Learning Operational Strategy Electing Robot

ECS170, Spring 2018

Group Members

Davey Jay Belliss	dybeliss@ucdavis.edu	912133893
Joshua Zhang	jykzhang@ucdavis.edu	912132516
Anjo Gabriel Cordero	ascordero@ucdavis.edu	912327813
Thomas Tran	tsmtran@ucdavis.edu	912205281
Matthew Stickle	mpstickle@ucdavis.edu	913359026
Alex Derebenskiy	avderebenskiy@ucdavis.edu	999814491

Problem and Contribution Statement

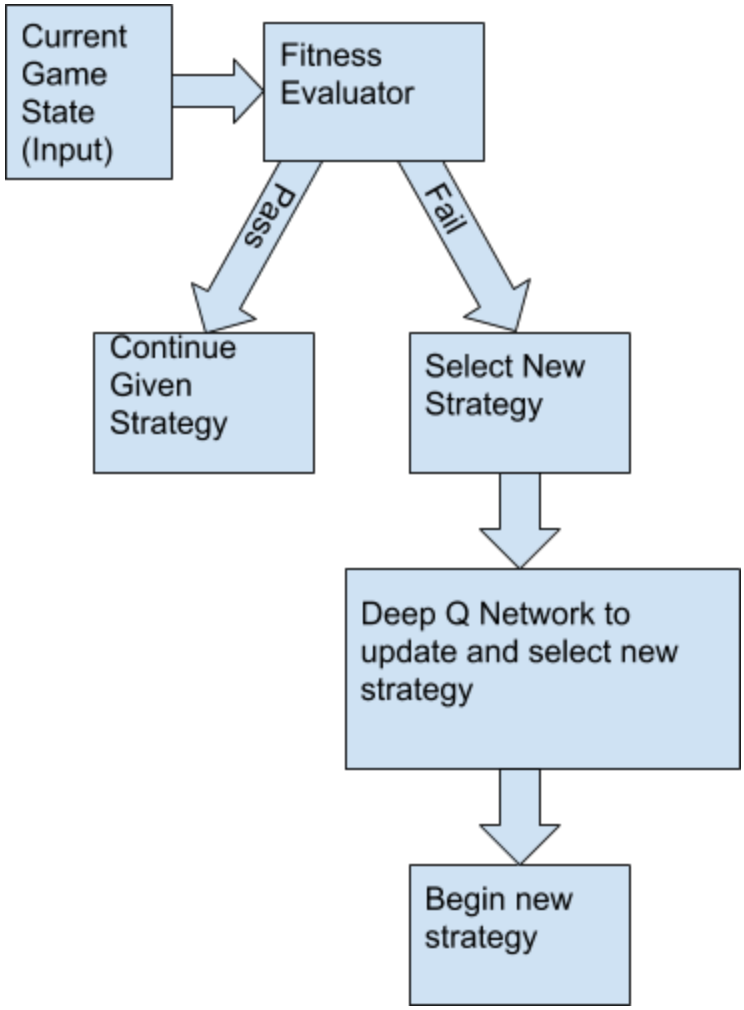
We are proposing to develop an artificially intelligent bot that will start a game with a default strategy, but then adapt to the opponent as more information is gathered. We chose to create a learning and scripted bot since it has the benefits of being both a scripted bot (using strategies known to be in the meta) and a learning bot (being able to adapt). The scripted portion of the bot will be broken down into a series of modules that the intelligent portion of the bot will be able to mix and match such that the bot will be more flexible to a given situation and harder to counter. Overall, our contribution to the problem is intelligent Starcraft 2 AI is a bot will be able to play a full game of starcraft in a (hopefully) intelligent way.

Since Starcraft II has several potential strategies for any given situation there is not a well defined “best solution” for how a bot should select a strategy nor how it will adapt if the opponent effectively counters the initial strategy. This will show that deep learning can be used in a competitive environment, even in a complex game like Starcraft II which is only partially observable and has a wide search space. Additionally it will show that a machine learning AI can adapt to new situations and be able to select a strategy on limited information given, and be able to adapt on the fly.

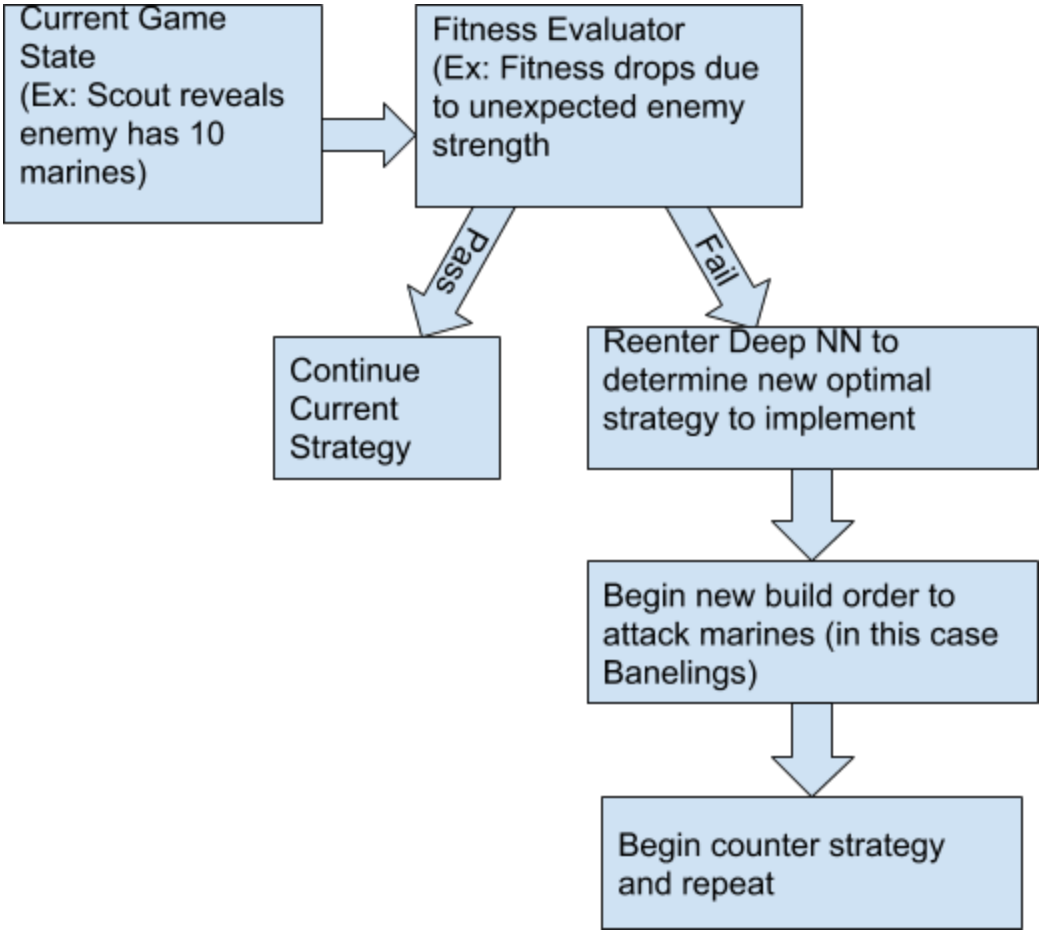
There has been work done by others that have developed similar bots that utilize more of a sparse strategy that focuses on winning with other races such as Terrans, as demonstrated by Steven Brown. However, instead of Steven’s approach where he checks to see if the given state he is in currently is the same as the previous step, we will use a combination of checking current states and evaluating overall “fitness” to help determine what the bot should do next. We hope to create an intelligent selection process that maintains its current strategy unless our fitness evaluation reveals that we need to change our given strategy to optimize our strategy and adapt to our opponent’s actions. For this intelligent selection process, we plan to use the AI approaches of neural networks and genetic algorithms.

The project was well scoped and held for the duration of the project.

Design and Technical Approach



Learning Operational Strategy Electing Robot Flowchart



Learning Operational Strategy Electing Robot Flowchart With Example

We will implement our AI in python with Pycharm, Github, a combination of python-sc2 with the pysc2 gui, OpenpAi's RL algorithms, keras, and TensorFlow technologies. PyCharm is a well known IDE that will provide useful programming features such as syntax correction, linting and debugging. We will use Github to provide version control with the ability to test our code changes and merge the work of our different programmers into the master branch. Our AI will be based on the skeleton provided by python-sc2 and facilitate the development of our machine learning algorithms, using Tensorflow or a similar machine learning library. To train the bot, we will use the replays from the Starcraft 2 protobuf client github repository and use a combination of training online using the google cloud and offline using a combination of the Nvidia graphics cards that we have available to us (2 970, 1 1050, 1 1060).

We plan on primarily utilizing a Deep Q Learning algorithm that uses a neural network to approximate the Q values such that we have a history of previous trainings so that we can continue to update our weights to optimize the performance of our bot, similar to the implementation by Steven Brown. However, in order to improve his approach, we plan to incorporate a fitness function that will help determine when we need to enter the neural network and choose a new strategy. This implementation is advantageous because it can be "lazy" in that we do not have to constantly input all of the data that we receive from the protobuf into our neural

network. However, in order to ensure that our fitness function does not encourage stagnation, the bot will enter the neural network every X iterations to verify that the modular strategy it has selected is still optimal. In order to determine fitness, we have defined the following functions:

$$Bot\ Fitness = (\%creep\ spread) + (\%vision) + econ(\#\ of\ buildings + active\ workers - inactive\ workers - (unspent\ resources/1000)) + (army\ size)$$

$$Estimated\ Opponent\ Fitness = (army\ size) + (building\ compositions) + (\#\ of\ expansions)$$

$$Overall\ Fitness = Bot\ Fitness - Estimated\ Opponent\ Fitness$$

If the fitness reveals that the current strategy is no longer deemed as optimal, we will pass our current army composition, buildings, and upgrades that we have as well as the same information regarding the opponents. The neural network will have access to a Q table, which will not only act as our history from past training sessions, but will also provide the weights that lead the bot to determine the optimal strategy. The Q space, in this case, will be the scripted strategies that the bot will have access to and excludes any of the scripted strategies that would be inefficient to perform from the current game state. The training will initiate a neural network with randomized weights which will be normalized between -2 and 2, as suggested by Josh Patterson, to aid in the convergence of our network.

Depending on the output of the neural network, the bot will piece together a scripted strategy that it thinks is optimal for the given situation. By scripted strategies, we mean to have our strategies categorized into, for example, build orders/timing, troop compositions for counters, and scouting strategies.

Our combat strategies will be a series of hard counters and unit compositions that the bot would choose.Examples would be banelings for massed marines or counters to flying units if the bot detects that the opponent has a flying unit or has begun building structures to allow for the creation of flying units.

Originally our plan was to use pysc2 to create the Starcraft 2 agents. Working with pysc2 was very difficult due to the strange nature of giving actions to units, and the lack of documentation or clear examples. Once we were exposed to python-sc2, we switched over immediately. This change set us back a little, but overall has helped us make much more progress on that project than if we had stayed with pysc2.

We have also split our original idea of strategies into 2 clearly defined parts. These two parts are the unit composition (build) and the tactic (attack, defend, harass, or scout). This allows the neural network a wider range of options to choose from, and helps with splitting the work among our group. It also allows each agent to only focus on building units, while attacking, defending, scouting, and harassing are all performed the same way for each agent.

Scope and Timeline

There are starcraft bots that use Q tables, games that use deep Q learning, and genetic algorithms to help explore large search spaces. Since Starcraft has a large search space and the potential to learn to adapt to strategies, we aim to fill the gap by making a single bot that can implement four different build orders, nine

different strategies, and the ability to adapt to any given game state by analyzing the current state, determining its own fitness, and using the Q network to decide the most optimal strategy.

There are a few existing Starcraft 2 bots that use deep learning, so we will use the python-sc2 framework as a base to start our development. Development can easily be segmented between the learning part of the bot and the scripted part of the bot, allowing our 6 person team to work in parallel to rapidly make progress. Some of our team members have experience with working with large groups through ECS160: Software development and ECS193: Senior Design. Many of us are also experienced with Git which will help us reduce time spent learning git. Below is an estimated timeline for the next 5 weeks of work on the project that will allow us to have our bot completed by week 10.

By Week 5 (Monday):

- Project Proposal

By Week 6:

- Installed PySC2 environment
- Ran the default learning agents
- Decide on properties for the fitness function
- Implement deep Q learning based on tutorials online in pysc2
- Begin implementing the default strategy bot
- Experiment with Starcraft 2 API

By Week 7:

- Finish translation of codebase from pysc2 to python-sc2
- Reimplement Q-learning for python-sc2
- Add different strategies to the bot
- Scouting
- Implement the fitness function

By Week 8:

- Tweaking the fitness function and introduce mutations for further explorations
- Train
- Add different strategies to the bot

By Week 9:

- Tweaking the fitness function
- Continue training
- Add different strategies to the bot

By Week 10:

- Bug fixes
- Finalizing github pages

Changing to python-sc2 forced us to push our timeline back by about a week to setup the new environment and convert our existing code to python-sc2. The separation of strategies into builds and tactics created additional tasks that needed to be completed. None of our changes affected the scope of

the project.

Documentation and Access

We will be using a public Github repo to facilitate version control for our project. Github will be used to keep a history of changes, merge the work of our programmers, and provide access control to our repository. We will include a readme in the repository which will explain our objective and contain instructions for running our program. On GitHub we will maintain a comprehensive README.md and CHANGE_LOG that details how to run the bot, how the bot works, and what we have tried so far in the bot.

We will also keep a weekly ToDo list that details who will be doing what for each week. This will help us keep track of how work is divided in an informal way that does not require creating a github issue ticket.

Our group never used a weekly ToDo list because our tasks were rapidly changing throughout the project. Instead, we kept track of each other's works by using a discord server with channels dedicated to specific parts of the project.

Github repo

Evaluation

Our criteria for success can be measured in two ways. The first, will be that the bot does change strategies based on what the opponent is building and what the bot's current strategy is. As a simple check, we can add a counter in our code that keeps track of the number of times the strategy changes. The second, is that the strategy the bot changes to is actually an improvement from its previous strategy. This will be verified by having the bot playing against an AI or human with an obvious build order (For example Marines, Marauders, Medivacs), and verify the bot begins building a counter (banelings and infestors).

We will also have a function that will record the results as we are training our bot so that we can monitor the progress of our bot over time. This function will append to a file that will keep track of the win/loss ratio and we expect that as the number of games that we train on that the win rate should increase. The record will also record the number of units built and lost, buildings raided, and other bits from the game summary to explore other interesting trends that may be observable as our bot becomes more intelligent. In order to ensure our bot is using units effectively, we will keep track of how many times our bot tells a unit to cast a spell.

Our bot did not actually end up changing strategies based upon the enemies buildings. Time constraints forced us to have the bot rely on a fitness function that only evaluated the opponents units, rather than also their buildings. (NEURAL NETWORK PEOPLE CHECK THIS OUT, AM I RIGHT? DID YOU INCLUDE A CATCH ALL FOR UNITS INCLUDING BUILDINGS? OR IS IT JUST UNITS?)

Our bot, however, does change strategies (randomly?), and then uses whether it won or lost the game to evaluate whether those changes were good or bad. (yeah maybe NN people should elaborate on this im just writing stuff down lol)

Our bot also does not build units specifically to counter other units, but rather has the ability to switch between build orders/agents over the course of the game, letting it learn whether switching the agent

under certain circumstances was good or bad.

Plan for Deliverables

We will make our github repository public as a usable example of a python-sc2 bot, however we will not submit our code to tournaments or write a conference paper unless our bot is very successful. Our GitHub repo will contain a README that details how to run the bot and how it works in order for others to learn about how to make a starcraft bot.

Separation of Tasks for Team

Task	People Assigned
Install Python-SC2	All
Create the Deep Learning section of the bot	Alex, Matthew, Thomas
Create the fitness function	Davey, Joshua, Anjo
Collect or develop scripted AIs	Davey, Joshua, Anjo
Basic Scouting (Scripting)	Davey, Joshua, Anjo
Train the deep learning AI	Alex, Matthew, Thomas
Tweak the fitness function	Davey, Joshua, Anjo
Advanced Scouting (Stretch: Machine learning)	Alex, Matthew, Thomas
Graphs of Data	Joshua
Code Review	**ALL**
Document code and write good code	**ALL**

References

Open AI's RL algorithms
https://github.com/openai/baselines

proto:
https://github.com/Blizzard/s2client-proto

Installing Headless SC2 on Linux
https://github.com/Blizzard/s2client-proto/blob/master/docs/linux.md

Example for Reinforcement Learning:
https://github.com/chris-chris/pysc2-examples

Python-SC2
https://github.com/Dentosal/python-sc2

StarCraft II RL Tutorial 1
http://chris-chris.ai/2017/08/30/pysc2-tutorial1/
https://github.com/skjb/pysc2-tutorial