Application of Approximate Q-learning to Simplified Macromanagement in StarCraft II

Michael Lee
Department of Computing
Andrews University
Berrien Springs, United States
lmichael@andrews.edu

Advisor: Rodney Summerscales
Department of Computing
Andrews University
Berrien Springs, United States
summersc@andrews.edu

***Abstract*—Contemporary strategies in Machine Learning that train using the real-time strategy video game StarCraft II have recently utilized the power of both neural networks and reinforcement learning in the form of "Deep Reinforcement Learning," and have risen greatly in popularity. Unfortunately, the use of neural networks in training comes with great costs in resources and requires expensive hardware to run in a manageable amount of time. Instead, we propose the use of a modified form of Q-Learning utilizing weighted features in linear approximation and forego the use of neural networks to explore the performance of non-neural network strategies in the StarCraft II environment in regards to outpacing an enemy in simplified macromanagement.**

I. Introduction

Reinforcement Learning research has taken advantage of the nature and conveniences of video game environments in order to test the robustness of algorithms. More recently, reinforcement learning research has gravitated toward the real-time strategy video game series StarCraft due to its tricky environmental complexities and vast state-space complexity. Much research that involves testing reinforcement learning strategies in StarCraft and its successor StarCraft II conflates two prominent subfields of Machine Learning, Deep Learning and Reinforcement Learning, in the form of Deep Reinforcement Learning in order to take advantage of the strengths of both fields when tackling many of the problems faced while playing the game. Although incredibly powerful, it unfortunately can result in substantial computational and temporal requirements. Our research focuses on the ability of reinforcement learning without neural networks (i.e. without Deep Learning) to test its ability in such a complex environment without relying on considerable computational resources. More specifically, we utilize a form of Q-learning called "Approximate" Q-learning, a modified form of Q-learning where values of Q are calculated based on a set of weighted features that correlate to important aspects of the environmental state, with the intention of observing considerable learning speed over its traditional counterpart. This is tested in a paradigm of StarCraft II gameplay called "Macromanagement" in order to focus on its ability to form policies regarding resource management and build timing rather than the mechanical and tactical ability associated with "Micromanagement."

II. Background

A. Related Work

Reinforcement learning research has utilized games as convenient training simulations due to their straightforward goals and reward structures, ease of manipulation, and direct access to information. Video games, for instance, often employ a point based scoring system that roughly exemplifies progression toward the goal of the game. Video games are also easily reproducible within computers, and an agent can directly access the information through the use of APIs. Board games have also been the subject of reinforcement learning research due to their potential for complex and deep strategic gameplay and vast complexity spaces, such as Chess or Go.

The field of Reinforcement Learning recently merged with Deep Learning to create Deep Reinforcement Learning, a subfield that utilizes the power of neural networks in conjunction with trial and error based learning. Deep Reinforcement Learning has become a prominent part of Machine Learning research, and research in its applicability and power is current and ongoing. Research conducted by DeepMind with Atari games in 2013 [10] showed promise in the ability with Deep Reinforcement Learning in relation to video games with traditional "score" systems. Following this, DeepMind applied Deep Reinforcement Learning to the board game Go in 2016 , creating an agent named *AlphaGo* that defeated the strongest players in the world, reaching a goal in RL research at the time considered to be unattainable at the time [4]. The group then moved onto applying Deep Reinforcement Learning to StarCraft II, a video game with a complexity space far more vast than Go and characteristics that are more akin to the real-world. DeepMind worked with Blizzard Entertainment to create the StarCraft II Learning Environment (SC2LE), an API that made it easier for Reinforcement Learning research to be done in the game [6].

StarCraft research was not limited to the period after the StarCraft II Learning Environment was released. Previous research has been done, and is still being done, with the predecessor to StarCraft II, StarCraft: BroodWar, with the unofficial API called the BroodWar API, or BWAPI [1]. Bots such as the UAlbertaBot created by David Churchill of the University of Alberta [3] have seen success in competitive

Fig. 1. Example of StarCraft II gameplay; Blue Player's base is being attacked by Red Player's marines.

StarCraft: BroodWar tournaments, and specific reinforcement learning research has been done on subproblems of the game, such as Wender and Watson's application of TD learning models to small-scale BroodWar combat scenarios [9].

StarCraft research gained momentum following the release of the SC2LE, and StarCraft II research has been done extensively, recently culminating in a deep reinforcement learning agent created by DeepMind titled *AlphaStar* that was able to defeat professional StarCraft II players after training from replay data [5].

B. Training Environment: StarCraft II

StarCraft II is a futuristic real-time strategy (RTS) video game developed by Blizzard Entertainment. In it, players choose one of three races (Terran, Protoss, and Zerg) to develop a base and army to defeat their opponent. For reinforcement learning training, StarCraft II is a challenging test-bed due to its environmental complexities and immense number of possible states. The game usually involves two players warring against each other, with the victor being the last one standing (often a player wins due to the opponent resigning). Unlike training environments used historically in reinforcement learning research, such as Chess, Go, or Atari video games, StarCraft II takes place in a continuous, real-time, semi-three-dimensional space with a multitude of unique units (similar to game pieces) spread across the three in-game races. These units are produced by buildings constructed in a player's base, and some buildings/units are chosen over others due to different strategies. The game's "Fog of War" mechanic further introduces complexity in removing perfect information; if there are no friendly units in a section of the map, it is darkened and no information pertaining to the enemy player is visible. With these attributes alongside the incredibly large number of possible states, StarCraft II is considered to be a significantly more complex environment than the other environments aforementioned, and has characteristics more similar to the real-world.

Gameplay in StarCraft II can be broken down into two major paradigms: Macromanagement and Micromanagement (often referred to as "Macro" or "Micro", respectively). Macromanagement involves making high-level decisions and actions in regards to the player's "economy" and the progression of an overarching strategy. This may include choosing which buildings to construct, improving resource gathering and management, and when to pick fights to lead to a better situation. Inevitably, the player with better Macromanagement will lead in overall resources and units, naturally causing that player to have an advantage over the opponent. Micromanagement, on the other hand, is the overall ability of the player to control specific units to do specific actions that may cause the player to gain some positional or combat-related advantage, such as moving a group of units in a way that minimizes the amount of damage received, or splitting an army such that it can attack from multiple angles. Having better Micromanagement than the opponent can turn the tables for an important battle. As such, being skilled in both Macro and Micro becomes important to winning games of StarCraft II.

C. Q-learning and Approximate Q-learning

With the introduction of Q-learning near the end of the 20th century [2], alongside the improvements of computational power, Reinforcement Learning research grew in momentum. Reinforcement learning research today often refers to and modifies this approach in some way. With Q-learning, the agent determines its optimal policy by calculating and updating a numerical value Q from the state and the actions associated with that state. These Q values are then stored to represent a state at a certain time and an action taken within that state. Q is updated as shown:

$$Q'(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (1)$$

Unlike traditional Q-learning, Linear-approximation Q-learning, or Approximate Q-learning, determines its Q value based on a list of features, chosen from the state to more succinctly describe critical state information, and the weights associated with each feature. By doing this, Q values do not have to be stored in a table, but are rather simply calculated from a static list of weighted features. This process saves a considerable amount of resources since only the weights associated with their respective features are updated, giving the agent the potential to learn much quicker than its traditional Q-learning counterpart.

III. Our Research in StarCraft II

Despite the amount of contemporary reinforcement learning research done in StarCraft II, much of the research done in regards to StarCraft II Macro gameplay often utilizes the power of deep reinforcement learning strategies. However, upon observation of published research, the presence of Macro research that foregoes the use of neural networks is lacking. Therefore, our research aims to implement an Approximate Q-learning approach that does not utilize neural networks toward a simplified Macromanagement scenario to observe its ability. We do this due to the benefits offered from relinquishing neural networks; while powerful, neural networks require

substantial amounts of computational resources and time to train. Foregoing the use of neural networks therefore enables the ability to train our agent on more accessible hardware in shorter amounts of time. We aim to see if predetermined chosen features are sufficient for our agent to learn, and if it can learn quickly.

The choice of avoiding neural networks is also due to the unknowns associated with neural networks identifying critical aspects of the environment. Our construction and control of features related to the states in StarCraft II enables our ability to directly manipulate and observe what information is accessed by the agent. This removes a layer of abstraction between the researcher and the agent, as the complexities of neural networks can obfuscate the learning process.

The implementation of Approximate Q-learning comes with challenges considering the complexity of the StarCraft II environment. Unlike turn-based board games or older video games, where extracting relevant information, determining features, and predicting future states is more straightforward, StarCraft II's continuous and real-time nature consequently forms a set of possible states that is exponentially larger. Since extracting the features of future states can be complicated due to the exorbitant amount of possible state permutations in a continuous environment, simplifications and approximations are implemented.

Q-learning and its relative algorithms run into issues when implemented in a real-time environment, as they need a different approach to an update structure when compared to their turn-based counterparts. Since StarCraft II is played in real-time, this led to challenges regarding implementation. Also, since linear-approximation requires features that generalize the state at any given moment, determining and keeping track of those features is a major requirement to implementing Linear-approximation Q-learning. As a result, the execution of this strategy in the real-time StarCraft II environment was considered to be one of the major goals of this project.

We aim to observe the power of our approach against baselines. As such, we train our agent in a Macro subproblem of StarCraft II involving the production of Marines, the most basic combat unit in StarCraft II for Terran, in combat with an adversary. The initial framework for this scenario was created by Steven Brown [8]. We train our agent in this sub-problem against a random agent that has access to the same units and actions as our agent, but chooses actions randomly, as well as against the built-in StarCraft II scripted AI.

IV. Methodology

The StarCraft II Learning Environment is used to interact with the game. This API extracts the relevant information that a player would normally see and exports that data as tangible information for a computer program to parse. Alongside SC2LE, DeepMind created a Python wrapper dubbed PySC2 as a framework for researchers to develop and test machine learning strategies with the StarCraft II API. This was therefore used to develop and test the approximate Q-learning agent in conjunction with the base agent framework, random agent, and basic Q-learning agent provided by Steven Brown. In each instance of training, two agents are placed in a small 64x64 map, with each agent's Command Center located either on the top left or bottom right corners of the map. Both agents play as the Terran race (humans). The agents are supplied with 12 SCVs which automatically begin mining minerals. Given the aforementioned base agent framework, two types of agents can be specified to play against each other; if wanted, a human player can play against an agent.

The units and buildings of StarCraft II have their own specific purposes for progressing toward winning the game. For the Terran race, players begin with SCVs to mine Minerals, a resource in the game used to construct buildings and units (the other is Vespene Gas, but it is not used in our tests). Supply Depots are buildings that increase a player's supply, a number that acts as a cap for training units (in our case, Marines). A Barracks is a building that trains units (again, in our case Marines) for combat; they can only be constructed once a Supply Depot is built. Finally, Marines are our main unit for combat and are used to fight enemy units and buildings. While more units and buildings exist for Terran, we limit our usage to these only. The framework created by Steven Brown distills some of the actions available in PySC2 into easily recognizable actions that an agent can perform, each with predictable outcomes:

- **Harvest Minerals:**

  The agent instructs an SCV to search for the closest mineral patch and begin mining minerals from it. This is usually done when an SCV is idling after constructing a building.

- **Build Supply Depot:**

  The agent instructs an SCV to build a supply depot at a random location near the Command Center. This action can only be done if there are enough resources available to do so.

- **Build Barracks:**

  The agent instructs an SCV to build a Barracks at a random location near the Command Center. This action can only be done if there are enough resources, and a Supply Depot exists.

- **Train a Marine:**

  The agent selects a Barracks and trains a Marine there. This action can only be done if there are enough resources and "supply" available, and a Barracks exists.

- **Attack:**

  The agent selects a marine and executes an Attack

Move action against the enemy base. This causes the marine to fight any enemy units or buildings it finds along the way (based on sight, which is infinite). Since a marine has limited attack range, it has to move close enough to an enemy unit in order to fight it.

- **Do Nothing:**

  The agent does nothing.

All agents use these six actions to fight their enemies. Modifications to these actions were made in comparison to the original framework; the number of buildings that can be constructed were changed from one to five for Supply Depots, and were changed to be unlimited for Barracks.

Traditional Q-learning determines the value of Q from the approximated values of the state and the action that produces a transition to a new state. The maximum value of this state-action pair is used for Q. The update equation for Q is shown in Equation 1. The linear approximation modified form of Q-learning makes a few changes in determining the value Q; rather than calculate the value from the max state-action pair, it is instead calculated from the sum of all features multiplied by their respective weights:

$$Q'(s_t, a_t) = \sum_{i=1}^{n} \omega_n f_n(s_t, a_t) \tag{2}$$

Where ω_x is the weight pertaining to feature f_x. In order to determine the Q value for a given state, the agent calculates the maximum Q value considering all possible legal actions from that state to the new state. The learned weights are carried over and are used to calculate Q for each possible future state. The maximum of all of these is returned, and the action pertaining to the greatest Q value is the action chosen by the agent. The weights are updated based on the following:

$$\omega'_x = \omega_x + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] f_x(s_t, a_t) \tag{3}$$

Where α is the learning rate, r_t is the reward at time t, γ is the discount factor, and a_t is the action taken in state s_t at time t.

To circumvent the challenge of each possible current and future state being completely different than any state previously experienced (if a unit moves one pixel in a direction, it is technically a completely different state), along with the difficulty in parsing the relevant information that pertains to the environment, appropriate approximation of the features of the state are needed for the agent to learn in a short amount of time. To do this, we created a simplified approximate feature extractor for future states that returns simplified numerical and incremental information of the state should an action be taken; instead of the agent calculating the future state explicitly and extracting features from the foresight, it instead calculates a temporary incremental increase of critical features. For example, if a Barracks is built, instead of having to run a simulation of the game environment for the future state where the action "Build Barracks" is executed, or calculate the specifics of the change of the environment, instead simply choosing the action as a part of maximizing the value of the state-action pairs would increment the feature pertaining to Barracks by one. These increments or decrements, along with other simple calculations, would approximate the features for the future state; having the features make generalizations in this fashion makes running the agent a much simpler task. The agent can easily observe what exists in their base by referencing the change of a particular feature and be able to determine the best action for a state in this fashion.

In order to properly discover potential winning states, an exploration function is needed. Unfortunately, a simple static epsilon-greedy approach is not sophisticated enough for the agent to properly explore the action space. Consequently, implementing a simple exploration strategy comes with the large risk of the agent settling for a strategy that does not remotely lead to winning in the short term, and may need a long period of training to arrive at a viable strategy. Instead, we used a dynamic epsilon-greedy approach, where the value of epsilon decreases over time and approaches zero. This allows the agent to explore very quickly, but also causes the agent to settle on a strategy after training for a while.

We determined features based on perceived relevant aspects of each state. Specifically, the relevant aspects of each state are stored in a large tuple. Parts of this state variable were re-purposed in the form of features for the agent to focus on and use for learning. These values get passed on to the agent and are used in calculating the value of Q. In guessing future states, the agent modifies some of these features given the action taken to arrive at the future state and temporarily uses that new feature value to calculate the maximum Q value of the possible state-action pairs in all future states.

We tracked ten features that we believed represented the most critical parts of the state:

- Friendly Marines
- Friendly Barracks
- Friendly Supply Depots
- Enemy Marines - Friendly Marines
- Enemy Barracks - Friendly Barracks
- Enemy Supply Depots - Friendly Supply Depots
- Active (Attacking) Friendly Marines
- Minerals
- SCVs
- Idle SCVs

These features are first divided by a fixed value in order to become a ratio less than one (to prevent diverging weight values), then multiplied by their respective weights before being added together to calculate Q.

The approximate agent is placed in a simple map (called "Simple 64", due to it's minimap representation being 64x64 pixels in resolution) on one of the two main corners, and the opponent is placed on the opposite, diagonally symmetric corner of the map. The Approximate Q-learning agent trained against a random agent and the easiest built-in scripted AI (named "very easy") of StarCraft II. It is important to note

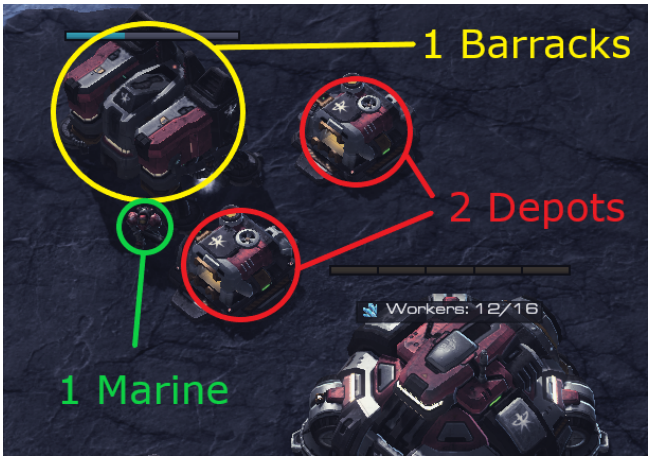

Fig. 2. Features extracted from a game state. The features are multiplied by a weight and added together to calculate Q.

that the random agent, despite its name, is not necessarily tenuous; while it takes its actions randomly, since the set of actions is small, and the step rate for the agent is fast, the random agent is able to create a large base and amass a large army in a short amount of time. The scripted AI on the other hand has access to all possible units, buildings and actions for the race chosen (in our case, Terran). Our Approximate Q-learning agent trained against both adversaries in "runs," where each run consists of 500 games, and weights are not carried over between runs. Wins, ties, and losses are recorded and fed back into each agent as rewards for learning (Win = 1, Tie = 0, Loss = -1). Training was done on a personal home computer, with an RTX 3070 Mobile Graphics Card and Ryzen 9 5900HS CPU; usage of more powerful hardware was avoided.

V. Data and Results

Fig. 3. Win percentage of Appx. Q-learning Agent against the random agent over the first 300 games. Each colored plot is an independent run. Win percentage increased dramatically after about a dozen training games.

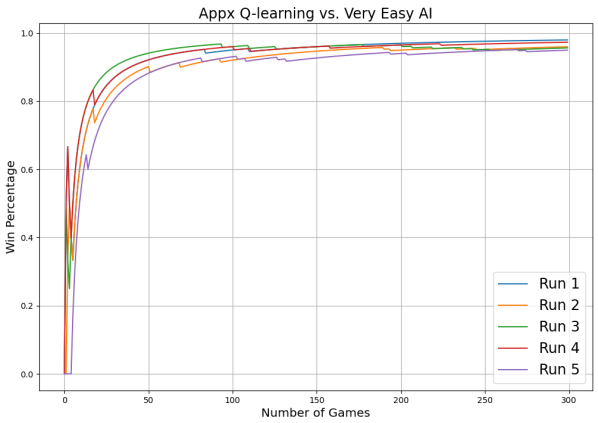

Fig. 4. Win percentage of Appx. Q-learning Agent against the "Very Easy" scripted AI over the first 300 games. Each colored plot is an independent run. Like Figure 3, win percentage increased dramatically after about a dozen training games.

Figures 3 and 4 represent our Approximate Q-learning agent's win percentage over number of games trained against the random agent and the "very easy" scripted AI, respectively. Each graph contains multiple independent plots depicting individual runs, with a run consisting of 500 games.

Our agent's progress in Macro gameplay was measured empirically. The first few training games, the agent predictably chose actions randomly for the sake of state exploration. This period therefore shared performance with the random agent and noticeably carried a middling win-rate before suddenly improving as the exploration function enabled a reduction in random actions taken. This early period also displayed low win-rate against the scripted AI, and the agent required a few games of training and reduction in the exploration function's random element before suddenly increasing its win-rate against the scripted AI. After a period of 10-20 training games, our agent dramatically increased its win-rate against both adversaries after settling on a policy that committed to only a few constructed buildings in favor of producing marines as fast as possible. Our Approximate Q-learning agent struggled against the stronger forms of the scripted AI; the agent still manages to create a policy similar to the policy formed from training against its easier adversaries, but can not overcome the stronger AI's defensive forces.

VI. Discussion and Future Work

The Approximate Q-learning agent was able to develop a policy of rapid marine production and began winning consistently against both the random agent and the easiest in-game AI after about 10-20 games of training. Our agent's win rate percentage abruptly increased and appears to asymptotically approach 100% before the run ended. Notably, even if the agent struggled to win from the beginning, it still manages to develop a policy that overwhelms its adversary with marine production.

Our agent demonstrates the ability to implement a non-neural network based reinforcement learning agent with only a small set of features representing a complex state space. Based on the quickly increasing win rate against both the random agent and the scripted AI, our agent also demonstrates the benefits of utilizing a reinforcement learning agent despite the small list of predetermined features.

More noticeable issues when training against the stronger scripted AI mainly stemmed from the limited set of actions. More specifically, the trained policy was based on the simple action of sending a single marine to attack the opponent each time the "Attack" action was chosen. This is in contrast to the more diverse methods of committing to combat demonstrated by both the stronger scripted AIs as well as more sophisticated human players; even if combat units are limited to marines, those units can be held as a defensive force, or sent as a large offensive wave. Due to our limited action set, this method of defense/offense appears to be difficult, if not impossible, to formulate. Therefore, implementing this option may boost the agent's ability to win games without dabbling too much in Micro gameplay.

An area of potential further work involves increasing the complexity of the state space and the list of actions to include making more units/buildings accessible to the agent. This would be an interesting area of research to test the limits of Approximate Q-learning's potential to handle increasing amounts of features simplifying a complicated environment, and it will be a challenge for the researcher to determine the appropriate features to exemplify the state well enough for the agent to learn appropriately.

Another area of possible research is testing the agent's ability to learn when placed in more challenging environments, such as including larger and more complicated maps. One major way of increasing environmental complexity would be to reintroduce Fog of War, effectively turning the problem into one that supplies imperfect information of the opponent. This would therefore require features and actions that incorporate methods for gathering the relevant information regarding the opponent in order to make strategic decisions.

The aforementioned areas of further research may require more robust exploration functions in order to more thoroughly explore the state space, as improper exploration may result in potentially missing important states that may reduce the chance of learning strong strategies.

Acknowledgments

I would like to thank Rodney Summerscales, PhD of the Andrews University Department of Computing for his mentorship, aid in guiding this research, and kindness and patience in professorship regarding Computer Science, the field of Artificial Intelligence, and scientific academia. I would also like to thank the J.N. Andrews Honors Department for allowing me the opportunity to conduct and produce research during my undergraduate years at Andrews University, despite the setbacks caused by the COVID-19 Pandemic and the resulting adjustments required.

References

[1] BWAPI: The BroodWar API. http://bwapi.github.io/, 2017.
[2] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol 8, no 3–4, bll 279–292, 1992.
[3] D. Churchill, *UAlbertaBot*. https://github.com/davechurchill/ualbertabot/, 2013.
[4] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol 529, bll 484–503, 2016.
[5] O. Vinyals *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol 575, bll 350–354, 2019.
[6] O. Vinyals *et al.*, "Starcraft II: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708. 04782*, 2017.
[7] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 2nd ed. Cambridge, MA: MIT Press, 2020, pp. 204-209.
[8] S. Brown. "Reinforcement Learning with Raw Actions and Observations in PySC2." ITNEXT. https://itnext.io/reinforcement-learning-with-raw-actions-and-observations-in-pysc2-af0b6fd8391f (accessed Sept. 17, 2021).
[9] S. Wender and I. Watson, "Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft:Broodwar," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 2012, bll 402–408.
[10] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312. 5602*, 2013.
[11] Z. Tang, D. Zhao, Y. Zhu, and P. Guo, "Reinforcement learning for build-order production in StarCraft II," in *2018 Eighth International Conference on Information Science and Technology (ICIST)*, 2018, bll 153–158.