Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5514--19-9980

BETASTAR

Leslie N. Smith
Anthony Harrison

NCARAI Branch
Information Technology Division

Evan Reilly
Luke Veenhuis
Regina Wang

Naval Research Enterprise Intern Program
Washington, DC

Alex Saam

Loudoun Academy of Science
Loudoun, VA

January 6, 2020

DISTRIBUTION STATEMENT A: Approved for public release, distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*
06-01-2020

2. REPORT TYPE
NRL Memorandum Report

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
BETASTAR

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Leslie N. Smith, Anthony Harrison, Evan Reilly*, Alex Saam**, Luke Veenhuis*, and Regina Wang*

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
1J06

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER
NRL/MR/5514--19-9980

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

10. SPONSOR / MONITOR'S ACRONYM(S)

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: Approved for public release distribution is unlimited.

13. SUPPLEMENTARY NOTES
*NREIP, Naval Research Laboratory 4555 Overlook Ave., S.W. Washington, DC 20375-5320
**Loudoun Academy of Science, 42075 Loudoun Academy Drive, Leesburg, VA 20175-4717

14. ABSTRACT

Games have been used for decades as an important way to test and evaluate the performance of artificial intelligence systems. StarCraft, considered to be one of the most challenging Real-Time Strategy (RTS) games has emerged as a "grand challenge" for AI research. DeepMind posted on their blog where they described a StarCraft II program they named AlphaStar, the first Artificial Intelligence to defeat a top professional player. The objective of our BetaStar effort was to replicate DeepMind's AlphaStar from the incomplete information available from their blog post. This NRL Memorandum Report presents detailed documentation by all of the project's team members on investigations performed and accomplishments.

15. SUBJECT TERMS
Reinforcement learning   Deep learning   StarCraft
Alpha Start   Intelligent agents

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
					Leslie N. Smith
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	Unclassified Unlimited	47	**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified Unlimited	Unclassified Unlimited	Unclassified Unlimited			(202) 767-9532

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

This page intentionally left blank.

BetaStar

Introduction

Games have been used for decades as an important way to test and evaluate the performance of artificial intelligence systems. As capabilities have increased, the deep learning research community has sought games with increasing complexity that capture different elements of intelligence required to solve scientific and real-world problems. After the conquest of Chess by AI in the 1990s and GO by machine learning (ML) in 2015, academic and commercial researchers have moved their focus to StarCraft II (SC2) with the goal of defeating the world champion. StarCraft, considered to be one of the most challenging Real-Time Strategy (RTS) games has emerged as a “grand challenge” for AI research.

DeepMind posted on their blog where they described a StarCraft II program they named AlphaStar, the first Artificial Intelligence to defeat a top professional player (see more information on their post at https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii).

The objective of our BetaStar effort was to replicate DeepMind’s AlphaStar from the incomplete information available from their blog post. Automating strategic decision making is critical for the US Department of Defense because this technology has the potential to revolutionize warfare. The motivation for our effort was derived from a Navy effort to examine the potential of machine-assisted learning and Automated Intelligence (AI) technologies to assist with assessing complex patterns and algorithms found in advanced systems, fusion systems, and unmanned systems. Currently, existing threat assessment approaches and tools do not fully leverage available advanced machine-assisted learning technologies. The integration of ML modeling tools into Naval Enterprise test beds, such as the Next Generation Threat System (NGTS), is a priority because NGTS is a tool which has significant importance to pilot training and will enable the assessment of human performance. An intelligent agent integrated within Naval Enterprise Modeling and Simulation tools, such as NGTS, will provide a challenging simulation environment for training future Department of the Navy warfighters. In addition, super-human tactics developed by such DRL agents can be analyzed as an additional learning experience.

One of our goals after this BetaStar work is to incorporate a Deep Reinforcement Learning algorithm into the Next Generation Threat System (NGTS) to provide threat-assessment capabilities into the core enterprise system. The objective is for the DRL agent to perform (near) optimal battlefield actions, even in novel situations. Techniques described in this report will be the starting point for developing this intelligent agent for NGTS.

Our efforts so far has focused initially on replicating AlphaStar and we have named this algorithm BetaStar. This NRL Technical Report presents detailed documentation by all of the project’s team members on investigations performed, accomplishments achieved, and dead ends encountered. The purpose is to make all of our work performed on replicating AlphaStar easily replicable by others. The team consists of the authors of this report (listed alphabetically): Anthony Harrison (NRL), Evan Reilly

Manuscript approved Month 00, 2019

(summer 2019 intern), Alex Saam (summer 2019 intern), Leslie N. Smith (NRL), Luke Veenhuis (summer 2019 intern), and Regina Wang (summer 2019 intern).

Evan Reilly

Utilization of Open Source Strategic AI Frameworks for Defense Applications

Executive Summary

After the conquest of GO and Chess by AI, academic and commercial researchers have moved their focus to StarCraft II (SC2) with the goal of defeating the world champion. From this bevy of academic and commercial research, open-sourced and freely available tools have been developed to optimize real-time strategic decisions for resource and knowledge constrained situations. The following report outlines the state-of-the-art tools and interfaces that can be leveraged at a low development and computation cost to solve scenarios of relevance to the DoD.

Gameplay Mechanics

StarCraft II: Wings of Liberty is a military science fiction real-time strategy (RTS) computer game developed and published by Blizzard Entertainment in July 2010. The game features three distinct races: Terran – humans with powered combat suits and futuristic tanks and battlecruisers, Protoss – a technologically advanced species with vast mental powers and energy shields, and Zerg – a swarming race of xenomorphic creatures that rapidly evolve and mutate to adapt to any situation. The player can choose any one race to control, and even though the races have greatly different units and abilities, the three races are about equal in power.

In the RTS genre, a game does not progress incrementally in turns like chess or checkers. Instead, each player manages resource gathering, base construction, technological development, unit production, and combat at the same time as their opponent. To win a game, a player must accumulate resources, construct buildings, amass an army, and eliminate all of the opponent’s buildings. A game typically lasts from a few minutes to one hour, and early actions taken in the game (e.g., which buildings and units are built) have long term consequences that are often unpredictable.

The tasks a player must perform to succeed at an RTS can be very demanding, and complex user interfaces have evolved to cope with the challenge. The screen is divided into a map area isometrically displaying the game world terrain, units, and buildings, and an interface overlay containing command and production controls and a small overview of the entire map.

In a 1v1 game of Starcraft II, two opponents spawn on a map which contains resources and other elements such as ramps, bottlenecks, and islands. Players do not begin with any information about their opponent due to the “fog of war” which obstructs the view on the map anywhere the player does not have units present. To dynamically understand and react to their opponents’ strategy, a player must send units to scout their opponent’s base regularly or risk being ill-equipped to handle an attack.

Gameplay Strategy

This game is more complicated than chess and go because .....

- The level of complexity of the game compared to chess and go
- Suite of actions
    - All actions are not immediately available
- Suite of units
    - Require specific buildings and add-ons
- Suite of upgrades
    - Cost resources
    - Require lower tier upgrades first
- Build orders?
    - Complexity human vs. ai implemented
        - Just a check-list
- Scouting?
- Comparison of existing AI
    - Lacks reactionary capabilities based upon units
    - Just sticks to build orders

The basis for decision making is…

API Library Layout

The primary reason for using open-source software libraries and projects is to cut down on in-house development cost while leveraging a continuous improving code base of new ideas and bug fixes. As part of Google's Deep Mind Group's research they have open-sourced their tool set to, one, drive publicity for the project but to more importantly to absorb ideas from other research groups that feel inclined to push updates. While this NRL research group has no initial desire or strategy for pushing sanitized updates to the larger code-base, the group does plan to continue to utilize any new features or algorithms that do become available.

The two primary open source projects that the group has implemented are PySC2 which is managed by Google's Deep Mind Group and OpenAI Gym which is managed by the non-profit AI research company OpenAI. The PySC2 library exposes core functions of Starcraft II commands and gameplay screens through a python wrapper for an agent to interact with. OpenAI Gym houses a suite of algorithms that generate agents to solve various games like Tic-Tac-Toe, checkers, and Chess to digital games like Pong, Space Invaders, and Paddle Ball. Therefore by connecting these two repositories together various algorithms can be quickly implemented and tested.

PySC2

The PySC2 API is a python library which contains the primary functions for collecting game play data and executing actions. Figure 1 is a chart referenced from [1] that depicts the workflow of the software framework that PySC2 exposes to a developer. The learning environment allows for observations of the game to be transformed into features of both the screen and the associated mini-map. An example of these screen features is shown in Figure 2 also from [1]. Following the computation of a reward function defined by the developer, an agent network is able to select an action a result of its policy and input an action back into the game. The cycle then continues until the conclusion of the match with new screen features and possible actions evolving based upon gameplay.

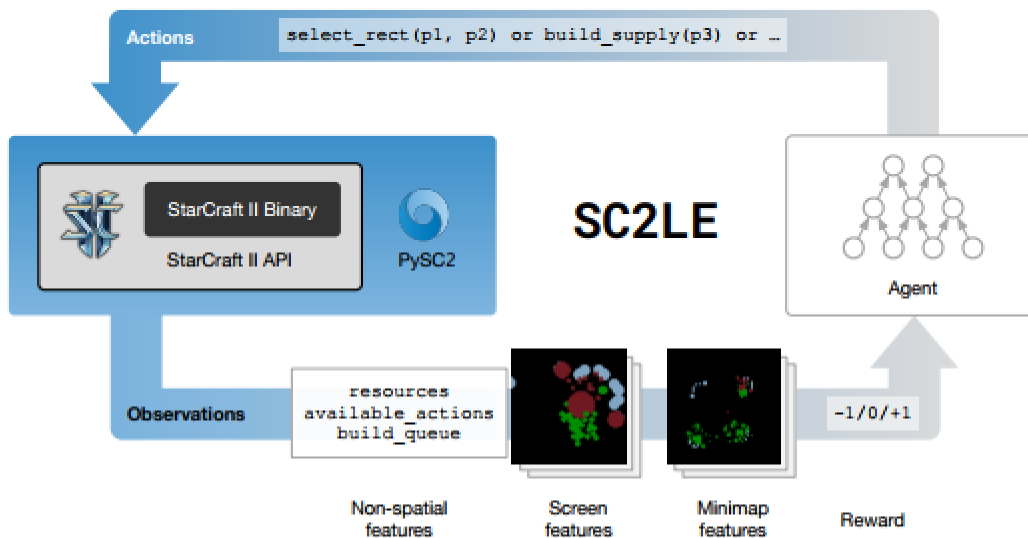


Figure 1 depicts the process flow of information from the game to an agent who then inputs and action back into the game [1].

The API is broken up into 5 main structural sections: agents, bin, env, lib, and maps.

The agents segment holds first the base class appropriately named base agent. This class creates the basic structure that can be used to create custom agents. The API also includes a random agent that selects a random action based on currently available actions as well as a random set of arguments for that action. This allows for comparison of any policy with that of a naïve policy.

Following these two basic classes, there are three specifically tailored agents in the custom_agents.py script to “solve” the test mini-game environments. While these agents are rudimentary, these example scripts also allow for well documented examples of reading a specific screen feature and generate an action with the API. Figure 3 taken also from [1] highlights the comparison between the human’s interface and the API’s agent interface.

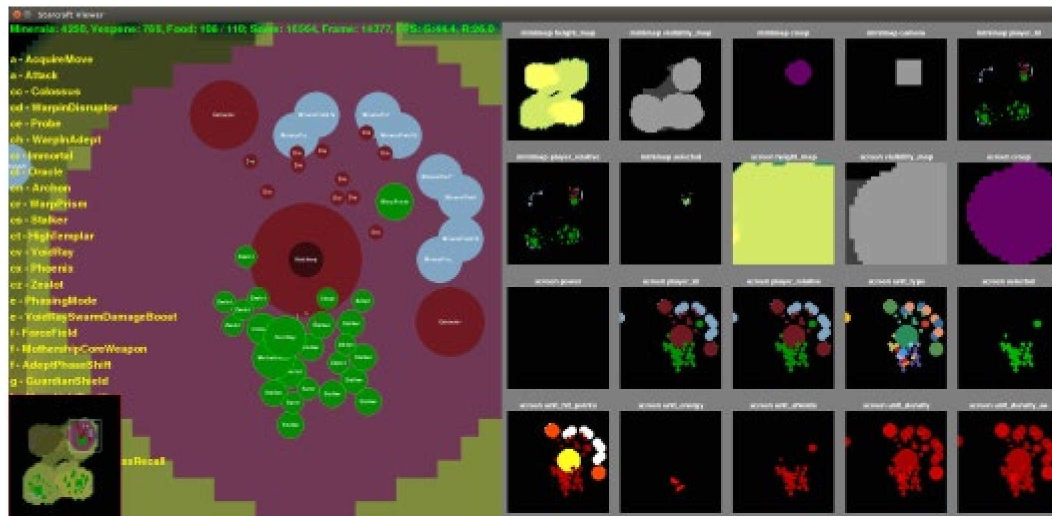


Figure 2 illustrates an example feature map that PySC2 generates based upon the game play map. These features can be processed by an agent before making an action [1].

The *MoveToBeacon* agent implements an index call for the unit's position and the location of the target beacon. After both points are found, the agent outputs a move command to the location half-way between those two points in the x-y plane. As the unit gets closer after each movement these process is repeated till the unit reaches the beacon at which time a new beacon is generated by the game.

The *CollectMineralShards* agent first puts all mineral shards into a matrix and calculates the x-y distance to each object from the player's current position. The agent then selects the closet mineral shard to their current location and moves to that location. At each future time step, the agent performs this calculation again and chooses to move to the next mineral shard until all shards have been collected. This simple pathing algorithm is rather inefficient but once again demonstrates the procedure for collecting a group of locations from a feature map object.

The final scripted agent the *DefeatRoaches* agent demonstrates the method for attacking an enemy unit. Like the *CollectMineralShard* agents, this agent also calls the location of the player's unit but now also gets the location of the hostile enemy unit. Upon identification of the enemy's location, an attack command is given by the agent until the enemy unit has been eliminated from the map. This agent once again does not incorporate advanced strategies that human players employ but none the less demonstrates the basic calls for a more advanced agent.

- Actions:
  - A useful file in the API is lib/actions.py. This file includes 523 pre-defined actions which appear to cover most (if not all) of the game's available action space (Will need to confirm every action possible).
- Bin/:

- Holds the top level run scripts. These may be good boilerplate for any development.
- Play.py can be used as the human agent interface and can run on linux. It is also used for watching replays.
- Agent.py is used to run agents. From the CLI, you can input the map name and agent name. Additional flags can also be input.

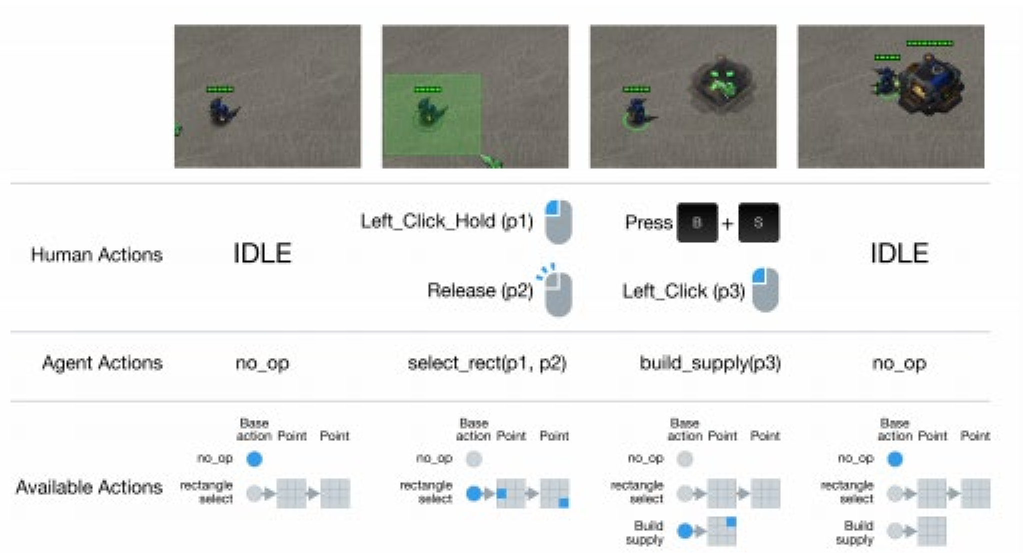


Figure 3 shows side-by-side the movements by a human player and the comparable API call of an agent in the same scenario [1].

- This files also can output the list of available maps and valid actions.

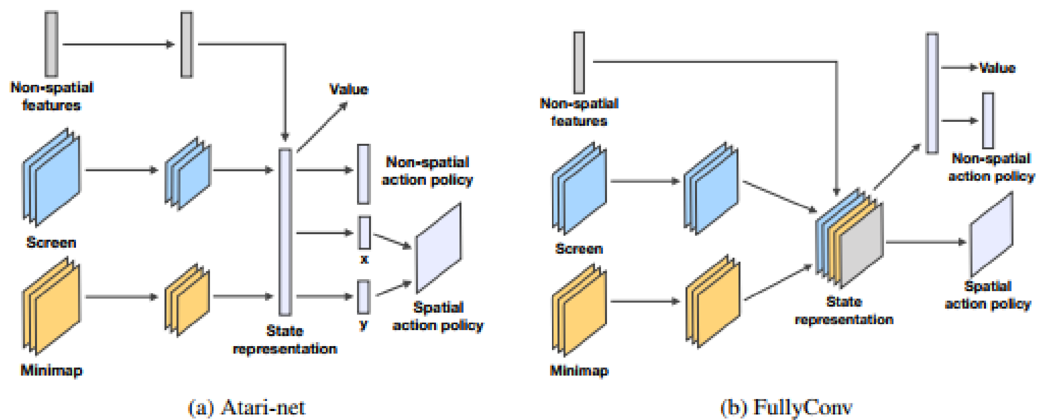
- Env/
  - The base environment is defined here. The sc2_env.py class SC2Env is a subclass.
    - In sc2env: there is an open TODO: How should we handle more than 2 agents and the case where the episode can end early for some agents?
    - This is a curious question.
  - There is also a base_env_wrapper.py which they describe as “A base env wrapper so we don't need to override everything every time.”
  - The run loop is hosted in this directory. It loops “…to have agents and an environment interact.”
- Lib/
  - Library of useful functions and classes. Need to sort them still.
- Maps/
  - Includes the mini-maps described in the paper.
  - Lib.py “The library and base Map for defining full maps.
    To define your own map just import this library and subclass Map. It will be automatically registered for creation by `get`.

```
class NewMap(lib.Map):
  prefix = "map_dir"
  filename = "map_name"
  players = 3
```

You can build a hierarchy of classes to make your definitions less verbose.

To use a map, either import the map module and instantiate the map directly, or import the maps lib and use `get`. Using `get` from this lib will work, but only if you've imported the map module somewhere."

  -
  -



(a) Atari-net (b) FullyConv

Figure 4: Network architectures of the basic agents considered in the paper.

- Open AI gym
  - Available Deep Reinforcement Learning (DRL) models
  - Generating of the cost function
    - $$\pi(a|s) = \prod_{l=0}^{L} \pi(a^l | a^{<l}, s).$$
    - $$\underbrace{(G_t - v_\theta(s_t))\nabla_\theta \log \pi_\theta(a_t|s_t)}_{\text{policy gradient}} + \beta \underbrace{(G_t - v_\theta(s_t))\nabla_\theta v_\theta(s_t)}_{\text{value estimation gradient}} + \eta \underbrace{\sum_a \pi_\theta(a|s) \log \pi_\theta(a|s)}_{\text{entropy regularization}},$$

      where $v_\theta(s)$ is a value function estimate of the expected return $\mathbb{E}[G_t \mid s_t = s]$ produced by the same network. Instead of the full return, we can then use an n-step return $G_t = \sum_{k=0}^{n} \gamma^k r_{t+k+1} + \gamma^n v_\theta(s_{t+n})$ in the gradient above, where n is a hyper-parameter. The last term is regularises the policy towards larger entropy, which promotes exploration, and β and η are hyper-parameters that trade off the importance of the different loss components. For more details we refer the reader to the original paper [19] and the references therein.

- Analyze framework, identify gaps in API
    - Layers represent multi-int data streams to be fused.
    - Implementing a probability of detection should be possible.

Generating DoD Relevant Scenario

The previously sections have laboriously laid out the free tools available to a developer. In this section, there will be a strict focus on the implementation of these tools on a problem of interest for the DoD. The specific mission investigated here is the general ISR requirement for any given area the DoD is required to operate in. This mathematical scenario of maximizing information gathered, area monitored, and personnel safety while minimizing cost and time can be broadly applied to a majority of DoD's operations.

One example that will be explored here is US and regional allies would like to monitor the area for any anomalous activity and intercept any perpetrators of illicit in the Gulf of Mexico and Eastern Pacific. At their disposal are a suite of mobile assets like vessels and UAVs along with sensor network arrays of costal radars, AIS base stations, and commercial satellites. The balancing of cost and benefit of each capability is a multi-variable optimization problem that a low-cost agent based simulation environment is well-suited to solve.

To model this situation, the proposed scenario will have two teams representing friendly "Blue" patrol assets and enemy "Red" smuggling forces that are trying to avoid detection while transiting across the map. The Blue agent will have at its disposal 2 medium speed mobile units for scouting purposes and 1 slow moving unit that serves as the arresting force. The difference in speed and function between the two unit types reflects the inability of air patrols to directly interdict before the slower moving naval vessels arrive on the scene. 3 Command Centers with the ability to scan specific places on the map at only select times for 1 second will also augment the Blue agent's surveillance capability. This will mimic the ability of using Satellite imagery to support the patrolling assets and better direct search patterns.

The Red agent will only command one unit at a time but it will be the fastest unit in the scenario. This unbalanced Cat-and-Mouse speed setup mimics the current real-world advantages smugglers have. The Red Agent will be tasked with moving towards 1 of 2 target locations that have different reward functions. The more valuable location will be at the opposite side of the map where the Blue forces start and is therefore harder to achieve. This reflects the high-pay out for a smuggler to deliver the cargo directly to shore but the high risk of being caught the closer one gets to the mainland. A defined location in the middle of the map will serve as the less valuable, yet still profitable, option for the agent to aim for. This region simulates an ocean drop-off for another vessel to later pick-up and return to shore with the cargo which is not always successful.

Once the Red agent has either moved a unit from the starting location to one of the two regions that unit disappears and another one is generated at the original starting point to begin the transit again. Therefore the scenario can be continuously run for the Blue agent to learn how to best allocate their assets and have them work together effectively. The RL process is propagated by the Red agent being

rewarded for successfully transiting the map while the Blue agent is penalized for allowing Red to escape and vice versa.

- Map editor
    - What properties of units
    - Blizzard has a map editor tool bundled with the game that can be used to make custom maps and modifications. Not only can one edit the terrain and layout of the maps, but it is also possible to modify existing data including unit health, armor, starting energy, weapon firing rates, scripted in-game event triggers, etc. This allows for nearly infinite map and scenario customization possibilities.
-

EV stations are Command Centers

Assets are Terran air units

Landscape is two slivers of land with the rest being ocean.

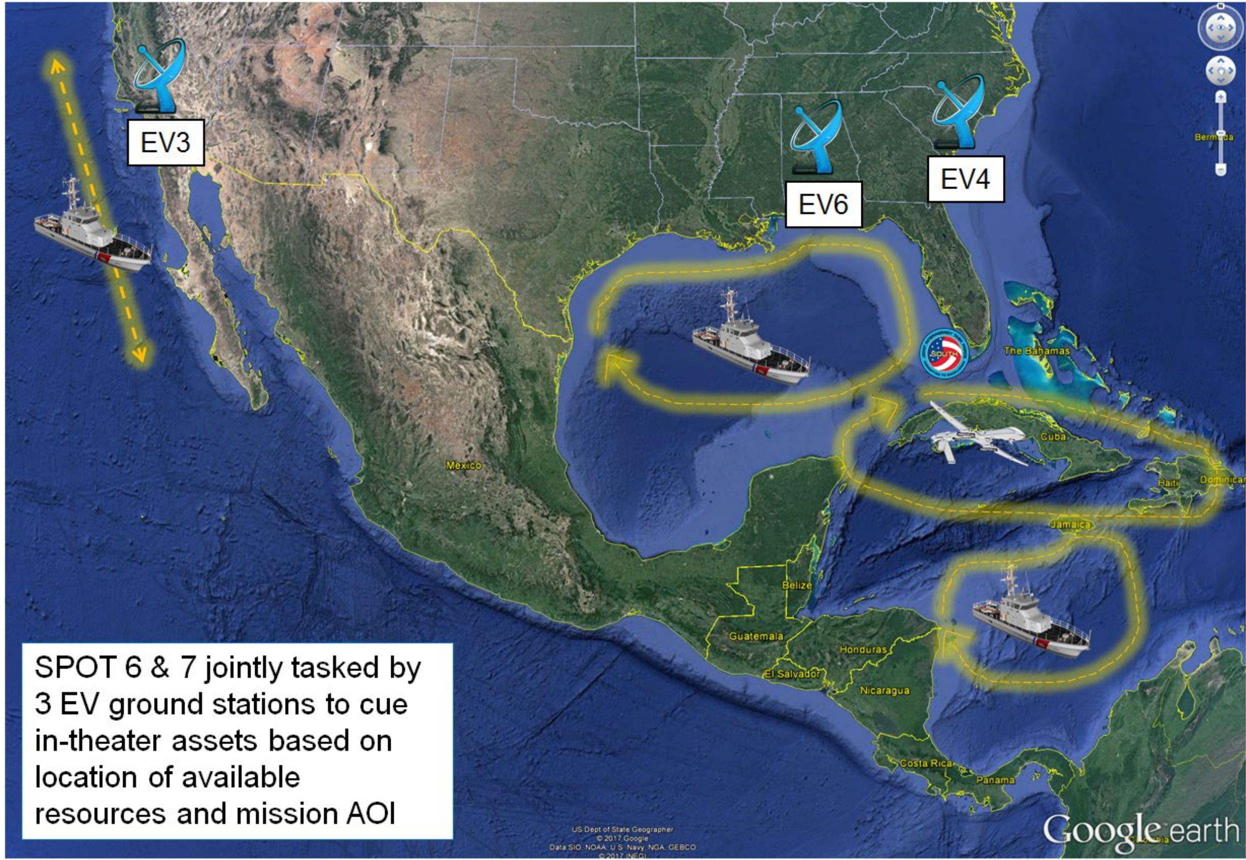
The Red team has to get to the other side without being detected by the blue team.

Restrict when the command centers can scan through energy regeneration pace.

Slow down the pace of all units to reflect naval speeds

Decrease the range of look for all units.

Construct the cost function.


EV3
EV6
EV4
The Bahamas
Cuba
Mexico
Belize
Guatemala
Honduras
El Salvador
Nicaragua
Costa Rica
Panama
Jamaica
Haiti
Dominican
Bermuda
SPOT 6 & 7 jointly tasked by
3 EV ground stations to cue
in-theater assets based on
location of available
resources and mission AOI
Google earth

1- Command Center with scanning capability to mimic satellite looks.
2- Interdictor by blue forces. Slowest moving unit but only one with attack function. Triggers blue force reward and red force penalty functions when successfully destroys red force units.
3- Surveillance by blue forces units. Medium speed unit with no attack function and low visibility. Has limited time before runs out of fuel and needs to return to base.
4- Neutral units that mimic commercial shipping traffic.
5- Smuggler unit for red forces. Unit with the fastest speed tasked with avoiding detection and delivery cargo.
6- Smuggler drop off point that triggers reward function for red forces and blue force penalty function.
7- Red force smuggler spawn point after successful drop-off or destruction by blue force interdictor.

All units have vastly decreased visibility and speed from standard game to mimic velocities and sensors in the maritime environment.

Proposed Future Study

Solve the proposed scenario with OpenAI Gym generated agents and translate that into JIATF-S strategy.

- Beat current list of mini-games
- Create newer mini-games based upon real-game mechanics

Bibliography

[1] O. Vinyals and K. Calderone, "StarCraft II: A New Challenge for Reinforcement Learning," DeepMind and Blizzard Entertainment, https://deepmind.com/documents/110/sc2le.pdf, 2017.

Evan Reilly

BetaStar installation instructions

1) Installation

a) First, you'll need a previous version of PySC2 to run the agent, specifically version 1.2.

pip3 install pysc2==1.2

b) Next, you will need tensorflow version 1.14.0rc1 in order to be compatible with the repository code as well as PySC2. In order to install this version of tensorflow, you will need the 64 bit version of Python, the default 32 bit will not work.

pip3 install tensorflow==1.14.0rc1

c) Install StarCraft II

a. Windows: https://starcraft2.com/en-us/

Note: You will have to create a Blizzard account to proceed to Windows download.

b. Linux: http://blzdistsc2-a.akamaihd.net/Linux/SC2.4.7.1.zip

i. The zip file is encrypted but may be unlocked with the key 'iagreetotheeula'

ii. Ensure that StarCraftII/ lives within your home directory

Instructions for pysc2 and Starcraft II installations are on Deepmind's pysc2 github page here.

d) In order to install the git repository, you must first get access to the 'Origin' server. Once you gain access, you will be able to do a git clone to access the repository. The command is as follows…

git clone ssh://USERNAME@l14gfe1.aic.nrl.navy.mil/opt/git/BetaStar.git

Just replace USERNAME with the actual username you signed up with when getting access to Origin.

2) How to run agents

a) Before running the agents on any of the seven mini-games, you will need to create a directory named 'mini_games' located in Program Files (x86) (or the like), go to StarCraft II, then Maps.

b) In order to run agents on these mini-games for training, you will need to download them here and place them in the 'mini_games' directory you previously created.

c) There are various custom parameters you should be familiar with before running agents. Here are a list of each with a description, default values for each may be viewed in run.py.

Experiment_id: The name of the experiment you would like to run. Separate name with underscores, group names for which summaries will be created are based on what come before the last underscore in the name. Ex) Experiment_id 'test_experiment_1' will have a group named 'test_experiment' created for it. Furthermore, if another experiment_id is named 'test_expirement_2', it will fall under this same group. Summary statistics are generated for these created groups.

--n1: Refers to the first number of n1 episodes summary statistics will be calculated for. Ex) n1 best (max score of first n1 episodes), n1 avg (average score of first n1 episodes).

--n2: Same as n1 but for a different number of episodes for which summary statistics will be calculated. Ex) n2 best (max score of first n2 episodes), n2 avg (average score of first n2 episodes).

--score: Score you would like your agent to achieve before it saves and exits.

--ma: Size of the moving average window for calculating summary statistics. Ex) Best last window (max score of the last 'ma' episodes), Avg last window (average score of the last 'ma' episodes). Furthermore, average score of the last window is the actual value compared with score to dictate whether the agent saves and closes or not (when Avg score last window > score).

--eval: If false, episode scores are evaluated.
--ow: Overwrite existing experiments (if –train=True).
--map: Name of SC2 map.
--vis: Render with pygame.
--max_windows: Maximum number of visualization windows open.
--res: Screen and minimap resolution.
--envs: Number of environment simulated in parallel.
--step_mul: Number of environments simulated in parallel.
--steps_per_batch: Number of agent steps when collecting trajectories for a single batch.
--discount: Discount for future rewards.
--iters: Number of iterations to run (-1 to run forever)
--seed: Random seed.
--gpu: GPU device ID.
--nhwc: Train fullyConv in NCHW mode.

--summary_iters: Record training summary after this many iterations.
--save_iters: Store checkpoint after this many iterations.
--max_to_keep: Maximum number of checkpoints to keep before discarding older ones.
--entropy_weight: Weight of entropy loss.
--value_loss_weight: Weight of value function loss.
--lr: Initial learning rate.
--save_dir: Root directory for checkpoint storage.
--summary_dir: Root directory for summary storage.
--load: Directory for checkpoint to be loaded.

d) Different running locally vs. running on the GP servers.

a. New Local Model Ex)

python3 run.py test_experiment_1 --n1 3 --n2 5 --score 1 --ma 5 --nhwc --envs 1 --map MoveToBeacon --lr 0.0007 --steps_per_batch 16 --entropy_weight 0.001 --value_loss_weight .05 --load out/models/test_experiment_1

Note: When running a brand new agent, you must still be sure to specify –load with 'out/models/' followed by the same experiment_id as the model you're beginning.

b. Loaded Local Model Ex)

Python3 run.py test_experiment_1 --n1 3 --n2 5 --score 1 --ma 5 --nhwc --envs 1 --map MoveToBeacon --lr 0.0007 --steps_per_batch 16 --entropy_weight 0.001 --value_loss_weight .05 --load out/models/test_load_1

Note: Just specify the path of the model you wish to load with –load. Default Models will save to the out/models directory. However, you may also load in models from elsewhere on the CPU by specifying an absolute path.

c. Running agents on GP servers Ex)

python3 run.py test_experiment_1 --n1 25 --n2 50 --score 25 --ma 10 --nhwc --envs 4 --gpu 3 --map BuildMarines --lr 0.0007 --steps_per_batch 16 --entropy_weight 0.001 --value_loss_weight 0.5 --load out/models/test_load_1 > out/summary/test_load_1.txt &

Note: It is helpful to add "> out/summary/EXPERIMENT_ID.txt &" at the end of the command as shown above, this way the output of your model will be written to a text file located in the "out/summary" directory instead of your console.

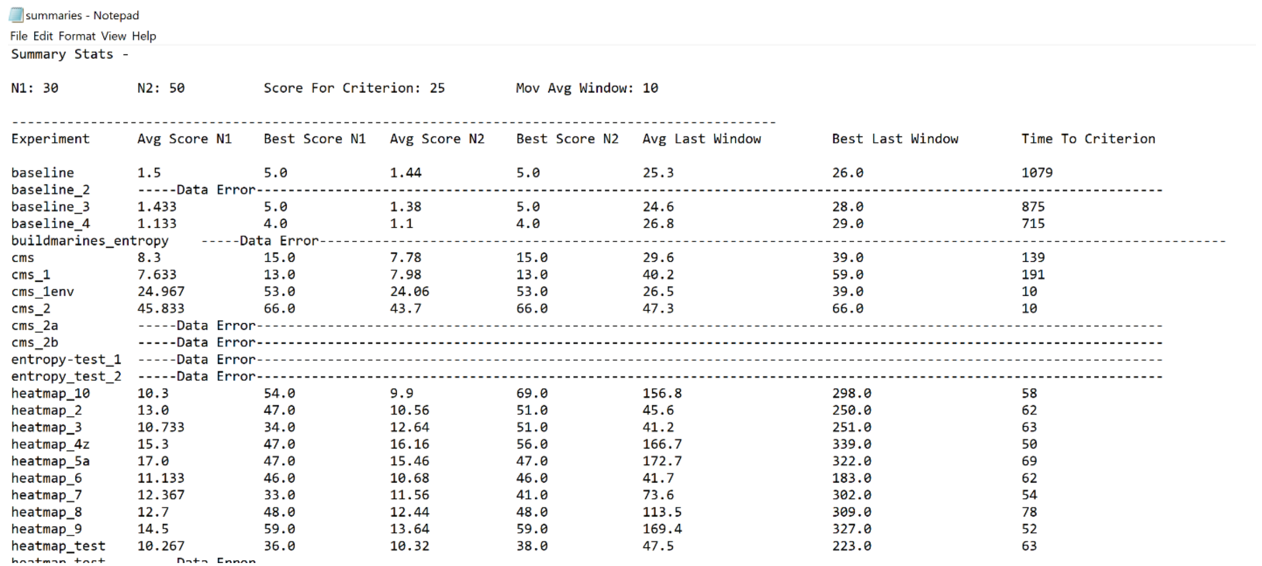
3) How to run visualize.py for summaries

Visualize.py is a file located within the BetaStar repository. When run as a stand-alone application, it generates summaries of the agent runs within "out/summary/"

It is run using four parameters mentioned previously (n1, n2, score, and ma).

Ex) Visualize.py –n1 30 –n2 50 –score 25 –ma 10

Running this command generates a couple of summary files. The first of which is "summaries.txt" which will be created in your BetaStar directory once the command is run.



summaries - Notepad

File Edit Format View Help

Summary Stats -

N1: 30 N2: 50 Score For Criterion: 25 Mov Avg Window: 10

Experiment	Avg Score N1	Best Score N1	Avg Score N2	Best Score N2	Avg Last Window	Best Last Window	Time To Criterion
baseline	1.5	5.0	1.44	5.0	25.3	26.0	1079
baseline_2	-----Data Error-----						
baseline_3	1.433	5.0	1.38	5.0	24.6	28.0	875
baseline_4	1.133	4.0	1.1	4.0	26.8	29.0	715
buildmarines_entropy	-----Data Error-----						
cms	8.3	15.0	7.78	15.0	29.6	39.0	139
cms_1	7.633	13.0	7.98	13.0	40.2	59.0	191
cms_1env	24.967	53.0	24.06	53.0	26.5	39.0	10
cms_2	45.833	66.0	43.7	66.0	47.3	66.0	10
cms_2a	-----Data Error-----						
cms_2b	-----Data Error-----						
entropy-test_1	-----Data Error-----						
entropy_test_2	-----Data Error-----						
heatmap_10	10.3	54.0	9.9	69.0	156.8	298.0	58
heatmap_2	13.0	47.0	10.56	51.0	45.6	250.0	62
heatmap_3	10.733	34.0	12.64	51.0	41.2	251.0	63
heatmap_4z	15.3	47.0	16.16	56.0	166.7	339.0	50
heatmap_5a	17.0	47.0	15.46	47.0	172.7	322.0	69
heatmap_6	11.133	46.0	10.68	46.0	41.7	183.0	62
heatmap_7	12.367	33.0	11.56	41.0	73.6	302.0	54
heatmap_8	12.7	48.0	12.44	48.0	113.5	309.0	78
heatmap_9	14.5	59.0	13.64	59.0	169.4	327.0	52
heatmap_test	10.267	36.0	10.32	38.0	47.5	223.0	63

It lists all experiments that are within 'out/summary/' calculating a variety of summary statistics for each agent based on the parameters you specify.

Avg Score N1: Average score the agent receives for the first N1 episodes.

Best Score N1: Best score the agent receives for the first N1 episodes.

Avg Score N2: Average score the agent receives for the first N2 episodes.

Best Score N2: Best score the agent receives for the first N2 episodes.

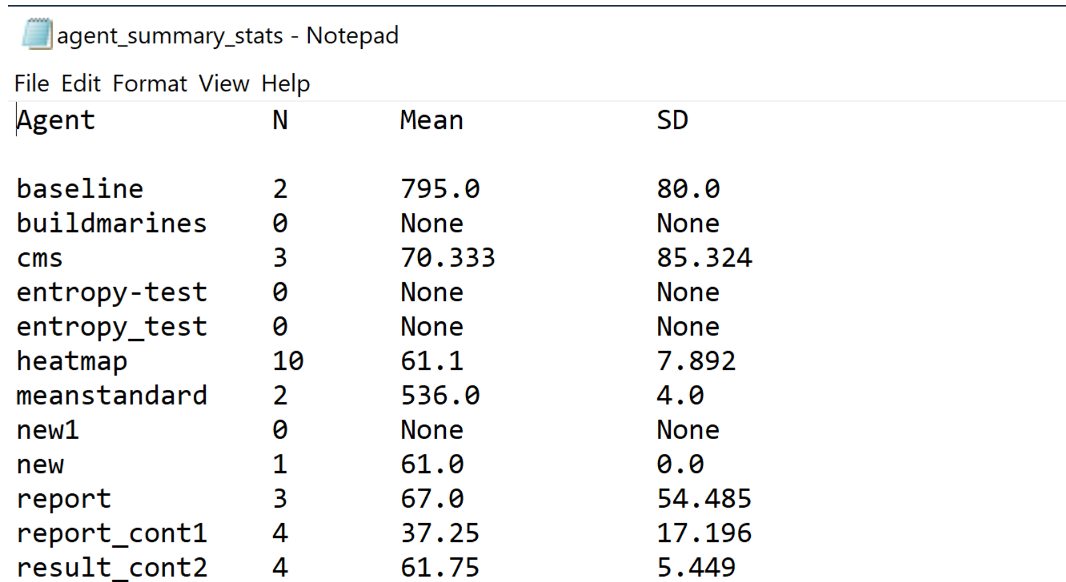
Avg Last Window: Average score of the last MA episodes (--ma).

Best Last Window: Best score of the last MA episodes (--ma).

Time To Criterion: Number of episodes that occur before "Avg Last Window" > score (--score).

“Data Error” occurs for experiments where the parameters you choose aren’t functional with the given experiment.

Next, within “BetaStar/out/summary/ there is a file named “agent_summary_stats.txt” that generates the mean and standard deviation of time to criterion for groups of agents within “out/summary”. Furthermore, the number of agents per group used to calculate these values is included.


agent_summary_stats - Notepad

File Edit Format View Help

Agent	N	Mean	SD
baseline	2	795.0	80.0
buildmarines	0	None	None
cms	3	70.333	85.324
entropy-test	0	None	None
entropy_test	0	None	None
heatmap	10	61.1	7.892
meanstandard	2	536.0	4.0
new1	0	None	None
new	1	61.0	0.0
report	3	67.0	54.485
report_cont1	4	37.25	17.196
result_cont2	4	61.75	5.449

Agents listed are actually grouped agents based on the name of the experiment_id. Agents are grouped based on the string that comes before the last underscore of the experiment_id. For example, an agent with experiment_id equal to “heatmap_1” will be grouped with agents with similar names such as “heatmap_2” or “heatmap_test”. These calculations only take valid values, so if one of the agents in the group doesn’t achieve the criterion, it will not be included.

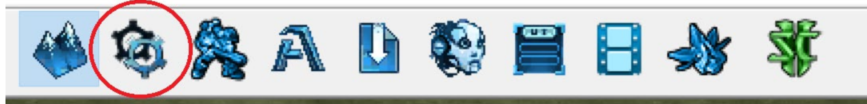
4) How to edit maps for reward shaping

a) To edit the mini-games, simply navigate to the same StarCraft II directory located in Program Files. There, you will see “StarCraft II Editor_x64”. Click on this link instead of the “StarCraft II Editor”. You may run into issues with the text appearing to be cut off in many parts initially. This issue may be fixed rather easily according to the video found here.

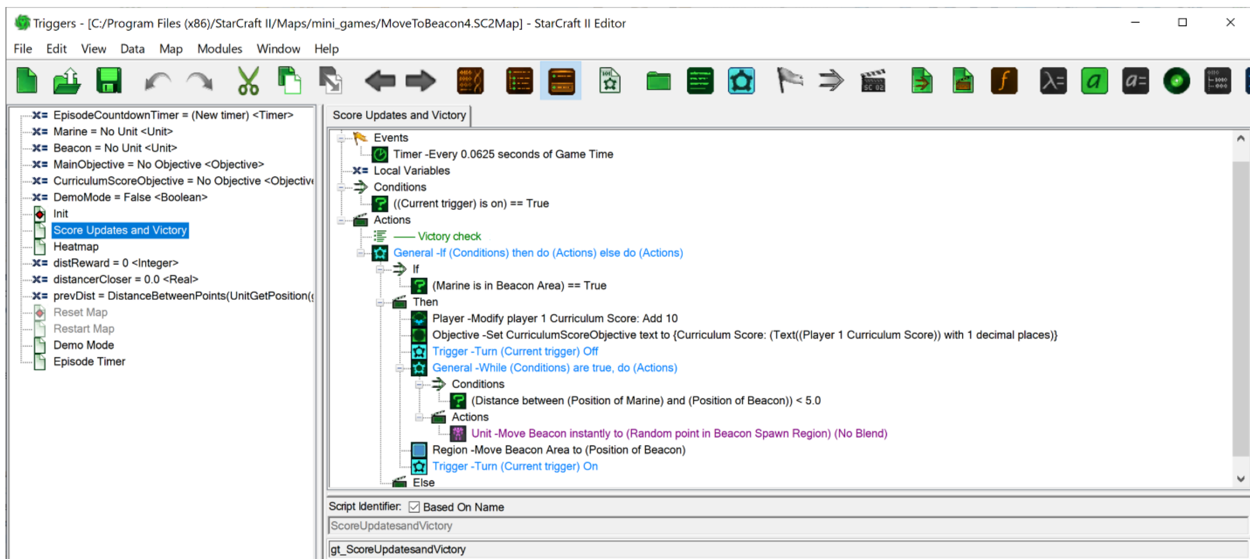
b) When the editor has loaded up, simply go to File -> Open and navigate to the StarCraft II directory and go to where your mini_games are saved.

c) It is advisable to keep copies of the original mini-games before experimenting with new rewards. Just copy and paste the map you would like to make changes to as well as rename the new map to something else.

d) To begin changing the structure of the rewards, you will need to go to the navigation bar at the top of the screen and click on the triggers tab indicated by the picture with the two gears.



e) Click on Score Updates and Victory, this will show you how curriculum score is modified. This value is the reward the agent currently receives from training on the map.



f) You may create variables and other triggers that dictate how curriculum score changes, primarily through working in the window to the left.

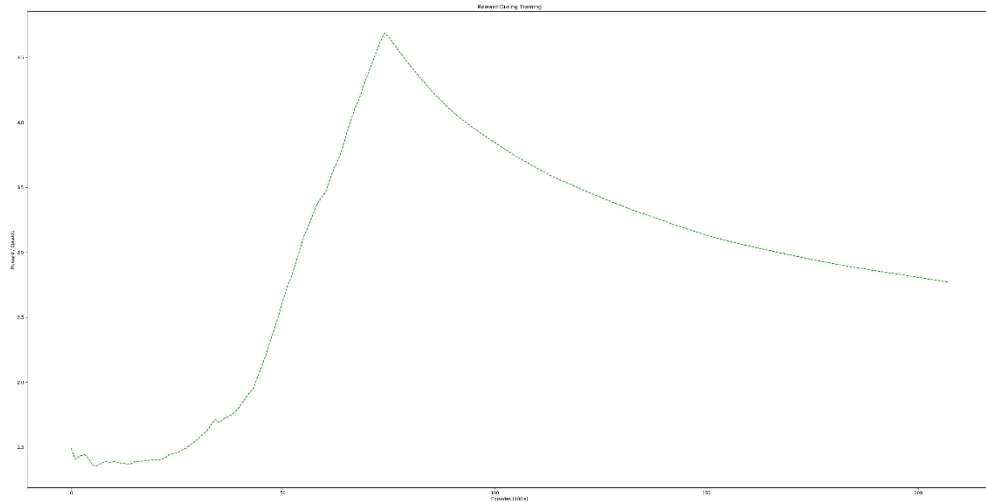
Evan Reilly

Progress report

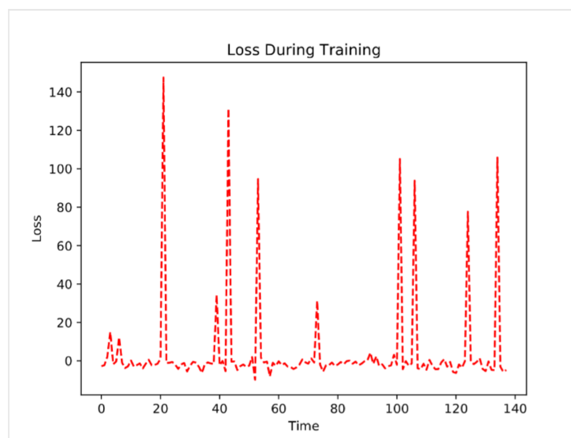
The first couple weeks of my ten week-long internship as an NREIP intern at the U.S. Naval Research lab consisted mostly of updating deprecated code from https://github.com/zachdj/Hasu to work with the latest version of PySC2 (v 2.02) using PyTorch. There were a variety of issues with this code, ranging from differences in naming conventions to the difference of formatting of certain datatypes to many others. After many hours of looking at code and coming up with clever fixes, we were finally left with an agent that compiled.

Besides working out the bugs of the program, I had worked on visualizations using the matplotlib module. The code I wrote would automatically output two graphs with valuable information, reward and loss. The reward graph would show the number of episodes on the x-axis and the instantaneous reward received and average reward received so far at each episode. The loss graph was similar in that it showed number of episodes on the x-axis, however, it displayed a combination of policy and value loss (total loss) on the y-axis.

Within the next couple weeks of the internship, the team was left with a new problem. Although our agent compiled and we were able to run on GP2 and GP3, our agent didn't seem to reach convergence on the first, simplest, of seven mini-games, "MoveToBeacon", only testing our agent's movement. Other researchers on the internet who trained similar agents reached convergence around approximately 1,000 episodes. However, our agent showed little signs of hope. Reward across episodes using certain different hyper-parameters looked like it may cause our agent to actually learn something. But after countless attempts of running the agent, the same result occurred. It looked as if it was learning, but after a certain point, our agent does absolutely nothing (as shown below in the graph with average reward across episodes).



Furthermore, our total loss (policy loss + value loss) was not functioning like how we wanted it to, staying stagnant with jumps throughout training. We would usually like to see total loss decrease over time. The graph below shows instantaneous total loss at each moment of time throughout training.



After thousands of episodes, still no luck. We continued looking for fixes that would allow our agent to train and keep training without crashing at a given point. However, with taking so long to figure out if one fix or the other could work, we were running out of time.

At this moment, I began working on a new mini-game for our agent to train. I would turn the sparse reward of the current MoveToBeacon into a much denser reward. Previously, agents training on the mini-game would only receive +1 reward each time they hit the beacon. However, we realized this

could be changed to improve training time. I started brainstorming ideas on how we could make this reward much denser. The initial thoughts that came to mind were to give the agent a small reward every second it is relatively close to the Beacon. Furthermore, it would receive a larger reward for being even closer, an even larger reward for being very close, and finally the greatest reward for actually getting to the beacon. I soon realized this reward could have flaws by being too discrete. I then came up with a new solution involving giving the agent +1 for being closer to the beacon than the last second. Similarily, the agent would receive a -1 penalty for being further away than the last second. Not only that, but I would give the agent a -10 penalty for not getting to the beacon at all during an episode in order to penalize the agent quickly for not doing what we want (standing still etc.). Also, the agent was to receive +10 each time it got to the beacon. Finally, I realized the agent may not necessarily benefit from the +1 / -1 reward because the agent may not explicitly know to go directly to the beacon if it receives a discrete reward like this. So, I came up with a final agent that made sense intuitively and was fairly simple. The new reward would calculate the Euclidean distance from the marine to the beacon at each second of game time, comparing this distance to the distance from the previous second. Then, based on this distance, the agent would receive a relatively larger reward for being closer and a relatively larger penalty for being further away than the last second. It was a complicated process working within the StarCraft II map editor, however. I ran into issues using "real" values in the StarCraft II map editor as rewards, only Integers were accepted. Furthermore, converting the real value of distance to an Integer value for reward, as far as I could tell, wasn't possible. I figured out the approximate maximum distance a marine could travel within a second of game time and essentially had to create various if/else statements in order to convert the "real" distance calculated into an Integer value of reward at each second to feed our agent. This is the densest reward I thought to be possible. However, when it came to testing with code we currently had on hand, the agent still didn't train. The issue wasn't with the map we were using, it was the agent itself. This PyTorch agent was not functional.

We decided to implement a different currently existing code base from https://github.com/simonmeister/pysc2-rl-agents using Tensorflow. Most of everything we worked on before this point was scrapped. Right away, we tested different instances of the agent on GP3 and it converged within around 600-1000 episodes like we were hoping.

Now that we had a working agent, we needed to create a set of tools for evaluating agent performance. I was tasked with first creating a –load argument for our command line so that we could load previous saves of our agents. This is useful for transfer learning. For instance, when we train on MoveToBeacon, we can load that previous model and see how it performs on the next game CollectMineralShards. I was then tasked to create a summary tool that would allow us to compare a variety of saved agents by parsing through their Tensorflow event files. But first, I was to create more command line arguments for our running agent. These include –n1, identifying the number of training episodes to be executed before creating summary statistics for those first n1 episodes. Secondly, --n2 was created for the same purpose but to do this for a different set of episodes. Next, --ma was introduced to the command line to identify the size of the moving window of episodes for us to calculate summary statistics. Finally, --score was introduced to be the given reward/score to be beaten by the moving average window of episode scores so that our agent will know when it has been trained to a sufficient level.

All of these newly introduced values allow our program to compute valuable statistics for us to analyze how some of our agents compare to others. I created the visualize.py module to do this. It allows us to create a “summaries.txt” file. The header of the file shows the corresponding n1, n2, score, and ma, from which the values in the file were calculated. Each line after the header of the file corresponds to a given agent. The summary statistics calculated include “Avg Score N1” which is the average score of the first n1 episodes, “Best Score N1” which is the best score of the first n1 episodes. Furthermore, “Avg Score N2” is the average score for the first n2 episodes, while “Best Score N2” is the best score for the first n2 episodes. Next “Avg Last Window” is the average score of the moving average (size specified by –ma). This is also the value that is compared with the value specified with –score to determine if the agent should stop what it’s doing, save and close. Next, “Best Last Window” is the best score of the last window also specified by –ma. Lastly, and most importantly, “Time To Criterion” is the number of episodes that have passed before the criterion has been met (The point where “Avg Last Window” > Score). If a summary statistic doesn’t have the necessary information to display, its value defaults to “None”. Also within visualize.py, a “name-of-episode_summary.txt” is generated within the corresponding summary folder of the agent. This file has similar summary statistics outputted as in summaries.txt, however, the results are calculated real-time while the agent is training. This is a valuable function of the program so that we may see if our agent is doing what it is expected to do and so we can see if it has achieved its criteria yet. The final visualize.py file generated from the program is “agent_summary_stats.txt”. The information within this file is particularly valuable because it groups similar agents with one another and calculates a group mean and group standard deviation of their time to criterion along with the corresponding “N” number of individual agents from which these values are calculated. Agents are grouped by the string that comes before their experiment_id’s last underscore. For example, if an agent has the experiment_id of “test_experiment_1”, while another one is named “test_experiment_2”, a group called “test_experiment” will be created. So, in the agent_summary_stats.txt file, there will be the corresponding “test_experiment” under the “Agent” column, “2” under the “N” column, as well as a corresponding mean under the “Mean” column and standard deviation under the “SD” column.

Now that we had a tool that allowed us to analyze our agents as well as a working model, I thought it would be a good idea to attempt to run a new set of agents on the MoveToBeacon map I had created previously with a very dense reward. I ran ten instances of this agent with the following parameters. There was a learning rate (--lr) of 0.0007, steps per batch (--steps_per_batch) of 16, entropy weight (--entropy_weight) of 0.001, and finally a value loss weight (--value_loss_weight) of 0.5. Furthermore, I specified the score (--score) to beat was 170 with a moving average window (--ma) of 1. This was chosen because it is the score that, once achieved, visually looks as if the agent is directly moving from beacon to beacon as a new one resets. Furthermore, the previously listed parameters were all default values given from the downloaded original code.

Summary Stats -

N1: 25 N2: 50 Score For Criterion: 170 Mov Avg Window: 1

Experiment	Avg Score N1	Best Score N1	Avg Score N2	Best Score N2	Avg Last Window	Best Last Window	Time To Criterion
heatmap_1	9.48	54.0	9.9	69.0	280.0	280.0	86
heatmap_2	13.12	47.0	10.56	51.0	250.0	250.0	62
heatmap_3	10.68	34.0	12.64	51.0	251.0	251.0	63
heatmap_4	17.92	47.0	16.16	56.0	339.0	339.0	59
heatmap_5	17.32	47.0	15.46	47.0	322.0	322.0	69
heatmap_6	11.04	46.0	10.68	46.0	183.0	183.0	62
heatmap_7	13.72	33.0	11.56	41.0	302.0	302.0	54
heatmap_8	11.08	45.0	12.44	48.0	309.0	309.0	78
heatmap_9	14.8	59.0	13.64	59.0	310.0	310.0	52
heatmap_10	10.16	36.0	10.32	38.0	223.0	223.0	63

Agent	N	Mean	SD
heatmap	10	64.8	9.887

From examining the above results, we were able to conclude that the newly shaped reward was a much better fit with a time to criterion of about 65 episodes than the previously sparse reward with a convergence ranging from 600 to 1000 episodes. Furthermore, the standard deviation associated with these runs seem relatively small when compared with the value of the mean. This makes our runs reproducible and consistently better than the sparse reward alternative.

While training on a dense reward is great, it also seemed necessary to attempt to train an agent on a reward that starts off dense, but then slowly turns sparse over time. For instance, we could train the marine starting with the Euclidean distance reward stated before. However, after every 16 or so episodes for instance, we could slow down how often the marine “looks back” to calculate its distance to the beacon. For this experiment, the marine would begin calculating its distance to the beacon every second of in-game time (This is also the interval our previous dense agent trained on the entire time). Every subsequent 8 episodes after the first 16 episodes would result in the agent looking back less often, waiting an additional 0.5 seconds each time. This causes the marine to start off with a very dense reward signal that tapers off as it learns. Results for this agent looked promising at first. The first couple agents of the “result_cont1” group (referring to continuously changing reward) looked promising with 54 and 55 episodes until reaching the desired score of 170. However, the next couple of training episodes resulted in 70 and 80 episodes until convergence. This reward structure was not statistically more significant than our dense reward group “heatmap”. Furthermore, the variance was larger which is undesirable. The second group of agents tested were “result_cont2”. This group varied from the first in that the first 16 episodes of training left the marine “looking back” at its distance every second, but then looking back every two seconds for the duration of the next 16 episodes, then finally, looking back every 3 seconds until convergence. Results for this agent weren’t great. In fact, the fourth run of result_cont2 went through hundreds of episodes with no convergence and was therefore scrapped (and not included in the results below).

Agent	N	Mean	SD
result_cont1	4	64.75	10.848
result_cont2	3	86.0	39.606

After testing with rewards that were more continuous in the way they changed how often they "look back" over time, results weren't pleasing. There was a sporadic change in the amount of episodes it took to train our agents until they reach convergence. For instance, a few of the tested runs wouldn't converge for a few hundred episodes, while some converged around our previous average of 64.8 for the "heatmap" agents. It therefore seemed like a much better idea to train on a dense reward, at least for the MoveToBeacon mini-game.

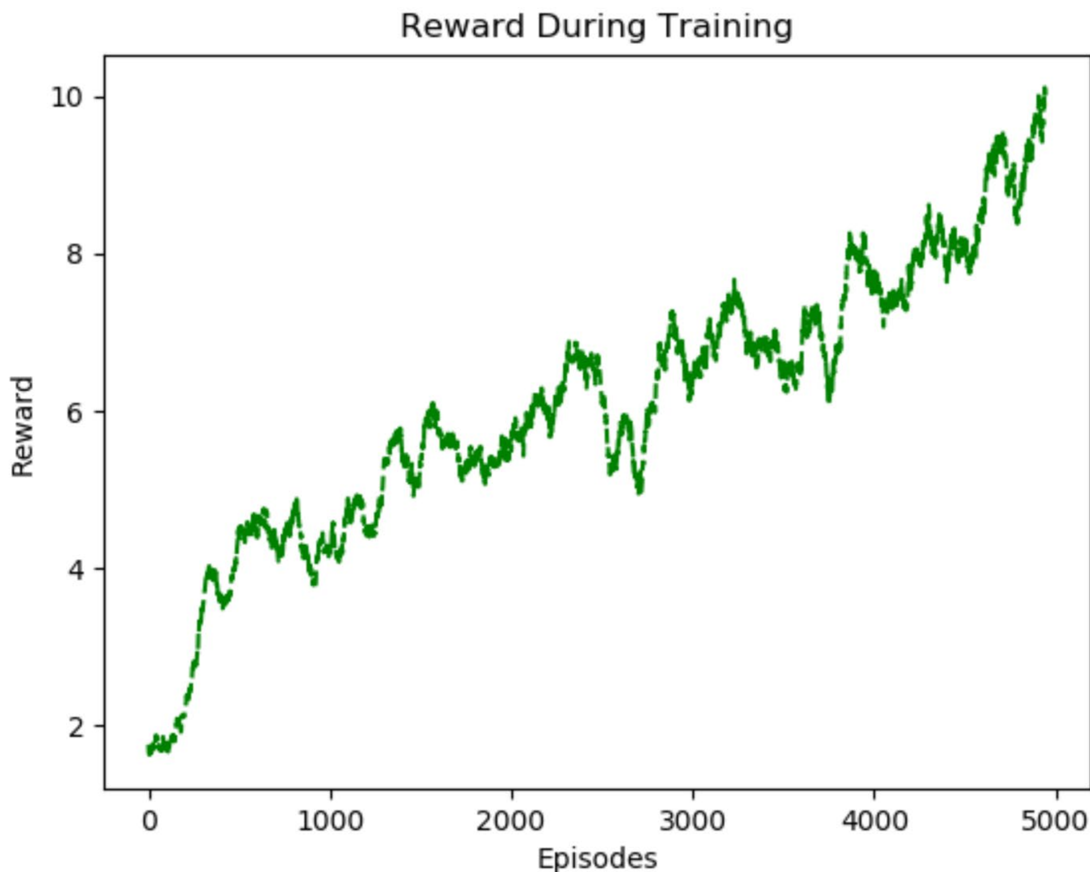
Besides the MoveToBeacon, there was seemingly only one other mini-game in PySc2 that had too sparse of a reward for our agent to train in a reasonable amount of time. This was BuildMarines, the last and most challenging mini-game of them all. This was given the fact that the agent only receives a reward when marines are created. However, there are other production requirements that must be satisfied before an agent can train them. For example, the agent must first use their Space Construction Vehicles to collect minerals. Using these minerals, agents must build supply depots in order to have enough "supply" to train marines. Furthermore, they must have enough minerals to build barracks, the buildings actually used to train marines. Only after the previous steps have been accomplished can the agent actually train the marine. One can see why this may be considered sparse. So, we introduce a denser alternative by giving the agent a certain reward at each step leading up to training the actual marine. This reward is structured as follows. The agent receives a reward of +1 for each new worker created (Workers automatically collect mineral shards so we don't need to introduce a reward for that). Furthermore, the agent receives a reward of +2 for building a supply depot. Additionally, the agent receives a reward of +4 for building barracks. Finally, a reward of +8 is given for every marine trained. The reward is structured in an incremental, multiplicative way. The hope is that the agent will prefer to pick actions that result in actual marines created. If possible, the agent will generally prefer to save minerals for productions that actually train marines. For example, if the agent has enough minerals to build a supply depot, but could achieve a better reward for saving up to build barracks, ideally the agent would choose to build barracks (or train a marine if possible).

Although this logic may make sense intuitively, it has been extremely difficult to test the hypothesis that this denser training is faster than the sparse, default alternative. The BuildMarines mini-game by far takes the longest amount of time of all the mini-games to achieve a significant amount of knowledge and perform well (taking about 250 million episodes for the fullyConv network to produce any significant change in performance). However, there are differences between the agents that are noticeable after our agent runs for only a short period of time.

After our agent trained for 2,663 episodes (auto saved after 75000 iterations), I decided to evaluate it by grabbing the checkpoint from the GP3 machine and running it locally. The agent showed significant differences from the random agent. Unlike the random agent, this agent wasn't building nearly as many

supply depots. It was however, creating much more barracks, a key structure for training marines. This much was reassuring. This agent also seemed to be training more marines, although, not significantly more. The random agent would train an average of 2.5 marines per episode while this agent was training about 2.8 per episode. However, shortly after calculating these averages, the agent started behaving unfavorably. For a few episodes in a row, all the agent would do is collect minerals and literally do nothing else while it hoards them away. Realizing this, I knew I must change the reward structure. We needed a penalty when our agent can afford to buy the most expensive building/thing available but chooses not to do so. The most expensive object in this mini-game was the barracks costing 150 mineral shards. Knowing this information, we created a kind of buffer for the agent, only giving the agent a penalty when they have more than 300 minerals on hand. The trigger we added to the edited map would penalize our agent's score by 4 every 10 seconds its total minerals were above this point (300 minerals). This seemed like a good fix, given the behavior we examined from the loaded agent we trained before. I then trained an agent from the previous model with these new map changes. I also loaded a fresh one to run simultaneously.

After looking at our brand new agent after running for 5,000 episodes, it appeared to do little to nothing. The penalty imposed for not using resources quick enough seemed to backfire. The agent appeared to just take the penalty and only collect minerals. This didn't seem to be the right starting point for our agent. However, after examining the loaded agent from the previous model, it was doing a noticeably better job at using its resources (minerals). It tended not to hoard them away as much as it did before this period of training. Furthermore, the agent looked like it knew what it was doing when it came to building the right amount of supply depots (relatively low amount considering the low amount of marines) and a lot of barracks (we need a lot of barracks to train a lot of marines). It was generating an average of 3 marines per episode. Although it was better than the previous period of training, there was further improvement needed. Because the agent appeared to have a good foundation for training marines, I decided to do another transfer of learning. I loaded this agent back onto the original BuildMarines mini-game with the sparse reward of only receiving a score every time a marine was trained. After training for another 5,000 episodes on this reward, our agent was creating around 10 marines per episode. This was significant progress considering the sparsity of the reward. It appeared that our agent was learning much faster after receiving a reasonable knowledge base than it would have if only trained on the sparse reward from the default map.



This is a graph of the 100 episode moving average score of the previously mentioned agent. This run only includes its progress after being loaded back onto the default BuildMarines map after receiving its knowledge base.

Now that there is a way for our agent to train halfway efficiently on most of our mini-games, we need a way to compare lots of agents against each other on a live leaderboard. Of course the summary text files that are generated from running the agent are sufficient enough for doing tests on the local machine. However, to train in bulk, we needed a tool that allowed us to train hundreds of agents with different parameters and needed a way to actively test them against one another. To tackle this task, we needed to figure out a set way of storing information. Therefore, I designed the structure of the database we would use for this leaderboard. It consists of a "Hyper-Parameters" table as well as a "Scores" table. It looks as follows.

Hyper-Parameters

<u>AgentID</u>	Learning Rate	Entropy Wt.	Val Loss Wt.	Steps Per Batch	N1	N2	Score	MA	Mini game	Loaded Agent

Scores

<u>AgentID</u>	N1_best	N1_avg	N2_best	N2_avg	Last_Window_Best	Last_Window_Avg	TTC

We separate these tables into two even though the unique ID is the same between them both. This is because the Hyper-Parameters table may be populated instantaneously upon running the agent. All of the information it needs is included within the parameters of run.py. The Scores table is used to store all of the statistics that are calculated while running our agent.

After the actual online leaderboard was built, we just had to implement it with my current code at the time. Sending data to the database after every X iterations. The leaderboard now worked with actively running agents and for the most part, was HPC compatible.

Regina Wang

Progress report

During this internship at NRL, I have been working on the project BetaStar. The objective of BetaStar is to replicate AlphaStar. AlphaStar is a deep learning agent that Deep Mind from Google developed that beats professional players at StarCraft II, a considerably difficult video game that has the player controlling hundreds of different units and 10 to the 26 actions at each time step. However, a big bottlecap to our ability to fully replicate AlphaStar is our lack of computing power; thus, BetaStar is more of a replication of the early stages of AlphaStar. This report covers the work I have done at NRL, summer of 2019, which entails:

1. New arguments
2. Details on Major Changes
3. Future Work

New Arguments:

--rms: The network runs with RMSprop optimizer. Without this "--rms" argument, the

network runs with the Adam optimizer and will batch normalize and clip the intermediate spatial and nonspatial outputs. See (1) for more information.

--lstm: The fully convolutional network will have a LSTM layer right after spatial

and nonspatial data are concatenated. See (2) for more information.

--transformer: The fully convolutional network will have a transformer layer before the

very last layer. See (3) for more information.

Major Changes:

Upgrade to Adam Optimizer (1):

In the beginning of this NREIP internship, I ventured to practice some hyperparameter tuning on the network in order to allow it to converge faster and obtain better results. While reading the code, I noticed that it used RMSprop optimizer rather than the standard Adam optimizer, which is generally considered a step up from RMSprop because it combines RMSprop and momentum. However, once the change was made, a minor error which would pop up in around 10% of runs with RMSprop and crash the entire program became major as it would crash the program in nearly 95% of the runs. Given that the

issue became very large and the Adam optimizer seemed to be converging faster in the time before the error came up, I decided to investigate this issue next.

Possible Future Improvements:

Checking out other optimizers to minimize convergence time and maximize average mean score.

Batch Normalization and Clipping (1):

At first glance, the issue was very strange—the network runner was receiving a command from the network that was out of bounds of the command list. At first, we believed that it was a result of improper connection of the network with the environment. After tracing through the error some more, I found that the network itself knew that it was returning something out of bounds and was doing so on purpose. The intermediate data inside the network was displayed and it was found that one of the layers in the network was consistently malfunctioning and returning negative indefinite numbers which caused the softmax to return an error result—the length of the possible results list itself. This issue could possibly be solved in multiple ways, including waiting out the error until it returns to normality and dealing with abnormalities with the maximum entropy reward or intermediate values.

Logically, waiting out the indefinite number error is not the best idea because once the network falls into the pit of indefinite numbers, the entropy reward becomes so high that the reward will be maximized as the entropy goes to infinity. The next possible solution following this is making entropy more robust. Clipping/normalizing the entropy would not work properly either since when the indefinite numbers error hits, the network's loss will be in a local minimum; in addition, it was affecting and slowing down performance before the error hit.

While considering the use of normalizing, it seemed promising to normalize and clip the result of the layer in the network that was acting up. After using batch normalization on the results of the layer before it went through the softmax and seeing that it cleanly cleared up the issue, it was important to see whether a different normalization technique or clipping would work better. Thus, I compared L1, L2, and L infinity normalization, with clipping the result to only be within the range [-10, 10].

Even while using a high coefficient of 0.1 times the result as a part of the loss function, L2 and L infinity would still jump to the error after some time. L1 norm affected the reward structure so much that results became more based off of minimizing L1 than maximizing entropy or the reward itself. Clipping was also very effective in both curtailing the error and keeping performance. To prevent batch normalization from ever falling into the error while also keeping the slightly superior performance of batch normalization, I combined batch normalization with clipping so that after batch normalization, the result would be clipped to be within [-100, 100].

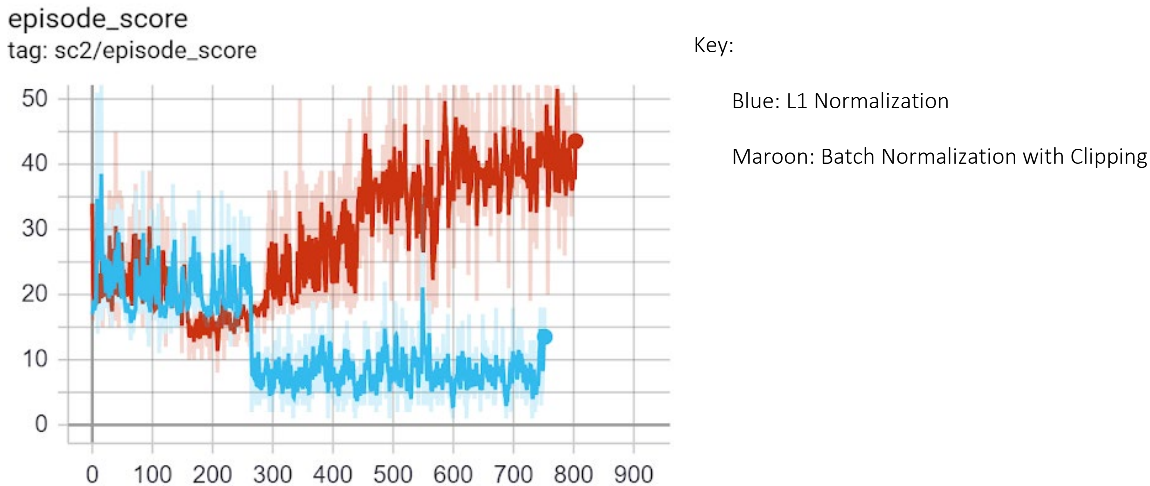


Figure 1.0: It is clear that while both start with similar results, the blue's score quickly deteriorates as the network converges to telling the agent to do nothing in order to minimize the L1.

Even though through fixing this error, the network converged faster on most minigames. The old model with RMSprop and 10% error rate is still saved and can be run with the tag, --rms. Otherwise, the network defaults to Adam optimizer and batch normalization with clipping.

Possible Future Improvements:

A possible countermeasure against the error coming up again in around 5% of the runs,

or for other reasons.

Add batch normalization and clipping between each intermediate value.

Change the clipping range, [-100, 100]

Long Short Term Memory (2):

Following the above, I had a project to add memory to the network, or, more specifically, add LSTMs to the network. LSTMs, like every recurrent network layer, have states that need to be saved and passed back in at every time step. However, because the environment medium, runner of the agent, and the agent were set up irregularly, we had a batch of environments to run at every step, and a batch of batches of environment steps to run at every training step. After tinkering around to resolve this difference, it was found that a built in tf.cond statement, the equivalent of an if statement built into the tensorflow network would allow for this difference. That is, we use the conditional statement to account for the differences in how the network trains and steps because of the batch versus batches of batches difference. However, tf.cond had many restrictions and there were many differences to resolve such as having to combine the batch of batches of hidden states into a single batch of hidden states, having to reset hidden states after episodes reset, which can happen in a staggered manner, and the results of the

two functions in the conditional statements having to be the exact same, down to the shapes of each result of the function. In addition, tensorflow's conditional is a more unused feature especially since it should not be used in well written networks.

In the end, to run an LSTM layer, the runner must pass the hidden state and a boolean list containing whether training or stepping is occurring and whether each episode is new. With this information, the network first determines whether it is stepping or training. If the network is stepping, then it zeros the state to be inputted if stepping and runs the LSTM layer with the existing state. Then, the result is passed to the next layer and the hidden state is returned to the runner to be saved for training and to be used in the next step. If the network is training, then the network first combines the batches of states into a single large state. Then, the network uses the large state to train, the result is passed to the next layer, and the large resulting state is split so that only the last normal sized state is passed to the next stepping of the network.

After finally dealing with connecting hidden states and batching abnormalities, each episode with a LSTM layer took nearly 15 minutes to run--in comparison to the original 3 seconds (a 300x difference).

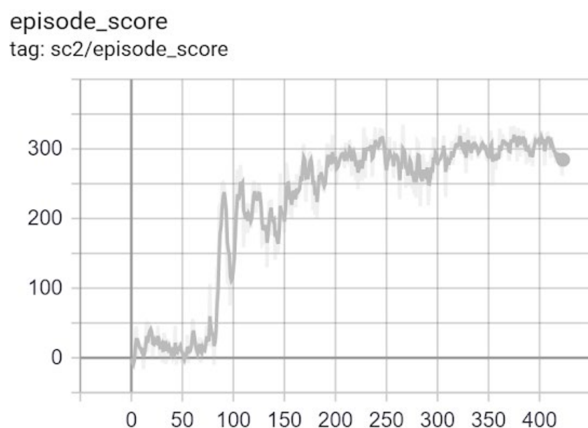


Figure 2.0: This training of a network with an LSTM layer would normally take three hours to converge without an LSTM layer, but in this case, it took over three days to get this graph.

After seeing how much longer an LSTM layer adds to the episode running time, I decided to add more parallelization for the batches during stepping. This allowed for the time to run an episode to decrease to 10 minutes rather than the original 15 minutes, which is very practically significant, though not significant numerically.

Possible Future Improvements:

To increase the number of layers and number of units in order to simply expand graph or for

other reasons,

Modify num_layers inside of runner.py, agent.py, and fully_conv_lstm.py.

Modify num_units inside of runner.py, agent.py, and fully_conv_lstm.py.

To further parallelize the LSTM to minimize time, modify the code under the if

statement “if isLSTM:” in fully_conv_lstm.py.

Fix environment to network connection so that the improper batches of batches and

differences between stepping and training will not be needed. By doing this, LSTMs will most likely not speed up or will speed up minimally; however, the network will be of proper form and will be easier to read and add layers to.

Transformer (3):

When I was finished with adding in the LSTM layer, I came upon a paper about relational deep reinforcement learning which had a much better result than the original paper introducing SC2LE, an environment to test reinforcement learning in StarCraft II, did. In addition, AlphaStar mentioned that they used transformers rather than relational deep reinforcement learning specifically. Following this, I looked more into the literature on transformers and found that they were still uncommonly used, but that they would be much faster than LSTMs, and be better suited for memory in BetaStar since it could draw from events from long ago clearly by saving its inputs and outputs at every time step, while LSTMs only used and saved a single state at each time step.

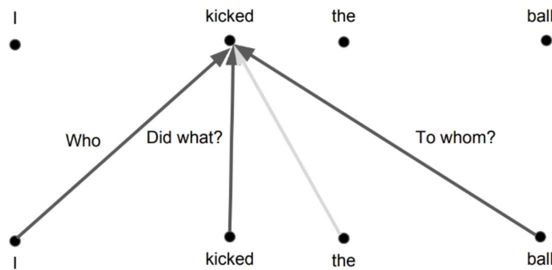


Figure 3.0: Self attention in a transformer layer which gives the network past inputs and outputs as different elements and allows it to decide what is important.

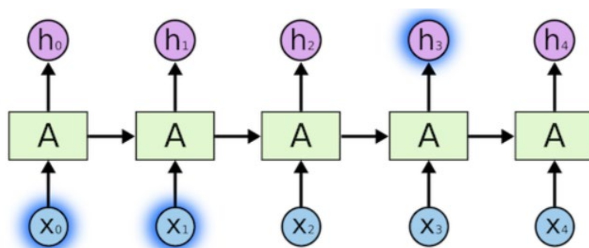


Figure 3,1: Recurrent neural networks, like LSTMs, only pass a single state to the next step, which may minimize the amount of information that is passed on, among other

Adding the transformer into the network proved to be a very hefty task. To begin with, all the official examples of the transformer, one created by the writers of the paper and one created by tensorboard itself, were created in keras, while our network was written entirely in tensorflow. After scavenging through other repositories, I came upon a transformer network written fully in tensorflow; however, it was a tutorial based transformer, had no batching available at all and was specific to language translation, the main application of transformer. Despite this, I managed to add in batching, patch up embeddings for the transformer to accept BetaStar's data, and tailor the transformer to other networks with time.

In the beginning, we wanted the transformer to be as high in the network as possible, so that relational information could be factored in before running through layers; unfortunately, when placing the transformer layer higher in the network, the entire network would crash as the dimension of the input would be way too high for our computational power to deal with since the dimensions were in the 10,000s range. Thus, I settled with putting it lower in the model, where the dimension of the layer output was 256. In addition, with the knowledge of having dealt with LSTMs previously, I was able to overcome a deeper issue than the LSTMs of using tensorflow conditionals and batching batches of lists of previous inputs and outputs of the transformer and padding lists for tensorflow to accept and other similar issues with the LSTMs.

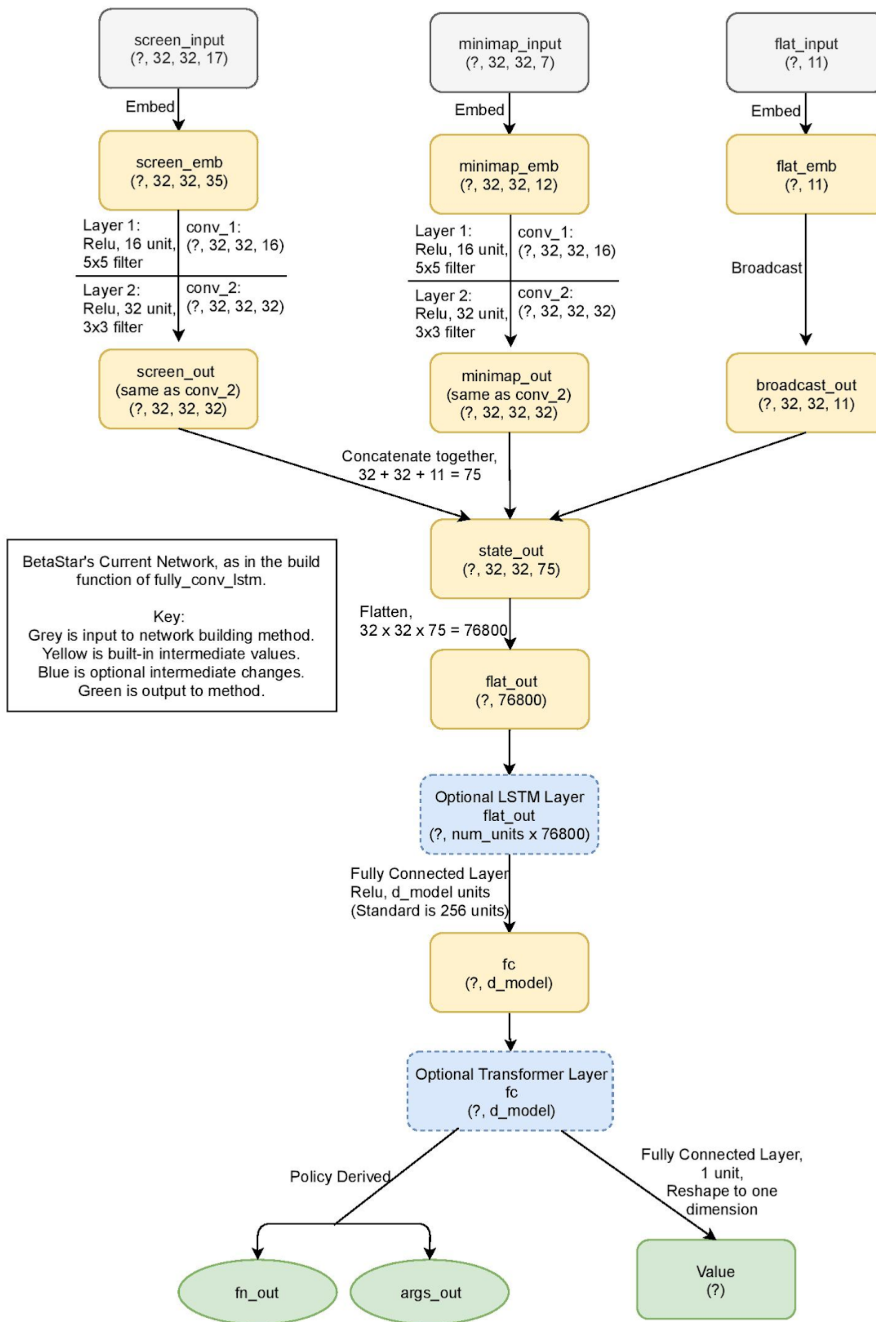


Figure 4.0: Diagram of full BetaStar Network, ? refers to batch size

As promised, transformers run much faster than LSTMs and its time is comparable to running just the fully convolutional network. Moreover, running the network with LSTMs takes the same amount of time to run the network with LSTMs and transformers, so it may be helpful to run LSTMs and transformers to improve the network.

	Time:	Slower by:
Fully Convolutional Network	3.33 sec	1x
Fully Conv with Transformer	10.33 sec	3.1x
Fully Conv with LSTM	600 sec	180x
Fully Conv with LSTM and Transformer	600 sec	180x

Figure 4.0: Time benchmark for transformer and LSTMs.

To test for any remaining issues, I checked whether staggered episodes would affect either LSTM or transformer runtime, and whether convergence could occur on Defeat Roaches. Currently, there are no known bugs in adding the transformer layer; however, further testing should occur to verify this.

Possible Future Improvements:

Increasing the dimension of the transformer model by:

Changing d_model in runner.py

Changing d_model in fully_conv_lstm.py

Note: The standard dimension for transformer models is 512; moreover, in the

paper that introduced transformers, it states that it used the dimension size 1024 for best results.

Further testing on speed of convergence with transformer and LSTM with transformer.

Fix environment to network connection so that the improper batches of batches and

differences between stepping and training will not be needed.

Future Work:

Overall, we have been closely following the network presented by paper that introduced PySC2. At the point where we are now, if we would like to improve our network further, it may be important to enlarge our network, and possibly follow a different network in a paper with better results in episode scores. It is also imperative that if we are to add new recurrent layers, or simply for readability and transferability, that we fix the improper connection between the environment and the agent so that the batching of batches is simply no longer an issue.

Additionally, we are at the stage where we could advance to the next stage of the replication of AlphaStar--extracting replay data and training our network off it. However, this is a very burdensome task which would take a while, and may simply not be worth the time and effort put in since the goal of the replication of AlphaStar is to apply what is learned to another system. A more feasible direction may be to continue Luke's work with upgrading psyc2 so that we can improve on making step by step progress with transfer learning

Finally, it would be very worthwhile to start agent versus agent tournaments in order to not just allow agents to train off each other and learn competitively, but also to benchmark how effective each of the major changes that all of the interns made. Besides these three major things, we could also add a pointer network, auto-regressive policy head, and other network upgrades stated in the AlphaStar blogpost in order to bolster the performance of BetaStar.

Luke Veenhuis

Progress report

During my short tenure here at the NRL, I accomplished three major tasks that I will expand upon in this report. These tasks were replay extraction; reward shaping and transfer learning; and dependency upgrading. In addition to these engineering tasks, this was also an incredibly educational summer in that I learned a great deal about Deep Reinforcement Learning through literature reviews and hands-on experience. With these new ideas, I've started to develop new and exciting areas of research I will be able to capitalize on moving forward.

Replay Extraction

In the first few weeks, we had some early aspirations of being able to tackle the idea of introducing supervised learning from replays. Though there were some setbacks (such as starting over with a new codebase), some work was accomplished towards this goal.

I have created a side project that extracts some metadata describing the games in each replay file. The features extracted include map name, number of frames, number of seconds, player 1 race request, player 1 race actual, player 1 result, player 1 MMR, player 1 APM, player 2race request, player 2 race actual, player 2 result, player 2 MMR, player 2 APM.

There had initially been an aspiration to transfer this data into a simple query-able database though no work has been done to achieve this goal. It is my understanding that this database would be useful in selecting individual matchups for BetaStar's training. There may be a time when training specifically on high-level Protoss vs. Terran fights is necessary, and this database would provide that with ease.

Reward Shaping and Transfer Learning

One of the significant limitations BetaStar will face the availability of computational resources. DeepMind primarily focused on reward systems that rewarded the integer value of the amount of time an end goal was accomplished. While this is perfectly viable on paper, in practice, it requires a tremendous amount of computation not easily found at the NRL. To reduce the training time of BetaStar, two methods have been proposed: sparse-to-dense reward shaping and transfer learning.

Reward Shaping

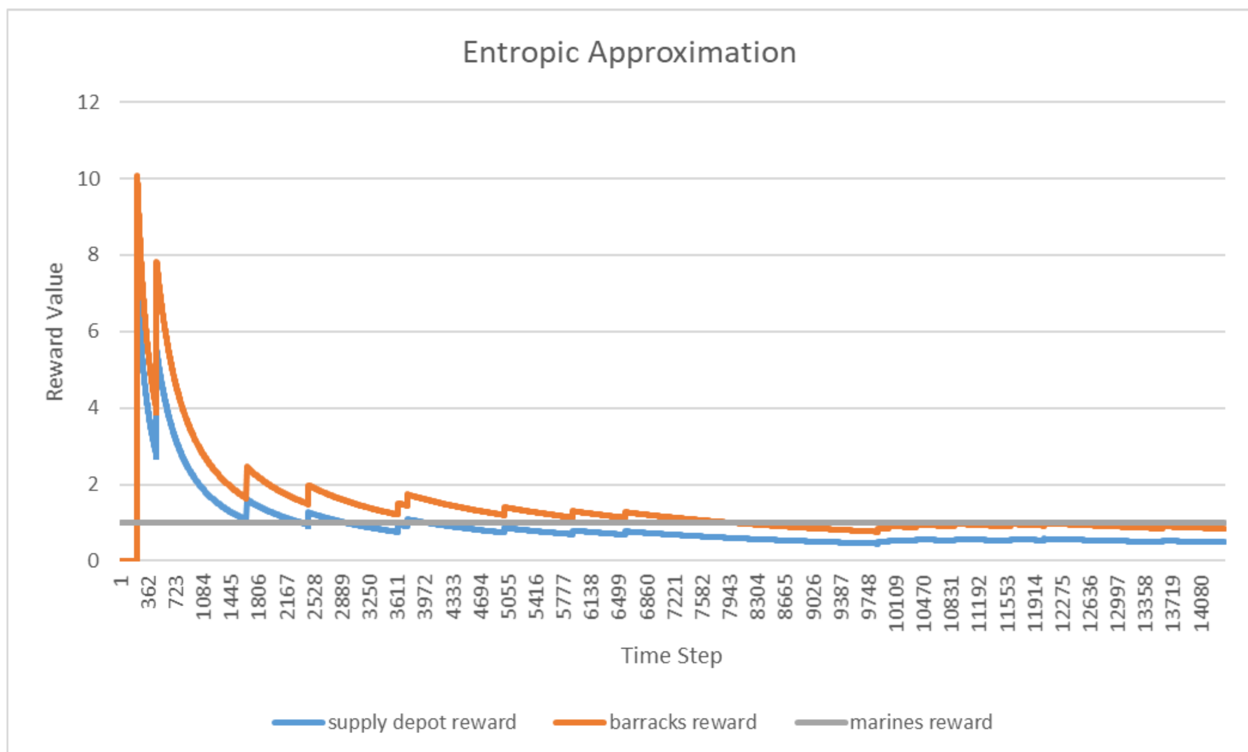
Our original proposal for reward shaping was to reward the agent in such a way that it was encouraged to reduce the entropy of its environment. In my previous experience, I have observed that agents in an environment, whether they be human or artificial, always act in such a manner that always result in a particular distribution known as a Zipf distribution. A Zipf distribution is a transformation of an entropic distribution in that you can convert a Zipf distribution to an entropic one through a simple negative log function.

Encouraging an agent to develop an optimal Zipf distribution in a small number of episodes would result in an agent that accomplishes its tasks in an ideal manner. To do this properly would require our reward function to remember the state at each frame of an episode for the totality of its training period. After weeks of trial and error, it was determined that this was not possible within the StarCraft environment simply because the game is not designed for existing within memory for a large number of episodes. Of course, one could write and read from a file before and after StarCraft exits and reenters memory, but it appears this functionality only exists in the windows version of the StarCraft binary which would increase our training costs far beyond practicality.

To circumvent these limitations, we elected to create an entropic approximation formula. The first iteration of which has taken the form of

$$r = \sum_{i=1}^{n} M(VT^{-x_i})$$

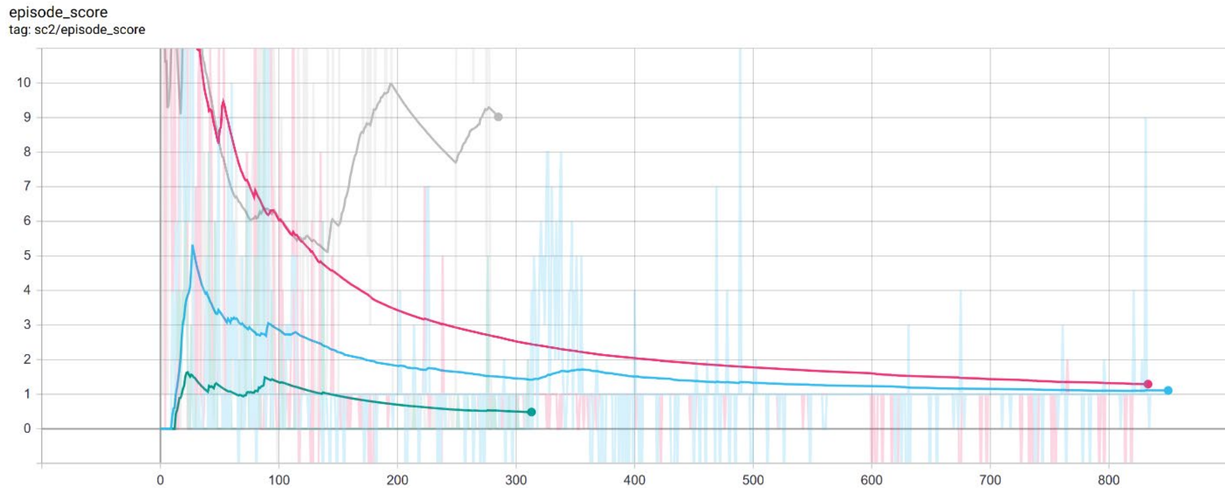
Where n is the number of subgoals, V is the value of the variable representing each subgoal, and T represents the time step. X is simply a free variable used to adjust the rate of reward decay for each subgoal. M is another multiplier used to scale the reward to a desirable level. The figure below illustrates an example of this.



A chart illustrating how subgoal rewards are designed to decay throughout an episode while the end goal reward remains constant. Supply Depot was calculated with M = 1000 and X = 1.327. Barracks was calculated with M = 1000 and X = 1.27

This reward shaping was designed in the hope that the agent would begin to recognize that after approximately 25% of the episode building supply depots become less rewarding than the end goal and that the same should happen to barracks approximately 50% of the way through an episode.

Few results have been collected thus far, and what has been collected seem somewhat inconclusive. Most agents perform suboptimally while a few still seem to perform well. To understand the effect of this entropic approximation better a broader set of rigorous tests are necessary. The graph below illustrates the results of agents shaped with entropic approximation.



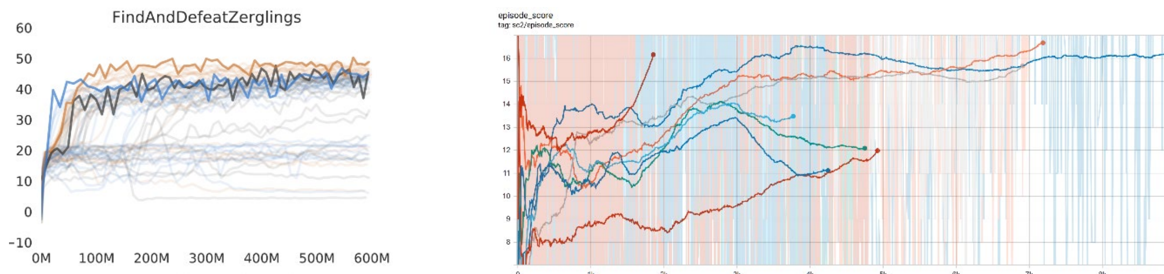
Transfer Learning

Another effort towards reducing training times came in the form of transfer learning. My approach to transfer learning was to exploit the initial convolution of screen and minimap data to encourage the agent to associate subgoals with larger goals.

For example, I sought to decrease the training time of the complex FindAndDefeatZerglings mini-game through the heat map variant of the MoveToBeacon mini-game. Doing so was a simple act of placing a zergling in the center of the beacon and making no other changes, as shown below. By associating a zergling with something it should move towards the “find” portion of FindAndDefeatZerglings becomes a pre-trained part of the agent’s policy.

The results of this transfer learning approach seem promising in that we are seeing similar results to AlphaStar in a much shorter time scale. While we have not run enough episode to replicate

the end level of efficacy that AlphaStar demonstrated I am confident that our first 3,000-8,000 episodes are quite comparable to what took AlphaStar ~10 million-~30million episodes to accomplish. This demonstrates that by breaking a large goal into easily trained subgoals training times can be diminished on the order of several orders of magnitude.



Dependency Upgrade

The original source code clone from Simonmeister's repository was a couple of years old, and with that, it was of course written to a now-outdated version of PySC2. While this was fine for many tasks we've performed this summer, there were some limitations that became apparent in the last few weeks of my internship. Most notably, as the curriculum score became a multivariate combination of many sub-goals rather than a single representation. By upgrading to PySC2, we now have access to a broader set of observations including a count of every unit at the end of each episode and as well as the value of any global variable stored within the map at the end of each episode. Instructions on how to accomplish this can be found in the git's documentation.

Future Work

For future work, I think it would be essential to elaborate on both transfer learning and complex reward shaping. A broader battery of tests would be necessary to determine if changing the values of the free variables have a significant effect on the performance of the model.

It would also be worthwhile to experiment with alternative network architectures. Exploiting the scaling methods detailed in the efficient net paper may provide significant improvement in performance that simply is not attainable in a network as shallow as ours.

As we expand into mini-games in which agents learn to compete against each other, it may be useful to explore some newer architectures that have been developed by DeepMind since AlphaStar was originally developed. Most notably, their FTW architecture seems to develop strategies strong enough to defeat humans in as few as 50 thousand games. With this speed increase and that the agent determines its own dense reward function, it could be a very useful architecture for those looking to save on a large amount of computing time.

Alex Saam

Progress report

In the beginning, time was spent mostly on learning about machine learning and the AlphaStar project while I attempted to receive work from Leslie.

The first task I was set on was to get a mean value from a series of files related to the research being conducted on adversarial attacks by Leslie and Adam. The task was lightweight, and took only an afternoon to make, however the rest of the day was not wasted; time was spent on office orientation, getting a grasp on my knowledge of machine learning, and completing the required government training courses. The program, although functional, was not used heavily and was instead replaced by the next program I would make.

The next task assigned to me was to create a program which would organize and format data related to the work being done on adversarial attacks by Adam and Leslie. The end-goal of the program was to provide summary statistics from which a table could be generated for use in a paper on the topic.

The program itself takes as input a folder of files, ignoring file name and type. The program then iterates through every file, scanning every line for one of a few keywords in order to obtain some values: the attack (thus the identification of the file), the clean value (performance of the network without any distortion), and the adversarial value (performance of the network with distortion). After obtaining and sorting all these values, the program then formats the compiled data into a csv file, and generates a summary table containing mean, standard deviation, and residual values. This summary table is formatted into a separate csv file.

As for internal organization, the program primarily uses dictionaries to associate attack name and value, however this was not always the case; before this development, I had tried to use nested dictionaries (i.e. dictionaries for every attribute e.g. {Attack:Hyper-parameter:Trial number:values}), which proved to be far too complicated, time-consuming, and bloated for practical usage. So a valuable lessons was learned: limit nesting.

This program was used for Leslie and Adam's research for the majority of my internship, however, due to recent changes in the data, this program has become obsolete and Maya has replaced it with her own program. Note: the development life of this program did not end once I was assigned a new task; I frequently looked back at the program to update it for new data formats, add versatility, and improve user-friendliness.

Then, I was assigned the largest task during my entire internship: creating a webserver to host a leaderboard for a network's agents during runtime. Previously, the work I had done with

networking involved only scraping websites, which is only the client-side of networking, so most of what I performed in this task was new to me. As such, I learned a lot from this one project.

Before development could begin, a plan of how components in the network would interact with one another was needed. So, ideas were thought through, and a simple network was designed: the client would hook onto an agent and send data to the webserver; the webserver would then receive the data, append it to the database, and update its webpage contents to match. All of the client's interactions were to happen in a non-blocking, graceful manner. This means that several agents can communicate directly to one file, as opposed to the agents spewing out files that then need to be amalgamated into a single file.

Due to the fact that the webserver was meant to host a leaderboard, a way to organize and store the data was necessary. SQL, a common database-creation tool, was chosen as the means to store the data. This was a perfect choice due to the fact that Python, with its SQLite module, supports SQL natively, meaning SQL functions could be directly integrated into the server.

The creation of the server itself was simple; Python easily supports http(s) servers with its https module, allowing one to quickly start a server and host webpages. Real development truly began when the SQL database was being formed. The SQL database started off in a separate file (PopulateDatabase.py) and then gradually moved on to being incorporated inside of the main server.py.

Separate files were created throughout the development process, and features were added as needed. E.g. when rows were appearing as duplicates, a conditional clause was added to UPDATE when the row was present, and INSERT when the row was missing.

A list of features are here below:

- Organization of the data into tables
- The ability of the program to create new tables and databases where they are lacking
- The ability of the program to update entries in the database as well as append new entries
- Sorting of the table by clicking on a table header (e.g. "agent_id")
- A button to toggle generic hyper-parameters
- A delete function, wherein one selects rows, then presses the red 'delete' button
- A custom error message, notifying the user of a disconnection between browser and server
- A remote kill command by URL
- A .txt file from which IP address and PORT are read

All of this was developed over the course of a week.

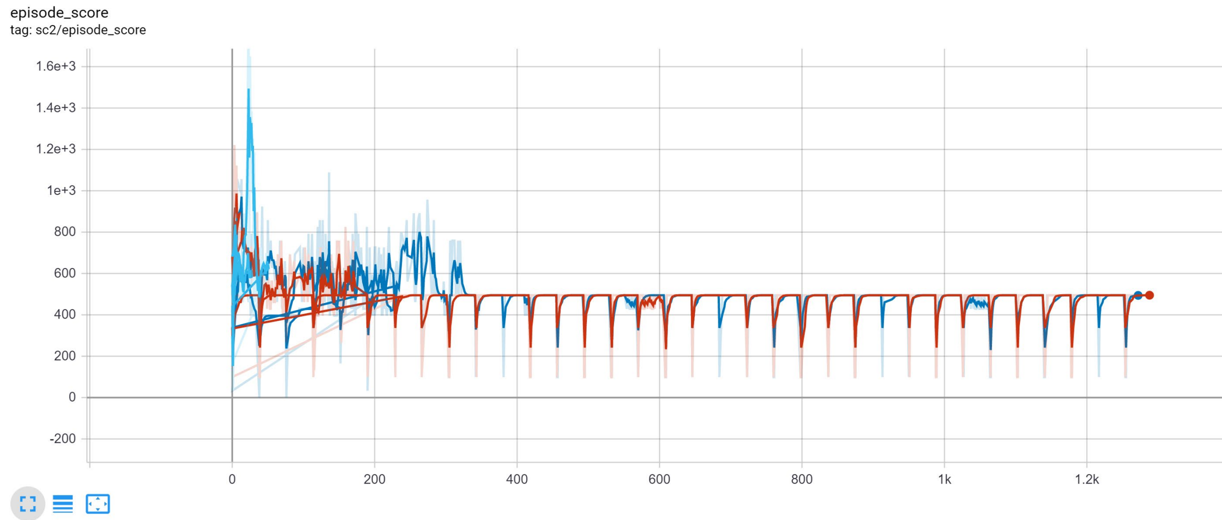
As a small, in-between task, I was also asked to create a file comparison program, in order to check if two files were equivalent in data. As a module for this already exists, such a program was trivial to develop, and so I leave it in here as a side-note.

As part of my recent work, I was assigned to evaluate a scenario: if the agent learns in steps - i.e. in a multi-step process, the agent is rewarded for the first step only, then the second step only, etc. – will it learn better or worse than other forms of reward shaping (e.g. exponentially increasing by step).

I hypothesized that learning by steps would be worse than other forms of rewarding wherein agents are rewarded for every step continuously. This would be due to the fact that the agent would no longer be thinking of the smaller steps, but rather focusing on optimizing the current step is on, potentially leading it to forget previously learned tasks.

In order to test this hypothesis, I designed several StarCraft 2 maps that incorporate step learning. These maps are all based around the map BuildMarines. The process was going to be such: 3 tests would be conducted for every map, with every agent, and the reward per step would be kept constant throughout the trials. Over the course of 4 maps (focusing on the steps of building workers, building supply depots, building barracks, and then finally building marines, in that order) their performance, given by their score, would be recorded and compared with other reward shaping methods in order to determine efficacy. Multiple instances of this process would have occurred, with different but constant rewards per step (e.g. 33 per step, 100 per step, $1x10^6$ per step, etc.) and with different rewards across steps (e.g. 33/33/33/100 across step, 25/33/33/100 across step, etc.).

Unfortunately, due to technical difficulties and due to constraints of time, none of these experiments have completed. Some of the difficulties encountered were: Python versions on GP3 that were not 64-bit, installing dependencies one level above where they were supposed to be located, having to alter mini_games.py inside of pysc2, and training on the building workers step not functioning as it should. As for the last difficulty, it was hypothesized by Luke that the performance of the agent was due to it starting off with 12 workers, leading it to neglect maximizing score (as it was always, on default, getting some) and experiment with offing its own workers, thus decreasing its score. The following image is of the score itself:



The hypothesis given by Luke has yet to be tested or affirmed by any visual evidence.

Lastly, the final project I worked on was a setup for a 2-agent environment. Similar to the last project, due to time constraints, this project was never finished either.

Two different ideas were bounced around for this project: a setup where 2 agents are initiated, gave actions to the environment, and then trained off of what they learned; the second idea was a setup where one agent is initialized and it is fed two different observations, from which it trains itself. This is a simpler method for achieving the multi-agent setup, and it has a process similar to batching. Both of these are included inside of my GP3 account. The issue with the first setup is that it got stuck on training the second agent. The issue with the second setup is that it doesn't already have several different observations to learn from. Supposing the first one became able to train the second agent, and the second setup was supplied with two different observations, they would both (probably) work.

Summary

Our efforts described here had focused on replicating AlphaStar. This NRL Technical Report presents detailed documentation by all of the project's team members on investigations performed, accomplishments achieved, and dead ends encountered. One purpose of this Report is to make all of our work performed so far on replicating AlphaStar easily replicable by others. In addition, we have learned a great deal while working on BetaStar and plan to continue this effort.

Our future plans include incorporating a Deep Reinforcement Learning algorithm into the Next Generation Threat System (NGTS) to provide threat-assessment capabilities into the core enterprise system. The objective is for the DRL agent to perform (near) optimal battlefield actions, even in novel situations. Techniques described in this report will be the starting point for developing this intelligent agent for NGTS. An intelligent agent integrated within Naval Enterprise Modeling and Simulation tools, such as NGTS, will provide a challenging simulation environment for training future Department of the Navy warfighters. In addition, super-human tactics developed by such DRL agents can be analyzed as an additional learning experience.