

# Learning Winning Policies in Team First-Person Shooter Games

Megan Vasta, Stephen Lee-Urban, Héctor Muñoz-Avila

Department of Computer Science and Engineering  
19 Memorial Drive West  
Lehigh University  
Bethlehem, PA 18015, USA  
{mev2,sml3,hem4}@lehigh.edu

**Abstract:** In this paper we present BLADE, an online reinforcement learning algorithm for developing winning policies in team first-person shooter games. BLADE uses a model for the learning problem that reduces the size of the learning space and a variation of Q-learning that allows the rapid exploration towards a winning policy against an opponent team. Furthermore in our empirical evaluation we demonstrate that BLADE adapts well when the environment changes. BLADE's variation of Q-learning may result in the algorithm not converging to an optimal policy or even not converging at all. We argue that this is needed to be able to obtain a winning policy sufficiently fast for the target domain. We also present results suggesting that traditional discount rates used in Q-learning (i.e., discount rates that lead to convergence) are inadequate in this domain.

## Introduction

Reinforcement learning (RL) has been successfully applied to multiagent systems in many instances such as game theory (Bowling & Veloso, 2002), and RoboCup soccer (Matsubara *et al.*, 1996; Salustowicz *et al.*, 1998). However, RL has seen little use in digital games aside from static, turn-based, fully observable games such as Backgammon. On the other hand, off-line techniques such as decision tree learning have been successfully applied in digital games (AI Wisdom 2, 2002). This lack of application of RL in complex digital games is surprising, given that game practitioners have pointed out the positive impact that on-line learning could have in the creation of more enjoyable games (AI Wisdom 2, 2002). Commercial games currently use predefined difficulty levels, which are recognized as very limited (AI Wisdom 2, 2002). As an alternative approach, RL could be used to dynamically adjust the difficulty level of AI opponents (Spronk *et al.*, 2004).

We conjecture that part of the reason why RL has not been broadly used for computer games is its well-known low convergence speed towards an optimal policy. The task is particularly daunting if we consider team-based first-person shooter (FPS) games. These are very popular kinds of games where teams consisting of two or more players compete to achieve some objectives. In team-based FPS games, individual players must have good reflexes and be able to make decisions on the fly. Also, players must work together to achieve the winning conditions of the game. The game is dynamic in the traditional sense of AI: world state changes occur while players deliberate about what to do. Moreover, FPS games are well-known for changes in the

world state occurring rapidly. Frequently, FPS games are zero-sum games, where one team's gains are the other team's losses.

As a result of these complexities in team-based FPS games, a RL algorithm used in this domain needs to explore both individual player behavior and team behavior as well as possible interactions between the two. Formulating a winning policy depends on the environment, which includes our own players, the opponent team (e.g., tactics), or the world state (e.g. the map where the game is played). The combination of these factors makes the problem of obtaining adequate team strategies even more daunting. A winning policy against one opponent might fail on another. Furthermore, an opponent's team may adjust its tactics to counter a winning policy.

In this paper we present BLADE (for: bounded learning algorithm for domination teams), an online RL algorithm for team-based FPS games. BLADE has the following characteristics:

- In BLADE the behavior of the individual team members is fixed, although this behavior is not explicitly known. This has two consequences. First, individual players work as "plug-ins" that can be easily exchanged. Second, BLADE concentrates on coordinating the team rather than controlling individual player's reactive behavior. Specifically the problem model describes game map locations of interest. This in turn reduces the possible state space that BLADE needs to explore, resulting in a faster formulation of a policy.
- BLADE eliminates traditional discount rates used in Q-learning (i.e., discount rates that leads to convergence). Instead it uses lower and upper bounds for the Q-values (and hence the name "bounded"). As a result, BLADE performs rapid exploration towards a winning policy in our problem model against an opponent team. However, BLADE may not converge, or if it does converge, the resulting policy might not be optimal. We argue that optimality is not needed and rather speed in formulating a winning policy is desired.

We demonstrated BLADE's speed in learning a winning policy using the Unreal Tournament™ (UT) game engine, published by Epic Games Inc. Against a static opponent, BLADE proves capable of developing a winning policy within the first game most of the time and always within two games. We also conducted an experiment showing that BLADE can adapt to a changing environment. Finally, we present results that show that traditional discount rates used in Q-learning are inadequate in our problem model. Furthermore, these results suggest that Q-learning is inadequate for online learning of winning strategies in team FPS games.

The potential commercial applicability of online learning is significant (van Lent *et al.*, 1999). FPS games typically provide a toolset allowing players to design their own maps. Therefore the potential number of different maps that can be generated can be considered, from a practical perspective, unbounded. As such, defining adequate policies in advance and the conditions under which they are applicable is an extremely difficult problem. This problem is compounded by the fact that in online games, teams are formed by combining players of varying capabilities; as a consequence, making a profile of team capabilities, which would allow the selection of adequate policies *a priori*, is almost impossible.

The paper continues as follows. We discuss related work in the next section, including a brief overview of reinforcement learning. Next, we discuss domination games in Unreal Tournament™ (UT), which is the kind of team-based FPS game we

used in our experiments. Following that comes the main section of this paper, which explains our model of the learning problem. Then we discuss our empirical evaluation. We conclude this paper with final remarks.

## **Related Work**

Computer games are considered by some to be the “killer app” for human-level AI, and coordinating behavior is one of many research problems they present (Laird *et al.*, 2001). FPS games are particularly interesting because they present real-time requirements and provide a rich, complex environment. Another advantage is that, unlike the RoboCup domain where an agent’s sensors can fail or provide inaccurate information, FPS games have perfect sensors that never fail. Also FPS games can avoid the need to consider team member communication, as is necessary in work described by Sen *et al.* (1994) and Lauer & Riedmiller (2000). This reduced complexity allows experiments to focus on coordination issues in a way not complicated by noise or communication protocols.

One example of coordinating bots was by Hoang *et al.* (2005). In that study, hierarchical task networks (HTNs) were used to encode strategic behavior that created cohesive behavior in a team of bots. The results demonstrated a clear dominance by the HTN team. In spite of the success, the strategy was encoded *a priori* specifically for the scenarios. No machine learning was involved.

RL has been used with success in certain classes of computer games. One of the most well-known examples is Gerry Tesauro’s implementation of a RL agent, called TD-Gammon, that plays backgammon at a skill-level equal to the world’s best human players. In addition to proving the power of the RL approach, TD-Gammon revolutionized backgammon when it arrived at game strategies previously unexplored by grandmasters, yet found to be equally effective (Tesauro, 1995). However, the backgammon domain is starkly different from the FPS domain in that backgammon is turn-based and the world is static and fully observable. In contrast, FPS games are real-time and the world is dynamic and only partially observable. Both backgammon and FPS games are stochastic.

Spronck & Ponsen (2004) use reinforcement learning to generate AI opponent scripts which can adapt to a player’s behavior in a real-time strategy game. A similar technique is used in a role-playing game (Ponsen *et al.*, 2004). This work differs with ours in the granularity of the learning problem. In Spronck’s work the units are learning individual policies such as which opponent to attack and which weapon or spell to use based upon the individual unit’s capabilities. In our work, BLADE is learning a team policy. Another major difference is the pace of the game. FPS games are well-known for having a faster pace than role-playing and real-time strategy games.

## **Domination Games in UT**

UT is a FPS game in which the usual objective is to shoot and kill an opponent. Players track their health and their weapon’s ammunition, as well as attempt to pick

up various items strewn about the map. Opponents may be other human players via online multiplayer action or computer-controlled bots. An interesting feature of UT is the ability to play several different game variants. One of these variants is a domination game, a feature offered by almost every team-based multiplayer game.

A domination game map has two or more domination locations, each of which can be owned by any team. A team scores points by controlling one or more domination locations. The last team that touches a domination location owns that domination location. For example, in a domination game with two teams and three domination locations, the team that owns two domination locations scores points twice as fast as the team that only owns one domination location. However, if the second team captures one of the first team's domination locations, the scoring reverses and the second team scores points twice as fast as the first team. A domination game ends when one team scores a predetermined number of points, in which case that team is declared the winner.

## RL for Coordinating Teams of Bots

We used reinforcement learning techniques to learn a winning policy to play a domination game in UT.

### Problem Model

A set of states and associated actions model the reinforcement learning problem domain, that we called henceforth the *problem model*. For an UT domination game, the states consist of the Cartesian product  $\prod_i o_i$  of the owner  $o_i$  of the domination location  $i$ . For instance, if there are 3 domination locations, the state (E,F,F) notates the state where the first domination location is owned by the enemy and the other two domination locations are owned by our friendly team. Other parameters were considered to increase the information contained in each state, however, through experiments, we found not only did this simpler definition greatly reduce the state space, but contained sufficient information to develop a winning policy. For three domination locations and two teams, there are twenty-seven unique states of the game, taking into account that domination locations are initially not owned by either team. The initial state is denoted by (N,N,N); indicating that the locations are not owned by any team.

States have an associated set of *team actions*. A team action is defined as the Cartesian product  $\prod_i a_i$  of the *individual action*  $a_i$  that bot  $i$  takes. For example, for a team consisting of three bots, a team action would consist of the Cartesian product of three individual actions. An individual action specifies to which domination location the bot should move. For example, in one team action, the three individual actions could send bot<sub>1</sub> to domination location 1, bot<sub>2</sub> to domination location 2, and bot<sub>3</sub> to domination location 3. A second team action might have individual actions sending all three bots to domination location 1. For a state with three domination locations there are twenty-seven unique team actions because each bot can be sent to three

different locations. If the bot is already in that location, the action is interpreted as to stay in its current location. Individual actions by different bots are executed in parallel.

Despite the simplicity in the representation of our problem model, not only it proves effective but it actually mimics how human teams play domination games. The most common error of novice players in this kind of game is to fight opponents in locations other than the domination ones. This typically leads to a severe warning by more experienced players as these fights generally do not contribute to win these kinds of games. Part of the reason is if a player is killed, it simply reappears in a random location and has no effect in the score. Also players communicate to which domination points other player should go. This is precisely the kind of behavior that our problem model represents.

Before continuing, we refer to *game instance* as one episode where both teams start with 0 points and the initial state is (N,N,N) and ends with one of the teams reaching the predefined number of points.

### The Blade Algorithm

Below we show the top level of the BLADE online learning algorithm. BLADE is designed to run continuously for multiple game instances. This is needed as there is never a final winning policy. Rather, the BLADE-controlled team adapts continuously to changes in the environment. These changes include: changes in our own players, changes in the opponent team (e.g., change of tactics), or changes in the world state (e.g., game instance is played in a new map). BLADE receives as parameters the upper and lower bounds for the Q-table (max and min), the number of games numGameInstances that the algorithm will be run (used in the experiments),  $\epsilon$ : rate of random exploration,  $\alpha_L$ , and  $\alpha_G$  the step-size parameters, which influence the rate of learning.

BLADE(max, min, numGameInstances,  $\epsilon$ ,  $\alpha_L$ ,  $\alpha_G$ )

**input:** max, min: upper and lower Q bounds

**output:** none

- 1 For every state  $S \in \{EEE, \dots, FFF\}$
- 2     For every action  $A \in \{<Go(x), Go(y), Go(z)> \dots\}$
- 3          $Q[S, A] \leftarrow (max + min) / 2$
- 4 For  $i \leftarrow 1$  to numGameInstances
- 5     Q-table[]  $\leftarrow$  runBLADE(max, min,  $\epsilon$ ,  $\alpha_L$ ,  $\alpha_G$ , Q-table[])

Blade starts by initializing the Q-values with a default value defined as the average between min and max (lines 1-3). Then runBlade is called subsequently for each new game instance, passing as parameter the Q-values from the previous run (lines 4-5).

```

runBLADE(max, min, ε, αL, αG, Q-table[1..#states,1..#actions])
input: max, min: upper and lower Q bounds, ε: rate of random exploration,
αL, αG: step-size parameter
output: updated Q-table[1..#states,1..#actions]
1 BeginGame()
2 k ← 0
3 StateActionList[] ← emptyList()
4 While not( EndGame () )
5   Sk ← currentState()
6   Ak ← actionsInState(Sk)
   Choose either:
7     i) Ask ← action A satisfying  $\max_{A \in A_k} Q[S_k, A]$ , with probability 1- ε
     ii) Ask ← random action A in Ak, with probability ε
8   StateActionList[k] ← <Sk, Ask>
9   Q[Sk-1, Ask-1] = bound(min, max, Q[Sk-1, Ask-1] + αL * (U(Sk) – U(Sk-1)))
10  executeAction(Ask) //blocking call
11  k ← k + 1
12 For each <Si, Ai> tuple in StateActionList[]
13   Q[Si, Ai] ← bound(min, max, Q[Si, Ai] + αG * ( ScoreOur – ScoreEnemy))
14 Return Q[]

```

Above we show the runBLADE algorithm, which runs during one game instance. First the game is started (line 1), the iterator k is initialized (line 2), and the StateActionList[] is initialize with the empty vector (line 3). This last variable stores the state that was reached and the action that was selected in each iteration of the algorithm. The following loop (lines 4-11) continues iterating while the game instance is not finished. The current state S<sub>k</sub> is observed and the set of possible actions A<sub>k</sub> for the state S<sub>k</sub> is assigned (line 5-6). Line 7 picks the team action A<sub>sk</sub> to be executed; the team action with the highest Q-value for the given state is chosen with a probability 1-ε, or a random team action from all those available is chosen with a probability ε. The StateActionList[] vector is updated with the team action and state in iteration k (line 8). Line 9 performs a local update on the Q-value. The Q-value for the pair (state,action) from the previous iteration (k-1) is updated (we discuss this formula in detail later on). BLADE executes the selected team action A<sub>sk</sub>. The algorithm stops running while the team action is executed (this is called a “blocking call”). Each bot can either succeed accomplishing its individual action or fail doing so (e.g., the bot is killed before it could accomplish its action). Either way executing a team action takes only a few seconds because the individual actions are executed in parallel. In the last instruction of the loop, the iterator k is augmented (line 11). The last part of the algorithm (line 12-13), a global update is made on the Q-values (We will discuss this formula below).

There are three aspects of the algorithm that we want to discuss in some detail:

- **Utility.** The utility u of state s is defined by the function  $U(s) = F(s) - E(s)$ , where F(s) is the number of friendly domination locations and E(s) is

the number of enemy-controlled domination locations. For example, relative to team A, a state in which team A owns two domination locations and team B owns one domination location has a higher utility than a state in which team A owns only one domination location and team B owns two.

- **Policy.** Step 7 of the runBLADE algorithm the  $\epsilon$ -greedy policy used by BLADE. Most of the time (with probability  $1 - \epsilon$ ) BLADE chooses the action for the current state  $S_k$  that has the maximum Q-value. But with probability  $\epsilon$  BLADE chooses a random action.
- **Local update.** In Step 9 of the runBLADE algorithm, the Q-value for the pair (state,action) from the previous iteration (k-1) is updated with the step-size parameter  $\alpha_L$  multiplied by the difference in utility between the current state and the previous state. If the utility increases, the Q-value are increased, otherwise it is decreased. The Q-value is bound by min and max; if the updated Q-value is larger than max, then max is assigned as the Q-value. Similarly, if the updated Q-value is smaller than min, then min is assigned as the Q-value.
- **Global update.** In Steps 12-13 of the runBLADE algorithm, the Q-values for each pair (state,action) selected during the game instance is updated using the same formula as with local update. The only difference is that the step-size parameter uses is  $\alpha_G$ , and instead of the difference between the utilities, BLADE uses the final score of the game instance. If the game was won (our score is higher than the one from our opponent), the Q-value is increased. Otherwise it is decreased. The idea is to reward actions taken at the various steps for a game won, and punish actions taken for a game lost. Either way, the parameter  $\alpha_G$  is set such that this update is small.

Both the local update and a global updates have the form:

$$Q(s,a) \leftarrow Q(s,a) + \alpha (R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

where  $\gamma = 1$  and  $R(S) = (U(S_k) - U(S_{k-1}) - \max_{a'} Q(s',a') + Q(s,a))$ , for the local update and  $R(S) = (\text{Score}_{\text{Our}} - \text{Score}_{\text{Enemy}} - \max_{a'} Q(s',a') + Q(s,a))$ , for the global update. The local update, which runs when a state transition is calculated after all bots have finished their actions or died in the process, provides a more significant change to the current Q-values based on the immediate state transition resulting from a given action. The global update, which runs at the end of the game instance allows wins and losses to affect Q-values but to a smaller degree than the local updates. One point to note is the diversion from the traditional discounting of rewards by setting  $\gamma = 1$  instead of a value less than 1. While this normally means that there would be no convergence of the algorithm, the Q-values were given artificial upper and lower bounds. This means that while the Q-function will converge for a given opponent on a given map quickly enough to be effective in a single game, a change in opponent strategy or the map will result in an again unstable Q-function and the exploration for a better strategy begins again. The traditional practice in reinforcement learning of setting  $\gamma$  to something less than one was not effective in this highly dynamic domain because a stable convergence was undesirable. The strategy needs the ability to adjust to changing opponents and environments quickly enough to be effective. Our experiments confirms the adequacy of setting  $\gamma = 1$  instead of a value less than 1

## Empirical Evaluation

We conducted a number of experiments to validate the following hypothesis: (1) BLADE is capable of converging to a winning policy quickly, (2) BLADE is capable of adapting to changes in the environment, and (3) traditional discount rates used in Q-learning are inadequate in our problem model.

### Experimental Setup

For this experiment we used the Gamebots distribution (Gamebot, 2005) and the UT game. All bots in this experiment use the same state machine to control their reactive behavior. For example, they fire at an enemy on sight, or pick up an inventory item if near one. This ensures a fair comparison of team strategies as no individual bot is better than the others. Pathfinding is determined by the UT game. While moving, the bots continue to react to game events as determined by their reactive state machine.

**Table 1.** Description of the 4 teams used in testing and training of BLADE.

Team Name	Description
HTNBot	The HTNBot is one described in (Hoang <i>et al.</i> , 2005); it uses HTN planning techniques to maintain control of half plus one of the locations.
OpportunisticBot	Does not coordinate among members whatsoever but each bot actively looks for a domination location to capture it.
PossesiveBot	Each bot is assigned a single domination location that it attempts to capture and hold during the whole game
GreedyBot	Attempts to recapture any location that is taken by the opponent

We used two maps that come with the Gamebots distribution, DOM-Stalwart and DOM-Lament. Each of these maps contains 3 domination locations. For every game instance of each experiment, we recorded the map used, the time taken, the score history, and the number of iterations of runBLADE.

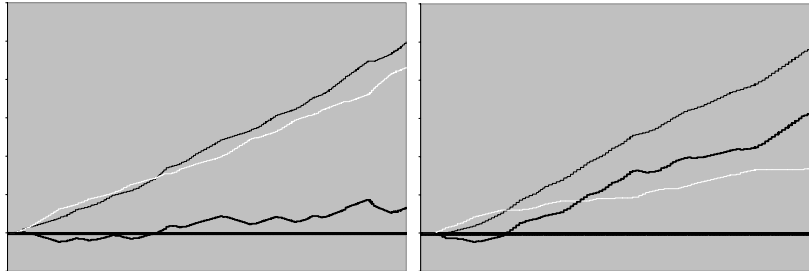
BLADE was called with the following values for the parameters. max: 150, min: 50,  $\epsilon$ : 0.1,  $\alpha_L$ : 0.2,  $\alpha_G$ : 0.8. These parameters were established by doing statistical sampling based on some trial runs.

### Fast Discovery of a Winning Policy

The hypothesis for this experiment is that BLADE is capable of converging to a winning policy quickly. The experiment was conducted on both maps with a new, untrained run of BLADE for each opponent team. An untrained run of BLADE has the iterator  $i$  on Line 4 equals to 1.

In the DOM-Stalwart map, untrained BLADE beat each opponent in the first iteration every time. Figure 1 (a) shows the results of a representative run of the HTNBot team versus the untrained BLADE team. Game instances between BLADE and the HTNBot team had the closest difference between final scores. The thin black

line graphs the score of BLADE, the white line graphs the score of HTNBot, and the thick black line graphs the difference of BLADE's score and the HTNBot score. BLADE began winning about a third of the way through the game after about 100 iterations of  $k$  in runBLADE and won the game 50 to 41.



**Figure 1** (a) HTNBot versus untrained BLADE, and (b) HTNBot versus trained BLADE. Both use the DOM-Stalwart map. The white line is the score for the HTNBots. The thin black line is the score for BLADE. The thick black line indicates the difference in score. The x-axis is time and the y-axis is the score

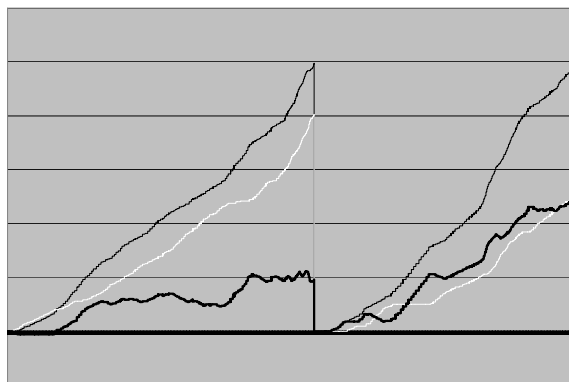
We also run untrained BLADE against the same opponents in the DOM-Lament map. Once again BLADE always beat the three weakest opponents. However, BLADE consistently lost the first game instance against the HTNBot team but always won in the second game instance.

Finally, we ran experiments where BLADE was trained by competing against the 3 weaker opponents in the following order (DOM-Stalwart map): OpportunisticBot, PossesiveBot, GreedyBot, OpportunisticBot, PossesiveBot. In the sixth iteration, BLADE ran against the HTNBots team. A representative last run is shown in Figure 1 (b). In this game instance, BLADE began winning a fifth of the way through the game (after about 60 actions) and defeated the HTNBots opponent 50 to 17, which shows a much stronger win for BLADE.

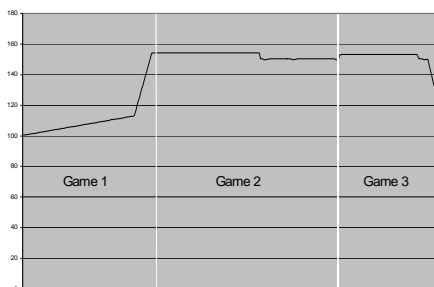
When we compare the plot lines that depict the difference in team scores on Figure 1 (a) and Figure 1 (b) we observe that when untrained, BLADE spends more time punishing its current policy than when trained. This is reflected by the fact that in Figure 1 (a) we observe 8 negatively sloped regions on the line plotting difference in score, whereas in Figure 1 (b) we observe only 3. Therefore it is not surprising that when BLADE is trained, it wins games faster (as evidenced by the larger score difference in this zero-sum game).

### Fast Adaptation to Different Environments

The hypothesis for this experiment is that BLADE is capable of adapting quickly to changes in the environment. We did two experiments to validate this hypothesis. In the first experiment we run two game instances, each on the DOM-Stalwart map: the first game instance against one opponent team, and the subsequent game instance against a different opponent team. For this second game instance, the final Q-values from the previous game instance were retained. In the second experiment, we observe how quickly the Q-value of an action changes when the map is changed.



**Figure 2.** Demonstrates the adaptability of BLADE during two games against two different opponents. Line and axis conventions are as in Figure 1.



**Figure 3** Weight of action 25 in state EFF over 3 game instances

Figure 2 shows a representative result for the first experiment. The first half of the graph plots BLADE score during a game instance against the OpportunisticBot team. The second half of the graph plots BLADE's score during a consecutive game instance against the HTNBots. In spite the environment change, BLADE again won the second game instance.

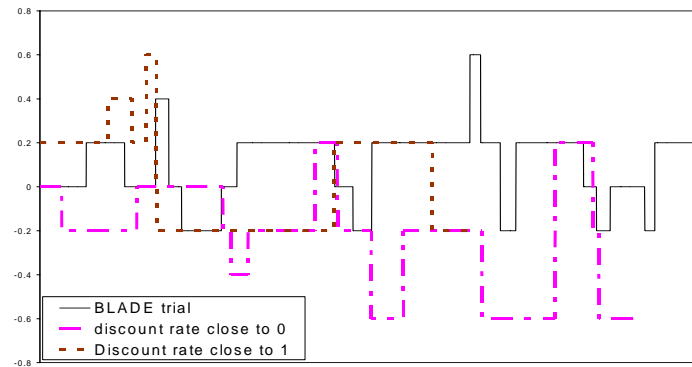
The second experiment was also conducted using DOM-Lament. Figure 3 shows the Q-value of action 25 in the state where the HTNBot team owns the first domination location and the BLADE team owns the last two domination locations (state (E,F,F) ). This team action is defensive; it sends one bot to guard one of the friendly domination locations and the other two bots to guard the second friendly domination locations. In the first game, the Q-value of the team action begins at 100, which is the initial Q-value for all actions. The team action results in a positive game state, so the Q-value is steadily increasing every time BLADE takes it. By the end of the first game, especially through the second global update function, the Q-value for the action has reached the upper bound. Throughout the second game, the action continues to result in a positive outcome most of the time, so the Q-value hovers around the upper bound. In the third game instance, we switched to the DOM-Stalwart map, so the current learned strategy is no longer effective. The Q-value for this (state, action) pair quickly descends towards the initial value.

### Alternative to Traditional Discounted Rewards

The hypothesis for this experiment is that traditional discount rates used in Q-learning are inadequate in our problem model. This experiment was conducted by using discounted rewards according to the formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha (R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

with various values of  $\gamma$  between 0 and 1. We tested on three values of  $\gamma$ : one closer to 0, one closer to 1 and an intermediate one. For these values neither an upper nor a lower bound on the Q-values were needed. In addition we used BLADE as the baseline; that is, the value of  $\gamma$  is set to 1. We ran the algorithm versus the HTNBot team in the DOM-Stalwart map. Figure 4 shows the results but excluding the intermediate value of  $\gamma$  to facilitate readability of the figure. The lines depict the slope of the difference in scores between the RL algorithm and the HTNBot. A value greater than 0 means that the score of the RL algorithm is increasing at a faster rate than that of the HTNBot. A value smaller than 0 means that the score of the RL algorithm is increasing at a slower rate than that of the HTNBot team. For the smallest value of  $\gamma$ , the majority of the running time the value is smaller than 0. For the intermediate value of  $\gamma$  (not shown), the majority of the value is smaller than 0 but not as bad as the smallest  $\gamma$ . For the highest value of  $\gamma$ , around half of the time is below 0 and the other half above. For  $\gamma = 1$  (BLADE), most of the time the value is above 0. Therefore, we conclude that using discount rates with our problem model is not effective to obtain a winning policy. In fact, none of these discount reward algorithms was able to beat HTNBot in the DOM-Stalwart map in the first game instance as opposed to BLADE, which always wins in the first game instance for this map.



**Figure 4** Results from trying different discount rates. Black line shows BLADE results.

### Final Remarks

We presented BLADE, an online reinforcement learning algorithm for team FPS games. BLADE voids discounted rewards in favor of speed in convergence towards a winning policy. This allows BLADE to quickly adapt to changes in the environment,

which is particularly crucial for these kinds of games where the world state can change very rapidly. We present a problem model that is simple yet effective. Our empirical evaluation confirms the effectiveness of BLADE to find winning policies for our problem model and that discounted rewards are not effective for this model.

## References

- Bowling, M. & Veloso, M. Multiagent learning using a variable learning rate. *Artificial Intelligence*, vol. 136, no. 2, pp. 215--250, 2002.
- Gamebot. URL: <http://www.planetunreal.com/gamebots/>. Last viewed: January 24, 2005.
- Hoang, H., Lee-Urban, S., and Muñoz-Avila, H. Hierarchical Plan Representations for Encoding Strategic Game AI. *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*. AAAI Press, 2005
- Kent, T. Multi-Tiered AI Layers and Terrain Analysis for RTS Games. In: *AI Game Programming Wisdom 2*. Charles River Media, 2003.
- Laird, J.E., & van Lent, M. Interactive computer games: Human-level AI's killer application. *AI Magazine*, 22(2), 15-25. 2001
- Lauer, M. & Riedmiller, M. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pp 535-542, 2000.
- Matsubara, H., Noda, I., & Hiraki, K. Learning of cooperative actions in multiagent systems: a case study of pass play in soccer. In *Adaptation, Coevolution and Learning in Multiagent Systems: AAAI Spring Symposium*, pgs. 63--67, 1996.
- Ponsen, M. and Spronck, P. (2004). Improving Adaptive Game AI with Evolutionary Learning. *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)*. pp. 389-396. University of Wolverhampton.
- Reynolds, J. Team Member AI in an FPS. In: *AI Game Programming Wisdom 2*. Charles River Media, 2003.
- Sen, S., Mahendra, S., Hale, J. Learning to Coordinate Without Sharing Information. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 426-431. 1994
- Spronck, P., Sprinkhuizen-Kuyper, I. and Postma, E. 2004. Online Adaptation of Game Opponent AI with Dynamic Scripting. *International Journal of Intelligent Games and Simulation*, Vol. 3, No. 1, University of Wolverhampton and EUROSIS, pp. 45-53.
- Sutton, R. S. & Barto, A. G. *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998.
- Tesauro, G. Temporal Difference Learning and TD-Gammon. In *Communications of the ACM*, March 1995 / Vol. 38, No. 3, 1995.
- van Lent, M., Laird, J. E., Buckman, J., Hartford, J., Houchard, S., Steinkraus, K. and Tedrake, R., Intelligent Agents in Computer Games. *Proceedings of the National Conference on Artificial Intelligence*, pp. 929-930, 1999.
- Watkins, C. J. *Models of Delayed Reinforcement Learning*. Ph.D. thesis, Psychology Department, Cambridge University, Cambridge, UK, 1989.
- Yiskis, E. A Subsumption Architecture for Character-Based Games. In: *AI Game Programming Wisdom 2*. Charles River Media, 2003.