Deep Imitation Learning for Playing Real Time Strategy Games

Jeffrey Barratt
Stanford University
353 Serra Mall
jbarratt@cs.stanford.edu

Chuanbo Pan
Stanford University
353 Serra Mall
chuanbo@cs.stanford.edu

Abstract

Competitive Computer Games, such as StarCraft II, remain a largely unexplored and active application of Machine Learning, Artificial Intelligence, and Computer Vision. These games are highly complex as they typically 1) involve incomplete information, 2) include multiple strategies and elements that usually happen concurrently, and 3) run in real-time. For this project, we dive into a minigame for StarCraft II that involves many engagement skills such as focus fire, splitting, and kiting to win battles. This paper goes into the details of implementing an algorithm using behavioral cloning, a subset of imitation learning, to tackle the problem. Human expert replay data is used to train different systems that are evaluated on the minigame. Supervised learning, Convolutional Neural Networks, and Combined Loss Functions are all used in this project. While we have created an agent that shows some basic understanding of the game, the strategies performed are rather primitive. Nevertheless, this project establishes a useful framework that can be used for future expansion. (This project was completed in tandem with a related CS221 project.)

1. Introduction

1.1. Background 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.

For our project, we take in as input the current state of the game. Using the PySC2 Library released by DeepMind, we can represent game states as "images", where each layer contains sptial information of the map (health, location, etc.). We use a Convolutional Neural Network (CNN) to produce outputs which represent the optimal action for the given input state.

1.2. Joint Project with CS221

While this paper will be solely submitted to CS229, we have used the same general infrastructure for both projects. However, we have applied different techniques and models to these problems between the two classes. We applied deep reinforcement learning to the problem for CS 221, while we applied deep imitation learning for CS 229. These models each present their own difficulties (data collection, algorithms, convergence, etc.).

2. Related Work

Games have long been the study of researchers in computer science, and machine learning has recently grown in popularity to apply to these problems. Computer vision and CNN's have very frequently been the methods applied to games. For example, Go, an ancient Chinese board game, has gained popularity because of DeepMind's work in the field with their recent success of AlphaGo [5] being able to beat the best player in the world. In fact, some research has been looked into not even exploring the game tree to play the game [1], an aspect of the game previously thought to take much thought in state exploration. Other than Go, there has been recent news about AlphaZero, a general reinforcement learning player that was able to beat top bots in Chess, Chinese Chess, and even Go [6] using the same learning infrastructure and model for all games.

For real-time games, much work has been done, most popularly in Atari games by DeepMind [4], where they used a DeepQ learning approach with experience replay.

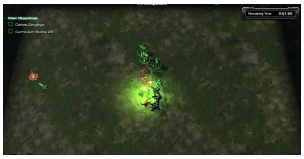
StarCraft: Brood War is the predecessor to StarCraft II and remains a popular AI testing environment. Much work has been done on this game because of its accessibility and APIs. Approaches to playing the game include work on macro-management[3], the management of all unit-creating and resource-gathering units and structures, and grouping of units for movement around the map[7]. However, virtually no computer vision techniques have been applied to the game despite the large amount of work and infrastructure, including an agent tournament circuit, created for the game.

Despite what has been mentioned, little work has been done within the space of StarCraft II, limited only to a few papers on datasets for macro-management [9] and using logistic regression to evaluate game balance [10]. These papers don't propose or evaluate any strategies for actually playing the game, only giving meta-analysis of the game itself in the form of datasets or regression. This lack of work is due to the relative difficulty to work on the game; before a few months ago, there was no available interface to the game, and the creators of the game worked hard to make sure there was no available entry point to the game to prevent the possibility of cheating by human players.

The only paper that we could find on this topic is DeepMind's own paper [8] which introduced PySC2 [2], an interface between StarCraft II and Python, and also provided some baseline reinforcement learning algorithms. These algorithms did not perform very well on various minigames proposed in the paper and available from PySC2. However, this paper was simply an introduction to the game and the environment itself, and StarCraft II is an active area of research for DeepMind, so further progress and publications are expected on this front.

3. Problem Definition

Because StarCraft II is a game comprised of multiple elements, we have decided 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 focused on the `DefeatZerglingsAndBanelings` minigame to model the complexities of battling with a group of Marines. At the start, 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 is 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 enemy defeated, and loses one point for each Marine lost.

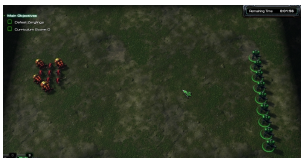


(a) When the marines are bunched up, they take splash damage from the baneling's dangerous area of effect (AOE) attack.

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

Figure 1: Lack of splitting leads to quick defeat.

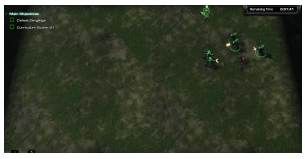
This approach shown in **Figure 1** is suboptimal for the problem at hand; a smarter approach is needed, as shown in **Figure 2**. Although this is one good way to improve the outcome of this particular map, it is a good way to demonstrate many skills in the game of StarCraft II: unit management and 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.



(a) The starting position of the minigame, with 9 marines and 10 enemies.

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

(c) Splitting the marines into smaller groups mitigates the effect of the splash damage inflicted by the AOE banelings, and takes precise mouse movement.



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

Figure 2: The second part of a skirmish between Zerglings and Banelings versus Marines, including the use of splitting and kiting to keep more marines alive.

4. Dataset and Features

4.1. Data Collection

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 (after pre-processing).

4.2. Data Preprocessing

StarCraft II replay data files are binary files that can't be trained on directly. Therefore, we used the PySC2 library to establish an interface between Python data structures and StarCraft II raw data at each given time step. The current state of the game is represented as an $84 \times 84 \times 17$ feature tensor, and is associated with a ground truth label representing the action to take. However, we still cannot train on this directly. Instead, we need to process the ground truth labels even further. Internally, PySC2 represents actions as function calls. For example, the `select_box` action (which selects a rectangle on the screen) can be represented as such:

```
func select_box(x, y, height, width);
```

There are hundreds of possible actions, each with a variable number of parameters. Sometimes, the replay will contain actions that bear no effect to the game but are for logging or human convenience. Therefore, for the scope of this project, we focus only on the five following actions shown in **Table 1**. For the purpose of our minigame, these are the only actions necessary. Our parser works by taking in a valid

Action	Parameters	Notes
`noop`	NONE	Do Nothing
`select_box`	`x, y, w, h`	Multi-Select
`select`	`x, y`	Single Select
`move`	`x, y`	Move To (x, y)
`attack`	`x, y`	Move and Attack

Table 1: A description of the actions used in this project.

action and then transforming the function-parameter representation into a 5×5 sparse matrix. The first column is a one-hot representing the action whereas the following four columns represent a parameter, if necessary. Additionally, we also wrote a reverse-parser, capable of transforming our network interpretation back into a function-parameter format for SC2. For the sake of convenience, we saved the pre-processed data as NumPy arrays for easier access.

4.3. Features

Unlike actions, which needs to be transformed into network-compatible matrices, the map data provided by PySC2 can be used directly. **Figure 3** shows features provided by the PySC2 interface. We see that the information

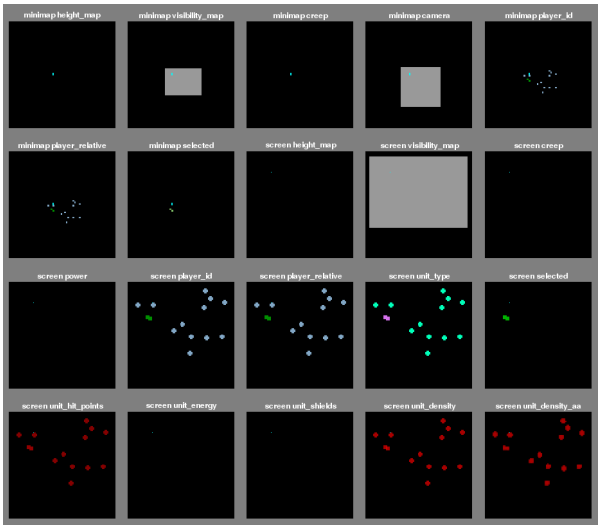


Figure 3: Features and their image representations

is contained spatially. Therefore, it is intuitive to use Convolutional Neural Networks to approach these features.

5. Methods

Figure 3 also shows us that the feature information is contained spatially within the state. Therefore, it is intuitive to use CNN's to approach these features. The parameter sharing properties of a CNN is also useful in the context of Marines and enemy units being in different locations.

5.1. Vanilla Model

The initial version of our model is shown 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. This network outputs a 5×5 action matrix. The action with the highest score is chosen and it's respective parameters are selected.

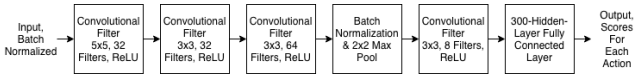


Figure 4: The vanilla actor model.

We defined the loss function as a combination of the Softmax Cross-Entropy loss on the action column, and the mean squared loss on the parameters. The softmax function

outputs probabilities for each action, and the loss calculates the deviation from ground truth. Mean squared loss is simply an average of squared euclidean distances. The loss is shown below. We denote $y_{i,j}$ as the column/row element.

$$L = \frac{1}{m}\left(\sum_{i=1}^{M}\sum_{c=2}^{5}||y_c^{(i)} - \hat{y}_c^{(i)}||_2^2 - \sum_{i=1}^{M}\sum_{k=1}^{5} y_{1,k}^{(i)} \log \hat{y}_{1,k}^{(i)}\right)$$

We also opted to use the relatively new batch normalization technique across training. As data passes through the network, it may be changed to become extremely large or small, an issue called "invariant covariate shift". Batch normalization accounts for this by adjusting layer outputs which helps lead to better training and improved accuracy. Lastly, we used regularization and compared regularized vs non-regularized models.

5.2. Model With Previous Actions

Our next model involved passing in previous actions as inputs along with the current state. The motivating reason behind this is that the Vanilla Network is completely reflex based. That is, it takes a state and outputs an action without any context of possible previous actions. In StarCraft II, it's common to play certain actions in a sequence. That is, given the previous action was selection, it's more likely that the following action is attacking or moving.

The structure behind the Vanilla Network is completely maintained. The 5×5 action matrix is flattened to a vector in $\mathbb{R}^{25}$ and fed through two fully connected layers with 100 hidden units each. The output is also a vector in $\mathbb{R}^{25}$ that's reshaped into a 5×5 action matrix and appended to the CNN output. We did not use convolutions for the actions because there is no spatial information to learn from. Therefore, a simple fully connected network (FCN) sufficed.

5.2.1 Variation on Number of Hidden Layers

Additionally, we experimented with the number of hidden convolutional and fully connected layers in the network. Specifically, we experimented with a "shallow" network which uses only one convolution and one fully-connected, as well as a "deep" network that uses five of each. The idea was to analyze the tradeoffs of faster reflexes vs more thinking. This is important in a real-time game.

6. Experiments

6.1. Model Evaluation

We evaluated our network through a combination of several techniques. Our primary test involved performing playouts on the minigame. This allows us to gauge practical performance of the model. To do this, we wrote a custom Python script that gathers score data at the end of every episode. In order to evaluate our performance during the learning process, we ran these playouts after every epoch.

Since our problem can be divided into a classification problem (for actions) and a regression problem (for parameter selection), our next set of evaluations involved looking at the accuracy of selecting the correct action and of choosing the correct parameters, respectively. These results provide insight into the training process and lets us know if training is actually successful.

For accuracy and loss evaluation, we also used a validation set. We ran our validation set at the end of every epoch. This allowed us to compare performance between training and validation and gave us insight into whether or not we were overfitting to our training set. Our Train-Validation-Test split was roughly 80-10-10.

6.2. Hyperparameter Selection

We chose our learning rate by running Minibatch SGD and examining the learning curve on just the training set. An example of the learning curves is shown in **Figure 5**. For all our networks, we found that a learning rate of around 0.001 was sufficient for around 30 epochs worth of training. The results were consistent, as opposed to lower rates.

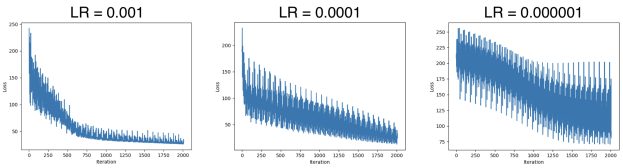


Figure 5: Learning Rates (Vanilla Network)

Due to the time limitations imposed and the use of personal GPU's, we capped our mini-batch size at 25.

We selected a regularization parameter of $\lambda = 0.1$ as a preliminary value to experiment regularization with. Batch regularization also has regularizing effects.

7. Results and Analysis

Figure 6 shows average performance after every epoch. We observe that our all models exhibit some form of learning. A random agent produces a score of around `21.664` over 155 games. Therefore, all our networks pass baseline performance being better than random. However, we note that after around 10 epochs, training seems to stagnate significantly. Judging by the error bars, most models seem to maintain the same level of performance after every iteration. Therefore, none of the models were able to reach the average 242.885 achieved by the human expert. To see why, it's best to observe what the agent was doing once it "converged." **Figure 7** depicts common strategy that the agent would consistently take, which is very similar to the strategy shown in **Figure 1**. The agent typically would not split

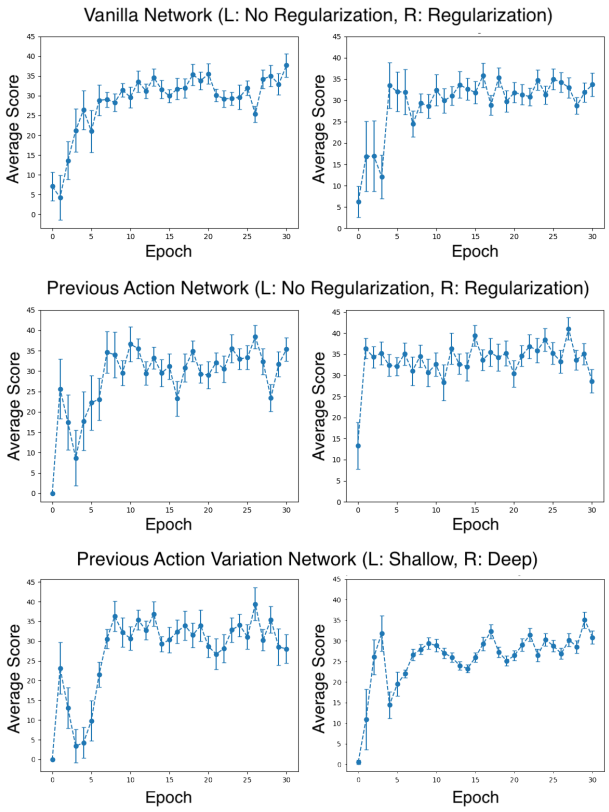


Figure 6: Playout results after every epoch.

after attacking. A possible explanation involves analyzing

Figure 7: Charging without splitting

the starting pattern. At the start of each episode, the agent's Marines are laid out in a very predictable way, and, as seen in **Figure 2**, it is customary to start the episode by charging. Therefore, going from a relatively consistent state to charging is expected. However, after charging, the states become inconsistent and more varied. Although moves and actions are deterministic, outcomes are not as there is some randomness when firing. Therefore, outcomes can vary rather quickly. The agent was able to imitate how to initiate the battle, but was unable to go any further.

This points to a bigger issue, namely overfitting. As seen in **Table 2** and **Table 3**, all our models show characteristics of high variance. Unsurprisingly, deeper networks tended to overfit more. In terms of performance, taking into account previous actions was beneficial in raising overall scores. This is due to the Vanilla Network occasionally getting in an infinite loop scenario, where the agent would go back and forth between two states.

Network	Reg	Train	Val	Test
VanillaNet	N	0.994	0.649	0.671
VanillaNet	Y	0.994	0.753	0.732
PrevAction	N	0.978	0.800	0.751
PrevAction	Y	0.904	0.691	0.718
PrevActTiny	Y	0.966	0.748	0.708
PrevActDeep	Y	0.938	0.682	0.647

Table 2: Action Accuracy Results (Accuracy Evaluation)

Network	Reg	Train	Val	Test
VanillaNet	N	4.719	139.589	125.173
VanillaNet	Y	7.826	100.374	112.902
PrevAction	N	14.149	90.891	103.006
PrevAction	Y	29.466	123.232	137.216
PrevActTiny	Y	18.589	102.945	101.718
PrevActDeep	Y	20.5747	126.859	124.518

Table 3: Final Param Loss Results (Parameter Evaluation)

Given that our issue appears to be high variance, we should look into gathering more data for this minigame as 10,000 state action pairs does not seem to be enough. Since no professional replays exist and we had to gather data on our own time, we were limited in this regard.

8. Conclusion and Future Work

We end our report with some general observations. Overall, we are not near maximum potential. Although the agent learned how to find and attack the enemies, it did not achieve an advance understanding of splitting and kiting. Additionally, the agent sometimes gets stuck moving back and forth between two states. That said, we were excited about the prospects of our what we can potentially achieve. Just by adding in previous states we were able to slightly increase performance. We hope to analyze features and develop better heuristics to improve performance even more.

Were we to have more time and resources to work on this project, we would consider evaluating more advanced models such as a Long-Short Term Memory Recurrent Neural Network (LSTM RNN) to help determine actions, especially in a sequence. We would work towards refining our abstraction to ensure stability during training. Finally, we would recruit more people to gather more training data to cover more possible cases, especially those cases that exhibit desired behavior.

References

[1] J. Barratt and C. Pan. Playing go without game tree search using convolutional neural networks. 2016.

[2] Google DeepMind. PySc2 – StarCraft II Learning Environment, 2017–. [Online].

[3] N. Justesen and S. Risi. Learning macromanagement in starcraft from replays using deep learning. 2017.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. 2013.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[6] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. 2017.

[7] G. Synnaeve and P. Bessiere. A dataset for starcraft ai & an example of armies clustering. 2012.

[8] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Kttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing. Starcraft ii: A new challenge for reinforcement learning. 2017.

[9] H. Wu, J. Zhang, and K. Huang. Msc: A dataset for macromanagement in starcraft ii, 2017.

[10] H. Yun. Using logistic regression to analyze the balance of a game: The case of starcraft ii, 2011.

9. Contributions

Name	SUNet
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 involved 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.

10. Combined Project

As per Piazza Post @2367, we will immediately release our CS 221 Paper, title `Deep Reinforcement Learning for Playing Real Time Strategy Games`, to the Teaching Assistants upon request.