Reinforcement Learning in StarCraft: Brood War

Abstract

This project was inspired by the recent advances in Starcraft AI research including the highly successful Berkeley Overmind agent, winner of the 2010 AI Starcraft Competition which implemented a genetic algorithm to learn a successful micro-management system for certain types of units within the game. We have injected reinforcement learning (RL) using neural nets (NNs) into an open-source Starcraft AI agent in place of the heuristics in an attempt to have the agent learn a more optimal build order for units. Our goals included gaining a deeper understanding of machine learning concepts in practice in addition to attempting to improve upon an already successful Starcraft AI.

1. Background

1.1 Motivation

Our work was based largely on that of Shantia, Begye, and Wiering in their *Connectionist Reinforcement Learning for Intelligent Unit Micro Management in StarCraft*. However, where they implemented two online learning methods, Q-learning with neural nets and simple Q-learning, to learn more optimal policies for controlling units during combat, we deferred to an open source agent for micromanagement of units. Instead, we focused our efforts with similar implementations on learning build orders for units given some state. Theoretically, one should be able to slowly improve policies for specific tasks such as build orders or micromanagement one at a time through learning, then incorporate these policies together into one much improved agent.

1.2 BWAPI[1]

BWAPI is an open-source API which allows developers to inject AI agents directly into the Starcraft: Brood War environment, giving the agents full access to every command that a real Starcraft player would have access to. This API allowed us to train our machine learning agent within the native Starcraft environment thereby allowing us to completely avoid simulation.

1.3 Open-source agents

The agent chosen for our implementation of reinforcement learning in Starcraft was the product of University of Alberta's Computer Science department and was moderately successful in the 2010 AI Starcraft Competition[2]. We chose this agent over Berkley Overmind agent for several reasons including the modularity of the code, as well as the language in which it was written. Overmind was extremely fragmented with no clear place within the code where we could replace the build order. In addition, it was written in Java and so required a bridge to interface with BWAPI to control the agent. UAlbertaBot, however, contained a Strategy Manager class within its combat controller which was wholly responsible for choosing which units to build and when. This allowed us the option to replace this class with our own strategy, learned through RL. Also, it was written in C++ which allowed our agent to directly interface with BWAPI.

1.4 Development Environment

As per BWAPI's installation instructions, our project was developed using Visual C++ 2008 Express Edition for primarily two reasons. First, this was the IDE that most of the open-source agents were developed in, and by using it we avoided compatibility issues. Second, this software was free.

1.5 Version Control

In order to allow both project members to develop code simultaneously, we opted to host our agent as an open source project on Google Code[3]. In addition, the software package, TortoiseSVN, provided a GUI for SVN that facilitated functions such as updating and committing the code.

1.6 Simplifications

[1] http://code.google.com/p/bwapi/downloads/list

[2] http://eis.ucsc.edu/StarCraftParticipants#UAlbertaBot

[3] http://code.google.com/p/ou-skynet/

To simplify the learning task, we have made some modifications to the environment. Namely, StarCraft contains a fog-of-war which limits visibility of unexplored areas. To implement our reward function, we have enabled a cheat flag which allows our agent perfect information of the map. Without this feature, a much more sophisticated reward must be developed. In addition, we've also limited our agent to competing against one type of opponent while learning, the Zerg. Although it would be simple to include knowledge of the opponent's race in the state space for both implementations, we've chosen to play against one race exclusively to accelerate the learning process.

2. Progress

2.1 NN Proof of Concept

In order to gain a firm understanding of the process of implementing Neural Nets, we first developed a simple NN to learn the OR function on binary inputs. This net

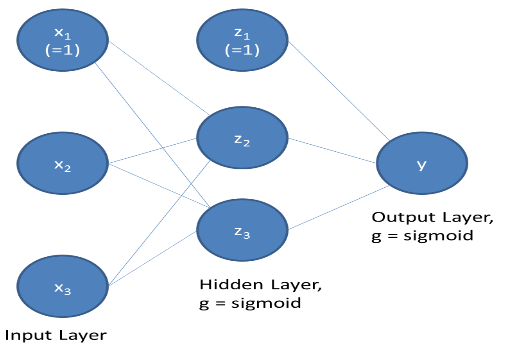


Figure 1. Graphical representation of our Proof of Concept Neural Net.

(shown in Figure 1) consisted of one output node, one hidden layer with three nodes (one bias), and one input layer with three nodes (one bias). The hidden layer and ouput layer both had a sigmoid activation function, and the net converged to a very low error model quickly.

2.2 NN Q-learning Implementation

2.2.1 NN Structure

Our agent learns using a set of four neural networks, one for each unit that we can build. Each net takes 33 inputs which together comprise the state space. These feed through seven hidden nodes to a single output: our estimated Q value.

The weights are updated throughout the game, and written to a file at the end. In this way they are made to persist from one game to the next. If no file is available for a

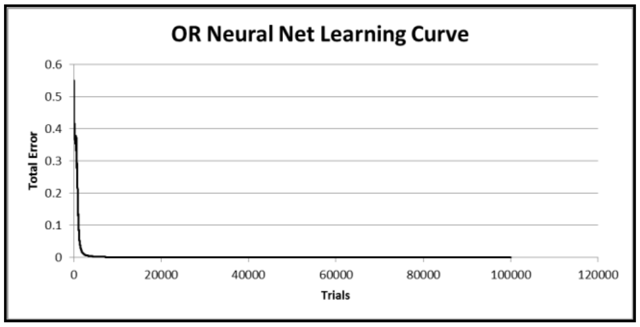


Figure 2. Errors for a Neural Net learning the OR function.

given net, it is initialized with random weights, taken from a Gaussian distribution between -1 and 1.

2.2.2 Q-Learning

Algorithm 1 Q-learning

```
repeat
obtain $s_t$
choose $a_t$ using $\pi(s_t)$
Observe $r_{t+1}$ and $s_{t+1}$
$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{i \in A} Q(s_{t+1}, i) - Q(s_t, a_t)]$
until $Q(s_t, a_t)$ converges
```

Every 20 seconds or so, our neural nets update themselves based on the Q-learning algorithm. This algorithm is shown above and involves updating each weight based on the actual reward received; the maximum value of the policy for any action; and the previous policy value from when the action was chosen.

2.2.3 State Space

Our state space is made up of 32 variables:

- Total number of units we control
- Available build capacity
- How many of each of 28 units types we control
- Log_{10}(Minerals)
- Log_{10}(Vespene Gas)

Build capacity refers is the number of units that we can control at a time, minus the number we actually have. Minerals and Vespene Gas are the resources used to create new units or buildings.

This state representation captures everything that we want to know about the current game state, although it contains a lot of extraneous details. As described below, the state space for our simple Q-Learning agent is much simpler, and it seems to perform slightly better than this implementation.

2.2.4 Reward Function

Our reward function is fairly simple at this point. We simply take the number of units and buildings that we control, and subtract the number of units and buildings controlled by our opponent. This is a fairly good way to boil a game state down to a single number, since the player with the biggest army is usually winning, and a player with no units or buildings at all has lost. However, this does reward our agent for building more probes (cheap, resource-gathering units that cannot fight well) than it probably needs. We might refine this later, perhaps by valuing our probes lower than other units, and/or giving the agent a bonus for winning the game.

2.3 Simple Q-learning Implementation

2.3.1 State Space Representation

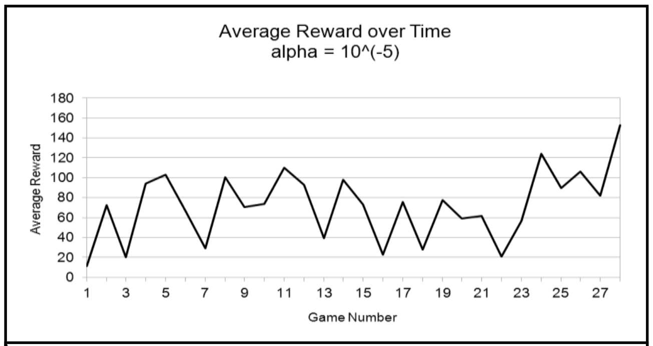
The state space representation for our Simple Q-learning agent was much simpler than that of our NN agent. We have divided the space with four binary classifiers (Does enemy have cloaked units, Does enemy have flying units, Can we see cloaked units, Can we shoot flying units) and two variables with five bins for minerals and vespene gas. The bins are less than 10, 11-100, 101-1000, over 1000 for both minerals and gas. In each of these 2*2*2*2*5*5 = 400 zones we randomly assigned a Q-value for each of the four actions available (build Zealot, build Dragoon, build Dark Templar, build Observer), and these Q-values are updated using the learning algorithm above.

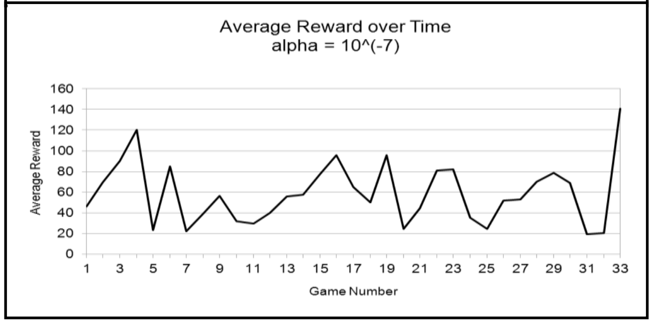
2.3.2 Reward Function

The reward function used for our simple Q-learning agent was the same as that used for the NN Q-learning implementation. Namely, we're rewarding the agent by the number of units and buildings we control minus the number of buildings and units the opponent controls

2.4 NN Q-learning Results

We tested different values of alpha (learning rate) to see what effect this had on our agent's performance. As you can see in figures 3 and 4 below, our reward function bounced around more or less randomly, although an alpha of 10^{-5} seems to work better than 10^{-7}. However, in both cases, the win ratio improved as time went on (figure 5). This suggests that our reward function does not correlate that strongly with whether the agent wins or loses a game, as indeed it does not, as shown in figure 6. We will examine possible explanations for this behavior later, as well as what it means for possible future work.





Figures 3 and 4. Average reward received over the course of a game as a function of number of games played. As you can see, there is no general trend up or down.

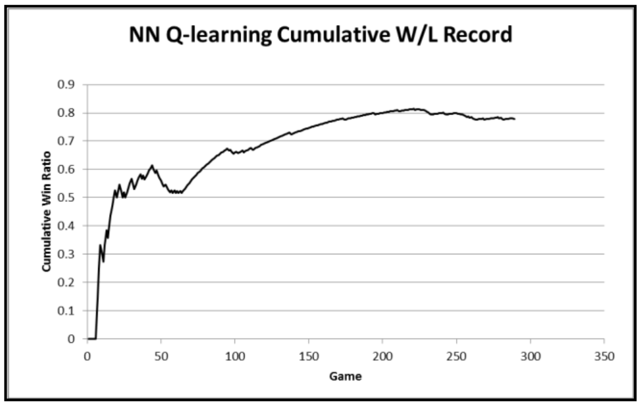


Figure 5. As you can see in the graph, the win percentage increases as the agent gains experience; thus the agent is learning.

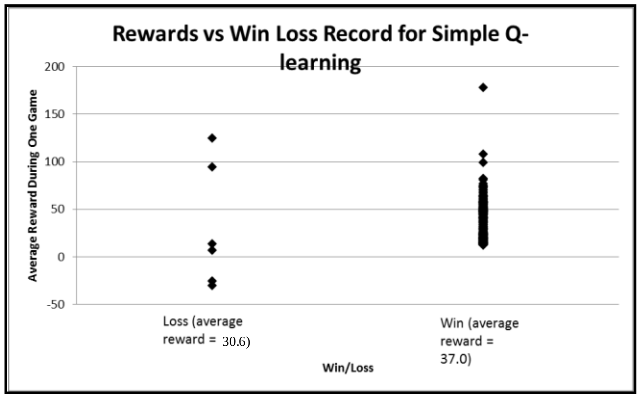


Figure 6. This graph show that the behavior we expect to see, namely observing a much higher reward for the games where the agent wins, is missing. Instead, the average reward for games where the agent won and loss are very near, 30.6 for losses and 37 for wins.

2.5 Simple Q-learning Results

As for the Neural Net implementation, we tested the simple Q-learning agent with several values of alpha. Again, the agent's average reward function did not increase over time (shown in figure 7), but the cumulative win loss ratio did (as seen in figure 8). Again, we hypothesize that this is a result of the stochasticity of the environment, and the fact that a high reward function does not correlate well with victory.

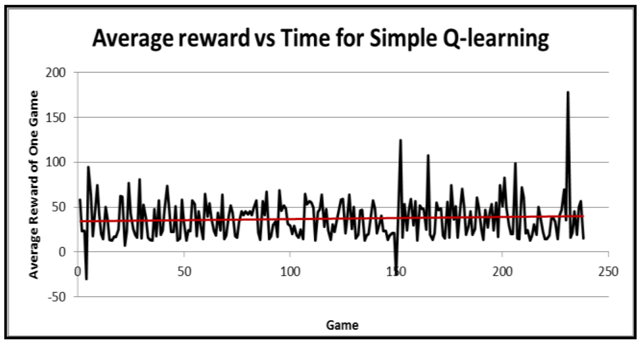


Figure 7. This graph show that the behavior we expect to see, namely observing a much higher reward for the games where the agent wins, is missing. Instead, the average reward for games where the agent won and loss are very near, 30.6 for losses and 37 for wins.

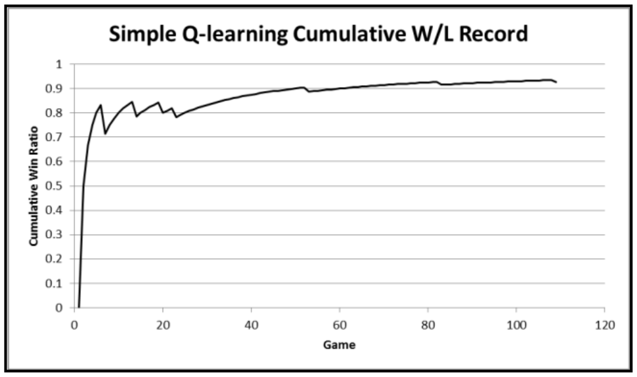


Figure 8. Again, the agent's performance increased with experienced, verifying that the agent was learning.

2.6 Comparison

While there was no strong correlation between the rewards being received and win percentage for either implementation, it is obvious that both agents are learning via their cumulative win percentages. One possible explanation for this behavior is the stochasticity of the environment. For example, our current reward function increases as we gain buildings and units and decreases as the opponent gains buildings and units. However, in several of the learning trials we observed that the agent would converge to a policy of building as many of the cheapest fighting unit as fast as possible. While this strategy does tend to do reasonably well (resulting in an 97% win ratio in one of our simple Q-learning experiments), it doesn't result in a high average reward over the course of the game since the agent ends the game quickly without building many units.

Another thing that we notice is that the simple Q-learning agent outperformed the NN implementation. We hypothesize that this resulted from the additional information given to the simple agent, namely the knowledge of enemy units such as flyers and cloaked units and our capacity to fight them. A combination of the implementations where Q values are approximated using Neural Net but with more appropriate input variables may outperform either individual implementation.

It is also interesting to examine the actual learning rates of the two implementations. The NN implementation achieved a win percentage of 80% after approximately 200 games, but the simple Q-learning agent hit 80% win percentage after only 10 games. We suspect this is because of the much simpler state space representation found in the simple Q-learning agent.

2.7 Conclusion

While there is certainly room for improvement in both of our agents, we were pleased to see both of their performances improve over time. It appears that as long as it captures the most important information, a simpler state space is better than a more complex one (as evidenced by the superior performance f the simple Q-learning agent). We also found that although our reward function is not that strongly correlated with actually winning the game, our agents still improved their win rates over time by trying to maximize that reward.

StarCraft is a very complicated environment, with many stochastic variables that affect the outcome of the game. For instance, at the beginning of each game, our agent sends a resource gathering unit to the enemy base, and tells it to attack their gatherers. Sometimes the enemy drones can gang up and kill our scot; other times they chase I around for up to a minute or two, completely neglecting their other duties. We feel that this wasted time is the single most important factor in whether our agent wins or loses, since it always wins almost immediately if the enemy drones are distracted for more than about 45 seconds. We feel that our inability to account for variables like this has been the biggest limiting factor on the learning of both of our agents.

3. Next Steps

3.1 Improve Reward Function

As described earlier, one promising avenue for improvement is refining our reward function. Having a lot of probes is beneficial, but toward the end of the game we would ideally be turning our resource advantage (from having so many gatherers) into fighting units.

3.2 Refine NN Implementation

We really only need to train observers (flying units which can see invisible enemies) if our opponent has lurkers (units which borrow underground to attack while invisible); hence, we should probably feed the number of enemy lurkers into the nets, and see if our agent will learn a more sophisticated observer-building policy as was observed in the simple Q-learning implementation. In addition, the NN agent would likely do better with knowledge of enemy flying units.

3.3 Experiment with Parameters

There are several parameters hard-coded into the agent, and we could experiment with any of these. We might try changing the epsilon value (how likely the agent is to take a random action at a given timestep) throughout the game. Perhaps it is best to explore early, and stick more closely to our policy later in the game. Maybe instead of the course of a game, we should change epsilon over the course of the agent's lifetime, reducing the value by some step at the start of every game. We could also adjust the learning rate and discount rate of the neural nets. Increasing alpha might cause the agent to learn faster, but if it is too high our policy will not converge. The discount rate affects how highly we value predicted future rewards. It seems worth investigating whether we can get a better policy by raising or lowering this value.