Deep Imitation Learning for Playing Real Time Strategy Games

Jeffrey Barratt
jbarratt@stanford.edu

Chuanbo Pan
chuanbo@stanford.edu

November 21, 2017

Introduction and Motivation

Competitive Computer Games, despite recent progress in the area, still remain a largely unexplored application of Machine Learning, Artificial Intelligence, and Computer Vision. Board games have long been the standard for advancements in game playing, However, the structure of these games is limited; they are for the most part turn-based and full information games where two players alternate moves and know the full state of the game. This shows limited application to the real world, as the real world operates in real time and it's impractical to know the whole state of the world in one computer.

Thus, Real Time Strategy (RTS) games such as Starcraft II provide an ideal testing environment for artificial intelligence and machine learning techniques, as they are performed in real time and the player cannot see the whole battlefield at once (incomplete information). They must balance making units, controlling those units, executing a strategy, and hiding information from their enemy to successfully win a game of Starcraft II.

Setup

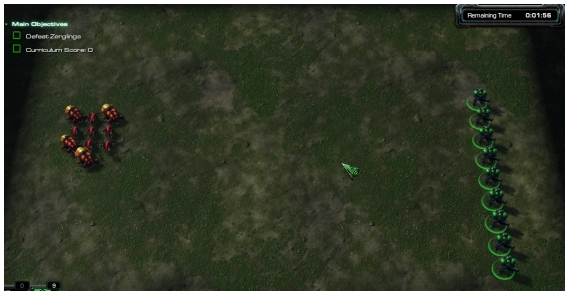
Because StarCraft II is a game comprised of multiple elements, we have decided for at least the milestone to focus only on a certain aspect of the game. This allowed us to set up our framework more easily so we can work on more complicated tasks in the future. Specifically, we decided to focus on the `DefeatZerglingsAndBanelings` minigame to model the basic complexities battling with a group of Marines. At the start of this minigame, the agent is given 9 preselected Marines and must defeat 6 Zerglings and 4 Banelings placed on the other side of the map. The agent can see the entire map and no 'Fog of War' mechanism is in place. When the agent defeats all the Zerglings and Banelings, a new group of Zerglings and Banelings are respawned (6 and 4 of each respectively) and the agent is given 4 additional Marines at full health. The destroyed marines are not re-spawned and the remaining Marines don't recover any health. This cycle continues for either 120 seconds or whenever the agent loses all its Marines. The agent is rewarded 5 points for each Zergling or Baneling defeated, and loses one point for each Marine lost. An example of the map is shown in the below figures.

When the marines are bunched up, they take splash damage from the baneling's dangerous area of effect (AOE) attack.

When the banelings (AOE melee attackers) are allowed to connect with many marines, it is much more difficult to win.

Figure 1: The mechanics of unintelligent use of a stalker vs. a zealot.

We see that this approach shown in Figure 1 is suboptimal for the problem at hand; a smarter approach is needed, as shown in Figure 2 and Figure 3. Although this is one good way to improve the outcome of this particular map, it is a good way to demonstrate many skills in the game of StarCraft II: unit management and prioritization in addition to precision with unit control. We set out to design and build an imitation learning algorithm to tackle this problem, described in more detail below.

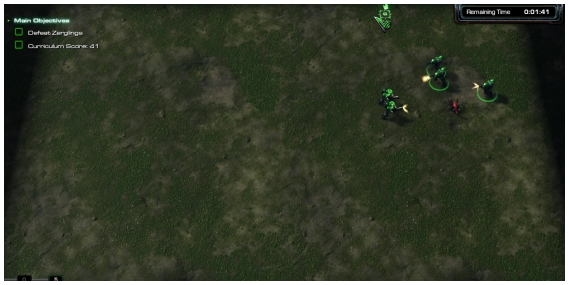


The starting position of the `DefeatZerglingsAndBanelings` minigame.

The marines can start out by attacking the more dangerous banelings first.

Figure 2: The first part of a skirmish between Zerglings and Banelings versus Marines.

Splitting the marines into smaller groups mitigates the effect of the splash damage inflicted by the AOE banelings.



Since zerglings are melee units, kiting (alternating between retreating and attacking with ranged units) can be used to trade more efficiently with the zerglings.

Figure 3: The second part of a skirmish between Zerglings and Banelings versus Marines, including the use of splitting and kiting to keep more marines alive.

Method

Model

We are planning on applying deep learning in addition to behavioral cloning to solve this problem. This will involve collecting human expert, or close to it, data and training via supervised learning to try and mimic those actions. Although maybe not an ideal way to ultimately tackle the problem of performance on StarCraft II in general, this sort of learning will provide a solid player that can play up to human level on certain minigames and possibly eventually full games.

The current version of our model, as seen below in Figure 4, we have four convolution layers, each with a ReLU activation layer, and two batch normalization layers, one at the start and one in the middle, and one fully-connected layer at the end, finally ending in a vector of scores for each possible action. We used a CNN because `PySC2` represents the state of the map as a $84 \times 84 \times 17$ feature tensor, which can be treated as an 'image'. Furthermore, the parameter sharing properties of a CNN is useful in the context of Marines and enemy units being in different locations.

The scores for each action is a softmax over all possible actions (select, move, attack, etc.). Each action has arguments, such as location to perform the action, or a pair of arguments which are the start and end of the rectangle selection to be made. The network is trained on the squared loss of the arguments and a softmax cross entropy loss across the actions.

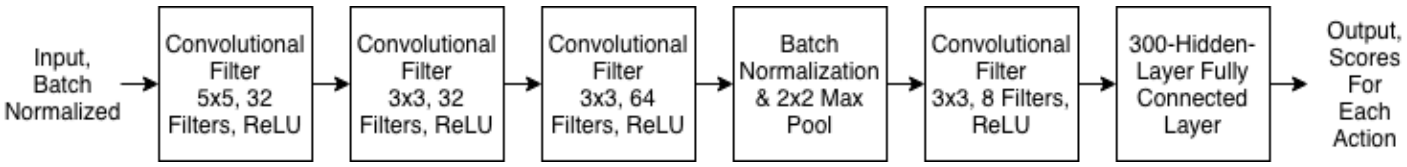


Figure 4: The current version of the model for our agent (Generation 1 Model) .

This initial version of our model allows us to work out the kinks in different parts of our infrastructure and allows us to have preliminary results so we can gauge further improvements to our training/testing process and overall model. Further models will likely include Recurrent Neural Network frameworks such as Long Short Term memory cells in addition to deeper networks.

Data Collection and Training

As with traditional supervised learning, we needed a sufficient amount of data in order to train our model. For games such as StarCraft II, this would involve collecting **Replay Data**. However, since this minigame is new and was created for deep learning purposes (specifically Deep Reinforcement Learning, which is the topic of our CS 221 counterpart), replay data was not readily available. Therefore, we had to collect our own replays by using `PySC2` Library to record the replays of a human agent playing the minigame. Overall, we have so far collected around 10,000 frame-action pairs worth of data.

We will train the network on these collected replays using variants of Stochastic Gradient Descent. As we build more models, we have multiple ways of evaluating them. First, we can compare the training, validation, and test loss of different models between each other. Second, we can compare statistics on what average scores or what distributions of scores different models get. Third, we can also watch the models play the mini-game and evaluate their performance holistically.

Preliminary Experiments (Generation 1 Model)

We were able to transcribe this replay of human expert data into a numpy format that was easy to use for training. This allowed us to train our network outside the framework of `PySC2` and StarCraft II.

We were able to let the training run for a small amount of time and collected data on the loss of the model on the given training data, seen in Figure 5. The loss is defined as the sum of softmax cross entropy loss between our model's probabilities for each of the actions plus the mean squared loss between the model's x,y coordinates for an attack, move, selection, etc. This allowed us to roughly evaluate its effectiveness in the mini-game given at the start of the paper.

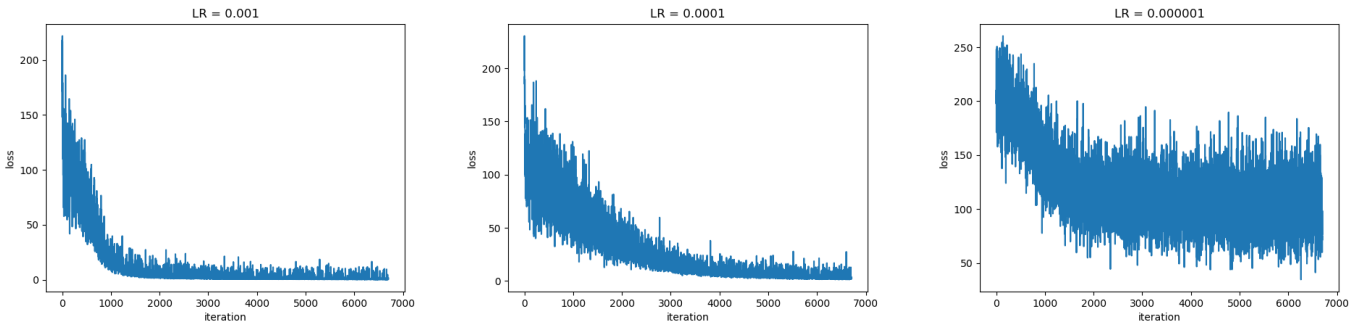


Figure 5: Loss curves for initial training tests across different learning rates.

We can see that we were able to lower the loss to highly satisfactory values given a proper learning rate. However, achieving a loss of around 1 did not translate to exceptional performance. Over 215 games, our agent produced an average score of `32.098`. For comparison, a random agent produced an average score of `21.6645` over 155 games. Such scores are not satisfactory for this project, especially since the best human score was around 750. Primary issues with the model included not splitting correctly and oscillating between two points. Further work on the model and training regimen, discussed below, can help us attain higher performance on our model.

Next Steps

In the coming weeks, we plan on advancing the scope of our dataset to include multiple mini-games as well as a mini full-featured map where the player can play against an easy pre-programmed computer agent. For this part of the project, we may be able to use actual replay data provided by Blizzard.

The bulk of the work for the project lies ahead; a mostly unexplored path of machine learning exists in this project realm and that allows us to have limitless expansion of our project, as much or as little as we can tackle in the given time frame. What we have so far is just a small step in the right direction towards intelligence within StarCraft II.

Contributions

Group Members: Jeffrey Barratt (jbarratt), Chuanbo Pan (chuanbo)
Although Jeffrey has significantly more experience with StarCraft II, the majority of the work involved with this project does not require knowledge of StarCraft II concepts. The only exception to this is data gathering, which will involve Jeffrey, an experienced StarCraft II player, playing the minigame repeatedly until enough data is gathered. Aside from that, both have contributed equally to the progress of the project, often working at the same time through TeamViewer.

Appendix

References and Resources

Below is a brief and informal list of resources we have used.

- **PySC2** (https://github.com/deepmind/pysc2) - StarCraft II Learning Environment created by Blizzard and DeepMind.
- **DeepMind StarCraft II Paper** (https://deepmind.com/documents/110/sc2le.pdf) - Original research paper outlining DeepMind's public research into StarCraft II.
- **TensorFlow** (https://www.tensorflow.org) - Open Source Machine Learning library created and maintained by Google.