Learning from abstract model in Games

Purvag Patel, Norman Carver, Shahram Rahimi
Department of Computer Science, Southern Illinois University

Abstract – Computer gaming environments are real time, dynamic, and complex, with incomplete knowledge of the world. Agents in such environment require detailed model of the world for learning good policies. Machine learning techniques such as reinforcement learning becomes intractable if they use a detailed model of the world. In this paper we tackle the well-known problem of low convergence speed in reinforcement learning for the detailed model of the world, specifically for video games. We propose training the agents with an abstract model of the world, and use the policy for training the agent with the detailed model of the world. The proposed technique is tested on a classic arcade game called Asteroids. Using this technique we observe that an agent can learn good policy in the abstract model and this policy when used to initialize the detailed model results in high convergence rate.

I. INTRODUCTION

Computer games have improved exponentially in graphics and sound capabilities, and with dedicated hardware for graphics, more CPU power is available for implementing AI techniques. Human players playing against or cooperatively with computer players have certain expectations, such as predictability and unpredictability, support, surprise, winning, losing and losing well [1]. If these expectations are not met, players may become bored and lose interest in the game.

Machine learning technique, such as *reinforcement learning* (RL) can potentially be used to address these concerns. There have been several recent attempts at using machine learning techniques--especially reinforcement learning--for developing learning agents for games [2,3,4,5,6]. The advantages of using machine learning methods are that: (1) this can reduce the time currently required by game developers for configuring crisp parameters of agents, (2) it would allow bots to evolve dynamically as a result of interaction with human players, and (3) performance can continue to improve even after the game is released.

Designing a learning agent for games can be a challenging task and RL has so far seen little use in commercial games. Agents in the game need to constantly interact with the humans, and adapt their game play without any guidance. There are wide varieties of situations to cope with. Even though these agents compete in a simulated environment, games can be close to the complexity of the real world, and can contain similar problems to those faced by robots in real-world applications such as navigation, identifying and selecting objects (weapons), making complex decision (when to hide), etc. Modifications to the environment may be needed for efficient use of an RL algorithm. For example, the choice of weapon used for training can be an issue, since too accurate a weapon might restrict the learning process [3]. It can be daunting to train a bot in a complex environment, because RL algorithms have low convergence rates towards an optimal policy when dealing with large state spaces [4]. One approach that has been tried to make learning game bots practical is to decompose the detailed problem space down in terms of smaller subtasks, and then combine the results into a complete system [7].

We propose a different approach: start smaller state spaces representing abstract models of the world, train the agent using the abstract state space, then use the results from this training as a starting point for training with the larger, more detailed model, hopefully drastically reducing the time required for training of the detailed model. Basically, we use the results from abstract learning to "bootstrap" the detailed learning. The size of the state space for the abstract models can be significantly less than for the detailed models, constructed such that the states in the detailed models are substates of the states in the abstract model. Due to lower state space size, an agent can relatively quickly learn a policy that is optimal for the abstract world model. This policy can then be used as the initial policy for the detailed problem space. The idea is that the abstract policy values get the initial utility values into the right ballpark, resulting in faster convergence with the more detailed models. This approach will also inhibit the agents from behaving randomly during initial game play, and hence, such agents could be provided as reasonable initial agents in the production version of a game.

In previous work [8,6], we demonstrated the effectiveness of a reinforcement learning technique, *Q-learning*, for evolving FPS bots for a testbed inspired from the game of Counter-strike. The primary reason for using a testbed instead of an actual game in both [8,5] was to limit the difficulties encountered trying to integrate RL into an actual game, and to limit the time required to run sufficient experiments (given the acknowledged low RL convergence rates for computer games [7,4]). Results from this research suggested that bootstrapping detailed learning from the results of abstract learning could be effective in dealing with RL convergence problems when developing bots for computer games.

In this paper, we present the results from evaluating our methodology with a second, real game. The game is a version of the classic arcade game Asteroids. In the game, a player controls a battleship in space, surrounded by asteroids. The game play is to destroy as many asteroids as possible. We automated the control of the battleship with a learning agent. The agent again uses Q-learning to improve its decision making. Experiments were conducted on three different state spaces representing increasingly detailed models of the world. The size of state space for the most abstract model is significantly less than that of the most detailed model. Our experiments demonstrate that there is improvement in the convergence speed when using detailed models when we take the policy learned from an abstract model to initialize the policy for the more detailed model.

Section II presents background and related work. Details of the simulation and the algorithm are presented in section III. In section IV we present the results of the experiments that show the effectiveness of the proposed technique, and finally we conclude in section V.

II. BACKGROUND AND RELATED WORK

In this section, we review the importance of learning techniques in computer games and discuss the current state of the art learning techniques used in them.

The goal of using learning techniques in games is not always to design "intelligent" agents capable of taking optimal action in all situations, but to design *believable* agents. Believability plays a major role for the characters in books and movies, even if fiction. Similarly, believability plays a major role in video games, especially with artificial characters. However, it is more challenging to design an artificial character for a video game compared to the characters in books and movies. The characters in games operate in dynamic worlds and are required to interact in real-time. The ability for agents to learn will increase believability of characters, and should be considered an important feature [9].

Hingston [10] organized a game bot programming competition, the BotPrize, in order to find answers to simple questions such as, can artificial intelligence techniques design bots to credibly simulate a human player ?, or simple tweaks and tricks are effective ? Competitors submit a bot in order to pass a "Turing Test for Bots". Bots were judged by a panel of 5 human players who play the game simultaneously with humans and bots. Each judge gave humanness rating to all the players. Bots were able to fool only one judge on average. Human players were having higher humanness rating then bots. It is really difficult to design a bots that can simulate human players. It is relatively easy to identify bots in the system. Some general characteristics used to identify bots were [10]:

- Lack of planning
- Lack of consistency – "forgets" opponents behavior
- Getting "stuck"
- "Static" movement
- Extremely accurate shooting
- Stubbornness

Several cutting-edge and interesting bots were submitted in the tournament, several of which are listed in [10]. Winner of the 2009 competition, *squitebot,* uses long-term memory stored in the SQL database to learn and remember the "hotspots"—places of interesting actions, and return to those places [11]. Other methods are also implemented to prevent the bots from repeating futile actions, avoid being predictable, and inhibit from being too proficient [10]. Bot that stood second, in both 2008 and 2009, used a finite-state machine prominently with two states Battle and Item collection [12]. Bot makes transition between two states under special events such as enemy lost or less energy.

In the tournament, a number of bots made use of learning techniques [10]. Among the learning bots, designed by Wang et. al. [13], stood third in 2008 tournament made use of FALCON and TD-FALCON, a neural network based reinforcement learning and TD-Learning techniques [14,15]. [13] design an agent that could evolve in real time using neural network based architecture (FALCON and TD-FALCON). Key components of the agent are behavior selection and weapon selection. Behavior selection handles running on the map, collecting items, escaping, and engaging in fire. Weapon selection handles selecting weapon to fire based on enemies distance and enemy itself. Contrary to most popular belief that machine learning techniques can be computationally expensive to use for online learning, [13] demonstrated the effectiveness of learning for bots. In this manner hard-coded bots with predictable behavior can be avoided and will evolve an impulsively.

There are several other recent attempts for using machine learning techniques, especially reinforcement learning, for developing a learning bot. [7] suggested a learning algorithm to investigate the extent to which RL could be used to learn basic FPS bots behaviors. The team discovered that using RL over rule-based systems rendered a number of advantages such as: (a) game programmers used minimal code for this particular algorithm, and (b) there was a significant decrease in the time spent for tuning up the parameters [7]. [5] demonstrates several interesting results using RL algorithm, Sarsa (λ) [16], for training the bots. Their experiments demonstrate the successful use of reinforcement learning to a simple FPS game [5].

Several recent efforts include the use of reinforcement learning in actual video game [17,18]. Wang et. al. [17] designed a bot, RL-DOT, for Unreal Tournament domination game. In RL-DOT, the commander NPC makes team decision, and sends orders to other NPC soldiers. RL-DOT uses Q-learning for making policy decision [17]. There are efforts to develop a NPC that would learn to overtake in The Open Racing Car Simulator (TORCS). It is suggested that, using Q-learning sophisticated behaviors, such as overtaking on straight stretch or tight bend, can be learned in a dynamically changing game situation [18].

There are only few attempts for using Neural Networks due to offline training and performance issues. For instance, in certain bots using neural networks for weapon selection and had to be trained using scripted behavior [19]. Similarly, Soni et. al. trained a bot using recorded human actions [20].

Other efforts include the use of genetic algorithm for turning the parameters and evolving different behavior of a bot. Genetic algorithms are relatively easy to implement for bots. Several features such as energy, healthy, distance, etc. necessary for decision making are selected as gene in the chromosomes. Fitness function is used to sort out good chromosomes which are subsequently used for future actions [21]. For example, different fitness function can be used so that bots would evolve with different behaviors such as survival, killing efficiency, or combination of both. Nevertheless, complex fitness function are needed for designing good bots [22]. Parameters for a team can also be tuned using genetic algorithm, wherein each team member contributes the learning of a team [23]. In a highly cited paper, [24] argues that in AI for bots, programmers spend a lot of time hard-coding the parameters of FPS robot controllers (bots) logic and consequently, a considerable amount of time is consumed in the computation of these hard-coded bot parameter values. Therefore, to save the computation and programmer's time, he presented an efficient method of using a genetic algorithm.

[25] implemented a bot that outperforms the existing automated players of a multiplayer real-time strategy game. The bots evolved to perform game actions including synchronizing attacks, assigning targets with intelligence, and usage of influence maps, along with using large-scale fleet movements.

Gamebots is a multi-agent system infrastructure derived from an Internet-based video game which is used as a test-bed for different multi-player games. Many of the bots presented in this section made use of GameBot to design bots for Unreal Tournament [11,26,12,23,22]. Its public availability in U.S. and overseas makes it convenient for game players to access it across the globe [27].

III. METHODOLOGY

Asteroids is a classic arcade game developed by Atari in the late 1970's. It was one of the most popular arcade games of the period. Since then, different variations of the game have been developed for different consoles, such as Playstation, Nintendo 64, etc. Desktop versions of the game have been implemented in different programming languages. Recently the game is being released for major smartphone platforms such as iOS, Windows Phone 7, and Android.

A player in the game controls the battleship in space, surrounded by moving asteroids. The player's goal is to shoot as many asteroids as possible without the ship getting hit by one. Bigger asteroids split into smaller asteroids on being shot by the player. The smallest asteroids vanish when hit by a missile.

For the simulation, we selected a cross-platform version of Asteroids developed in Java. For details on the implementation refer to [28]. A screenshot of the simulation environment is shown in Figure 1. We designed a learning agent to control the battleship. We modified the engine of the battleship, and added extra sensors to the battleship, so that the agent could calculate the distance and rotation angle from the asteroids. The agent can still use the preexisting controls for shooting, changing angles and boost.

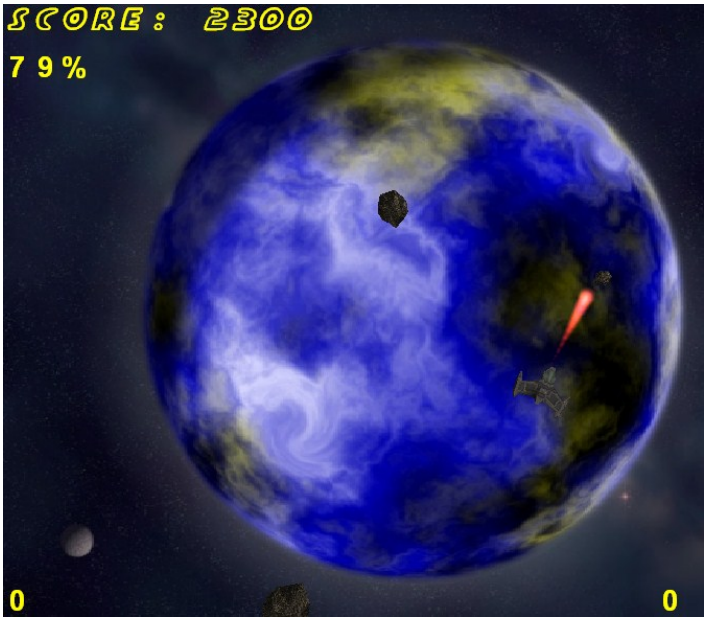


Figure 1 The Asteroids Game

The agent has three actions to choose from:

- $Attack\ (A)$,
- $Boost\ (B)$, or
- $RotateAngle\ (R)$

The agent can shoot the asteroids to destroy them, boost (its speed) to navigate, or rotate the battleship for aiming at an asteroid. The best actions will depend on the relative position and distance of the nearest asteroid from the battleship's shooting angle. Information regarding the position of the nearest asteroids is encoded by the current state. Details of the different state space models are provided in section IV.

An agent using Q-learning learns a mapping for which action he should take when he is in one of the states of the environment. This mapping can be viewed as a table, called a Q-table - $Q(s,a)$, with rows as states of the agent and columns as all the actions an agent can perform in its environment. Values of each cell in a Q-table signify how favorable an action is given that an agent is in a particular state (an estimate of the expected utility for taking the action in the state). The Q-table represents in tabular form the function that gives the quality of each state-action combination: $Q: S \times A \rightarrow R$. To achieve the best possible utility, an agent should select the highest value action for his current state: $arg\,max_a Q(s,a)$.

Actions taken by an agent can affect the environment, which may result in a change of state. Based on his action, the agent's learning component receives a *reward* (a number). Rewards may be positive or negative (negative rewards may be termed *punishments*). The rewards are used by the agent to learn the expected utility values for the state-action pairs. The goal of an agent is to maximize the total reward that he achieves, by learning the actions which are optimal for each state (these actions form the *policy*).

Typically, the Q-table is initialized with random values. Thereafter, each time an agent takes an action and receives a reward, this information is used to update the values in the Q-table. The formula for updating the Q-table is given by:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha_t \times [r_{t+1} + \gamma\ argmax_a Q(s_{t+1},a_t) - Q(s_t,a_t)],$$

where r_t is the reward at any given time t, α_t is the learning rate and γ is discount factor.

Algorithm 1 Q-learning algorithm

```
1   Initialize Q(s,a)
2   for every simulation step
3       s = getPreviousState()
4       a = getPreviousAction()
5       s' = getCurrentState()
6       r = getRewards()
7       Q(s,a) = update(s',s,r)
8       a' = selectAction()
9       execute(a')
10      save (s',a')
11  end for
```

The basic Q-learning algorithm is described above as Algorithm 1. In many environments, it is not possible to immediately determine the rewards for an action an agent took in

the current state (s_t). Therefore, as shown in Algorithm 1, the current state-action pair is stored and is used to update the Q-table once the resulting, next state (s_{t+1}) has been determined. In state s_{t+1} the action taken in previous state s_t is evaluated and rewards are assigned. One of the most difficult issues for RL is when the reward that results from an action is not received for several cycles. An example of this case is when a missile is fired, but it will be two or three time steps in the future before it can be determined if the missile actually hit the target, say at state s_i. In such case, it is typical to update the state at s_i considering $argmax(s_{i-1})$ and the rewards will eventually propagate to s_t (the state in which missile was fired) if learning continues long enough. This can result very slow learning, and is a problem in games, where actions like firing missiles or shooting are the primary source of rewards. In our learning agent, we treat the Attack actions specially, bypassing intermediate states and updating state s_t directly based on the rewards determined in state s_i.

The choice of rewards plays a major role in influencing the behavior that an RL agent learns. We experimented with different rewards, and thereafter used the same rewards structure for all the simulations. Table 1 summarizes the reward structure. Only one event yields a positive reward that is, when a missile hits or destroyed the asteroids. The agents receive a small negative reward for executing any of the three actions. The action $RotateAngle$ is given higher value than $Attack$ and $Boost$ action, since we observed better performance with these settings. We tried different values for rewards, but the relative changes in rewards did not have a significant effect on Q-values; Q-values easily converge irrespectively. It was necessary to have large rewards when a missile hits asteroids and a negative reward for $RotateAngle$ that was less than for $Boost$ in order to ultimately obtain desirable behavior from the agent controlling the ship.

Table 1 Reward Structure

Action	**Rewards**
Missile Hit Asteroid	100
RotateAngle	-5
Attack	-15
Boost	-15
Asteroids Hit Battleship	-20

A single *round* in our simulation experiments lasted 60 seconds, and after each round we reset the environment. The battleship was brought to the initial position, and all the asteroids were placed randomly. Whenever the game was reset, we removed all the existing asteroids and randomly placed four large asteroids in the space. In each round, we continue using the Q-table values learned from the previous round. A single experiment, or "*simulation*," consisted of five rounds. We ran each simulation multiple times since random placement of asteroids leads to different outcomes in each round.

In all the simulation we choose to use a conventional learning rate $\alpha_t = 0.1$, and a discount factor $\gamma = 0.05$. The learning rate, α_t, is progressively decreased by 25% after every round. Such a learning rate and not always suitable for game, but is fine here given that our purpose is only to investigate convergence rate. More discussion about configuring these Q-learning parameters can be found in our previous work [6].

IV. EXPERIMENTS AND RESULTS

A. Learning the Abstract Model

In the initial experiments, we set out to demonstrate that Asteroids agents using Q-learning could learn appropriate policies for shooting at the asteroids even when using a very abstract model of the game world. In addition, the Q-tables learned from this small abstract state space would later be used to initialize the Q-tables for learning larger, more detailed state spaces.

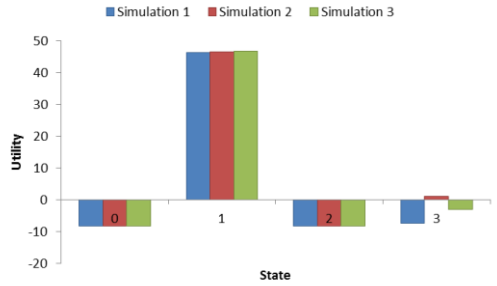
The state space for the most abstract world model consisted of just two attributes: *Distance* and *FireAllowed*. *Distance* represents the distance of the nearest asteroid from the battleship. To keep the state space extremely small, just two discrete values were used: *Near* and *Far*. The physics of the game restricts the number of missiles that can be fired in a second; hence, the Boolean attribute *FireAllowed* is needed to determine whether it is possible to shoot a missile or not in the current decision cycle, allowing the agent to be aware of the limitation exerted by the environment. Such a restriction on the number of shots per second also eliminates the advantage players might get from simply continuously shooting missiles.

The above model of the world results in just four states: cross-product of the *Distance* and *FireAllowed* attributes, each with two possible values. In each state, there are three actions to choose from: $Attack$, $Boost$, or $RotateAngle$. Table 3 in the Appendix summarizes the state space. As described in section V, a single simulation consists of five rounds of 60 seconds each. All the Q-table values are set to zero before the start of a simulation run.

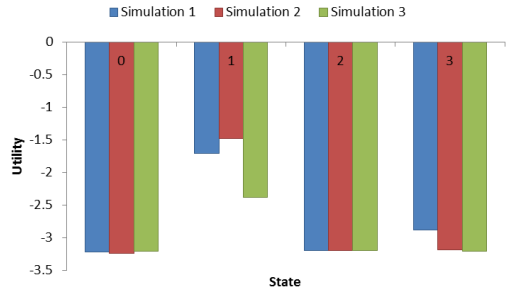
Figure 2 presents the Q-table utility estimates learned in three different simulations. The bars represent the final utility values for each of the four states and the three possible actions. There are only fairly minor differences in the values learned during each of the different simulations, with the exception of the Attack action in state 3 (Figure 2(a)). Although, this utility value is still less than state 3 in Rotate action for simulation 2 (Figure 2(c)), signifying that the policy for simulation 2 is the same as in the other two simulations. These simulations signify that the Q-values are converging to yield identical policies (best actions).

Convergence rates of the Q-values for all the state-action are plotted in Figure 3. There are irregular drops in the Q-values for the Attack action in state 1 (Figure 3(b)), because of the nature of the state. In this state, the asteroids are very close to the battleship, and hence there are higher chances of getting hit. When an asteroid hits the battleship, the agent receives a negative reward, causing sudden changes in the Q-values. Moreover, as the player shoots the asteroids, the larger ones split into two smaller asteroids increasing the chances of getting hit. Even if the agent took the correct action in previous state, that is shooting a missile, there is a fairly high chance that an asteroid

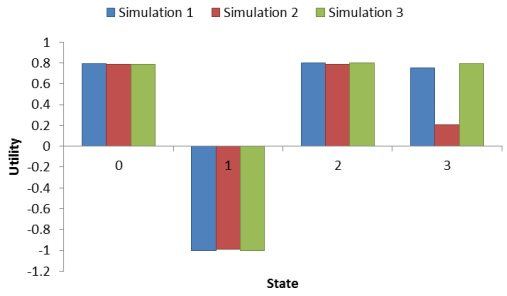
will hit the battleship; hence, we see a dip in Q-values. Similar is the case with the *RotateAngle* action in state 0 (Figure 3 (a)). It is necessary to rotate the battleship for aiming at the asteroid, but it would be difficult to rotate in a timely manner if the asteroid is very close to the battleship. Finally, for the action Attack in the state 3 (Figure 3 (d)) there are abrupt changes in Q-values due to the nature of the rewards. There is small negative reward for firing a missile, but there is a large reward if that missile ends up destroying the asteroid. Even though, a missile will not hit the asteroids that are further away, it might sometimes hit borderline asteroids that are neither close nor far, yielding a large positive reward. Eventually the Q-values seem converges to a negative value as a result of shooting a missile. Most of the Q-values converge by around the 2000th simulation step, with no significant changes in later steps.



(a) Attack Action



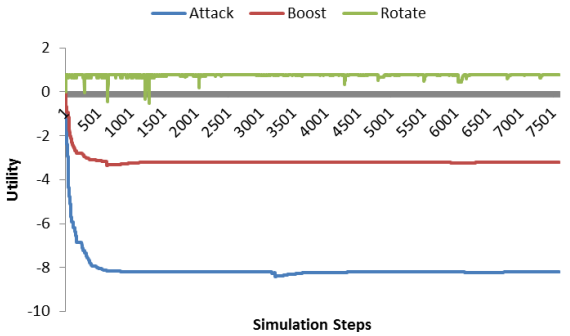
(b) Boost Action



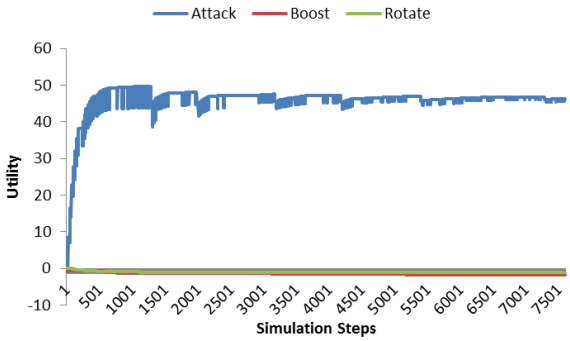
(c) Rotate Action

Figure 2 Policy comparisons for different simulations

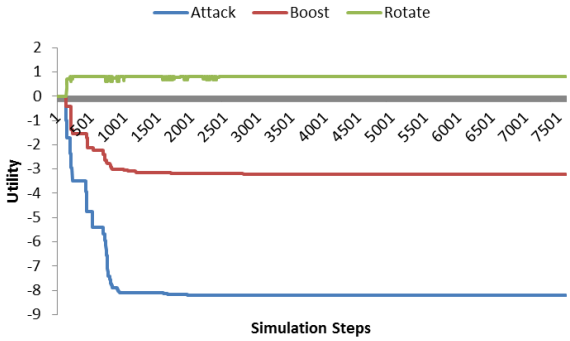
Table 2 shows the final Q-table for the most abstract model. On the whole, the agent learns to Attack when the asteroids are near and they are allowed to attack, and to rotate the ship in order to aim at the rock, otherwise. Notice that values for the *RotateAngle* action are positive, although an agent never receives a positive reward for taking this action (according to the reward structure in table 1). The learned Q-values in Table 2 are used as to initialize the Q-table for the larger state spaces described below in subsection B.



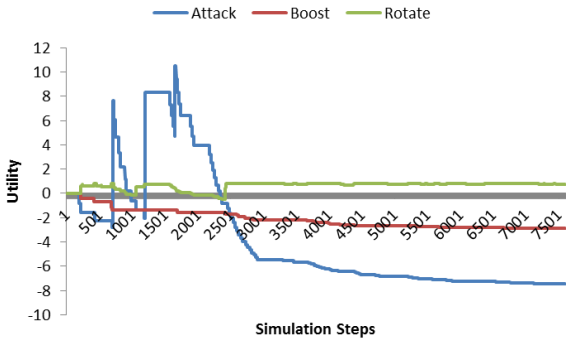
(a) State 0: Distance=Near/FireAllowed=No



(b) State 1: Distance=Near/FireAllowed=Yes



(c) State 2: Distance=Far/FireAllowed=No



(d) State 3: Distance=Far/FireAllowed=Yes

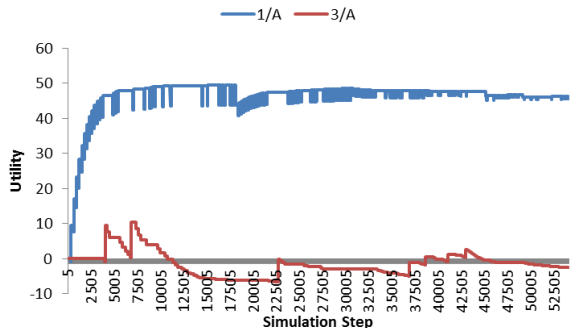
Figure 3 Convergence of q-values for abstract model

Table 2: Final Q-table for Abstract Model

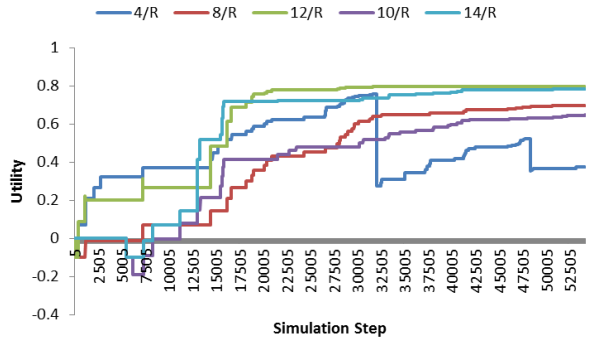
State	*Attack*	*Boost*	*RotateAngle*
0	-8.2071269134	-3.2140404374	0.79140303
1	46.3515048623	-1.7090302259	-0.9996560938
2	-8.1999687967	-3.1998971581	0.7999999523
3	-7.448342897	-2.8768396499	0.7553860125

B. Learning using the Abstract model

The experiments in the previous section demonstrated that starting with a random policy, Q-learning can be used to relatively quickly learn a decent policy for an Asteroids agent with a very modest size state space. However, most games will require much more complex and larger state spaces, potentially resulting in slow learning (and poor game play during the learning process). In this section we will present the results from experiments to evaluate our proposed method, in which the policy (Q-table values) learned in the abstract model are used to initialize the Q-table for a more detailed model of the world, with the goal of speeding up learning with the more detailed model. The more detailed model is derived from the abstract model such that the states in the detailed model are substates of the states in the abstract model. The Q-values from each of the states in the abstract model are copied to their corresponding substates in the detailed model. In the rest of the section, we demonstrate that using this methodology, the Q-values converge much more rapidly for the detailed model, as compared to the case when the agents starts with random initial values.



(a) Convergence of Q-values for *Attack* action



(b) Convergence of Q-values for *RotateAngle* action

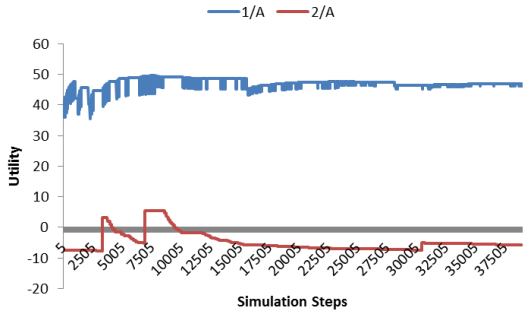
Figure 4 Convergence of q-values for DM1 with Q-table initialized with random values.

We designed two more detailed model, *Detailed Model 1 (DM1)*, and *Detailed Model 2 (DM2)*. Two types of simulations were conducted on both the models: (1) initializing the Q-table with random values, and (2) initializing the Q-table with the values learned policy using the abstract model. All the parameters, such as rewards, were kept same for all the experiments.

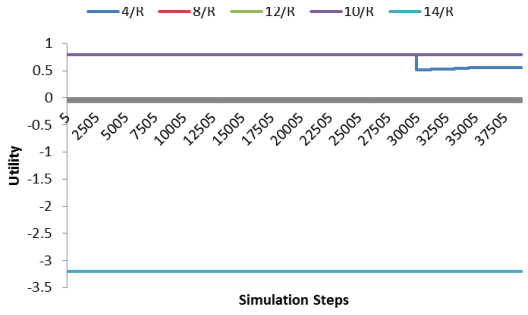
a. DM1: States with Angle

For DM1, the basic state space is expanded in order to include new attribute, called Angle, which indicates the relative rotation angle of the battleship as compared to the closest rock. The *Angle = {Front, Right, Left, Back}* is measured relative to the current rotation angle of the battleship. The game space is divided into four quadrants, and with rotation of battleship the axis of the quadrants also rotates. This results in a state space of size 16: cross-product of *Distance*, *FireAllowed* and *Angle*, as shown in Appendix Table 4. The parent state in Table 4 indicates the super-state from the abstract model.

Figure 4 shows the changes in Q-values over time for the simulation started with a random policy. We have selected seven state/action pairs out of forty-eight pairs for discussion. These are the pairs that either take longer to converge or do not seem to converge. In Figure 4, the notation 1/A and 2/A represents the action *Attack* for the state 1 and 2 respectively. Similarly, 4/R, 8/R, 12/R, 10/R and 14/R represent the *RotateAngle* action for the states 4, 8, 12, 10, and 14 respectively. As seen in Figure 4(a), the pair 1/A seems to converge around 5000th simulation step while, the pair 3/A does not seem to converge. All the pairs for *RotateAngle* action, in figure 4(b), seem to converge after the 40,000th simulation step.



(a) Convergence of Q-values for *Attack* action

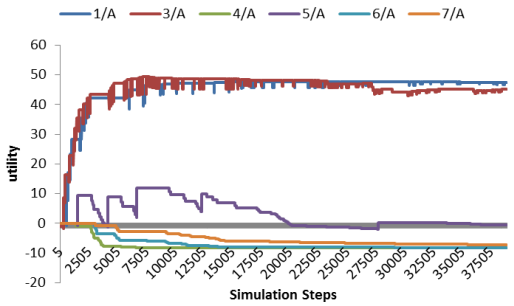


(b) Convergence of Q-values for *RotateAngle* action

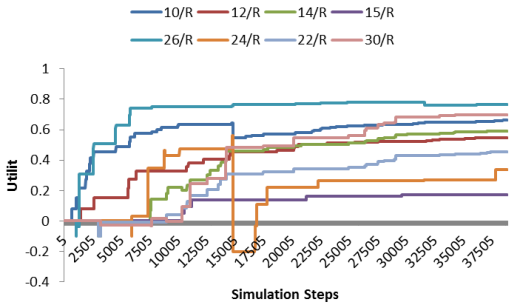
Figure 5 Convergence of q-values for DM1 with Q-table initialized with values from Abstract states

Figure 5 shows the convergence pattern for the simulation started with the policy learned from the abstract model. For comparison all the state/action pair shown in the Figure 5 are the same as the ones shown in Figure 4, but the initial values are obtained from the abstract model. For example, the initial value of the pair 10/R is 0.799999952—the value from the parent state, that is the state 2 for the action *RotateAngle* from Table 2. There is no change in Q-values for most of the actions for the *RotateAangle* angle, as shown in Figure 5(b), signifying that the initially assigned values is a good policy. Similar is the case with the pair 1/A in Figure 4(a), which converges immediately. Notice that the pair 2/A seems to be converging, while in the previous experiment (figure 4(a)) the pair did not converge.

In fact for most the states, no training is required when the Q-table values form the abstract model are used as initial value. So while some training is still required when starting from the abstract model values, significantly fewer cycles are required than to train DM1 from scratch a random policy: convergence of DM1 from a random policy is after the 40,000th simulation step vs. convergence after the 2000th simulation step when starting from the abstract model policy. This is a significant improvement in training time required for DM1.



(a) Convergence of Q-values for *Attack* action



(b) Convergence of Q-values for *RotateAngle* action

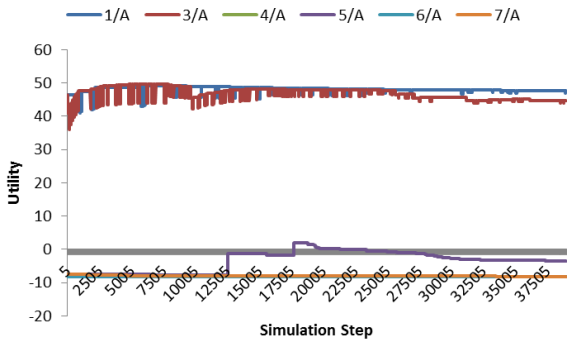
Figure 6 Convergence of q-values DM2 with Q-table initialized with random values.

b. DM2: States with extended Distance

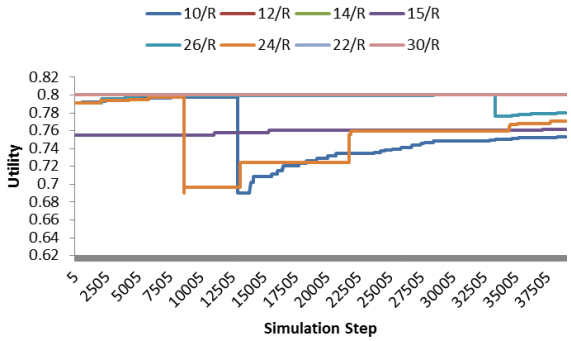
The experiments described in previous section were next repeated with the even larger state space of DM2. The size of the state space for DM2 was increased by increasing the number of svalues for the *Distance* attribute. The *Distance* attribute for DM2 has four values rather than two, to represent the distance of the nearest asteroid from the battleship. The values are: *Near1*, *Near2*, *Far1*, and *Far2*. This increases the size of the state space to 32 (four states from *Distance*, four from *RotateAngle*, and two from *FireAllowed*). The entire state space is summarized in Appendix Table 5. As in the previous section, two types of simulation were conducted: (1) initializing the Q-table with random values, and (2) initializing the Q-table with values learned from the most abstract model.

Figure 6 shows the convergence of Q-values when the Q-table is initialized with random values, while Figure 7 shows convergence when initialized with the values learned from the abstract model.

As shown in Figure 6, Q-values took a longer time to converge when the values were randomly initialized. The values seem to converge after about the 30,000th simulation step for the *RotateAngle* action, which is longer than all the previous simulations. These timings are as expected, since the state space is the largest. We again see an improvement in convergence rate when using the policy learned with the abstract model, as shown in Figure 7. For most of the state-action pairs, almost no learning is necessary, and since abstract model converged after the 2000th simulation step, the actual convergence can be achieved around same time. These speedup results are similar to the results observed with DM1.



(a) Convergence of Q-values for *Attack* action



(b) Convergence of Q-values for *RotateAngle* action

Figure 7 Convergence of q-values DM2 with Q-table initialized with values from Abstract states

Thus, our experiments with the Asteroids game again demonstrate that bootstrapping the Q-learning process using Q-values learned quickly with an abstract world model, imparts a significant boost when learning with more detailed models. The total number of simulation cycles required to end up with a final detailed policy is less than if one directly attempted to learn the detailed policy. Furthermore, since many of the optimal actions for the detailed states remained unchanged from what was found

for the superstates, agent actions during the learning process with the detailed model remain reasonable rather than being random.

V. Conclusion

The computer and video game industry has become a billion dollar industry, which is still growing rapidly. Most modern games utilize some techniques to simulate intelligence. We believe that machine learning method such as reinforcement learning hold much potential for producing intelligent game agents (bots).

However, basic RL techniques will often be intractable (or at least too costly) to apply to games, given that the problem spaces in many games are extremely large and complex. This leads to huge state-action spaces, resulting in slow learning (slow convergence rate). Low convergence rate is not usually acceptable in games since an agent can behave randomly during the game play. We presented a solution to this problem wherein the agents learn the policy for abstract model of the problem space and then use this policy as a starting point with more detailed model.

In our experiments, this approach resulted in significant improvements in the convergence rate when learning more detailed models. For instance, when the policy learned from abstract model with four states was used to initialize the Q-table for the more detailed model with 31 states, learning converged after the 2000th simulation step rather than converging after the 30,000th simulation step (as occurred when initializing the detailed model randomly). Another important advantage was that the agents behaved in a reasonable manner from the onset of the detailed simulation. From this experiment we could conclude that the partially trained agents from abstract states can be delivered to the players, and thereafter the agents continue learning, adjusting to the human player's behavior.

In the experiments reported on here, the abstract and detailed models of the environment were manually designed. In future work we would like to explore how to automate the process of generating the detailed model from abstract states. In addition, it would be interesting to expand the current asteroid game to include multiple players and design cooperating agents that have a shared goal of destroying the asteroids. To conclude, we should leverage the endless opportunities provided by computer games for designing and testing artificial intelligence techniques in the simulated environments that are similar to the real world.

VI. References

[1] Bob Scott, "The Illusion of Intelligence," in *AI Game Programming Wisdom*, Steve Rabin, Ed., 2002, pp. 16-20.

[2] Thore Graepel, Ralf Herbrich, and Julian Gold, "Learning to Fight," in *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004.

[3] Remco Bonse, Ward Kockelkorn, Ruben Smelik, Pim Veelders, and Wilco Moerman, "Learning Agents in Quake III," 2004.

[4] Megan Smith, Stephen Lee-Urban, and Héctor Muñoz-Avila, "RETALIATE: learning winning policies in first-person shooter games," in *Proceedings of the 19th national conference on Innovative applications of artificial intelligence*, 2007, pp. 1801-1806.

[5] Michelle McPartland and Marcus Gallagher, "Reinforcement Learning in First Person Shooter Games," *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, vol. 3, no. 1, pp. 43-56, 2011.

[6] Purvag Patel, Norman Carver, and Shahram Rahimi, "Tuning Computer Gaming Agents using Q-Learning," in *Proceedings of the Federated Conference on Computer Science and Information Systems*, 2011, pp. 581–588.

[7] M. McPartland and M. Gallagher, "Learning to be a Bot: Reinforcement learning in shooter games," in *4th Artifical Intelligence for Interactive Digital Entertainment Conference, Stanford, California*, 2008, pp. 78-83.

[8] Purvag Patel, "Improving Computer Game Bots' behavior using Q-Learning," Master Thesis 2009.

[9] Fabien Tencé, Cédric Buche, Pierre De Loor, and Olivier Marc, "The Challenge of Believability in Video Games: Definitions, Agents Models and Imitation Learning," in *GAMEON-ASIA'2010, France*, 2010, pp. 38-45.

[10] P Hingston, "A new design for a Turing Test for Bots," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on* , 2010, pp. 345-350.

[11] J Cothran. (2009, November) AIGameDev.com. [Online]. http://aigamedev.com/open/articles/sqlite-bot/

[12] D. Hirono and R. Thawonmas, "Implementation of a Human-Like Bot in a First Person Shooter: Second Place Bot at BotPrize 2008," in *CD-ROM Proc. of Asia Simulation Conference 2009 (JSST 2009), Shiga, Japan*, 2009, p. (5 pages).

[13] Di Wang, Budhitama Subagdja, Ah-Hwee Tan, and Gee-Wah Ng, "Creating Human-like Autonomous Players in Real-time First Person Shooter Computer Games," in *Twenty-First IAAI Conference* , 2009.

[14] Ah-Hwee Tan, "FALCON: a fusion architecture for learning, cognition, and navigation," in *IEEE International Joint Conference on Neural Networks*, 2004, pp. 3297-3302.

[15] A H Tan, N Lu, and D Xiao, "Integrating temporal difference methods and self-organizing neural networks for reinforcement learning with delayed evaluative feedback," *IEEE transactions on neural networks*, vol. 19, pp. 230-244, 2004.

[16] G. A. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," technical report no.166 1994.

[17] Hao Wang, Yang Gao, and Xingguo Chen, "RL-DOT: A Reinforcement Learning NPC Team for Playing Domination Games," *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, vol. 2, no. 1, pp. 17-26, 2010.

[18] D. Loiacono, A. Prete, P.L. Lanzi, and L. Cardamone, "Learning to overtake in TORCS using simple reinforcement learning," in *2010 IEEE Congress on Evolutionary Computation (CEC)*, 2010, pp. 1-8.

[19] Stelios Petrakis and Anastasios Tefas, "Neural networks training for weapon selection in first-person shooter games," in *Proceedings of the 20th international conference on Artificial neural networks: Part III*, 2010, pp. 417-422.

[20] B. Soni and P. Hingston, "Bots trained to play like a human are more fun," in *IEEE International Joint Conference on Neural Networks*, 2008, pp. 363-369.

[21] D.M. Bourg and G. Seemann, *AI for Game Developers*.: O'Reilly Media, Inc., 2004.

[22] A.I. Esparcia-Alcázar, A. Martínez-García, A. Mora, J.J. Merelo, and P. García-Sánchez, "Controlling bots in a First Person Shooter game using genetic algorithms," in *2010 IEEE Congress on Evolutionary Computation* , 2010, pp. 1-8.

[23] A.M. Mora et al., "Evolving the Cooperative Behaviour in Unreal Bots," in *2010 IEEE Conference on Computational Intelligence and Games*, 2010, pp. 241-248.

[24] N. Cole, S. J Louis, and C. and Miles, "Using a Genetic Algorithm to Tune First Person Shooter Bot," in *In the Proceedings of the International Congress on Evolutionary Computation.*, 2004, pp. 139-154 Vol. 1.

[25] Robin Baumgarten, Simon Colton, and Mark Morris, "Combining AI Methods for Learning Bots in a Real-Time Strategy Game," *International Journal of Computer Games Technology*, 2009.

[26] Koen V. Hindriks et al., "UnrealGoal Bots Conceptual Design of a Reusable Interface," in *Agents for Games and Simulations II: Lecture Notes in Computer Science*.: Springer Berlin / Heidelberg, 2011, vol. 6525, pp. 1-18.

[27] Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, and Sheila Tejada, "Gamebots: A 3D Virtual World Test-Bed For Multi-Agent Research," in *In Proceedings of the Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, 2001, pp. 47-52.

[28] Kevin Glass. (2012, May) Asteroids - 3D Rendering in Java. [Online]. http://bob.newdawnsoftware.com/~kevin/drupal/asteroidstutorial

VII. APPENDIX

Table 3 State Space for Experiment 1

State Number	State
0	Distance=Near/FireAllowed=No
1	Distance=Near/FireAllowed=Yes
2	Distance=Far/FireAllowed=No
3	Distance=Far/FireAllowed=Yes

Table 4 State space for Detailed Model 1 (DM1)

State Number	Parent State Number	State
0	0	Distance=Near/FireAllowed=No/Angle=Front
1	1	Distance=Near/FireAllowed=Yes/Angle=Front
2	2	Distance=Far/FireAllowed=No/Angle=Front
3	3	Distance=Far/FireAllowed=Yes/Angle=Front
4	0	Distance=Near/FireAllowed=No/Angle=Right
5	1	Distance=Near/FireAllowed=Yes/Angle=Right
6	2	Distance=Far/FireAllowed=No/Angle=Right
7	3	Distance=Far/FireAllowed=Yes/Angle=Right
8	0	Distance=Near/FireAllowed=No/Angle=Back
9	1	Distance=Near/FireAllowed=Yes/Angle=Back
10	2	Distance=Far/FireAllowed=No/Angle=Back
11	3	Distance=Far/FireAllowed=Yes/Angle=Back
12	0	Distance=Near/FireAllowed=No/Angle=Left
13	1	Distance=Near/FireAllowed=Yes/Angle=Left
14	2	Distance=Far/FireAllowed=No/Angle=Left
15	3	Distance=Far/FireAllowed=Yes/Angle=Left

Table 5 State space for Detailed Model 2 (DM2)

State Number	Parent State Number	State
0	0	Distance=Near1/FireAllowed=No/Angle=Front
1	1	Distance=Near1/FireAllowed=Yes/Angle=Front
2	0	Distance=Near2/FireAllowed=No/Angle=Front
3	1	Distance=Near2/FireAllowed=Yes/Angle=Front
4	2	Distance=Far1/FireAllowed=No/Angle=Front
5	3	Distance=Far2/FireAllowed=Yes/Angle=Front
6	2	Distance=Far3/FireAllowed=No/Angle=Front
7	3	Distance=Far4/FireAllowed=Yes/Angle=Front
8	0	Distance=Near1/FireAllowed=No/Angle=Right
9	1	Distance=Near1/FireAllowed=Yes/Angle=Right
10	0	Distance=Near2/FireAllowed=No/Angle=Right
11	1	Distance=Near2/FireAllowed=Yes/Angle=Right
12	2	Distance=Far1/FireAllowed=No/Angle=Right
13	3	Distance=Far2/FireAllowed=Yes/Angle=Right
14	2	Distance=Far3/FireAllowed=No/Angle=Right
15	3	Distance=Far4/FireAllowed=Yes/Angle=Right
16	0	Distance=Near1/FireAllowed=No/Angle=Back
17	1	Distance=Near1/FireAllowed=Yes/Angle=Back
18	0	Distance=Near2/FireAllowed=No/Angle=Back
19	1	Distance=Near2/FireAllowed=Yes/Angle=Back
20	2	Distance=Far1/FireAllowed=No/Angle=Back
21	3	Distance=Far2/FireAllowed=Yes/Angle=Back
22	2	Distance=Far3/FireAllowed=No/Angle=Back
23	3	Distance=Far4/FireAllowed=Yes/Angle=Back
24	0	Distance=Near1/FireAllowed=No/Angle=Left
25	1	Distance=Near1/FireAllowed=Yes/Angle=Left
26	0	Distance=Near2/FireAllowed=No/Angle=Left
27	1	Distance=Near2/FireAllowed=Yes/Angle=Left
28	2	Distance=Far1/FireAllowed=No/Angle=Left
29	3	Distance=Far2/FireAllowed=Yes/Angle=Left
30	2	Distance=Far3/FireAllowed=No/Angle=Left
31	3	Distance=Far4/FireAllowed=Yes/Angle=Left