen in a single dimension, and a Snitch that scrolls in the same dimension at a slower speed. The goal is to press the space bar as the player approaches the Snitch in order to catch it.

The helicopter game, designed by a user named Bungle, is a Scratch version of the classic game, Copter. It is significantly more complicated than the Harry Potter game because it has two possible control inputs, and many factors that affect whether the player wins or loses. Helicopter is essentially a scrolling world where rectangular obstacles appear every few seconds and move towards the player's helicopter. The player must dodge each obstacle by pressing the up arrow to fly the helicopter higher, or by releasing the up arrow to allow the helicopter to drift lower. The player must also avoid flying too high or too low, so that the copter will not crash into the ceiling or the floor. The game continues to scroll indefinitely until the helicopter crashes, so performance can be measured by calculating the distance the helicopter travels through the world.

Our programs are unique because reinforcement learning has never, to our knowledge, been implemented in Scratch. This is important because Scratch offers a simplified programming environment for games and other visual applications. This means that reinforcement learning is more accessible than ever before, which will allow a new demographic to understand and develop it. Development of reinforcement learning could lead to advances in neuroscience, intelligent machines, and eventually a world where machines will be independent.

2 Background

2.1 Reinforcement Learning Agents

Reinforcement learning (RL) is a process in which a computer program called an agent interacts with its environment and compiles data based on its experience. This data allows it to solve a problem on hand. The problem in RL is defined by a numerical reward that the agent receives from its environment as shown in Figure 2.1. The learner, known as the agent, attempts to maximize this reward by performing actions that will result in the greatest long term reward. The function that assigns reward to the learner is preset to represent the goals of the agent. For example, a chess game may have a positive reward assigned to winning the game and taking the opponents pieces, or similarly, a program designed to drive a car may give huge negative rewards for crashes [1]. The reward function essentially defines the agent's problem and goals.

Figure 2.1: This shows the agent/environment interactions. An agent receives a state and reward from the environment, performs an action on the environment, and receives the next state and reward from the environment.

To accomplish this goal and maximize accumulated reward, the agent must discover a policy that maximizes its long-term reward starting from the current state of the environment. In RL,
A policy is defined as a set of actions to take based on the state of the environment. The policy may take the form of a table, a set of rules, or any other mathematical expression that assigns an optimal action to an environmental state. The environmental state, or simply, the state, is a collection of all the data that the reinforcement-learning agent extracts from the environment at a given moment. Thus, the state space refers to all of the variables in the environment that the agent receives as inputs (shown in Figure 2.1). States are important to policies because they define when a given action can and should be taken. These policies, which are often represented as probabilities of future reward, are constantly updated as the agent learns, and should eventually converge to an optimal policy which expresses the best possible set of actions to take given certain states.

The optimal policy is defined as the policy which represents the optimal value function. Value is a measure of long-term total reward, and the value function is an expression which represents this quantity. It is often expressed formally as

$$V^\pi(s) = E^s_-\sum_{t=0}^{\infty} \gamma^t r_{t+k+1} \mid s_t = s$$

where the value of policy \( \pi \) is equal to the discounted sum of the rewards at states defined by the time interval \( t \) and state number \( k \) [1]. Basically, value is described by the sum of the rewards applied with a discount factor \( \gamma \).

Essentially, the value function is the core of RL because an optimal value function represents a perfect learner. This means that using algebra and mathematics to solve a value function will accomplish the same goal as an RL algorithm. This equivalence between RL and mathematics is key in deciding which problems are most efficiently solved by RL. For example, the sport of archery can be modeled at a basic level using physics to relate the archer’s bow angle, the target distance, the target height, and the wind speed. If we wanted to create an optimal archer, it would be much more efficient to program it with physics equations, instead of an RL algorithm that learns these relationships on its own. Therefore, archery is a problem where RL methods would not be an ideal solution. In contrast, a game such as backgammon has about \( 10^{20} \) different possible board configurations, and would require solving what are known as the Bellman optimality equations for an incredible number of variables. This would take years even on today's fastest computers, and is clearly not feasible [1]. Backgammon, however, is relatively feasible in an RL situation because it has discrete states and actions.

Q learning is an RL algorithm based largely off of the value function. Specifically, Q-learning is based off of what are known as action-values, which are represented by the variable Q. State action-values simply refer to the value of an action “a” given a state “s”. The Q-learning algorithm is:

$$Q[s, a] := (1 - \alpha)Q[s, a] + \alpha \left( r + \gamma \max_{a'} Q[s', a'] \right)$$

where \( \alpha \) is the learning rate, \( \gamma \) is the discount factor, and \( \max \) refers to the maximized value of Q [2]. \( \alpha \) controls how quickly a new experience affects Q values by weighting a known experience and a new experience inversely. \( \gamma \) is important because it controls how much the agent considers future rewards. If \( \gamma \) is high, then the agent will become foresighted, weighing the consequences of future states heavily into its decisions. If \( \gamma \) is low, the agent will be shortsighted,
only considering very immediate rewards. This is one of the key factors of Q-learning, and allows an agent to make decisions based on the probabilities of encountering good and bad states, not only in the next state, but also further into the future.

Greediness is another key factor in an RL agent. Naturally, in order to learn, an RL agent must spend a portion of its time exploring all possible actions so that the best ones can be discovered. At the same time, optimal decisions should be made as often as possible, so that the agent can maximize its reward. This is known as the explore/exploit dilemma, and is an important consideration in any reinforcement learner. Often this dilemma is solved by forcing the agent to take a random action a certain percentage of the time. This percentage is slowly tapered off as the agent converges to an optimal policy. Some situations, like our Snitch game, can be infinitely greedy because future states are not impacted by current actions. This is because capturing or missing the Snitch has no effect on the Snitch or the seeker’s positions, which are the only indicators of state. In this case, a random exploration period was implemented in the beginning, and was completely shut off once five Snitches were caught. The helicopter game required a clever workaround to the standard explore/exploit ratios, because an unnecessary crash could not be tolerated once a good policy was achieved.

2.2 Differences in RL Problems

In addition to all of the factors that make up agents, there are also many aspects to RL problems. There are two essential categories of RL problems: Markov decision processes and non-Markov decision processes. Markov decision processes (MDPs) consist of states that all have the Markov property. The Markov property means that each state contains all of the information necessary to make an optimal decision. Chess is an example of an MDP because each board state contains all the information the player needs to make the best move possible. This ignores the other player’s behavior and tendencies, which are non-Markov. An example of a non-MDP is the card game Go Fish. In Go Fish you draw cards from either the deck or another player in order to make pairs with all the cards in your hand. The current state space contains the cards in your hand and any pairs that players have placed down. Because not all of the information needed to make an optimal decision is in the current state space, Go Fish is not considered an MDP. RL is most often implemented in MDPs and near-MDPs because it is much simpler to write algorithms for them since they only need to take current and future states into account. Some algorithms (most notably model-building algorithms) can handle non-MDP reinforcement learning, but this paper deals solely with MDPs.

MDPs can also be categorized by other distinguishing criteria. All RL problems are either episodic tasks or continuing tasks. Episodic tasks have terminal states, where they stop and reset to a starting configuration. These tasks usually do not implement discounted reward because the reward must be maximized over the discrete period of time present in each episode of the task. Sometimes reward is assigned at the end of each task only, and the reward is therefore assigned to the total set of decisions that were made within the episode. An example of an episodic task
is Tic Tac Toe. An agent plays through each game individually, and is assigned reward based on whether it won, lost, or tied. The set of experience over the course of many games of Tic Tac Toe is used to find an optimal policy [1].

Continuing tasks are the opposite of episodic tasks. They consist of a single episode that continues to play out infinitely, or until it is interrupted. This means that rewards within the episode must be discounted so that more imminent rewards are weighted higher in the decision making process than rewards in the more distant future. The act of balancing a pole is an example of a continuing task given by Sutton and Barto. Balancing a pole can go on indefinitely, and requires future states to be discounted so that the immediate next state is considered with the most weight. A pole balancing agent should either be assigned positive reward for not dropping the pole over a given interval of time, or a negative reward for dropping the pole. Either way, the maximum reward comes from balancing the pole indefinitely [1]. Both the Snitch and Copter games researched in this paper are examples of continuing tasks.

Another small differentiating factor in RL problems is whether state transitions are stochastic or deterministic. Stochastic state transitions have a probability that a certain next state will occur, given a current state and action. This makes for a significantly more complicated game than one with deterministic (fixed) state transitions, where an action in a particular current state will lead only to one next state. Solving these games mathematically can be highly complex, but with RL algorithms can become simpler, and so are often used in RL.

2.3 Brief History of Reinforcement Learning

RL as we know today started in the 1980s through the union of two main ideas: learning by trial and error and optimal control. Learning by trial and error has its roots in the study of psychology and animal learning while optimal control concerns the use of value functions and dynamic programming in order to define long term learning. Trial-and-error learning in psychology led to the development of the Law of Effect, which involves selecting actions, comparing their consequences, and associating them with particular situations. RL uses the need for greater reward or satisfaction to foster favorable actions and reject other undesirable ones. Optimal control problems were solved by dynamic programming, using the Bellman Optimality equation, which was created during the 1950s. This equation defines the policy that returns the most reward. Dynamic programming is more efficient than most other methods, although its computational time grows exponentially with more states, accounting for its widespread use. RL solves problems related to optimal control problems, especially MDPs [1].

3 Method/Design Decisions

Feasibility was a major factor in the decision to implement RL in the Harry Potter and Copter games. Some initial ideas regarding applications of RL included games like foosball, Guitar Hero, and minesweeper. However, after some thought, we realized that the best option given the project time constraints was to pursue an algorithm that would teach an agent without the use of complex hardware or environments. Ideally, this environment would be pre-designed so that most of our efforts
could be focused on programming the agent, instead of designing a proper game.

3.1 Environment Decisions

Foosball, in particular, was rejected due to the complexity of building a robot that could not only make contact with the ball, but also see and recognize ball movements accurately. Also, the number of states in foosball is enormous, and state transitions follow very complex patterns based on subtle geometry and physics that would not be picked up by a simple RL agent.

Guitar Hero would have been reasonably simple to implement, due to pre-existing work that has been done with Guitar Hero playing robots. A Guitar Hero robot called DeepNote can play Guitar Hero perfectly by reading the television screen for notes using five photodiodes (discrete electronic components that sense the presence of light) [4].

Using photodiodes would be perfect for RL because they output discrete digital signals which would define a simple state space. The game also outputs a sound, which could serve as an input to the agent as feedback for when it makes a mistake. The reason Guitar Hero was not chosen as an environment for RL is because Guitar hero is not a typical RL problem. It can be played by simply adding a latency constant to the photodiode outputs in order to hit notes on time. Therefore, RL adds a substantial amount of calculation and coding that is not necessary in a game as straightforward as Guitar Hero.

Minesweeper was not used as an environment because it is a partially observable MDP and also a complex problem in theoretical computer science [3]. A partially observable MDP is an environment that contains only Markov states, but the entirety of each state cannot be read by the agent. This makes designing an RL algorithm highly complex, and beyond the scope of our project. Designing a minesweeper agent has been proven possible by researchers from U.C. Berkeley, but it is highly complex, and therefore doesn’t fit within the goals and constraints of our work [3].

Finally, we considered the possibility of having multiple agents. One example would be to have multiple exploring agents report back to a main robot which would accomplish the given task. This hierarchical structure’s implementation was discussed in the context of a Warcraft-like game where a main agent, that simulates the human player, controls lesser agents, which would perform simple functions such as gathering wood, gold, and building structures. To control the subordinate agents, the main agent would manipulate reward functions in order to assign tasks. The problem with multiple hierarchical agents in the Warcraft-like game was that they would require dynamic reward functions. This would have confused the lesser agents because every time a reward function got changed, the policies would no longer describe an optimal value function. This would cause the agent to behave irrationally and the agent would need to relearn its task.

Besides the time and complexity constraints, programming languages also played an important role in deciding which problems were feasible. For the Warcraft-like game, Java and Matlab were considered as they both provided an easy way to create two-dimensional arrays. The simplicity of array functions in Matlab was an integral attribute that
allowed its simple implementation. Because the Warcraft-like game needed dynamic rewards, the idea was abandoned. In addition, our lack of experience with Java and the relatively short time constraint made it difficult to implement a successful algorithm. After some research and exploration, we found Scratch, which proved to be an easily learned interface that would still allow us to work with complex visual environments, and implement different types of agents.

3.2 Scratch Games

After a cursory search for Scratch games online, we arrived at a game for which we could easily implement a simple algorithm.

The Harry Potter Snitch game formed a good foundation for learning to program an agent in Scratch. The only attribute of the state that was reported to the agent was its distance away from the Snitch. This made for a direct correlation between the state value and the optimal action. Computing a value function or using traditional algorithms such as Q-learning was not necessary for such a simple problem. We solved the Snitch game by computing an optimal policy through the running average of successful Snitch captures. The agent would first attempt random catches in an “explore” mode until it made five successful catches. It would then use the average state value of all the successful Snitch captures as the only state in which to attempt to capture the Snitch. This allowed the agent to capture as many Snitches as possible.

At this point, we wondered if indeed we could implement a more complex algorithm in a game using Scratch. We searched for a game with a more dynamic environment and a vast number of possible states. We also wanted to make sure that the agent was required to take more than one action. A popular game that satisfies these constraints is Copter, a famous Flash-based game recently adopted for Scratch. In this game, a helicopter navigates a cave where there are several rectangular obstacles hindering its path. When the helicopter comes in contact with an obstacle or hits the floor or ceiling of the cave, it crashes and the game is ended.
In order to make the actual game “agent-friendly”, we flattened the floor and ceiling of the environment to simplify the game, and defined quantitative states. Initially, we identified the parameters that an agent would find useful for determining its state in the game. We called these parameters relative distance x, relative distance y, close to top, and close to bottom. The relative distance parameters defined the state of the environment with a 3x3 grid. Relative distance x told if the helicopter was far, close, or very close to the wall in the x direction. Relative distance y told whether the helicopter was above, in line with, or under the wall in the y direction. The “close to” parameters told the agent if the helicopter was about to hit the floor or ceiling of the cave.

Originally, we designed the reward function for Copter so that the helicopter would receive a reward of -10 for crashing and zero reward for surviving. This function, however, did not yield good results. The helicopter had tendency to stay very close to the floor, and frequently crashed in the meantime, as it tried to avoid the walls. Because of this, we decided to apply a larger negative reward (-50 as opposed to the usual -10) for crashing into the floor or ceiling. In order to allow the helicopter to run autonomously for hours, the game also needed to be modified so that crashes did not reset the game. This was essential in allowing us to efficiently allow the agent to learn.

The agent still did not learn to avoid walls and ceilings very effectively, regardless of the tuned reward function. We decided that we needed to re-define the states because some states turned out to be identical. For example, the parameter “relative distance y” did not specify the magnitude of the vertical distance the block was from the helicopter and thus distorted the values of certain states. We first defined three more levels for this parameter so that there would be six total levels of “relative distance y” which could now better approximate the vertical distance of the helicopter from the wall. We then abandoned that system in favor of an absolute coordinate system that would define the distance of the wall and helicopter objectively relative to the environment. Because on occasion the helicopter would be the same relative distance away from the wall but would be in a completely different position in the cave, the agent was led to several erroneous decisions that failed to take into account the proximity of the floor and ceiling. For the absolute coordinate system, we discarded “relative distance y” and used the new parameters “helicopter y” and “wall y.” These new absolute parameters would define the y positions of the helicopter and wall, with each parameter being broken down into three levels. We later found that three levels were still not enough to properly define the states and subsequently increased the number of levels for these parameters to six, and finally twelve.

In order to promote exploration,
we had to devise a clever method of convincing the helicopter to explore new states, without causing it to crash unnecessarily. The nature of Copter doesn’t allow for a typical explore/exploit ratio because random exploration would result in unnecessary crashes once a good policy is found. This problem was solved by initializing all of the Q values at a high number (one), relative to the estimated optimal Q values. This means that, as negative rewards are obtained, the helicopter will tend to choose actions it has not yet tried because they will have higher Q values. This encourages maximum exploration in the beginning, which tapers off as an optimal policy is learned.

Initially, we found the agent sinking limitlessly into the floor as it was exploring the effects of various actions in this state. The agent was always helpless in this scenario, as both up and down actions would be equally bad in the short-term. This meant that the agent did not learn how to escape. Thus, we limited how far the helicopter could sink into the cave after crashing.

With the game finally defined, we were able to create a learning algorithm that could determine a near-optimal policy for the helicopter. We decided to proceed with Q-learning, a simple model-free algorithm that could lead to such a policy. The parameters that we used to define the learning algorithm were \( \alpha \) (the learning rate) of 0.4 and \( \gamma \) (discount factor) of 0.8. The agent was programmed to identify its current state and take the action (up or down) that had the higher Q-value for the particular state.

Later, we tried to get the helicopter to learn faster by tweaking the alpha and discount values. We originally tried modifying a fixed alpha value, but found that a high fixed alpha would have the helicopter forget useful past experience while a low fixed alpha would not have the helicopter learn at all. Thus, we adopted a cooling alpha that would decrease exponentially, allowing the helicopter to learn a lot at first and then gradually base its actions solely on its previous experiences.

We used the Q-learning equation

\[
Q[s, a] := (1 - \alpha)Q[s, a] + \alpha \left( r + \gamma \max_{a'} Q[s', a'] \right)
\]

and computed the value of the states in two tables: the Q-value for the up action and the Q-value for the down action. We assigned every combination of variables defining each state a number which corresponded to the quantitative representation of the state. The state’s number was used to index it in the Q-tables and when the agent was faced with a situation, it used the variable information coming from the environment to compute the state’s number and determine which action in the current state had the greater Q-value and take that action. It would then take the reward for its next state and use that to update the value of the previous state.

Copter still needed some more tuning as taking no action in this game was equivalent to falling due to “gravity.” The helicopter was always forced to push the up key several times in order to remain at the same altitude, causing it to appear very jittery and often hindering its performance.

We realized adding a short delay after deciding on an action would cause the helicopter to commit to the action for a period of time and make the helicopter’s actions more meaningful. Instead of each up action moving the helicopter by only a few pixels, taking an action would now cause the helicopter to “jump”. This allows the
helicopter to change states when it takes actions, ensuring that it knows moving up or down will result in a better reward.

In order to test our variables, we created a list of the cumulative reward at every two hundredth iteration (we used a modulus of two hundred) of the program. We had a variable that stored the cumulative reward and added that value to the list every time a counter variable was divisible by two hundred. We were then able to export and graph the list. Variables that quickened the processing time of the algorithm converged to a smaller negative cumulative reward. We used these graphs to determine the best combination of variables that most efficiently solved the Q-learning algorithm.

4 Results

The agent in the Snitch game created a policy for capturing the Snitch relatively quickly. After approximately thirty failed attempts, it reached the quota (five successful attempts) that we had set for it and stopped exploring random catches. These five attempts were averaged to form an optimal catching distance for the agent, which consequently never missed another Snitch. This exceeded our expectations because averages tend to waver significantly when calculated from few elements of a set. Evidently, the spectrum of possible catching distances was quite narrow, leading to a relatively precise average distance even after a small number of successful catches.

In Copter, the cumulative reward variable and algorithm allows us to experiment with variables and determine which combinations are beneficial to the Q-learning algorithm. We first experimented with the size of the walls, seeing if their thickness had any effect on the agent’s learning. Figure 4.1 shows our results.

![Graph showing cumulative reward over iterations for thick and thin walls.](image)

4.1: This chart shows that over time the cumulative reward for the thick wall and thin wall are similar, but that the thick wall reaches an optimal policy more quickly. (Iterations is an arbitrary value that defines the passage of time)

For the first hundred values, the curve for the algorithm is steep because the agent does not know the consequences of its actions and is exploring all the states. However, the change in cumulative reward decreases as time passes as the agent learns which actions to take.

Surprisingly, Figure 4.1 shows that the cumulative reward for the thicker wall is smaller than that for the thinner wall. We expected the copter to hit the thick wall longer with each crash and get a larger negative reward, but this doesn’t seem to be the case. It appears, however, that due to the helicopter’s bobbing motion, it sometimes randomly hits or misses the wall while in the same state. This causes the agent to learn sporadically and decreases the accuracy of the Q values. The helicopter will not dodge the thick wall randomly, due to its depth, so it correctly learns that the particular state is undesirable.

We were also able to plot the
difference in the cumulative reward between the algorithm with a delay and the algorithm that omitted delay in Figure 4.2.

![Figure 4.2](image)

**Figure 4.2**: This graph shows the massive difference in cumulative reward for when the helicopter delays its actions slightly versus when it does not delay. It is apparent that delaying the actions of the helicopter improves its performance drastically. (Iterations is an arbitrary value that defines the passage of time)

Figure 4.2 shows that the cumulative reward values for the algorithm with no delay are drastically worse than those for the algorithm with a delay. The curve for the algorithm with delay drops sharply initially, but eventually approaches a horizontal asymptote. This shows that the negative reward per iteration is approaching zero and that the Copter agent has found a near-optimal policy. The curve for the algorithm without the delay is almost linear and the slope is steep. This means that the negative reward per iteration isn’t changing, and that the agent is not learning effectively.

5 Related Work

Before Governor’s School we did research from a variety of sources; we all watched an introduction to reinforcement learning by Satinder Singh and read chapters of *Reinforcement Learning: An Introduction* by Richard S. Sutton and Andrew G. Barto and Michael Littman’s thesis, “Algorithms for Sequential Decision Making.”

We were also presented with many different RL problems that have already been studied and experimented with. The first game that we were shown was called Taxi. It was essentially a grid with a few walls, colored blocks, a circle, and a movable colored block. All the agent knew was the functions that it could attempt and that the goal of the game was to exit out of the screen. The agent had to learn, through random actions, the task of transporting the circle from one colored block to another...
and how it should accomplish it.

Other smaller applications of RL include the classic game of Pitfall, a robotic dog that had to learn how to climb over rocky terrain, and an RC helicopter that learned how to fly upside down despite a person trying to turn it right side up.

Among the challenges faced by RL developers today is tackling the prisoners’ dilemma. This is a multiple agent game played by two people, where one prisoner could either “rat out” the other prisoner or help him out. It was a difficult situation because if both prisoners helped the other out they would get a reasonable amount of reward, but if one ratted the other out the tattletale would get a great reward and the prisoner who had been told on would receive a bad reward. Then again, if both prisoners try ratting the other out then each would receive a horrible reward. The game represented the problem of trying to maximize reward without knowing the other prisoner’s actions. The prisoners’ dilemma is similar to problems developers face when they attempt to have multiple agents either cooperate or work against each other. Occasionally, the agents are faced with conflicting decisions in which sometimes they choose to make choices that actually yield much less reward. Finding a solution to this problem is a challenge that RL developers are trying to tackle.

## 6 Conclusion

In essence, the successes and failures of our experiences with Scratch originated from two basic concepts in RL: proper state space and action space definitions. In the Harry Potter game, these concepts were largely overlooked due to the simplicity of the game. The state space of the game was simplified because the agent could only move along one axis and could only press the space bar. The game could have been made more complex by allowing the agent and the “Snitch” to move freely in two dimensions or by adding a second “Snitch” which moved at a different velocity. This is similar to the Copter game because of the dynamic nature of the helicopter and the more complex environment. In the Copter game, we only defined twelve relative levels of height in the environment. We could have easily defined hundreds more which would have refined the decisions of the helicopter, but that would have required sacrificing a significant amount of time for learning. With more powerful computers, we could design a “Daredevil Copter” that would fly as close as possible to each obstacle and fly away at the last instant for stylistic purposes. In terms of gaming applications, our experiences have shown that in the future, it will be possible to design a “perfect opponent” for virtually any non-NP video game. In future homes, RL can be used to program a computer that learns users’ preferences and takes actions to assist the users in their daily tasks. The innovative applications of RL can be applied in several fields through a variety of programming languages and can impact the world in ways never seen before.

## 7 Acknowledgements

We would like to thank our project mentors Carlos Diuk, Thomas Walsh, and Mike Littman and our advisor Jameslevi Schmidt for their help and guidance. We would also like to thank Donald M. Brown the Director and Blase Ur the Program Coordinator.
and Deans Dr. Yogesh Jaluria and Dr. Thomas Farris of the Rutgers University School of Engineering for making the NJ Governor’s School of Engineering and Technology a possibility. We would also like to thank our sponsors Rutgers University, the Rutgers University School of Engineering, the Motorola Foundation, Morgan Stanley, PSEG, Silver Line Building Products, and the families of 2001-2008 program alumni for providing us with this enrichment opportunity. Last but not least, we would like to thank Curtis Giddings for allowing us the use of his computer for collecting data for this project.

References
Appendix A.
Snitch Game Code

Seeker Code

when ⬐ clicked
forever
move 15 steps
wait 0.05 secs
if x position < -295
set x to 300

when ⬐ clicked
go to x: -20 y: -18
set grabbed to 0

when I receive grab
set grabbed to 0
switch to costume reach
wait 0.1 secs
switch to costume grab
wait 0.5 secs
switch to costume grab
wait 0.1 secs
switch to costume fly
Stage Code

Snitch Code

when clicked
point in direction -90°
go to x: 97 y: -82

when clicked
forever
next costume
wait 1 secs

when clicked
forever
move 5 steps
wait (0.05) secs
if x position < -240
set x to 255
Appendix B.
Copter Game Code

Wall Code

when 

set x to 320
set size to 30
go back 1 layers
forever
move 0 speed steps
set dist_x to x position of Wall + 300
if dist_x > 420
set rel_dist_x to 2
else
if dist_x > 150
set rel_dist_x to 1
else
set rel_dist_x to 0
end if
set Wall_y to round y position of Wall
set dist_y to round y position of Wall
set dist_x to round y position of Wall
set last_state to state
set state to 1 + rel_dist_x + Wall_y + 3 + heli_y + 0
if touching edge
set z to 0
set y to pick random 160 to 170
switch to costume costume
else
set rel_dist_x to 0
end if

replace item last_state of 0,0,0 with 0.6 item last_state of 0,0,0 + 0.4 reward + 0.6 item state of 0,0,0
Copter Code

when [ ] clicked
reset timer
switch to costume helicopter
set y to 0
set cum_reward to 0
set move_y to 5
set speed to 1
set axiom to 5
wait 0.2 secs
forever
set coop_y to y position
if up = 1
set y to y position + move_y
else
if y position > -1.6
set y to y position - move_y
if touching color red
set reward to -10
else
if touching color green
set reward to -10
if timer < best
set best to timer
broadcast crashed and wait
else
set reward to 0
set cum_reward to cum_reward + reward

when [ ] clicked
delete all of rewards
set ct to 0
forever
if ct mod 200 = 0
add cum_reward to rewards
set ct to ct + 1