Agent Learning using Action-Dependent Learning Rates in Computer Role-Playing Games

Maria Cutumisu, Duane Szafron, Michael Bowling, Richard S. Sutton

Department of Computing Science, University of Alberta
Edmonton, Canada
{meric, duane, bowling, sutton}@cs.ualberta.ca

Abstract

We introduce the ALeRT (Action-dependent Learning Rates with Trends) algorithm that makes two modifications to the learning rate and one change to the exploration rate of traditional reinforcement learning techniques. Our learning rates are action-dependent and increase or decrease based on trends in reward sequences. Our exploration rate decreases when the agent is learning successfully and increases otherwise. These improvements result in faster learning. We implemented this algorithm in NWScript, a scripting language used by BioWare Corp.'s *Neverwinter Nights* game, with the goal of improving the behaviours of game agents so that they react more intelligently to game events. Our goal is to provide an agent with the ability to (1) *discover* favourable policies in a multi-agent computer role-playing game situation and (2) *adapt* to sudden changes in the environment.

Introduction

An enticing game story relies on non-player characters (NPCs or agents) acting in a believable manner and adapting to ever-increasing demands of players. The best interactive stories have many agents with different purposes, therefore, creating an engaging complex story is challenging. Most games have NPCs with manually scripted actions that lead to repetitive and predictable behaviours. We extend our previous model (Cutumisu et al. 2006) that generates NPC behaviours in computer role-playing games (CRPGs) without manual scripting. The model selects an NPC behaviour based on motivations and perceptions. The model's implementation generates scripting code for BioWare Corp.'s *Neverwinter Nights* (NWN 2008) using a set of *behaviour patterns* built using ScriptEase (ScriptEase 2008), a publicly available tool that generates NWScript code. The generated code is attached to NPCs to define their behaviours. Although ScriptEase supports motivations to select behaviours, a more versatile mechanism is needed to generate *adaptive* behaviours.

A user describes a behaviour motivation in ScriptEase by enumerating attributes and providing them with initial values. Behaviours are selected probabilistically, based on a linear combination of the attribute values and the attributes are updated to express behaviour consequences. For example, the motivation of a guard relies on the duty, tiredness, and threat attributes that control the selection of the patrol, rest, and check behaviours. When patrol is selected, duty is decreased and tiredness and threat are increased. An agent that selects behaviours based only on motivations is not able to quickly discover a successful strategy in a rapidly changing environment. Motivations provide limited memory of past actions and lack information about action order or outcomes.

In this paper, we introduce reinforcement learning (RL) to augment ScriptEase motivations. An agent learns how to map observations to actions in order to maximize a numerical reward signal (Sutton and Barto 1998). Our extension of the ScriptEase behaviour model provides agents with a mechanism to *adapt* to unforeseen changes in the environment by learning. The learning task is complicated by the fact that the agent's optimal policy at any time depends on the policies of the other agents, creating "a situation of learning a moving target" (Bowling and Veloso 2002). More specifically, the learning task is challenging because (1) the game environment changes while the agent is learning (other agents may also change the environment), (2) the other story agents and the player character (PC) may also learn, (3) the other agents may not use or seek optimal strategies, (4) the agent must learn in real-time, making decisions rapidly, especially to recover from adverse situations, because the system targets a real-time CRPG, and (5) the agent must learn and act efficiently, because in most games there are hundreds or thousands of agents. RL is not used in commercial games due to fears that agents can learn unexpected (or wrong) behaviours and because of experience with algorithms that converge too slowly to be useful (Rabin 2003).

We introduce a variation of a single-agent on-line RL algorithm, Sarsa(λ) (Sutton and Barto 1998), as an additional layer to behaviour patterns. To evaluate this approach, we constructed some experiments to evaluate learning rates and adaptability to new situations in a changing game world. Although our goal is to learn general behaviours (such as the guard described earlier), combat provides an objective arena for testing, because it is easy to construct an objective evaluation mechanism. In addition, Spronck (NWN Arena 2008) has provided a pre-built arena combat module for *NWN* that is publicly available and has created learning agents that can be used to evaluate the quality of new learning agents. We evaluate our learning algorithm using this module.

Our experiments show that traditional RL techniques with static or decaying RL parameters do not perform well in this dynamic environment. We identified three key problems using traditional RL techniques in the computer game domain. First, fixed learning rates, or learning rates that decay monotonically, learn too slowly when the environment changes. Second, with action-independent learning rates, the actions that are rarely selected early may be discounted and not "re-discovered" when the environment changes to be more favorable for those actions. Third, a fixed exploration rate is not suitable for dynamic environments.

We modify traditional RL techniques in three ways. First, we identify "safe" opportunities to learn fast. Second, we support action-dependent learning rates. Third, we adjust the exploration rate based on the learning success of the agent. Our agent learns about the effect of actions at different rates as the agent is exposed to situations in which these actions occur. This mirrors nature, where organisms learn the utility of actions when stimuli/experiences produce these actions as opposed to learning the utility of all actions at a global rate that is either fixed, decaying at a fixed rate, or established by the most frequently performed actions. This paper makes the following contributions: (1) provides a mechanism for increasing the learning rate (i.e., the step-size) in RL when prompted by significant changes in the environment; (2) introduces action-dependent learning rates in RL; (3) introduces a mechanism for decreasing the exploration rate when the agent learns successfully and increasing it otherwise; (4) evaluates an implementation of an RL algorithm with these enhancements in the demanding environment of commercial computer games (*NWN*) where it outperforms Spronck's dynamic rule-based approach (Spronck et al. 2004) for adaptation speed; and (5) integrates this RL algorithm into the ScriptEase behaviour code generation.

Related Work

There have been several efforts directed at improving the behaviours of NPCs that have appeared in the literature. Games that use AI methods such as decision trees, neural nets, genetic algorithms and probabilistic methods (e.g., *Creatures* and *Black & White*) use these methods only when they are needed and in combination with deterministic techniques (Bourg and Seemann 2004). RL techniques applied in commercial games are quite rare, because in general it is not trivial to decide on a game state vector and the agents adapt too slowly for online games (Spronck et al. 2003). In massively multiplayer online role-playing games (MMORPGs), a motivated reinforcement learning (MRL) algorithm generates NPCs that can evolve and adapt (Merrick and Maher 2006). The algorithm uses a context-free grammar instead of a state vector to represent the environment. Dynamic scripting (Spronck et al. 2004) is a learning technique that combines rule-based scripting with RL. In this case, the policy is updated by extracting rules from a rule-base and the value function is updated when the effect of an entire sequence of actions can be measured, not after each action. Also, states are encoded in the conditions of the rules in the rule-base. For a fighter NPC, the size of a script is set to five rules selected from a rule-base of 21 rules. If no rule can be activated, a call to the default game AI is added. However, the rules in the rule-base have to be ordered (Timuri et al. 2007), the agent cannot discover other rules that were not included in the rule-base (Spronck et al. 2006), and, as in the MRL case, the agent has only been evaluated against a static opponent.

The ALeRT Algorithm

We introduce a step-size updating mechanism that speeds up learning, a variable action-dependent learning rate, and a mechanism that adjusts the exploration rate into RL. We demonstrate this idea using the Sarsa(λ) algorithm in which the agent learns a *policy* that indicates what *action* it should take for every *state*. The *value function*, $Q(s,a)$, defines the value of action a in state s as the total reward an agent will accumulate in the future starting from that state-action. This value function, $Q(s,a)$, must be learned so that a strategy that picks the best action can be found. At each step the agent maintains an estimate of Q. At the beginning of the learning process, the estimate of $Q(s,a)$ is arbitrarily initialized. At the start of each episode step, the agent determines the state s and selects some action a to be taken using a selection policy. For example, an ε-greedy policy selects the action with the largest estimated $Q(s,a)$ with probability $1 - \varepsilon$ and it selects randomly from all actions with probability ε.

At a step of an episode in state s, the selected action a is performed and the consequences are observed: the immediate reward, r, and the next state s'. The algorithm selects its next action, a', and updates its estimate of $Q(s,a)$ for all s and a using the *ALeRT* algorithm shown in Figure 1, where for simplicity we have used $Q(s,a)$ to denote the estimator of Q. This algorithm is a modified form of the standard Sarsa(λ) algorithm presented in (Sutton and Barto 1998) where changed lines are marked by **.

The error δ can be used to evaluate the action selected in the current state. If δ is positive, it indicates that the action value for this state-action pair should be strengthened for the future. Otherwise it should be weakened. Note that δ is reduced by taking a step of size α toward the target. The step-size parameter, α, reflects the learning rate: a larger α value has a bigger effect on the state-action value. The algorithm is called Sarsa because the update is based on: s, a, r, s', a'. Each episode ends when a terminal state is reached and, on the terminal step, $Q(s,a)$ is zero because the value of the reward from the final state to the end must be zero.

To speed up the estimation of Q, Sarsa(λ) uses *eligibility traces*. Each update depends on the current error combined with traces of past events. Eligibility traces provide a mechanism that assigns positive or negative rewards to past eligible states and actions when future rewards are assigned. For each state, Sarsa(λ) maintains a memory variable, called an *eligibility trace*. The eligibility trace for state-action pair *(s,a)* at any step is a real number denoted *e(s,a)*. When an episode starts, *e(s,a)* is set to 0 for all *s* and *a*. At each step, the eligibility trace for the state-action pair that actually occurs is incremented by 1 divided by the number of active features for that state. The eligibility traces for all states decay by $\gamma * \lambda$, where γ is the discount rate and λ is the trace decay parameter. The value of γ is 1 for episodes that are guaranteed to end in a finite number of steps (this is true in our game).

```
Initialize $\vec{\theta}$ arbitrarily
** Initialize $\alpha(a) \leftarrow \alpha_{\max}$ for all $a$
Repeat (for each episode)
  $\vec{e} = \vec{0}$
  $s, a \leftarrow$ initial state and action of episode
  $\mathcal{F}_a \leftarrow$ set of features present in $s, a$
  Repeat (for each step of episode)
    For all $i \in \mathcal{F}_a$:
**    $e(i) \leftarrow e(i) + \frac{1}{\|\mathcal{F}_a\|}$ (accummulating traces)
    Take action $a$, observe reward, $r$, and next state, $s$
    $\delta \leftarrow r - \sum_{i \in \mathcal{F}_a} \theta(i)$
    With probability 1-$\varepsilon$:
      $\forall a \in \mathcal{A}(s)$:
        $\mathcal{F}_a \leftarrow$ set of features present in $s, a$
        $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
      $a \leftarrow argmax_a Q_a$
    or with probability $\varepsilon$:
      $a \leftarrow$ a random action $\in \mathcal{A}(s)$
      $\mathcal{F}_a \leftarrow$ set of features present in $s, a$
      $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
    $\delta \leftarrow \delta + \gamma Q_a$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha(a)\delta\vec{e}$
    $\vec{e} = \gamma\lambda\vec{e}$
**  $\Delta\alpha = \frac{\alpha_{\max} - \alpha_{\min}}{\alpha_{\text{steps}}}$
**  if $\overline{\delta(a)}\delta > 0 \wedge \left|\overline{\delta(a)} - \mu_{\overline{\delta(a)}}\right| > f\sigma_{\overline{\delta(a)}}$
**    $\alpha(a) \leftarrow \alpha(a) + \Delta$
**  else if $\overline{\delta(a)}\delta \le 0$
**    $\alpha(a) \leftarrow \alpha(a) - \Delta$
  end of step
until $s$ is terminal
** $\Delta\varepsilon = \frac{\varepsilon_{\max} - \varepsilon_{\min}}{\varepsilon_{\text{steps}}}$
** if $\sum_{step} r_{step} = 1$
**  $\varepsilon = \varepsilon - \Delta\varepsilon$
else
**  $\varepsilon = \varepsilon + \Delta\varepsilon$
end of episode
```

Figure 1. The ALeRT algorithm.

For example, assume $\gamma = 1$, $\lambda = 0.5$, and the *melee* action is taken in state *s* at some step. Assume that the state has three active binary features, then *e(s,melee)* = 1/3 and the eligibility traces for the rest of the actions in state *s* are zero. In the next step, if (s,*melee*) does not occur again, its eligibility trace decays to $\lambda = 0.5$ and in the next step it further decays to $\lambda^2 = 0.25$.

Traditionally, α has either been a fixed value or decayed at a rate that guarantees convergence in an application that tries to learn an optimal static strategy. As stated in the introduction, one of the two main issues with using RL in computer games is the slow learning rate in a dynamic environment. In a game environment, the learning algorithm should not converge to an optimal static strategy because the changing environment can make this static strategy obsolete (not optimal any longer).

The first change we make to traditional RL algorithms is to speed up the learning rate when there is a recognizable trend in the environment and slow it down when the environment is stable. This supports fast learning when necessary, but reduces variance due to chance in stable situations. The problem is to determine a good time to increase or decrease the step-size α. When the environment changes enough to perturb the best policy, the estimator of *Q* for the new best action in a given state will change its value. The RL algorithm will adapt by generating a sequence of positive δ values, as the estimator of *Q* continually underestimates the reward for the new best action until the estimate of *Q* has been modified enough to identify the new best action. However, due to non-determinism, the δ values will not be monotonic. For example, if a new powerful range weapon has been obtained, then the best new action in a combat situation may be a *range* attack instead of a *melee* attack. However, the damage done/taken each round varies due to non-determinism. The trend for the new best action will be positive, but there may be some negative values.

Conversely, when the environment is stable and the policy has already determined the best action, there is no new best action, so the sequence of δ values will have random signs. In this case, no trend exists. When a trend is detected, we increase α to revise our estimate of *Q* faster. When there is no trend, we decrease α to reduce the variance in a stable environment. We recognize the trend using a technique based on the Delta-Bar-Delta measure (Sutton 1992). We compute *delta-bar*, the average value of δ over a window of previous steps that used that action, and then compute the product of the current δ with delta-bar. When the product is positive, there is a positive correlation between the current δ and the trend, so we increase α to learn the new policy faster. When the product is negative, we reduce α to lower variance. However, we modified this approach to ensure that variance remains low and to accommodate situations where the best policy may not be able to attain a tie (non-fair situations for our agent).

We define a *significant trend* when delta-bar differs from the average delta-bar ($\mu_{\bar{\delta}(a)}$) by more than a factor *f* times the standard deviation of delta-bar ($f * \sigma_{\bar{\delta}(a)}$). The average and standard deviation of delta-bar for that action are computed over the entire set of episodes. Initially, $\alpha = \alpha_{max}$, because we want a high step-size when the best policy is unknown. In the stable case (no trend), α decreases to reduce variance so that the estimate of Q does not change significantly. However, α does not get smaller than α_{min}, because we want to be able to respond to future changes in the environment that may alter the best policy. If we identify a trend that is not significant, we do not change α. If there is a significant trend, we increase α to learn faster. We change α using fixed size steps ($\alpha_{steps} = 20$) between α_{min} and α_{max}, although step-sizes proportional to delta-bar would also be reasonable.

The second change we make to traditional RL techniques is to introduce a separate step-size *α(a)* for each action *a*. This allows the learning algorithm to establish separate trends for each action to accommodate situations in which a new best action for a particular state replaces an old best action for that state, while a different action for a different state remains unaffected. For example, when an agent in *NWN* acquires a better range weapon, the agent may learn quickly to take a range action in the second step of the combat instead of taking a melee action because the α(*range*) value is increased due to trend detection. The agent may still correctly take a speed potion in the first step of a combat episode because the α(*speed*) parameter is unaffected by the trend affecting α(*range*). We expect α(*a*) to converge to a small value when the agent has settled on a best policy. Otherwise, α(*a*) will be elevated, so that the agent follows the trend and learns fast.

An elevated α value often indicates a rare action whose infrequent use has not yet decayed its step-size from its high value at the start of training. The agent's memory has faded with regards to the effect of this action. When this rare action is used, its high α value serves to recall that little is known about this action and its current utility is judged on its immediate merit. This situation is reminiscent of the start of training when no bias has been introduced for any action. In fact, as shown in Figure 1, we combine action dependent *alphas* with trend-based *alphas* in that there is a separate delta-bar for each action.

Our trend approach to the step-size parameter (α) is consistent with the WoLF principle of "learn quickly while losing, slowly while winning" (Bowling and Veloso 2001). We also use this principle in our third change to traditional RL techniques. We vary the exploration parameter (ε) in fixed steps ($\varepsilon_{steps} = 15$) between ε_{min} and ε_{max}. Initially, $\varepsilon = \varepsilon_{max}$ for substantial exploration at the start. Then, ε increases after a loss to explore more, searching for a successful policy, and decreases after a win, when the agent does not need to explore as much. Exploration is necessary to discover the optimal strategy in dynamic environments, therefore, we set a lower bound for ε.

NWN Implementation

We define each NPC to be an *agent* in the *environment* (*NWN* game), controlled by the computer, whereas the PC (player character) is controlled by the player. The NPCs respond to a set of game events (e.g., *OnCombatRoundEnd* or *OnDeath*). If an event is triggered and the NPC has a script for that event, then that script is executed.

NWN combat is a zero-sum game (one agent's losses are the opponent's wins). We define an *episode* as a fight between two NPCs that *starts* when the NPCs are spawned and *ends* as soon as one or both or the opponents are dead. We define a *step* as a combat round (i.e., a game unit of time that lasts six seconds) during which the NPC must decide what action to select and then execute the action.

We define a *policy* as the agent's behaviour at a given time. A policy is a rule that tells the agent what action to take for every state of the game. In our case, the policy is selected by estimating our Q_a using a $\vec{\theta}$ that has one feature for each state-action pair. The *state space* consists of five Boolean features: 1) the agent's HP are lower than half of the initial HP and the agent has a potion of heal available; 2) the agent has an enhancement potion available; 3) the agent has an enhancement potion active; 4) the distance between the NPCs is within the melee range, and 5) a constant. The *action space* consists of four actions: *melee*, *ranged*, *heal*, and *speed*. Therefore, there are 20 features in $\vec{\theta}$, one for each state-action pair. For example, if features 1, 3 and 5 are active and a *melee* action is taken, three components of $\vec{e}$ will be updated in the step: *1-melee*, *3-melee* and *5-melee*. These updates will influence the same components of $\vec{\theta}$ and will affect the estimator of Q_{melee} in future steps.

When the agent is in a particular game state, the action is chosen based on the estimated values of Q_a using an ε-greedy policy. When we exploit (choose the action with the maximum estimated Q_a for the current state) and there is a tie for the maximum value, we randomly select among these actions. We explore using a uniformly random approach (irrespective of the estimated Q_a values).

We define the *score* of a single episode as 1 if the agent wins the episode and -1 if it loses. The agent's goal is to win a fight consisting of many consecutive episodes. We define the immediate reward, *r*, at the end of each step of each episode as: *r = 2*[HPs' /(HPs' + HPo') – HPs/(HPs + HPo)]*, where the subscript *s* represents the agent (self), the subscript *o* represents the opponent, a prime (') denotes a value after the action and a non-prime represents a value before the action. Note that the sum of all the immediate rewards during an episode amounts to 1 when our agent wins and -1 when our agent loses. Moreover, the sum of all the rewards throughout the game amounts to the difference in episode wins and losses during the game.

Experiments and Evaluation

We used Spronck's *NWN* combat module to run experiments between two competing agents. Each agent was scripted with one strategy from a set of seven strategies. NWN is the default NWN agent, a rule-based probabilistic strategy that suffers from several flaws. For example, if an agent starts with a sword equipped, it only selects between *melee* and *heal*, never from *ranged* or *speed*. RL_0, RL_3, and RL_5 are traditional Sarsa(λ) dynamic learning agents with $\alpha = 0.1$, $\varepsilon = 0.01$, $\gamma = 1$, $\lambda = 0$, $\lambda = 0.3$, and $\lambda = 0.5$ respectively. ALeRT is the agent that uses our new strategy with action-dependent step-sizes that vary based on trends, with the parameters initially set to $\alpha = 0.2$, $\varepsilon = 0.02$, $\lambda = 0$ (fixed), and $\gamma = 1$. M1 is Spronck's dynamic scripting agent (learning method 1), a rule-based strategy inspired by RL, called dynamic scripting, that uses NWN version 1.61. OPT is a hand coded optimal strategy, based on the available equipment, which we created.

Each experiment consisted of 50 trials and each trial consisted of either one or two phases of 500 episodes. At the start of each phase, the agent was equipped with a specific configuration of equipment. We created the phases by changing each agent's equipment configuration at the phase boundary. In the first phase, we evaluated how quickly and how well an agent was able to *learn* a winning strategy from a starting point of zero knowledge. In the second phase, we evaluated how quickly and how well an agent could *adapt* to sudden changes in the environment and discover a new strategy. In essence, the agent must overcome a bias towards one policy to learn the new policy. Each equipment configuration (*Melee*, *Ranged*, *Heal*) has an optimal action sequence. For example, the melee weapon in the *Melee* configuration does much more damage than the ranged weapon so the optimal strategy uses the melee weapon rather than the ranged one. The optimal strategy for the *Melee* configuration is *speed*, followed by repeated *melee* actions until the episode is finished. Similarly, the *Heal* configuration has a potion that heals a greater amount of HP than the healing potions in the other two configurations so that the optimal strategy is *speed*, repeated *melee* actions until the agent's HP are less than half of the initial value, *heal*, and repeated *melee* actions. We recorded the average number of wins for each opponent for each group of 50 episodes. The two competing agents had exactly the same equipment configuration and game statistics. However, the agents had different scripts that controlled their behaviours. In the experiments illustrated in Figure 2 and Figure 3, we ran only a single-phase experiment.

Each data point in a graph represents an agent's winning percentage against its opponent after each group of fifty episodes. The x-axis indicates the episode and the y-axis indicates the average winning percentage for that episode. For example, the data point at x=200 is the average winning percentage of all of the episodes between episode 151 and episode 200 over all trials for that experiment.

Motivation for ALeRT

Originally, we thought that traditional RL could be used for agents in NWN. In fact, RL_0 defeated NWN as shown in Figure 2. RL agents with other RL parameter values also defeated NWN. However, although we tried many different parameter values, we could not get RL to converge to the optimal strategy against OPT. For example, Figure 2 shows that RL_0 and RL_3 could attain only 34% and 38% wins respectively against OPT with the *Melee* configuration after 500 episodes, and 41% and 40% wins respectively for *Ranged*. Experiments with various other parameter values did not yield better results. For example, RL_5 did better than RL_0 and RL_3 for *Melee*, but did worse for *Ranged*. It was clear that we had to change the Sarsa(λ) algorithm in a more fundamental manner.

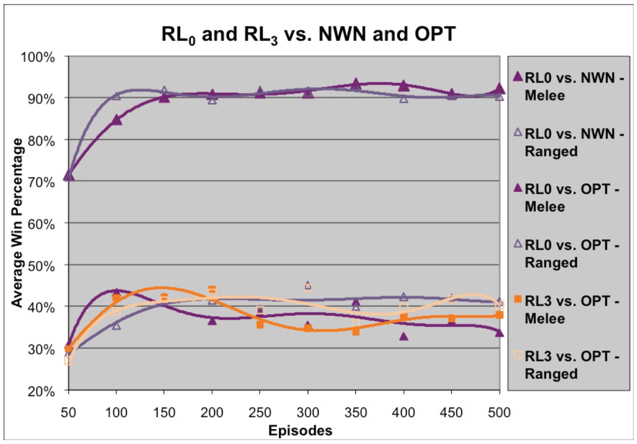

Figure 2. Motivation for ALeRT: RL_0 and RL_3 vs. NWN (upper traces) and OPT (lower traces).

It is important for a learning agent to be able to approach the skill level of any agent, even an optimal one. If the learning agent is the opponent of a PC that has a near optimal strategy, the learning agent should provide a challenge. If the learning agent is a companion of the PC and their opponents are using excellent strategies, the player will be disappointed if the companion agent causes the PC's team to fail. Therefore, we developed the action-dependent step-sizes in the ALeRT algorithm to overcome this limitation. ALeRT and RL defeated NWN, although after 500 episodes ALeRT won 70% for *Melee* and 78% for *Ranged* (Figure 3), while RL_0 won 92% and 90% (Figure 2). However, ALeRT's behaviour is more suitable for a computer game, where it is not necessary (and usually not desirable) for a learning agent to crush either a PC or another NPC agent by a large margin. The important point is that ALeRT converged to the optimal strategies for *Melee* and *Ranged* configurations against OPT, while RL_0 and RL_3 did not converge. The action-dependent step-sizes in ALeRT are responsible for this convergence.

Figure 3 shows traces of M1 (Spronck's learning method 1) versus NWN and M1 versus OPT. M1 defeated NWN by a large margin, 94% for *Melee* and 90% for *Ranged*. These winning rates were more than 20% higher than ALeRT's winning rates, but as stated before, winning by a large margin is not usually desirable. M1 converged to the optimal strategy against OPT for the *Melee* configuration. However, as shown in Figure 3, M1 converged much more slowly than ALeRT. The latter achieved a win rate of 48% after the first 100 episodes and did not drop below 46% after that. M1 won only 30% at episode 100, 40% at episode 200 and did not reach 46% until episode 450.

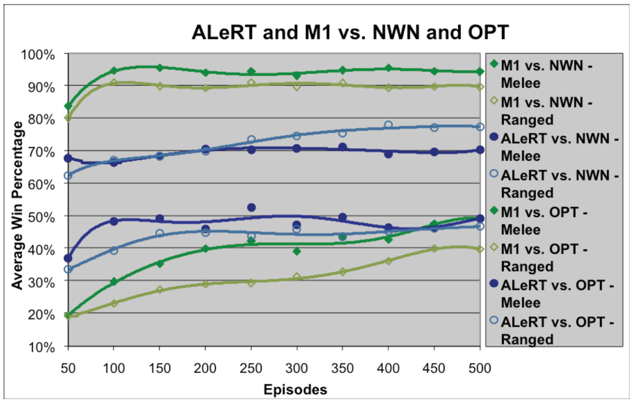

Figure 3. ALeRT and M1 against static opponents – NWN (upper traces) and OPT (lower traces).

M1 did not converge to the optimal strategy against OPT in the *Ranged* configuration. After 500 episodes it only attained 40% wins. The *Melee* configuration was taken directly from Spronck's module (NWN Arena 2008), but the *Ranged* configuration was created for our experiments to test adaptability. It is possible that M1 was tuned for the *Melee* configuration and re-tuning may correct this problem. Nevertheless, ALeRT converged quickly to the *Ranged* configuration, attaining a 44% win rate by episode 150 and 46% by episode 450.

Adaptation in a dynamic environment

To test the adaptability of agents in combat, we changed the equipment configuration at episode 501, and observed 500 more episodes. Each agent was required to overcome the bias developed over the first phase and learn a different strategy for the second phase. NWN is not adaptive, therefore, we compared ALeRT and RL_0 to M1. We used the following combined configurations: *Melee-Heal*, *Melee-Ranged*, *Ranged-Melee*, *Ranged-Heal*, *Heal-Melee*, *Heal-Ranged*, where the configuration before the dash was used in the first phase and the configuration after the dash was used in the second phase. The learning algorithms were not re-initialized between phases, so agents were biased towards their first-phase policy. In the first phase, we evaluated how quickly an agent was able to learn a winning strategy without prior knowledge. In the second phase, we evaluated how quickly an agent could discover a new winning strategy after an equipment change.

We ran 50 trials for each of the *Melee-Ranged*, *Melee-Heal*, *Ranged-Melee*, *Ranged-Heal*, *Heal-Melee,* and *Heal-Ranged* configurations. Figure 4, Figure 5 and Figure 6 illustrate the major advantage of ALeRT over M1. ALeRT adapts faster to changes in environment (equipment configuration) that affect a policy's success. ALeRT increased its average win rate by 56% (the average over all six experiments), 50 episodes after the phase change. By episode 1000, ALeRT defeated M1 at an average rate of 80%.

The features of ALeRT that contribute to this rapid learning are the trend-based step-sizes and win-based exploration rate modifications to Sarsa(λ). In fact, RL_0 defeated M1 at a slightly higher rate (84%) than ALeRT after the phase change, but RL_0 won only 42% in the first phase (Cutumisu and Szafron 2008). As stated in the previous section, RL_0 won only 34% and 41% against optimal strategies with *Melee* and *Ranged* configurations respectively, which is not an acceptable strategy. Therefore, for added clarity of the graphs, we do not show RL_0 traces in the graphs.

Rather than showing six separate graphs, we combined the common first phase configurations so that the experiments can be shown in three graphs: *Melee-Ranged&Heal*, *Ranged-Melee&Heal*, and *Heal-Melee&Ranged*. The data points from two separate experiments were combined into one trace in the first phase of each graph, therefore each data point represents the average winning percentage over 100 trials. Each second phase data point represents the average winning percentage over 50 trials.

Figure 4 shows the *Melee-Ranged&Heal* results. ALeRT did almost as well as M1 in the first phase with a 3% deficit at episode 450, recovering to a 49.3% win rate at episode 500. M1 did very well in the initial *Melee* configuration, perhaps due to manual tuning. Nevertheless, ALeRT's winning rate was very close to 50% throughout the first phase and it dominated M1 during the second phase.

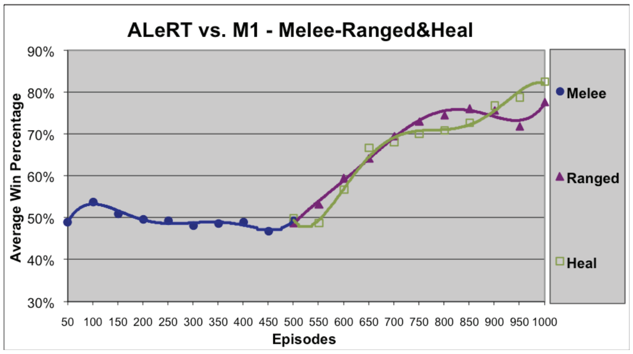

Figure 4. ALeRT vs. M1 – *Melee-Ranged* and *Melee-Heal.*

In the *Ranged-Melee* and *Ranged-Heal* configurations (Figure 5), the agents tied in the first phase, while ALeRT clearly outperformed M1 in the second phase. One of the reasons for the poor performance of M1 is that when it cannot decide what action to choose, it selects an attack with the currently equipped weapon.

In the first phase of the *Heal-Melee* and *Heal-Ranged* configurations (Figure 6), ALeRT and M1 tied again, but ALeRT outperformed M1 during the second phase. In each configuration, the major advantage of ALeRT over M1 is that ALeRT adapts faster to a change in environment, even though it does not always find the optimal strategy.

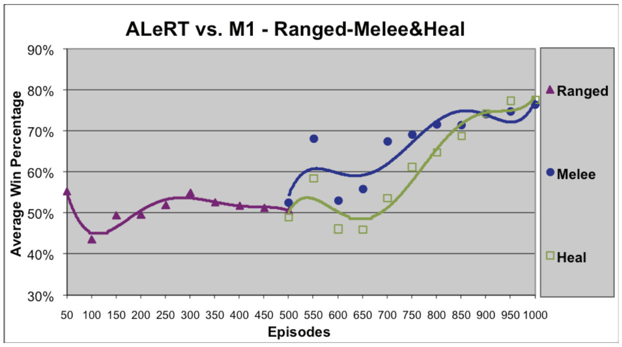

Figure 5. ALeRT vs. M1 – *Ranged-Melee* and *Ranged-Heal.*

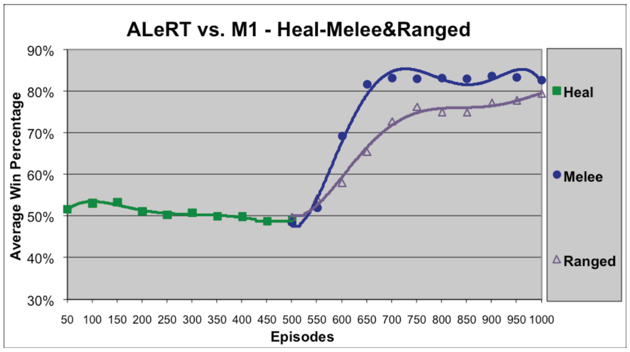

Figure 6. ALeRT vs. M1 – *Heal-Melee* and *Heal-Ranged.*

Observations

ALeRT is based on Sarsa(λ) and the only domain knowledge it requires is a value function, a set of actions and a state vector. Unmodified Sarsa(λ) does not perform well against either an optimal strategy (OPT in Figure 2) or against the dynamic reordering rule-based system, M1, in the first phase (Cutumisu and Szafron 2008). ALeRT overcomes this limitation, using three fundamental modifications to traditional RL techniques. ALeRT uses 1) action-dependent step-size variation, 2) larger step-size increases during trends, and 3) adjustable exploration rates based on episode outcomes. While conducting our experiments, we made several observations that may explain why ALeRT adapts better to change than M1.

ALeRT achieves a good score even when the opponent is not performing optimally and it does not attempt to mimic the opponent. Although ALeRT may not find the optimal solution, it still finds a good policy that defeats the opponent. In the games domain, this is a feature, not a bug, as we do not aim to build agents that crush the PC.

ALeRT works effectively in a variety of situations: short episodes (*Melee* configuration), long episodes (*Ranged* configuration), and time-critical action selection situations, such as taking a speed action at the start of an episode and a heal action when the agent's HP are low.

The ALeRT game state vector is simple (5 binary features), so each observation is amortized over a small number of states to support fast learning. Although the game designer must specify a state vector, "obvious" properties such as health, distance, and potion availability are familiar to designers. M1 relies on a set of 21 rules; discovering and specifying rules could be challenging.

Although ALeRT may select any valid action during an episode, M1 only chooses one type of attack action (ranged or melee) per episode in conjunction with speed/heal, but never two different attack actions. This restriction proved important in the *Melee-Heal* experiments, where although M1 discovered heal in the second phase, it was sometimes selecting only ranged attacks and could not switch to melee during the same episode. In some trials, this allowed ALeRT to win even though it did not discover heal in the second phase in that particular trial. Moreover, when M1 cannot decide what action to choose, it selects an attack with the currently equipped weapon (calling the default NWN if there is no action available). This is a problem if the currently equipped weapon is not the optimal one. Conversely, ALeRT always selects an action based on the value function. If there is a tie, it randomly selects one of the actions with equal value. There is no bias to the currently equipped weapon.

ALeRT uses an ε-greedy action selection policy which increases ε to generate more exploration when the agent is losing and decreases ε when the agent is winning. We experimented with several other ε-greedy strategies, including fixed and decaying ε strategies, but they did not adapt as quickly when the configuration changed. We also tried *softmax*, but it generated differences between estimated values of Q that were too large. The result was that the agent could not recover as fast once it selected a detrimental action. M1 uses *softmax* from the Boltzmann (Gibbs) distribution. Most importantly, ALeRT's action-dependent step-sizes provide a mechanism to recover from contiguous blocks of losses. ALeRT's trend-based step-size modification is natural, flexible and robust. In addition to allowing ALeRT to identify winning trends and converge fast on a new policy, it smoothly changes policies during a losing trend. M1 appears to use a window of 10 losses to force a radical change in policy. This approach is rigid, especially when the problem domain changes and the agent should alter its strategy rapidly.

Conclusions

We introduced a new algorithm, ALeRT, which makes three fundamental modifications to traditional RL techniques. Our algorithm uses action-dependent step-sizes based on the idea that if an agent has not had ample opportunities to try an action, the agent should use a step-size for that action that is different than the step-sizes for the actions that have been used frequently. Also, each action-dependent step-size should vary throughout the game (following trends), because the agent may encounter situations in which it has to learn a new strategy. Moreover, at the end of an episode, the exploration rate is increased or decreased according to a loss or a win. We demonstrated our changes using the Sarsa(λ) algorithm. We showed that variable action-dependent step-sizes are successful in learning combat actions in a commercial computer game, *NWN*. ALeRT achieved the same performance as M1's dynamic ordering rule-based algorithm when learning from an initial untrained policy. Our empirical evaluation also showed that ALeRT adapts better than M1 when the environment suddenly changes. ALeRT substantially outperformed M1 when learning started from a trained policy that did not match the current equipment configuration. The ALeRT agent adjusts its behaviour dynamically during the game. We used combat to evaluate ALeRT because it is easy to assign scores to combat as an objective criterion for evaluation. However, RL can be applied to learning any action set, based on a state vector and a value function, so we intend to deploy ALeRT for a variety of NWN behaviours. ALeRT will be used to improve the quality of individual episodic NPCs and of NPCs that are continuously present in the story. Before a story is released, the author will pre-train NPCs using the general environment for that story. For example, if the PC is intended to start the story at a particular power level, the author uses this power level to train the NPCs. During the game, when an NPC learns a strategy or adapts a strategy, all other NPCs of the same type (e.g., game class, game faction) inherit this strategy and can continue learning. Each of these vicarious learners jump-starts their learning process using the $\vec{\theta}$ vector generated by the experienced NPC. Ultimately, these improved adaptive behaviours can enhance the appeal of interactive stories, maintaining an elevated player interest.

Future Work

Next we intend to consider a team of cooperating agents, instead of individual agents. For example, we will pit a fighter and a sorcerer against another fighter and a sorcerer. Spronck's pre-built arena module will support experiments on such cooperating agents and ScriptEase supports cooperative behaviours, so generating the scripts is possible. We are also interested in discovering the game state vector dynamically, instead of requiring the game designer to specify it. We could run some scenarios and suggest game state options to the designer, based on game state changes during the actions being considered.

We are currently developing simple mechanisms for the game designer to modify RL parameter values. We also intend to provide the designer with a difficulty level adjustment that indicates a maximum amount that an agent is allowed to exploit another agent (usually the PC) and to throttle our ALeRT algorithm so that the agent's value function does not exceed this threshold. This can be done by increasing exploration or by picking an action other than the best action during exploitation.

We have experimented with a *variable* lambda that automatically adjusts to the modifications in configuration, but more experiments are necessary. We expect to learn faster in some situation if lambda is a function of the number of steps in an episode (e.g., directly proportional). For example, on average, in the *Melee* phase of a *Melee-Ranged* experiment there are 3.5 steps per episode and in the *Ranged* phase there are 5.5 steps. Lambda is responsible for propagating future rewards quickly to earlier actions, therefore, it needs a higher value to propagate to the start action in a longer episode. Finally, we are developing evaluation mechanisms for measuring success in non-combat behaviours. This is a hard problem, but without metrics, we will not know if the learning is effective or not.

References

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

Bowling, M., and Veloso, M. 2001. Rational and Convergent Learning in Stochastic Games. In *Proceedings of the 7th International Joint Conference on AI*, 1021-1026.

Bowling, M., and Veloso, M. 2002. Multiagent Learning Using a Variable Learning Rate. *Artificial Intelligence* 136(2): 215-250.

Cutumisu, M., Szafron, D. 2008. A Demonstration of Agent Learning with Action-Dependent Learning Rates in Computer Role-Playing Games. *AIIDE 2008*.

Cutumisu, M., Szafron, D., Schaeffer, J., McNaughton, M., Roy, T., Onuczko, C., and Carbonaro, M. 2006. Generating Ambient Behaviors in Computer Role-Playing Games. *IEEE Journal of Intelligent Systems* 21(5): 19-27.

Merrick, K., and Maher, M-L. 2006. Motivated Reinforcement Learning for Non-Player Characters in Persistent Computer Game Worlds. In *ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, Los Angeles, USA.

NWN. 2008. http://nwn.bioware.com.

NWN Arena. 2008. http://www.cs.unimaas.nl/p.spronck/GameAIOnlineAdaptation3.zip.

Rabin, S. 2003. Promising Game AI Techniques. *AI Game Programming Wisdom 2*. Charles River Media.

ScriptEase. 2008. http://www.cs.ualberta.ca/~script/.

Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., and Postma, E. 2006. Adaptive Game AI with Dynamic Scripting. *Machine Learning* 63(3): 217-248.

Spronck, P., Sprinkhuizen-Kuyper, I., and Postma, E. 2003. Online Adaptation of Computer Game Opponent AI. *Proceedings of the 15th Belgium-Netherlands Conference on AI*. 291-298.

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* 3(1): 45–53.

Sutton, R.S. 1992. Adapting Bias by Gradient Descent: An Incremental Version of Delta-Bar-Delta. In *Proceedings of the 10th National Conference on AI*, 171-176.

Sutton, R.S., and Barto, A.G. eds. 1998. *Reinforcement Learning: An Introduction*. Cambridge, Mass.: MIT Press.

Timuri, T., Spronck, P., and van den Herik, J. 2007. Automatic Rule Ordering for Dynamic Scripting. In *Proceedings of the 3rd AIIDE Conference*, 49-54, Palo Alto, Calif.: AAAI Press.