Finding Competitive Angles of Engagement in Starcraft using Neural Networks

By Thomas Cummings

The project isn't quite finished. Implementing the data collection has turned out to be more difficult than I realized. I will be turning in a new version tomorrow to hopefully get more points.

Background

Starcraft is a beloved classic in the genre of Real-Time-Strategy games. Starcraft was created by Blizzard Entertainment company in 1998 and later that year expanded with the Starcraft: Brood War expansion pack. In 2017, Starcraft: Remastered was released, which did not change the gameplay but instead updated the graphics and allows the game to be played in much higher resolution. Starcraft is played by collecting two different types of resources, constructing buildings, and training combat units to attack the opponent. The game starts with each player having a home base and a few resource collecting units. As the game progresses, each player must create new buildings that can be used to train new combat units, such as soldiers and tanks, and building new bases for collecting resources when resources in the home base are depleted. The game can be played using many different strategies, but most strategies involve building an army of combat units and using that army to destroy the opponent's army and buildings. The game is won when the opponent no longer controls any buildings. A typical game lasts between 20 minutes and 30 minutes.

Strategy Basics

Strategy in Starcraft can be divided into two different categories, macro-management and micro-management. Macro-management (or macro) deals with strategies about how many resources to gather, which buildings to create, which combat unit upgrades (technologies) to research, how many combat units to build vs how many resource collecting units to build, and what time should units be built. There are three main macro strategies utilized in in the early stages of the game, economy, tech, and rush.

In an economy strategy, the player focuses on maximizing resource collection in the early game, and then using those resources to build a large army of combat units to attack the opponent in the middle game. In a tech strategy, the player focuses on constructing technology buildings which allow the player to build high-tech combat units, and then attack the opponent in the middle game with these high-tech units. In a rush strategy, the player focuses on training as many combat units as possible as early as possible and overwhelming the opponent in the early game with a superior army. This paper focuses on a common early game scenario where the defending player is using an economy or tech strategy (and thus having a smaller army in a defensive position), and the attacking player is using a rush strategy (and thus having a larger army moving out to attack). The second scenario this paper studies is a common middle game scenario where one player has used an economy strategy (and thus has a large army) and the other player has used a tech strategy (and thus has a smaller, but high tech army).

Micro-management (or micro) is the tactical aspect of controlling combat units in the most effective way possible. Since the game is played in Real-Time (as opposed to a turn-based game), the human player is greatly limited in time that can be used in order to micro combat units properly. Combat units can only be issued commands in groups of 12 at a time. This means that when larger amounts of units are involved, it requires many mouse-clicks and keyboard button presses to issue commands to all those units. Professional Starcraft players train to increase the speed that they can click the mouse and press buttons on the keyboard in order to micro combat units faster. Player's speed can be measured in terms of actions per minute, where each action is a command that has been issued in the game. Novice players will have speeds of less than 60 actions per minute and will see large improvements in win rate as they increase their speed up to 150 actions per minute. After 150 actions per minute, there are diminishing returns since not all the actions end up being effective. However, many professional players find improvements as the actions per minute increase well beyond 150. Many professional players can average sustained speeds of over 300 actions per minute and short bursts of over 500 actions per minute. These speeds can be absolutely overwhelming to witness for players new to the game, but these speeds are needed in order to macro and micro at the same time.

Even with this amazing speed, players are constantly having to make compromises in macro or micro because there is simply not enough time to do everything. And in Starcraft, if you must choose between compromising on macro management or compromising on micro, you should compromise on micro. Because of this, there is a great deal of strategy improvement that can be done in micromanagement using machine learning because the computer speed means it doesn't need to compromise on micro.

Micro plays such a key role in successfully winning battles that strategies that distract the opponent, such as attacking in multiple places at once, can be extremely effective. These strategies can often outright win games against all but the most skilled opponents. Because micro requires such a large time investment for humans and there is such a limited amount of time in each game, this category of Starcraft strategy is the part where machine learning will find results most quickly. This project focuses on the results of improving micromanagement in two common and pivotal engagements in Starcraft when good micro will even be at its most important.

==

Methodology:

The project has two parts to it. The first part generates a dataset using a program written in Java to repeatedly setup the two common combat engagements described above.

Setting up the simulated combat scenario:

The battlefield in Starcraft is determined by the "map" (a file created by the Starcraft Map Editor program with a file extension of SCM or SCX) and contains all the battlefield terrain (and in this case the combat units needed for the combat scenarios). The map file is selected at the beginning of each game. A custom map was created for the purpose of this project for each of the combat scenarios. The contents of the first map are: 8 defending combat units (Zealots), and three defending buildings (Nexus, Pylon, and Photon Cannon), and 24 attacking combat units (Zerglings).

[Include Picture of Combat setup]

For the second combat scenario, a separate map was created. This map contains 24 attacking combat units (12 Zealots and 12 Dragoons) and 20 defending combat units (8 Siege Tanks and 12 Vultures).

[Include Picture of Combat setup]

In these pictures, the combat scenario has not been set up and the combat units are not in their starting positions.

Interfacing Programmatically with Starcraft:

Before combat occurs, the combat units will need to be moved to their starting location. For this a Java program was written, that utilizes the BWAPI tool to interface with Starcraft.

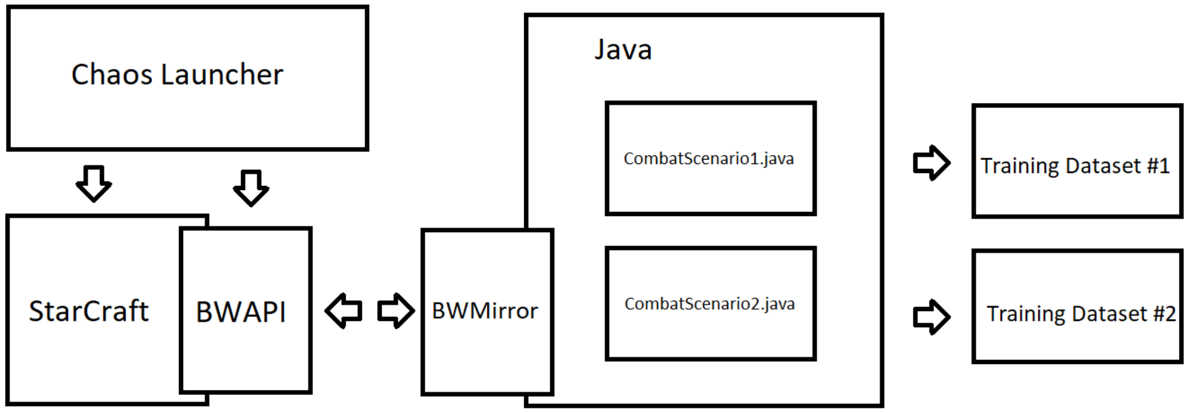

Figure 2: Starcraft Broodwar version 1.16.1 was used because it is compatible with BWAPI 4.12. Both C++ and Java can be used with BWAPI. Java was used which required that we also use BWMirror.

StarCraft is started with the Chaos Launcher tool and enables the BWAPI interface. The BWAPI tool allows a C++ interface to interact with Starcraft. BWMirror is a Java wrapper for the C++ interface, allowing for two custom Java programs to control the combat scenario. The two Java programs each position combat units for the engagement, command the combat units to engage in battle, and then stores the result in a CSV to be used as training data.

Setting up the Combat Scenario:

One hurdle that had to be overcome with this project was setting up each individual run of the combat scenario. For this project, each unit will be placed randomly within a specific area. For the first combat scenario, the defending combat units will be placed randomly in the blue grid and the attacking combat units will be placed randomly in the red grid.

[Include Picture of Combat setup showing the spawning grids]

At the beginning of the combat, the Java program is used to move all the combat units into position. Then one army is given the command to attack.

One problem is our Java program can only control one player at a time. However,to create the combat scenario, armies from both players need to be positioned. In order to do this, “triggers” were added to the map that would temporarily give control of all combat units to the player being controlled by the Java program. After positioning both armies, a second map trigger returns control of the combat units to their original player. There is room for improvement in this part of the project since triggers don’t operate as reliably at the same game time each time. This could be improved by having two Java programs connect to the game using network play.

Evaluating Result of a Combat Scenario:

Each battle is scored based on the number of combat units that survived, and how healthy they are. For scenario 1, if the buildings are destroyed there is a penalty. Each combat unit that is still alive will be worth 50 points. Each remaining hit point will be worth one point, and each remaining shield point will be worth 0.5 points. Also, I may decide to award extra points for combat engagements that end quickly (meaning the combat units are then freed to do other tasks).

Collecting the dataset:

To collect the dataset, I’m going to setup the two scenarios based on a certain time in the game where it is very common for a combat engagement to occur. I’m going to be using the BWAPI program to control the Starcraft game from an external program written in Java. This Java program will use the BWAPI library to automatically start games of Starcraft, issue commands in game, and then read the state of the game after the combat engagement has completed. My plan is to run the StarcrtaftScenarioRunner for several days in order to get a large enough dataset to start training a NN to pick the best commands to issue in the combat engagement. I believe that the first scenario will take less than 30 seconds to complete, and for the second scenario to take less than 60 seconds to complete. However, I believe it may be possible to use BWAPI to speed up the game speed, so I may be able to shorten the time it takes to collect data on each instance of the dataset.

Training the Model:

The “Keras” library is used to create the neural network. The neural network has 66 inputs for the first combat scenario and 114 inputs for the second scenario. The inputs are the coordinates of the combat units for both sides and the command that the combat units were sent to attack towards. The neural network’s output is the estimated score of the battle. The inputs were normalized using the “sklearn” library.

Using the Model to Select the Best Command:

When confronted with a scenario where the defenders and attacker’s positions are known, the best command to control the defenders (or attackers) is done using a grid search. Each possible coordinate for the command is added to the inputs and an estimated battle score is returned from the neural network. The command coordinate that receives the highest estimated battle score is the one chosen as the command for the combat units.

Evaluating Result of a Combat Scenario:

Each combat unit that is still alive will be worth 50 points. Each remaining hit point will be worth one point, and each remaining shield point will be worth 0.5 points. Also, I may decide to award extra points for combat engagements that end quickly (meaning the combat units are then freed to do other tasks).

These values I may decide to tweak, but they will form a basis for scoring the resulting position, similar to how positions are scored in chess. The better the final position is for each in game scenario, the more we want to train the neural net to give the in-game command that will result in that final position.

I’m going to be using Starcraft Broodwar version 1.16.1 because it is compatible with BWAPI 4.2. Both C++ and Java can be used with BWAPI, I plan to use Java.

Inputs and Outputs:

For the two engagements, the inputs are going to be very similar and consist mostly of the position of the units on the battlefield.

First Combat Engagement Scenario:

For the first combat engagement scenario, the NN is going to be faced with a defensive position, where some number of combat units (zerglings) are going to be attacking the NN’s defensive position. The zerglings will be given an attack-move command to a specific grid on the battlefield. There also will be some number of combat units (zealots) that the NN will be able to control. The position of the offensive combat units and defensive combat units will be within a specific area of the battlefield. So the input to the NN will be the position of the combat units, and the specific grid tile that the offensive units were commanded to attack towards. The output of the NN, will be the specific grid tile that the defensive units should be commanded to attack towards. I don't know yet how much computational power will be needed in order to train a NN such as this. However, I will be able to increase or decrease the number of inputs by changing the number of combat units that are be used in the combat engagement. Ideally, we would be looking at there being 5-10 defending combat units, and 12-24 offensive units. Each one of these would be an input into the NN. So likely a minimum number of inputs being 5+12+1=18, and a maximum number of inputs being 10+24+1=35 for the first combat engagement scenario.

Second Combat Engagement Scenario:

Here's a video of a similar combat scenario. The combat engagement lasts about 30-40 seconds. You don't need to watch any longer than that.

https://youtu.be/cu9N9jy43Z0?t=819

For the second combat engagement scenario, the NN will be faced with an offensive position, where the NN is going to be given some number of combat units (zealots and dragoons) and is going to be tasked with picking the best grid tile to attack towards the defending side's army. The defending side is going to include some number of combat units (vulture's, spider mines, and siege tanks) in a stationary position (since these specific combat units have extremely low mobility and generally fight in a stationary position). The inputs to the scenario are going to be the locations of the combat units for both sides, and the output will be the specific grid tile of the battlefield that the NN should command the offensive units to attack towards. Again, the size of the input can be increased or decreased by changing the number of combat units put onto the battlefield. This combat engagement would happen during a time in the game when the armies would be larger, so there would be more inputs to this scenario. Ideally, we would have about 12-30 attacking combat units, and 12-30 defending combat units. So the minimum input to the NN would be 12+12=24, and the maximum would be 30+30=60 for the second combat engagement.

I could also do each engagement multiple times with a different number of combat units (different number of inputs) each time. However, I'm anticipating that I won't have extra time to collect data and train the NN on all these different combat engagement scenarios. So, I will probably just stick with one of each.

Benchmark to beat:

I'd like to compare the output of my project's NN's selected grid tile to picking a random grid tile. I'd also like to compare how the number of instances of data collected affects the effectiveness of the NN.

For the project paper and presentation:

There will be an excellent visual aspect to the report because I can show the actual combat engagements, and several of them can be watched side-by-side to see a visual comparison of the results.

References:

I haven't consulted any literature about this project. I've looked briefly at the Student Starcraft AI Tournament (SSCAIT) before taking this class. But haven't looked up any papers about developing Starcraft AI or using neural networks to control them.

Architecture Diagram for Collecting Training Data:

ChaosLauncher, BWAPI, BWMirror, Java Controller/Data logger

Architecture Diagram for Training Neural Network:

Training Data, Python, Keras

Heatmap diagram for a few sets of starting locations

Table comparing % of victorious battles for random command vs NN command

Table comparing battle score for random command vs NN command, and show how larger training data size improves score

BWAPI:

https://github.com/bwapi/bwapi/releases/tag/v4.1.2

https://sscaitournament.com/index.php?action=tutorial