Continuous and Reinforcement Learning Methods for First-Person Shooter Games

Tony C. Smith and Jonathan Miles

Abstract—**Machine learning is now widely studied as the basis for artificial intelligence systems within computer games. Most existing work focuses on methods for learning static expert systems, typically emphasizing candidate selection. This paper extends this work by exploring the use of continuous and reinforcement learning techniques to develop fully-adaptive game AI for first-person shooter bots. We begin by outlining a framework for learning static control models for tanks within the game BZFlag, then extend that framework using continuous learning techniques that allow computer controlled tanks to adapt to the game style of other players, extending overall playability by thwarting attempts to infer the underlying AI. We further show how reinforcement learning can be used to create bots that learn how to play based solely through trial and error, providing game engineers with a practical means to produce large numbers of bots, each with individual intelligences and unique behaviours; all from a single initial AI model.**

Index Terms—**reinforcement learning, game AI, adaptive control**

1. INTRODUCTION

Machine learning approaches to non-player character (NPC) control have long been cited as the future of game AI (Millington 2006, Palmer 2002). Many methods have been demonstrated to produce reasonably competent static models using neural nets (Thurau et al. 2003, Dawes and Hall 2005), candidate selection decision models (Smith et al. 2007, Thurau et al. 2004), and dynamic scripting (Spronck and Jaap van den Herik 2004), to name but a few. This paper contributes to this area by describing methods to create continuously adapting NPC control models within a first-person shooter (FPS) game. Specifically, we detail experiments of two kinds: a continuous learning approach for creating non-static game AI bots, and a reinforcement learning approach to making bots that by themselves develop individual behaviour models through trial and error, starting from a relatively naive state. The game used for this particular study is BZFlag 2.0.10—a free, open source, cross-platform, multiplayer 3D tank battle game widely used in real-time competitions.

Existing standard paradigms for NPC control are typically based on either *scripting* (where *bot* behaviour is dictated without regard to the current game state) or *rule-sets* (where the NPC executes one of a number of alternative actions or plans based on current conditions during game play). Both approaches rely on a developer's ability to design and codify *a priori* NPC behaviours that deliver satisfying experiences for all gamers—a distinctly challenging and time-consuming task equivalent to expert system development. Even when successful, such static models often succumb to either direct or indirect inference of their underlying algorithm by experienced gamers, who may thereafter take advantage of predictability in bot behaviours, possibly undermining continued enjoyment of the game. The study reported here looks to circumvent this problem by designing bots that continuously adapt in response to game events, allowing them to improve with experience and exhibit some amount of on-going unpredictability that might enhance or extend overall game-play and user enjoyment.

For epistemological reasons, we restrict the set of features made available to the *bot* during learning to just those also available to a human player—such that cheating is not allowed. Moreover, we look only at machine learning methods that can yield human readable models (e.g. no neural networks) so that we can actually see a characterization of what the *bot* has learned at any given time. This also implies a possible practical benefit of allowing game engineers to pre-train bots for a time, then inspect the model and make manual changes that might help a *bot* re-focus toward specific elements of the environment or its experience (with the option to resume learning afterward).

One major benefit specifically offered by a reinforcement learning approach is that it presents a tool to help game developers more easily create individual AI systems for a very large number of bots, such that each *bot* evolves its own distinct playing characteristics. That is, as commercial 3D "virtual world" immersion games continue to grow in complexity and detail (e.g. GTA-IV, Modern Warfare 2, etc), the overhead entailed in developing satisfactory static AI models for a great many different bots and bot-classes becomes onerous. Reinforcement learning makes it possible to create any number of individual bots whose initial state is effectively one of zero-knowledge. These bots can then play each other (at CPU speed) and acquire their own distinct behaviours through trial-and-error. And, of course, any adaptive models can ship with the game, meaning *bot* learning continues for the lifetime of the product.

Three general approaches were trialed and reported here. The first is a basic proof-of-concept attempt to infer a static model that can play BZFlag with satisfactory competence, confirming that machine learning is a viable approach to BZFlag and suggesting which learning schemes

Manuscript received June 30, 2010.

Tony C. Smith and Jonathan Miles are with the Department of Computer Science, University of Waikato, Hamilton, New Zealand (phone: +64-7-838-4453; fax: +64-7-858-5095; e-mail: tcs@cs.waikato.ac.nz).

DOI: 10.5176_2010-2283_1.1.02

© 2010 GSTF

are perhaps best to pursue for adaptive modeling. Following this, continuous learning is employed to show how adaptive behaviour modeling can be realized with simple modifications to static model induction. And third, reinforcement learning is demonstrated as a means to create bots that start with no initial experience and then gradually learn better and better ways to achieve their goals.

2. GAME ENVIRONMENT

The basic game-play of BZFlag is to have two or more tanks whose objective is to shoot each other. BZFlag uses a client-server architecture, though both the client and server programs can run on the same machine. The client is a 'fat client', whereby a large amount of processing is done in the client program while the server program mainly handles synchronization of the game state between multiple clients. The learning component is attached to a *bot* client as an external process so that induction can be carried out as a separate thread.

The world configuration refers to the characteristics of the virtual world created by the server, and includes aspects such as size, obstacles, tank abilities, flags, and game-play modes. Due to the large number of parameters that can be set in BZFlag, only a brief overview of the capabilities is given here.

World size is measured in 'BZFlag units' which have no real-world counterpart (though it is suggested that if the tank was life-sized one BZFlag unit would be approximately one meter). The world size is set for the X and Y coordinate planes, but a constant in the Z axis. The coordinates on all three axes can be positive or negative, so a world with size 200x200 is effectively 400x400 units on the X-Y plane with coordinate values ranging from -200 to +200.

Games in BZFlag are one of two varieties; death-match or capture-the-flag. Our investigation is restricted to death-matches, which are simply free-for-all games where every tank is trying to shoot any other tank. BZFlag comes with two built-in computerized players. We refer to them here as basic-pilot and autopilot. Both use rule-sets to determine their behaviour, though their rule-sets are different. Basic-pilot is the standard computer opponent during single player games. It has simple *dodging* code but overall performs poorly and is easily beaten by a human. Autopilot exists to take over a human player's tank when an in-match break is needed. Its performance is generally better than basic-pilot in that it easily beats basic-pilot in a one-on-one match; but its simple, fixed rule-set creates predictable behaviour that renders it easily beaten by an intermediate human player.

3. LEARNING ENVIRONMENT

This study aims to determine if a computer controlled opponent can adapt to a human player's style of game-play using the same level of information and control that the human player is given (i.e. no cheating). We explore conventional supervised learning schemes and reinforcement learning. The basic process of supervised learning is to provide the learner with a set of training instances (recorded during some prior play), where each instance is a vector of game conditions and an action taken under those conditions. The induction algorithm seeks a terse characterization of those instances (e.g. a decision tree) such that the inferred model is subsequently able to choose an effective action in future and potentially novel circumstances.

The well-known open-source WEKA workbench includes just about every learning scheme in the public domain and is employed for our experiments, both for static model induction and continuous learning. A comprehensive description of WEKA and the algorithms included can be found in Witten & Frank [2005]. Reinforcement learning (RL) is a method that attempts to match a situation (world state) to an action so as to maximize some reward function. Unlike in supervised learning, the learner (agent) is not explicitly told the right action to take at any time but rather learns through trial and error (i.e. no training examples). The lack of known 'correct' examples often results in slower learning than for supervised schemes, but in principle, given sufficient learning time, RL is capable of exploring the entire search space and so is able to find the optimum solution (in the limit).

The underlying approach utilizes state-action pairs, where [notionally] all possible combinations of states and actions are kept in memory along with the accumulated reward for each state-action pair based upon what happened whenever this action was taken from this state in the past. Several reinforcement learning schemes exist to choose from, and we utilize PIQLE (Platform Implementing Q-Learning) in this study. PIQLE is a Java framework designed to separate problems from algorithms, allowing researchers to test new algorithms on standard problems easily. It includes implementations of various RL algorithms (generally those described in Sutton and Barto, 1998), but only the state-action pair algorithm was trialed. It stores all combinations of states and actions along with the maximum expected reward for each, but uses hashing to reduce the memory requirement so that only observed state-action pairs are stored. This approach works well on small or simplified problems such as BZFlag, but has difficulty scaling to more complex domains. That is, memory requirements increase exponentially with the number of states and actions, and all state-action pairs, in principle, must be visited repeatedly in order for the algorithm to converge. PIQLE allows the number of actions available to be set on a state-by-state basis, which is in practice sufficient to keep requirements manageable.

4. CONTROLS AND ATTRIBUTES

BZFlag allows players to control a tank inside the virtual world created by the BZFlag server. For this study, overall control is separated into three distinct classes; speed, shooting, and rotation. Models are inferred for each class and combined together to form a complete *bot* AI. Such separation makes it possible to focus on each decision process independently, simplifying the learning objectives without undermining the overall goal of having adaptive *bot* behaviour.

© 2010 GSTF

Speed is the tank's velocity along the line it is facing and is adjusted by setting a floating-point number representing the fraction of the maximum possible speed. This can be set to a maximum of 1.0 and a minimum of -0.5, with 1.0 being full speed ahead and -0.5 being full speed backwards (the tank can only go half as fast in reverse). Changes to the speed happen virtually instantaneously (that is to say, to the user the acceleration appears to happen instantly).

Shooting is the ability to fire a projectile from the tank. Once fired the projectile continues along a straight-line path until it either hits something (an obstacle, tank, or wall) or reaches its maximum range. A reloading mechanism prohibits the tank from always being able to fire, unlike the speed and rotation controls that are always available. The shooting control is also different from speed and rotation in that it is a binary variable and thus can simply be toggled as needed.

Rotation is the tank's orientation in the virtual world. As with speed this is adjusted by setting a floating-point number representing the fraction of the maximum possible turn speed. It has a maximum of 1.0 and a minimum of -1.0, where 1.0 is turning as fast as possible to the left and -1.0 is turning as fast as possible to the right. Unlike speed however, turning does not happen instantaneously—it takes time for the tank to rotate; approximately 3 seconds to turn 360 degrees.

The training data is gathered from a one-on-one match between the autopilot and the default robot player. The decisions made by the autopilot player are output at each time-step of the game. We concede that the maximum level of skill learnable by the *bot* is potentially limited by the skill of its teacher, and that it would therefore be preferable to learn from expert humans than from either the autopilot or default robot. However, gathering training data from people is somewhat harder; and unnecessary given that we are only demonstrating the overall approach in this paper.

Table 1 itemizes the attributes in the training data. Position attributes (*MyPosition* and *EnemyPosition*) are the absolute position of a tank (autopilot or opponent) on the world axes. The X and Y coordinates have a range of -200 to +200 in our experiments, the Z coordinates have a minimum value of 0 and a maximum of 30 (n.b. there is a server option that allows tanks to jump on top of obstacles in the world). Velocity attributes (*MyVelocity* and *EnemyVelocity*) are the velocities of the tank along the three axes, as given by the attribute's annotation. These have a range of -25 to +25. *EnemyDistance* is the straight-line distance to the opponent's tank in BZFlag units. In the world configuration used here, this has a minimum value of 0 and a maximum of approximately 565 (i.e. maximum Euclidian distance on the diagonal). *AngleDifference* is the difference in angle between the tank's current orientation and the orientation required for it to face straight at the opponent. This is measured in radians and so has a minimum value of 0 and a maximum of approximately 3.14 (just under 180 degrees). *isObscured* is a Boolean value that is true if the opponent's tank is obscured by an obstacle. The *EnemyDistance*, *AngleDifference*, and *isObscured* attributes are all generated by functions that are built-in to the autopilot's logic. The attributes *isObscured* and *Fire* are Boolean values and all the rest are floating-point values.

TABLE I
ATTRIBUTES AND CLASSES

Attribute	Description
MyPositionX	- The position of the autopilot's tank on the X axis
MyPositionY	- The position of the autopilot's tank on the Y axis
MyPositionZ	- The position of the autopilot's tank on the Z axis
MyVelocityX	- The velocity of the autopilot's tank along the X axis
MyVelocityY	- The velocity of the autopilot's tank along the Y axis
MyVelocityZ	- The velocity of the autopilot's tank along the Z axis
EnemyPositionX	- The position of the opponent's tank on the X axis
EnemyPositionY	- The position of the opponent's tank on the Y axis
EnemyPositionZ	- The position of the opponent's tank on the Z axis
EnemyVelocityX	- The velocity of the opponent's tank along the X axis
EnemyVelocityY	- The velocity of the opponent's tank along the Y axis
EnemyVelocityZ	- The velocity of the opponent's tank along the Z axis
EnemyDistance	- The straight-line distance to the opponent's tank.
AngleDifference	- How far the tank must rotate to be facing the opponent tank
isObscured	- *true* if opponent's tank is obscured, *false* otherwise.
Fire (class)	- *true* when a shot is fired, *false* otherwise.
Speed (class)	1.0 = full speed ahead, 0 = stopped,-0.5 = max. reverse speed
Rotation (class)	1.0 = rotate as fast as possible left, -1.0 = as fast as possible right

5. INITIAL STATIC MODELS

Initial static learning was attempted first as proof-of-concept that supervised learning was capable of achieving satisfactory bot behaviour in BZFlag, and to gain insights as to which learning schemes worked best. Separate models were learned to independently control tank speed, the decision to shoot, and the angle of rotation (i.e. direction). An initial dataset of 150,000 training instances was generated, then sampled to obtain random stratified subsets of 1500 examples (i.e. equal numbers of positive and negative instances). Ten-fold cross-validation was used to train and evaluate a wide range of learning schemes available in WEKA, and results from 27 of the more commonly known (and best performing) ones are outlined below.

© 2010 GSTF

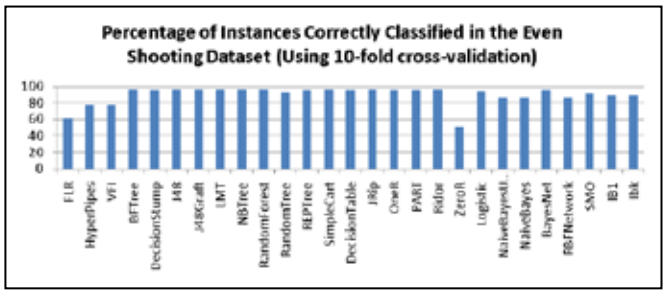

Fig. 1. Classification accuracy for shooting models.

Shooting: The results for the shooting model are shown in Figure 1. Note that ZeroR (i.e. baseline naïve classifier) scores close to 50% as expected, and the majority of the classifiers score over 90%. This suggests that either the problem of controlling tank shooting is a relatively simple one, or over-fitting of the data has occurred. To test for over-fitting, one of the classifiers was inserted into the autopilot *bot* while retaining the original bot code for controlling speed and rotation. The PART classifier was selected because it is a rule-based learner and therefore the model is easily integrated into the autopilot's native IF...THEN rules, and because it has the smallest and least complex set of rules of all the rule-based learners. This modified autopilot *bot* was pitted against the default robot player in a one-on-one best-of-a-hundred death match, with the goal of seeing how well it did in novel situations. Repeated matches produced more or less even performance, suggesting the inferred shooting model was not over-fitting.

Speed control is more complex than shooting because a floating point numeric value must be predicted, rather than a binary value. Very few classification algorithms are able to predict numeric values, and the majority of the classifiers in WEKA must have a nominal class. As per standard procedure, we discretized the class value (i.e. the speed). Recall that changes to tank speed happen almost instantaneously. This greatly simplifies descretization since virtually all values in the dataset are either 1.0 (full speed ahead), 0.0 (stopped), or -0.5 (full speed backwards). This was made effectively uniform for all instances using the Discretize filter available in WEKA (using equal-width binning), such that the bins generated by the filter are; < -0.315415, -0.315415 to 0.342293, and > 0.342293.

Once again, ten-fold cross-validation was used to evaluate many different classifiers in WEKA. Unfortunately space limitations preclude including the analogous graph here (though all results will be presented at the conference). Many of the classifiers scored well over 90%, with JRip performing at nearly 99%. As with shooting, an empirical test was devised to guard against possible over-fitting by inserting the JRip model into the autopilot *bot* (leaving the rest of the original code in place) and having it compete against the default robot player in a series of best-out-of-100 death matches. The modified *bot* averaged 43 kills per match, indicating a successful model had been learned to work in novel situations.

Rotation, like speed, is a numeric value so must be discretized for the learning schemes that require a nominal class. A crude discretization might place each value into one of the three groups; -1.0 (turn left) 0.0 (go straight) and 1.0 (turn right). However, because rotation values in the training data are more evenly spread than the speed values, we empirically settled on hand-set bins of < -0.01, -0.01 – 0.01, and > 0.01. A broad array of learning schemes was once again evaluated under ten-fold cross-validation. Many decision tree models performed well in experimentation (but, once again, space limitations preclude printing them here), with RandomForest winning by a small margin at around 98% accuracy. Empirical tests for over-fitting through the one-on-one death-matches saw RandomForest score an average of 41 kills. REPTree actually scored better in similar tests (averaging 46 kills per match), but RandomForest generally performed better in subsequent death-match tests with bots that had all three controls (shooting, speed and rotation) operated by inferred models.

6. CONTINUOUS LEARNING MODELS

The preceding experiments establish that satisfactory control mechanisms can be successfully learned. But because such models are inferred from a finite set of example instances, their maximum performance tends to be inherently limited (assuming the training sample is not a *characteristic set*). One way to overcome this is to allow the learning algorithm to continue to consider new sample instances (without limit) as it plays. Each instance is annotated with a feature that records the associated outcome, using +1 if a hit is scored on the opponent, -1 if the bot is itself hit, -0.1 if the *bot* is reloading (to slightly penalize just continuously shooting whenever it is possible to do so), and 0 all other times. All decisions made by the bot *and those made by the opponent* are sent to the classifiers, which update their models for shooting, speed and rotation accordingly. If memory (or update time) becomes an issue, older instances (or poorly classified ones) can be deleted from the training set, such that the models are always inferred from the most recent (or most successful) experiences.

The process, shown in Figure 2, is as follows. The *bot* sends world information from BZFlag to the WEKA-Server and awaits a response. All instances received by WEKA-Server are sent to ClassifierBuilder. ClassifierBuilder reruns the learning algorithm and sends the updated model to WEKA-Server, which then issues control instructions to the bot.

Fig. 2. Communication topology for all continuous learning models.

In principle, any ML algorithm can be used to build the classifier. A set of experiments was devised to determine which combination of algorithms (i.e. one each for the rotation, shooting and speed models) showed the best overall trend for improving over time. In each test, the learning *bot* played the robot-pilot in a death-match until the total kill count reached

© 2010 GSTF

300 (chosen arbitrarily). After every ten kills (called a 10-kill block), the number of times the *bot* scored a kill was recorded, creating 30 data points per match. The ability for each algorithm to improve is measured by plotting these data points and comparing the slope of the linear trend lines.

Experiments with a many combinations of learning algorithms indicated that DecisionTable was clearly the best for controlling rotation, improving steadily in every iteration. OneR (a one-level decision tree) always gave the best control results for shooting (although REPTree was close), indicating that the decision of whether to shoot or not is probably quite trivial. Two algorithms, JRip and RandomTree, showed the best trends for improvement when controlling speed, and this can be seen in Figure 3.

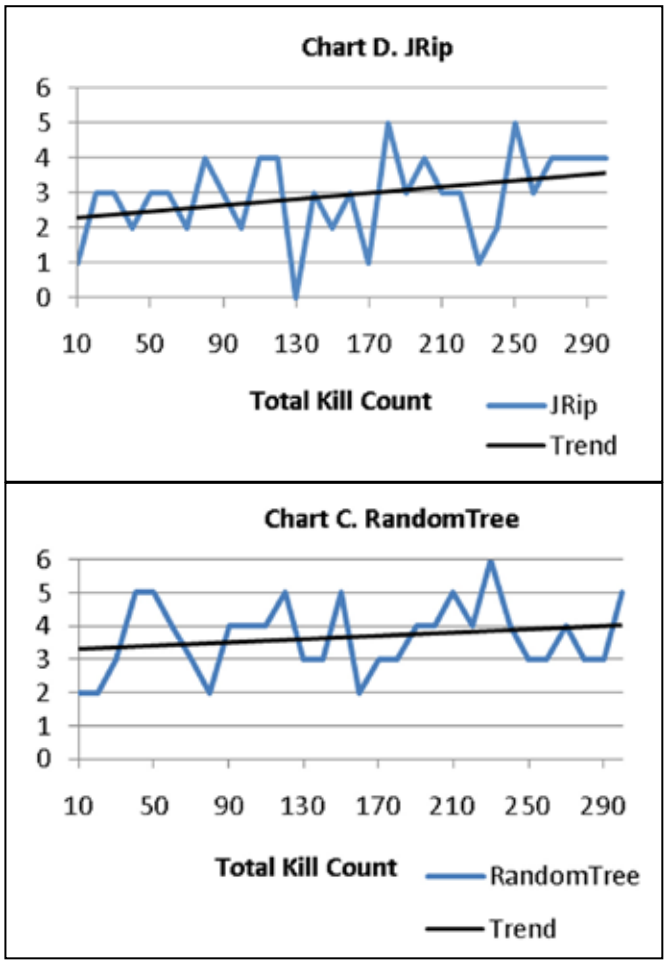

Fig. 3. Trends for two continuous learning bots.

Although RandomTree ultimately achieved a higher average score, JRip showed the best trend towards improvement.

We note that we were unable to get a combination of models to match or exceed the success rate of the robot-pilot. This possibly suggested that the models are at best able to converge on the playing ability of their teacher, which was robot-pilot itself. To breach this limit, the *bot* must be able to explore new behaviours outside of the examples provided by its teacher. Such learning by trial-and-error is the domain of reinforcement learning, discussed in the next section.

7. REINFORCEMENT LEARNING

PIQLE utilizes a so-called Q-learning framework that randomly explores (in a biased way) the behavior space, recording the outcome of chosen actions with some kind of reward or penalty (where penalty is typically just a lesser or negative reward). PIQLE creates a multidimensional table (by hashing to reduce memory demands) where each cell is indexed by a state-action pair and maintains a record of the total reward observed when that action was previously taken in that state. Actions are chosen using random proportionate selection over possible actions from the current state based on the reward each has received. That is to say, selection is biased towards successful action but never precludes any possible action.

At least two reviews are required for every paper submitted. For conference-related papers, the decision to accept or reject a paper is made by the conference editors and publications committee; the recommendations of the referees are advisory only. Undecipherable English is a valid reason for rejection. Authors of rejected papers may revise and resubmit them to the TRANSACTIONS as regular papers, whereupon they will be reviewed by two new referees.

TABLE II
ATTRIBUTES AND CLASSES FOR REINFORCEMENT LEARNING

Attribute	Description
RelativePositionX	The position of the opponent's tank on the X axis, relative to the autopilot's tank
RelativePositionY	The position of the opponent's tank on the Y axis, relative to the autopilot's tank
AngleDifference	The difference between the current rotation of the agent's tank, and the rotation which would point the agent's tank straight at the opponent's tank. (How far the agent's tank must rotate to be facing the opponent tank)
FiringStatus	Integer value, tank can only fire when value is 1 (meaning 'ready')

Table 2 shows the values used to represent world state for the PIQLE agent. This is a much smaller set of attributes than those used previously but should be the minimum data required to accurately distinguish world states. However, using the raw values creates too large a state-action pair space for the Q-learning framework to operate effectively, so substantial rounding is necessary. Specifically, the *RelativePosition* attributes are rounded off to the nearest 40, giving 21 possible values using the world configuration described earlier. The *AngleDifference* attribute is rounded to the nearest 0.5, giving a range of 12 values (0 to +6.0). *FiringStatus* is an integer value with only three possible values so is left unchanged.

Figure 4 shows a graph plotting PIQLE's success rate for 1000-kill blocks over a match that terminates after 100,000 kills. Although the average performance does not

© 2010 GSTF

reach 50%, every test we performed produced a positive trend that continued until memory was exhausted. The implication is that bot performance could conceivably continue to improve until above average performance is attained. Indeed, from a practical perspective, the fundamental problem may turn out to be one of inhibiting the models from getting too good.

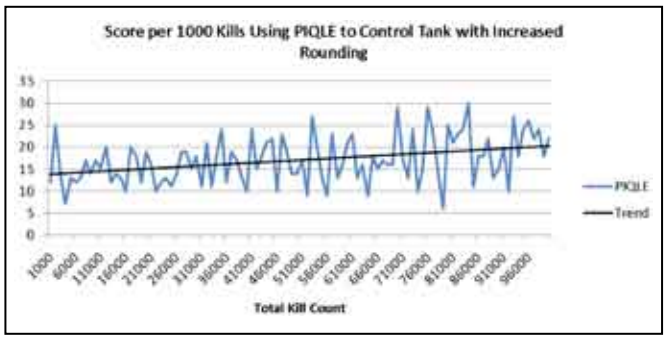

Fig. 4. Positive trend by reinforcement learning.

8. CONCLUSION

We have shown that first-person shooter *bots* can be created using machine learning methods. More importantly, we have demonstrated how continuous learning can be employed to created bots that continuously adapt their behavior in response to their experiences, allowing them to change their actions and responses over time and toward different players. We have also shown how reinforcement learning can be applied to first-person shooter games as a means to make bots that learn how to play a game all by themselves, demonstrating a technology of considerable practical benefit for game engineers in that an unlimited number of bots can now be developed, each with independent and continually adapting behaviours, without the need to code expert game bots at the outset. Overall, these methods can both improve game development and extend gameplay experiences for gamers.

REFERENCES

[1] Thurau, C., Bauckhage, C., & Sagerer, G. Imitation learning at all levels of game-AI. In Proceedings of the International Conference on Computer Games, Artificial Intelligence, Design and Education, 2004, pp. 402–408.

[2] Pieter Spronck and Jaap van den Herik. Game Artificial Intelligence that Adapts to the Human Player, 2004. http://www.ercim.org/publication/Ercim_News/enw57/spronck.html.

[3] Thurau, C., Bauckhage, C., and Sagerer, G. Combining self-organizing maps and multilayer perceptrons to learn bot-behavior for a commercial computer game. In Proceedings of the GAME-ON, 119–123.2003.

[4] Smith, M. Lee-Urban, S. Munoz-Avila, H. RETALIATE: Learning Winning Policies in First-Person Shooter Games, Proceedings of the National Conference on Artificial Intelligence, 2007.

[5] Millington, I. Artificial Intelligence for Games. Series in Interactive 3D Technology. Morgan Kaufmann. ISBN 0-12-497782-0, 2006.

[6] Palmer, Nick. Machine Learning in Games Development, 2002. http://ai-depot.com/GameAI/ Learning.html.

[7] Dawes, Mark and Hall, Richard. Towards Using First-Person Shooter Computer Games as an Artificial Intelligence Testbed. In *Knowledge-Based Intelligent Information and Engineering Systems*, (pp. 276-282), Springer Berlin/Heidelberg. 2005.

[8] Witten, I.H. and Frank, E. 2005. *Data Mining: Practical machine learning tools and techniques*. 2nd Edition, Morgan Kaufmann, San Francisco. 2005.

[9] Sutton, R.S. and Barto, A.G. *Reinforcement Learning: An Introduction*. Bradford Book, The MIT Press, MA. 1998.

Tony C. Smith is a Senior Lecturer in the Department of Computer Science at the University of Waikato, where he is also an active member of the WEKA Machine Learning Group. He is an associate editor of the International Journal on Intelligent Data Analysis, and is the co-founder of several computer technology companies, including Reel Two Inc, SureChem Inc., Rifleman Systems Ltd. and most recently Proofstone Ltd. He is a member of the Language Technology Association and serves on the programme committee for a number of international conferences.

Jonathan Miles recently completed his Masters degree in computer science at Waikato University under the supervision of Dr. Smith.

© 2010 GSTF