Learning Operational Strategy Electing Robot

ECS170, Spring 2018

Artificial Intelli-Gents: Group Members

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

Problem and Contribution Statement

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

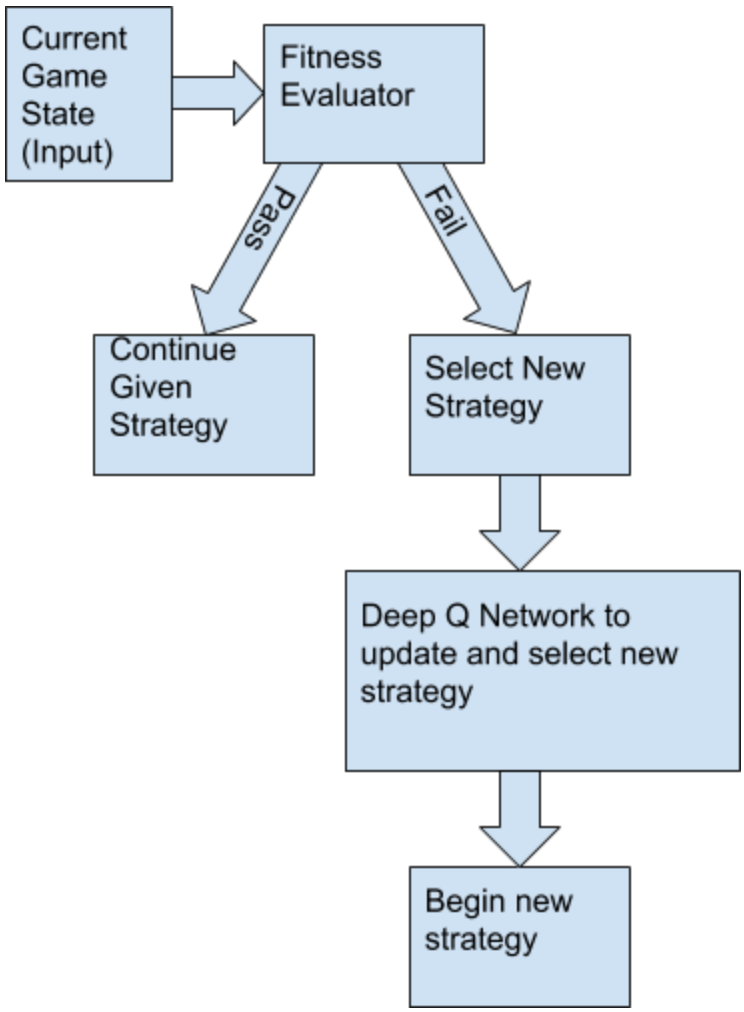
Since Starcraft II has several potential strategies for any given situation there is not a well defined “best solution” for how a bot should select a strategy nor how it will adapt if the opponent effectively counters the initial strategy. This will show that deep learning can be used in a competitive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

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

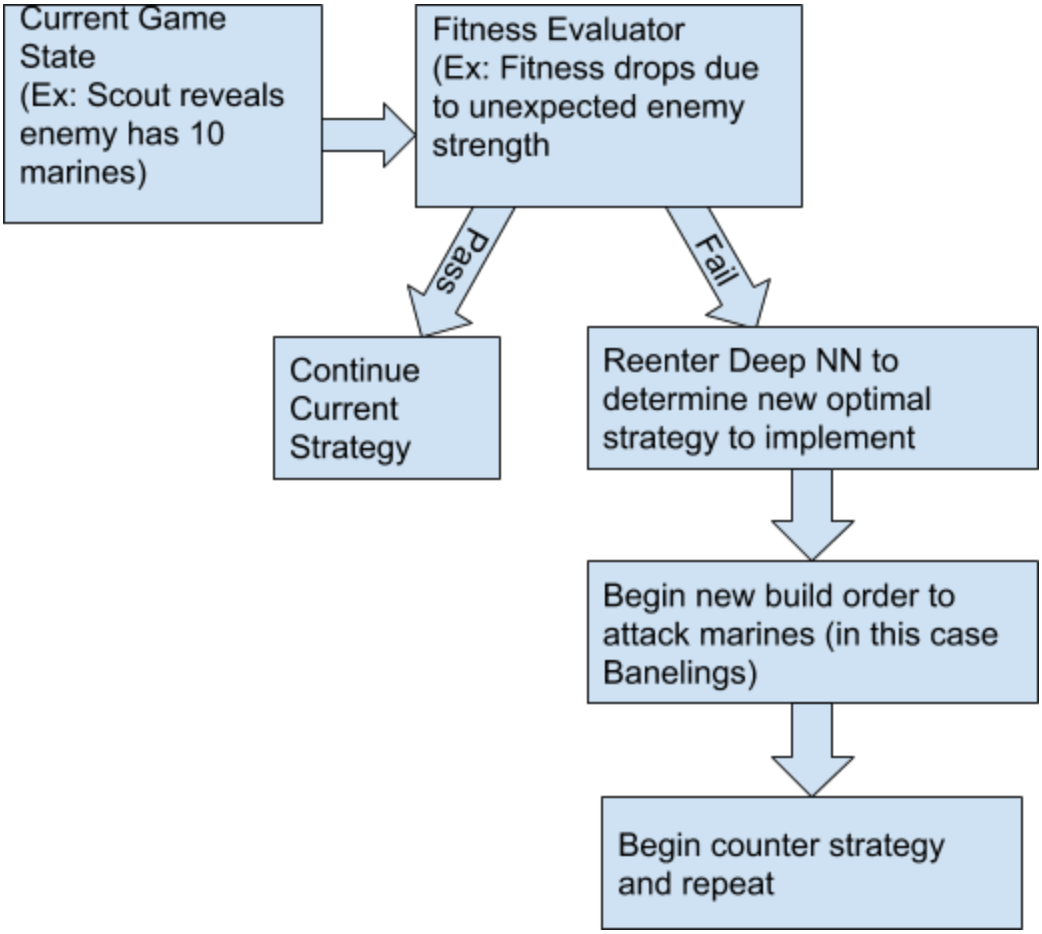
The project was mostly well scoped and held for the duration of the project. We did not use Steven Brown’s code as a reference in the end either because we switched over to python-sc2, but we did use a

similar reward strategy. Fitness also changed by evaluating past decisions and penalizing the bot if our fitness greatly dropped. We then update our neural network and predict how to improve for the next iteration.

Design and Technical Approach



Learning Operational Strategy Electing Robot Flowchart



Updated Learning Operational Strategy Electing Robot Flowchart With Example

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

We plan on primarily utilizing a Deep Q Learning algorithm that uses a neural network to approximate the Q values such that we have a history of previous trainings so that we can continue to update our weights to optimize the performance of our bot, similar to the implementation by Steven Brown. However, in order to improve his approach, we plan to incorporate a fitness function that will help determine when we need to enter

the neural network and choose a new strategy. This implementation is advantageous because it can be “lazy” in that we do not have to constantly input all of the data that we receive from the protobuf into our neural network. However, in order to ensure that our fitness function does not encourage stagnation, the bot will enter the neural network every X iterations to verify that the modular strategy it has selected is still optimal. In order to determine fitness, we have defined the following functions:

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

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

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

Updated Fitness Equations

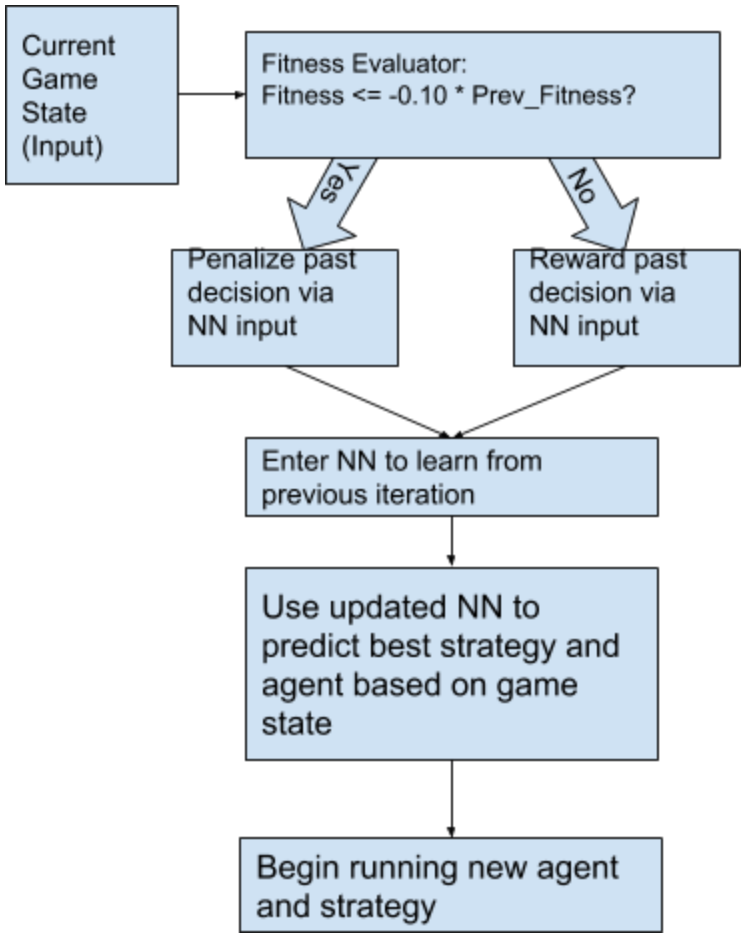
$$Fitness = (\#Defensive\ buildings)(4) + (\#Production\ buildings)(2) + (\#Upgrade\ buildings)(2) + (\#Technology\ buildings)(3) + (\#Other\ basic\ buildings) + (\#\ Other\ advanced\ buildings)(2) + (\#Fully\ upgraded\ bases)(3) + (\#\ of\ army) + (\#\ of\ workers)$$

$$Overall\ Fitness = Fitness(self) - Fitness(Bot)$$

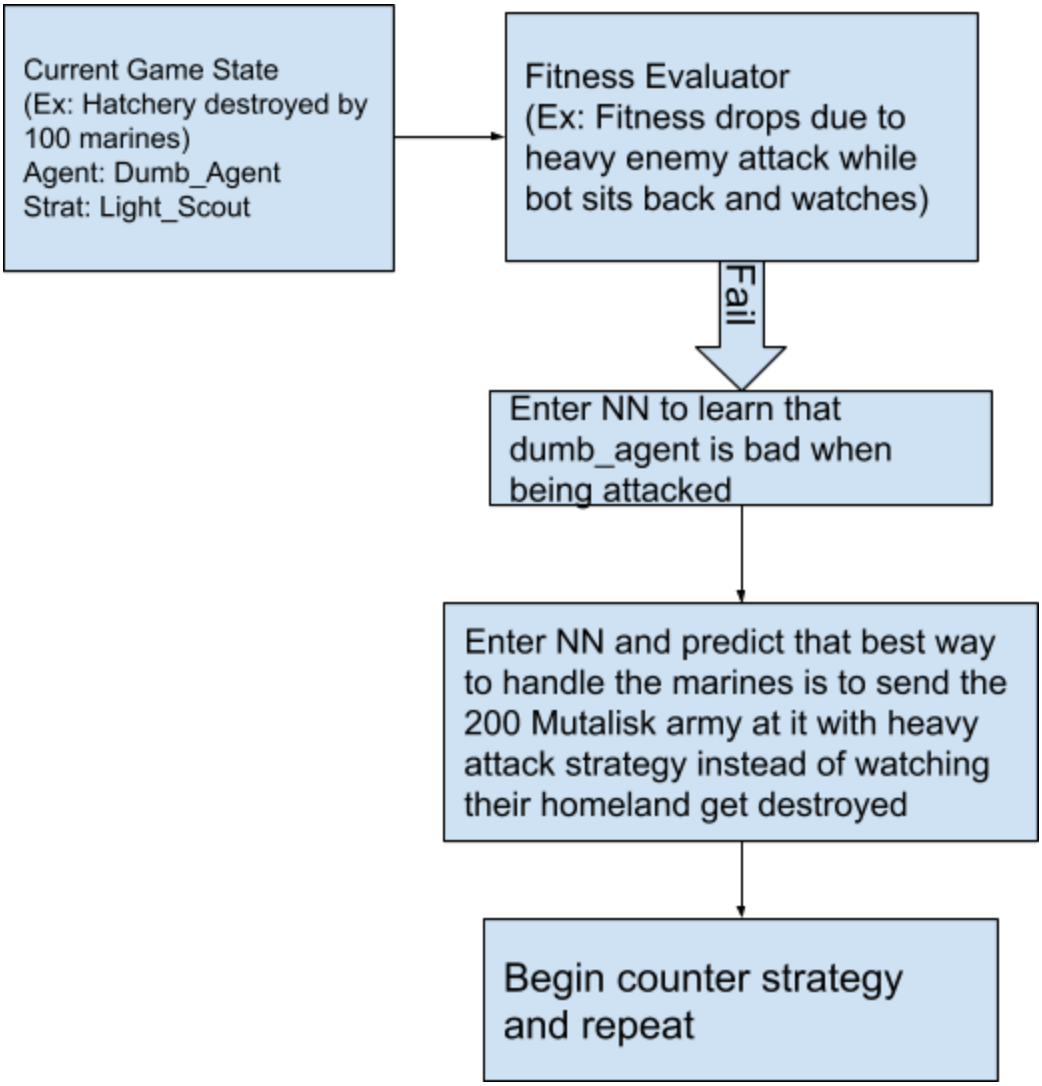
If the fitness reveals that the current strategy is no longer deemed as optimal, we will pass our current army composition, buildings, and upgrades that we have as well as the same information regarding the opponents. The neural network will have access to a Q table, which will not only act as our history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

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

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



Learning Operational Strategy Electing Robot Flowchart



Learning Operational Strategy Electing Robot Flowchart With Example

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

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

Fitness was also changed to give feedback on the previous decision to help our bot learn throughout the game, as can be seen in out flowchart examples.

Scope and Timeline

There are starcraft bots that use Q tables, games that use deep Q learning, and genetic algorithms to help explore large search spaces. Since Starcraft has a large search space and the potential to learn to adapt to strategies, we aim to fill the gap by making a single bot that can implement four different build orders, nine different strategies, and the ability to adapt to any given game state by analyzing the current state, determining its own fitness, and using the Q network to decide the most optimal strategy.

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

By Week 5 (Monday):

- Project Proposal

By Week 6:

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

By Week 7:

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

By Week 8:

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

By Week 9:

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

By Week 10:

- Bug fixes
- Finalizing github pages

Changing to python-sc2 forced us to push our timeline back by about a week to setup the new environment and convert our existing code to python-sc2. The separation of our original idea of strategies (Builds and playstyle), into two separate parts created additional tasks that needed to be completed. None of our changes affected the scope of the project.

Documentation and Access

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

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

Our group never used a weekly ToDo list because our tasks were rapidly changing throughout the project. We also did not keep track of changes with a change log. Instead, we kept track of each other's works by using a discord server with channels dedicated to specific parts of the project.

Github repo
Github Pages

Evaluation

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

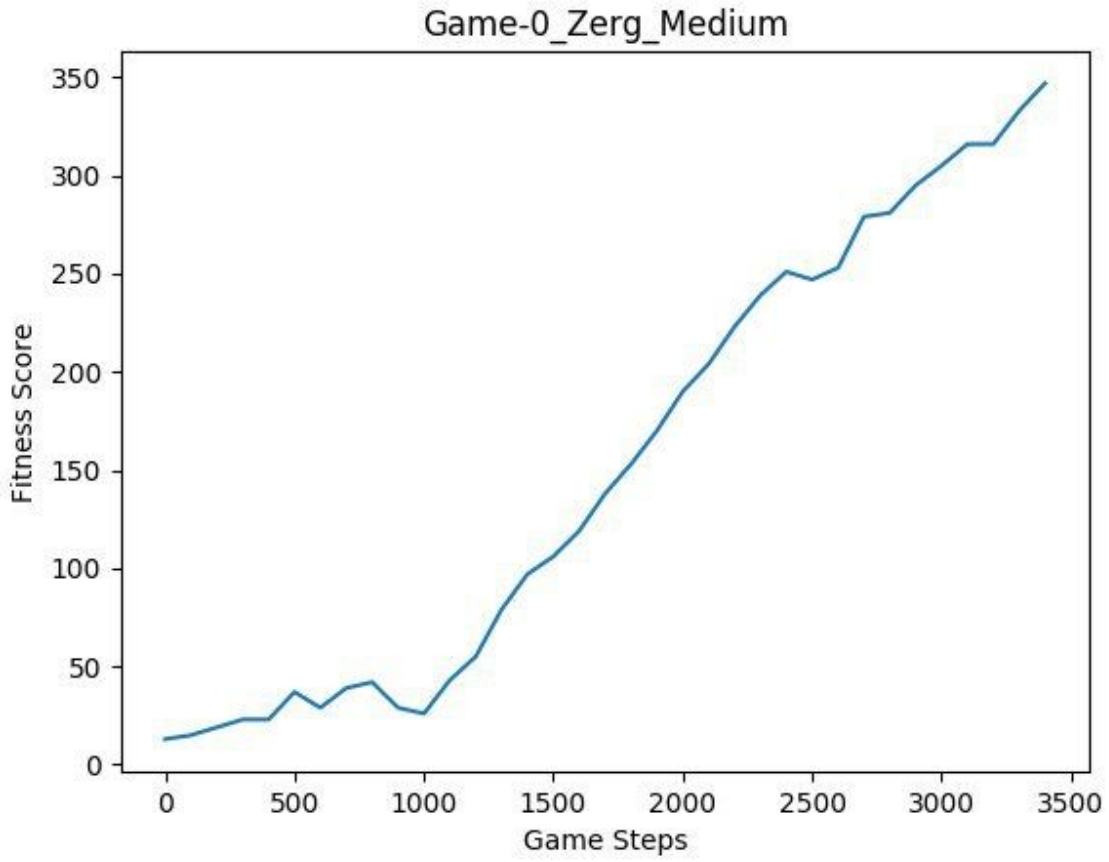
We will also have a function that will record the results as we are training our bot so that we can monitor the progress of our bot over time. This function will append to a file that will keep track of the win/loss ratio and we expect that as the number of games that we train on that the win rate should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

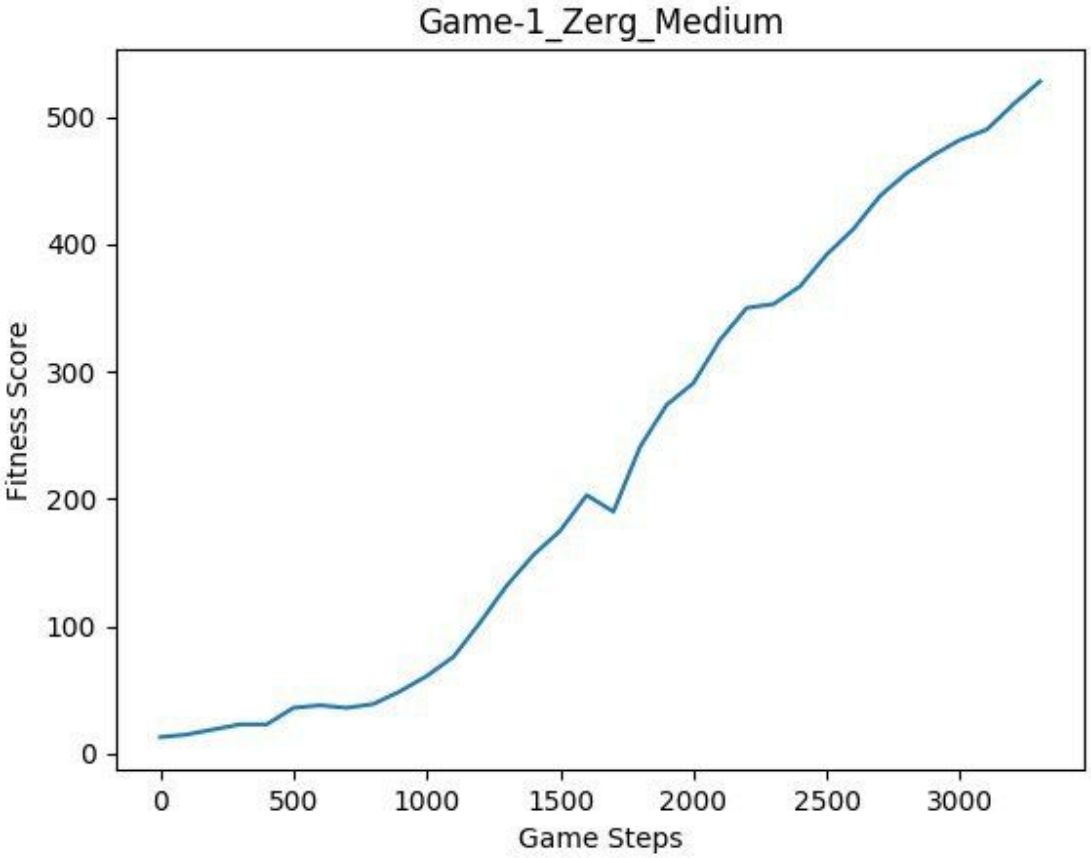
Our bot determines fitness by looking at its own units and what it knows about the enemy, The buildings were broken into several tiers depending on their uses (ex: upgrades versus static defense),

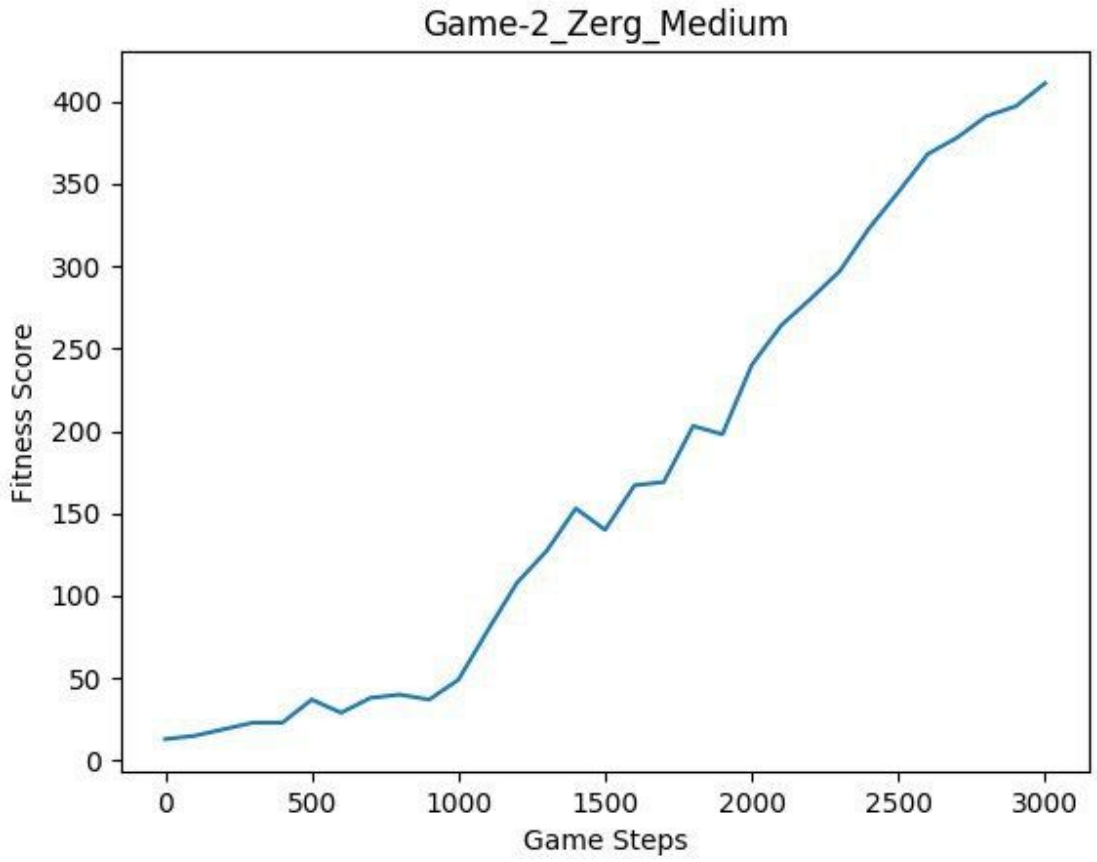
but the army units were equally weighted. In reality, all units in an army are not equal, but due to time constraints, we were unable to effectively group the different units into more meaningful weights.

Our bot, however, does change strategies, and then uses whether it won or lost the game to evaluate whether those changes were good or bad.

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



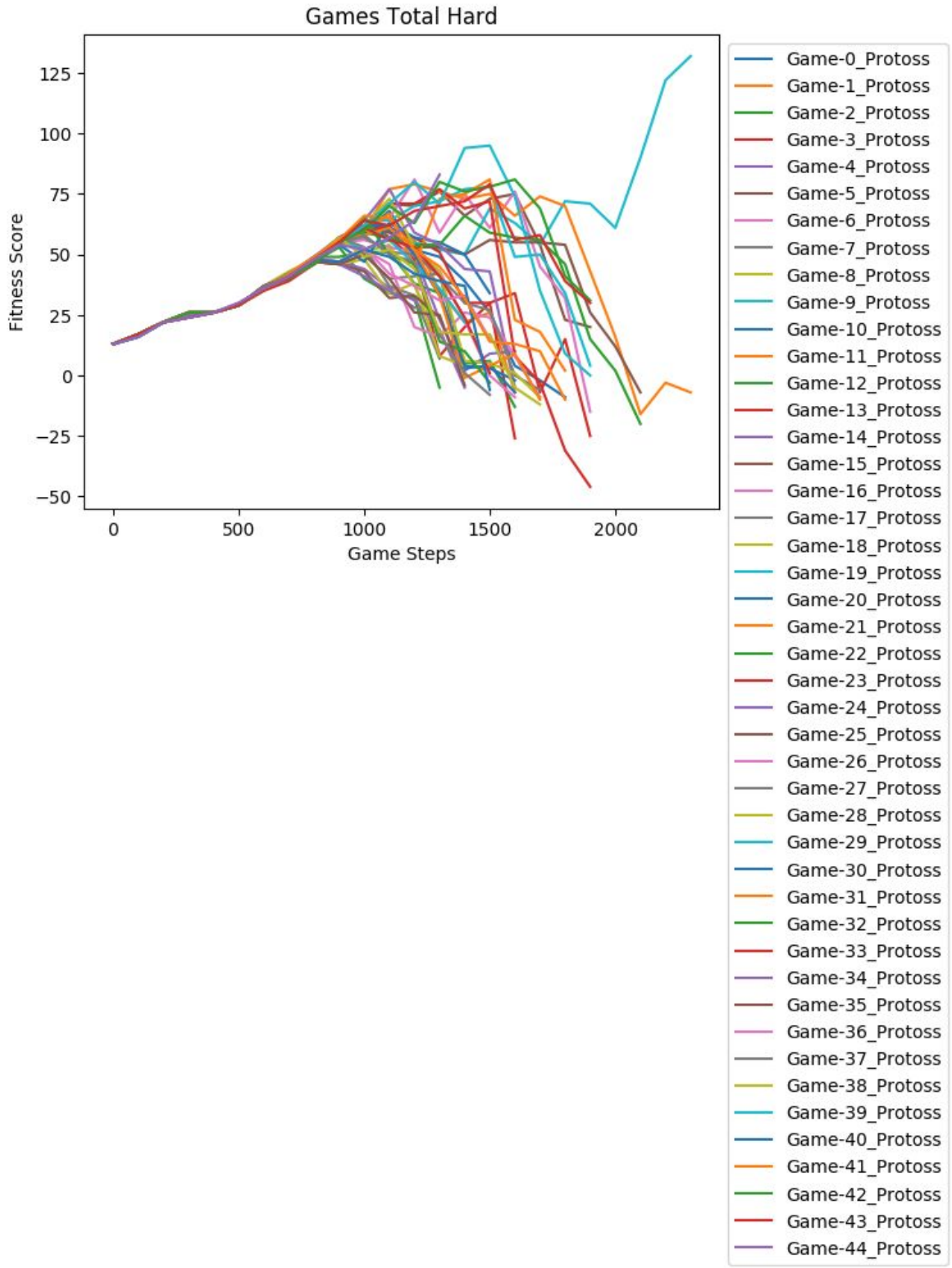

Game-1_Zerg_Medium
Fitness Score
0
100
200
300
400
500
0
500
1000
1500
2000
2500
3000
Game Steps



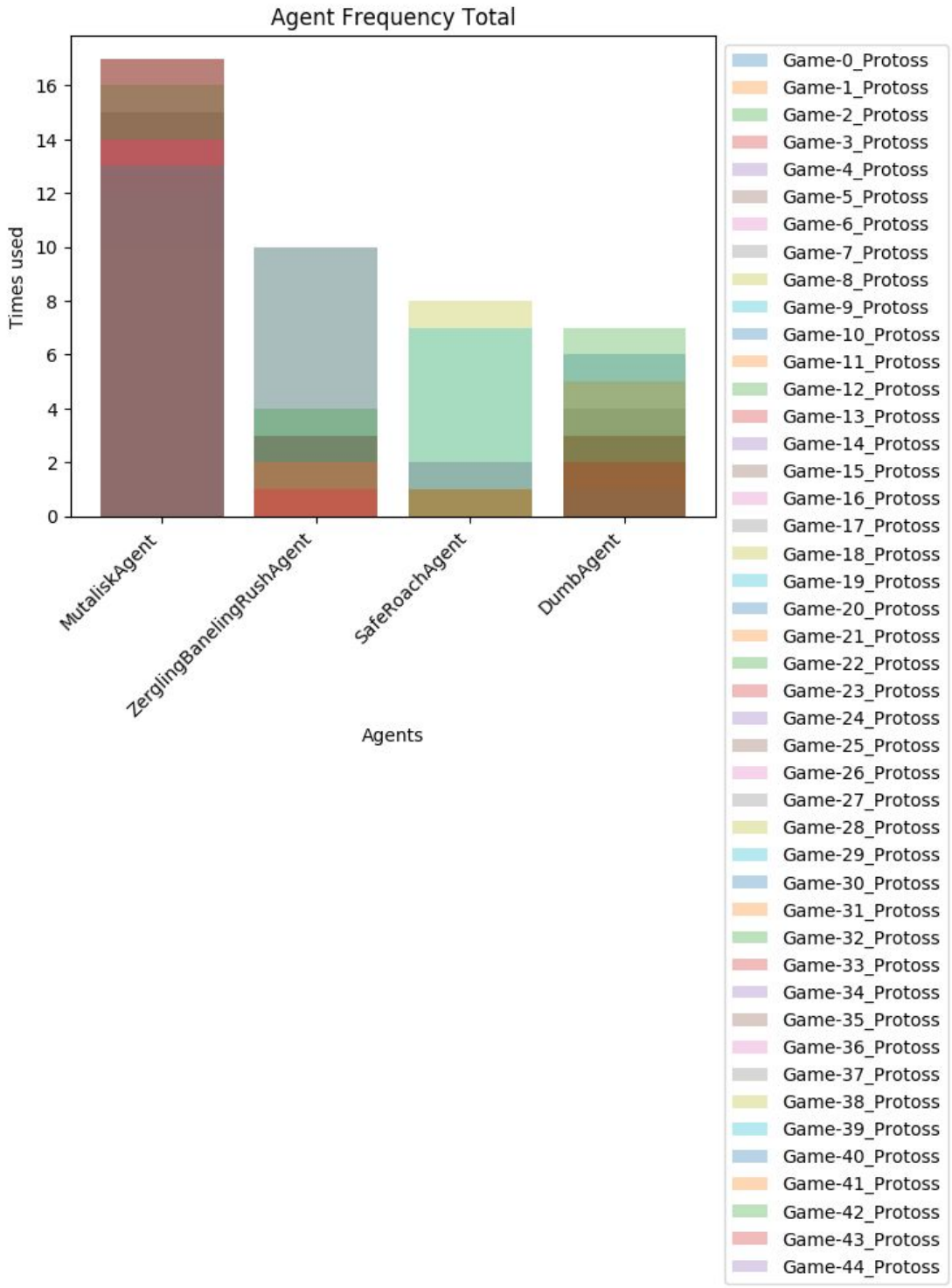
The diagrams above detail how the fitness changed for our bot when running against a medium zerg bot. Most of our strategies are able to beat easy and medium level bots quite easily, so to see if our bot learns, we ran it against a hard AI. We also see that our bot learned to play a little more effectively in this small test set by about roughly 200 game steps.

Hard Protoss Opponent Results:

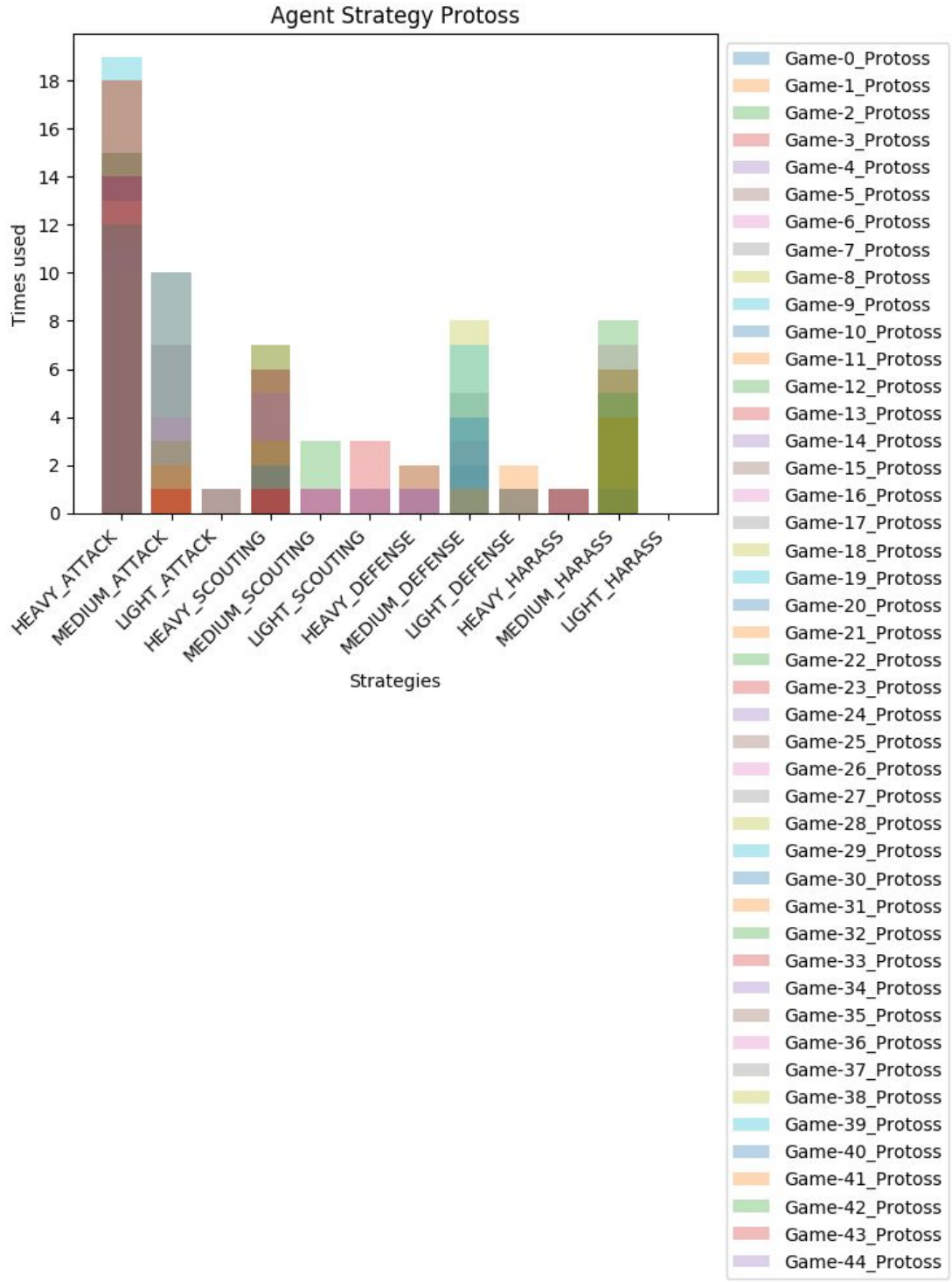
Fitness:



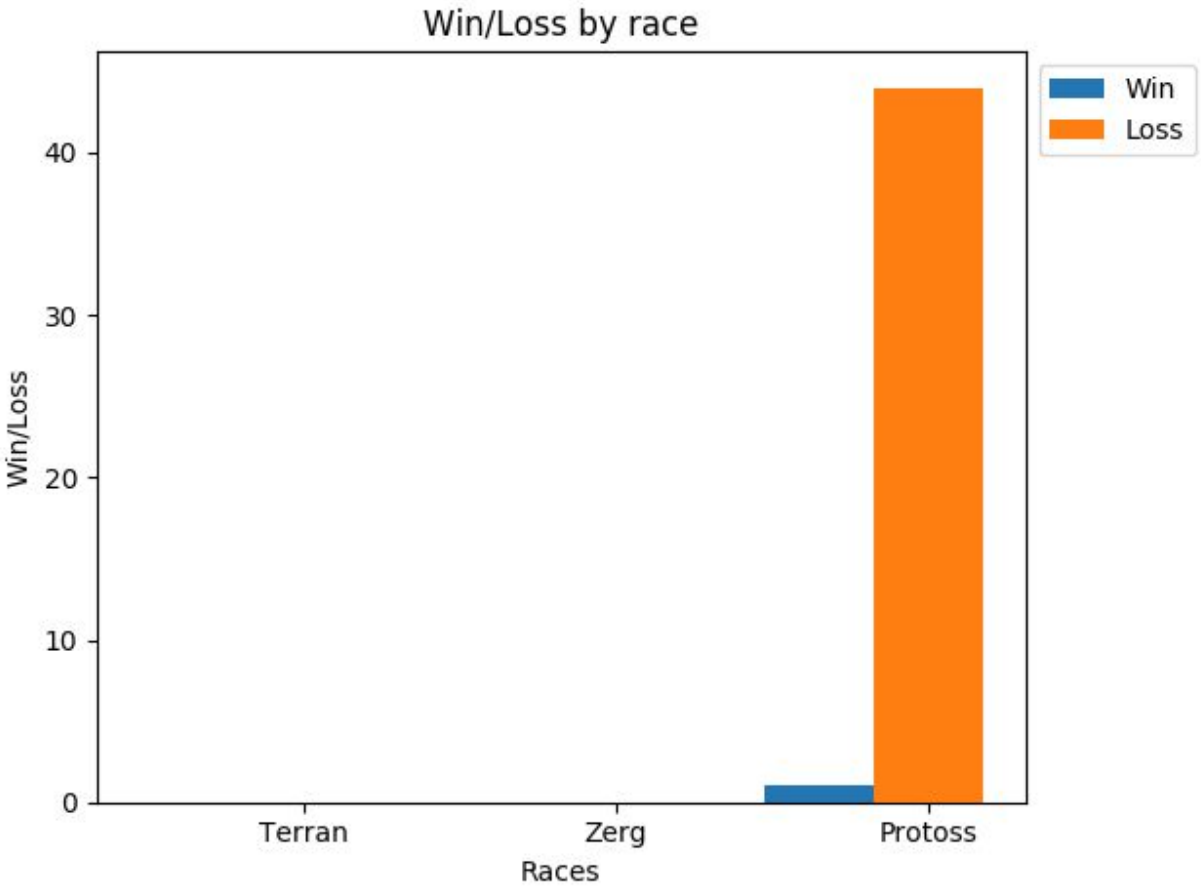
Agent Frequency:



Strategy Frequency:

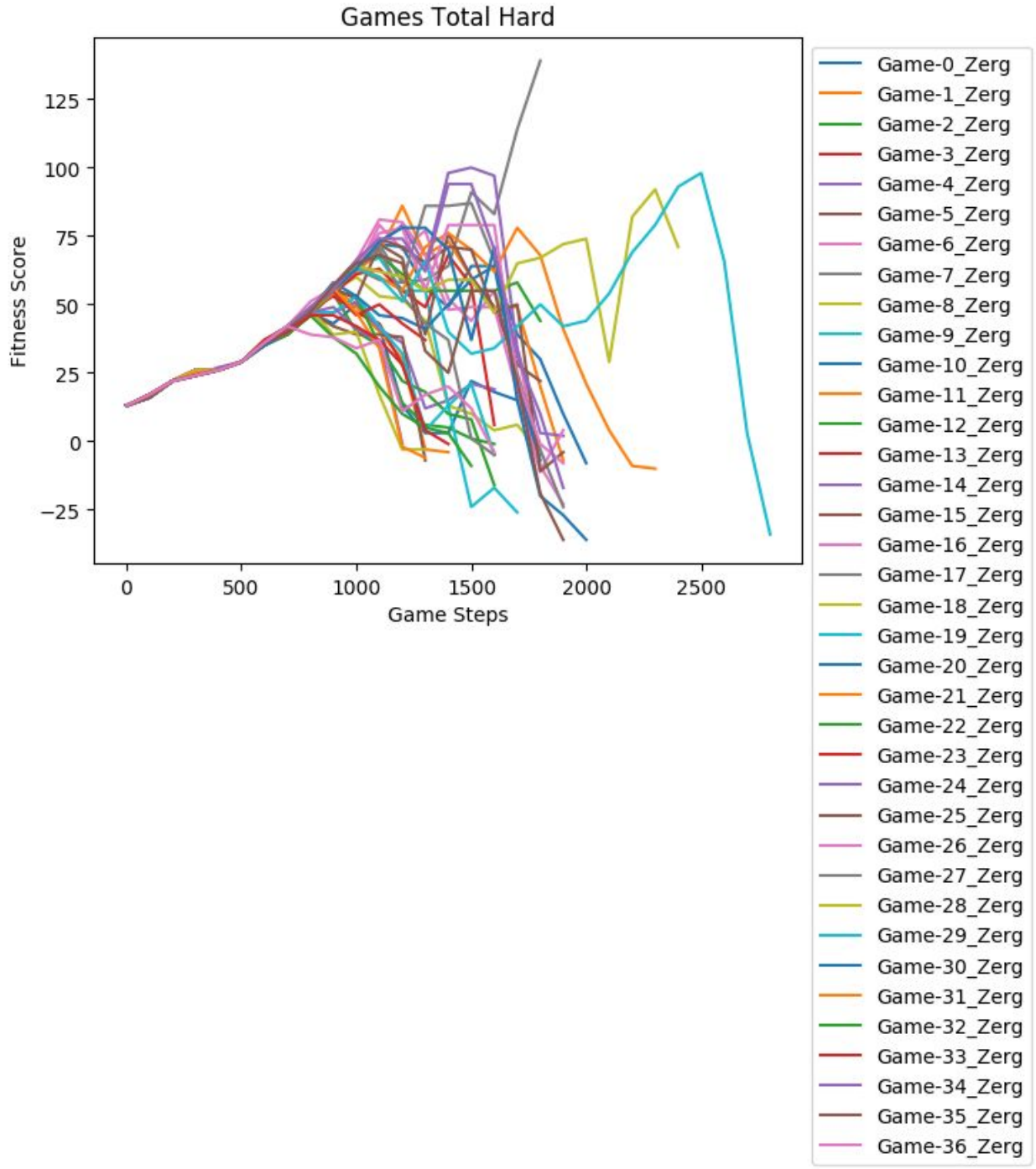


Wins and Losses:

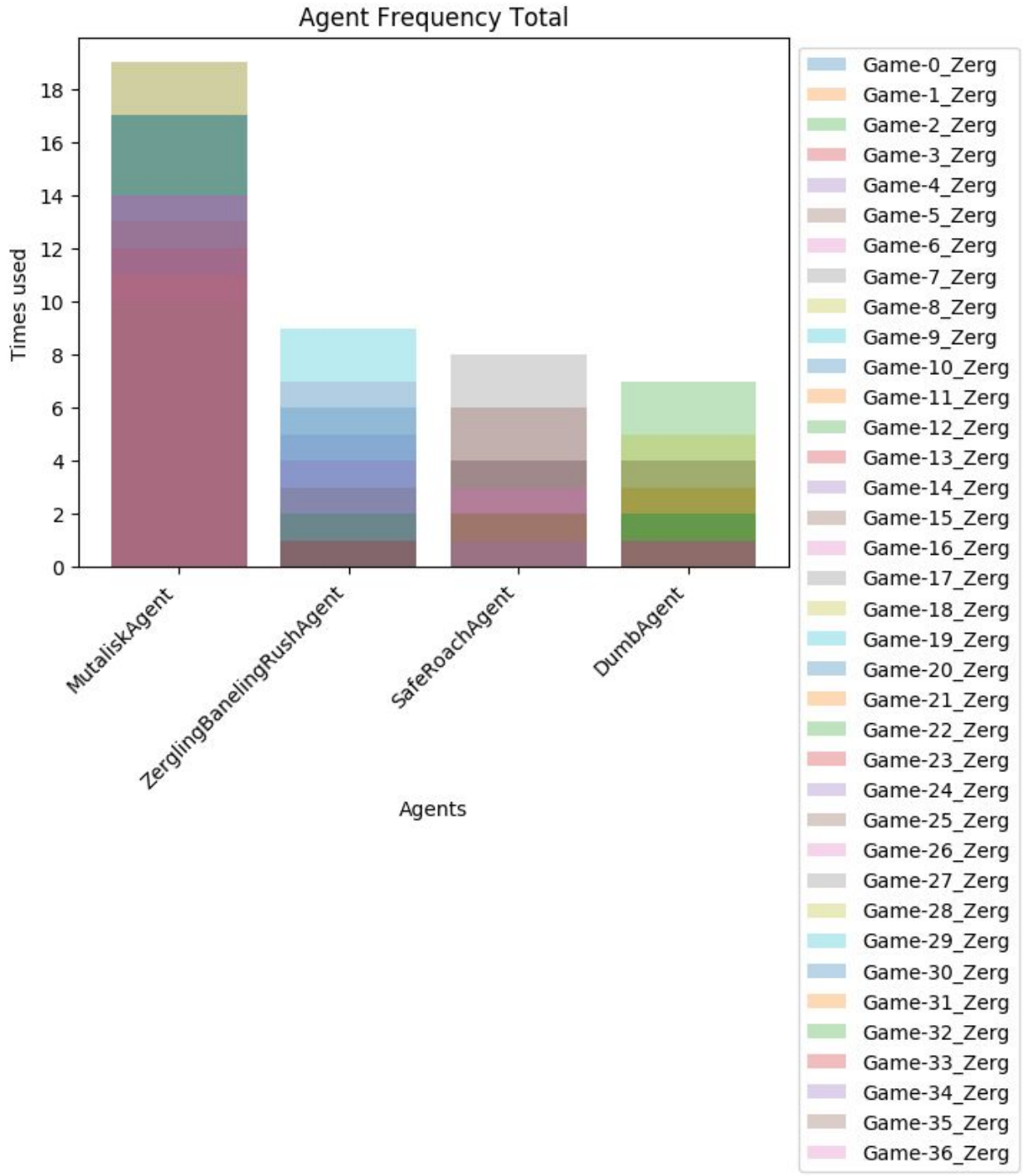


Hard Zerg Opponent Results:

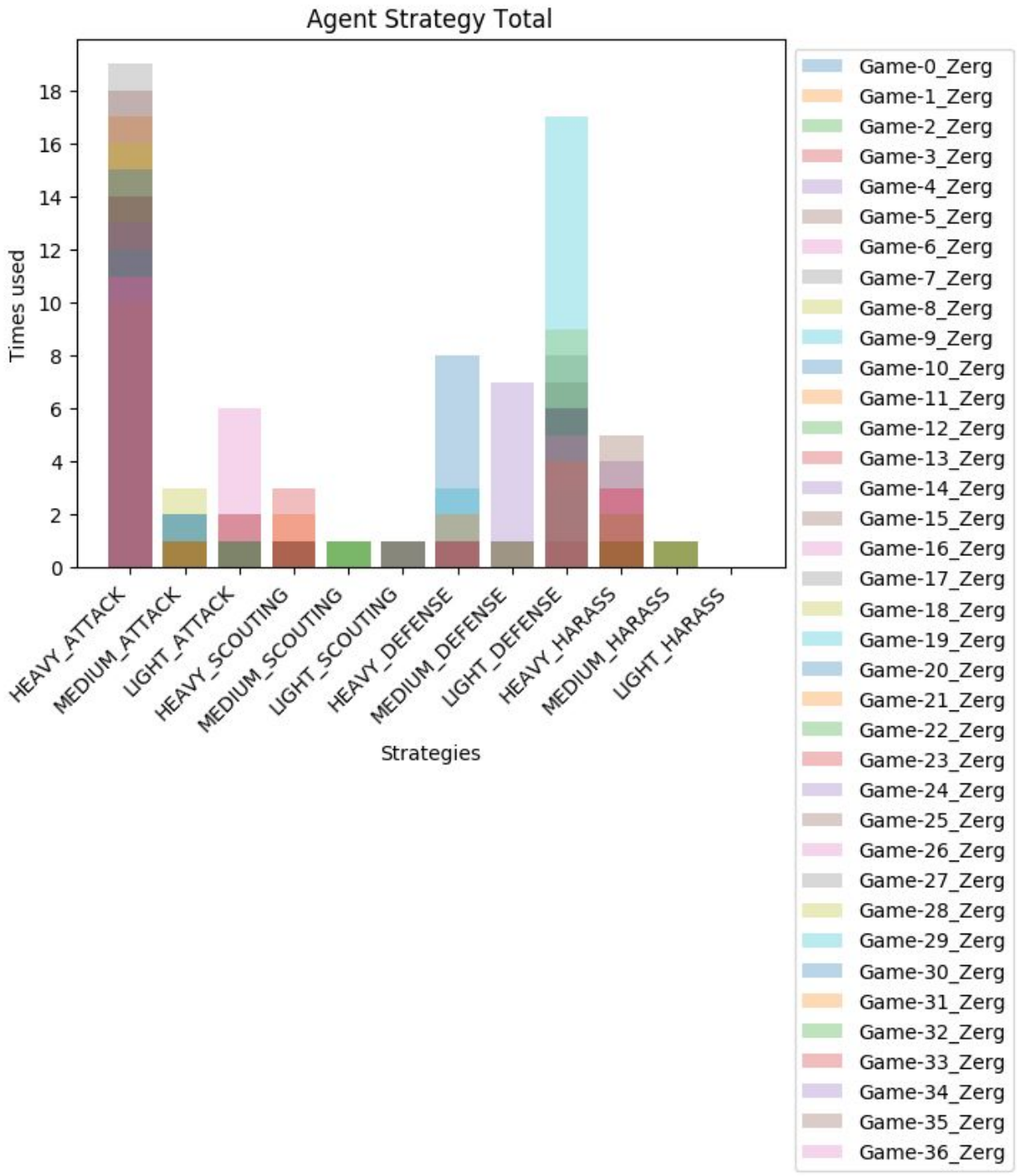
Fitness:



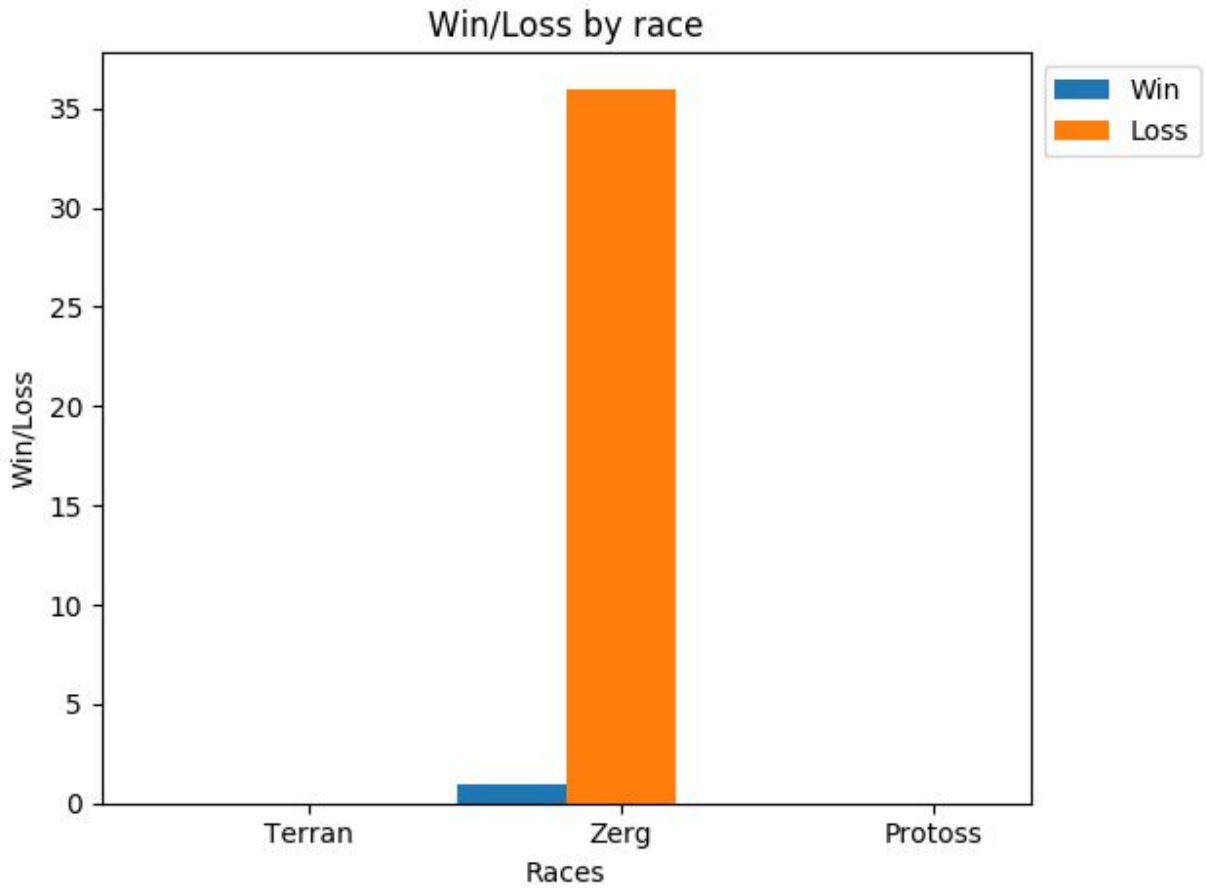
Agent Frequency:



Strategy Frequency:

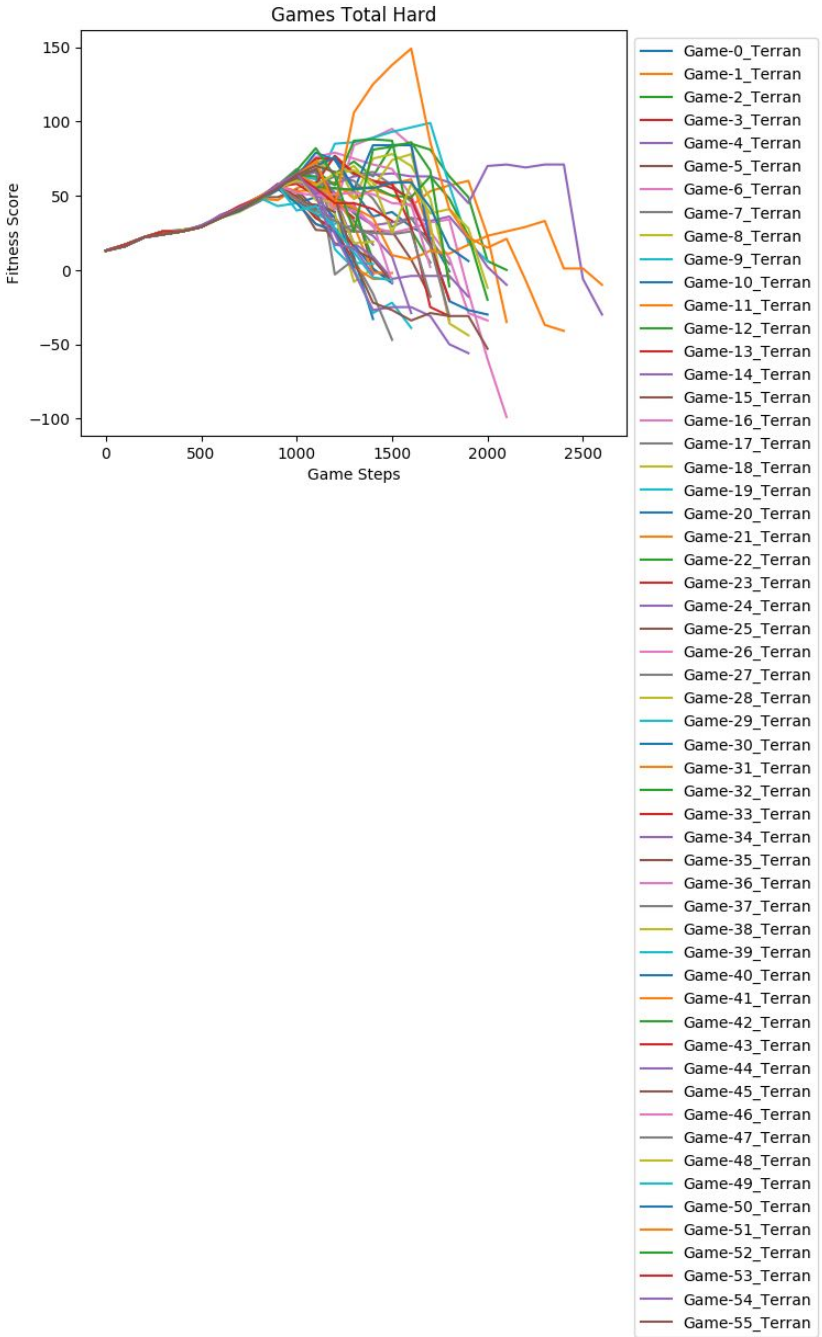


Wins and Losses:

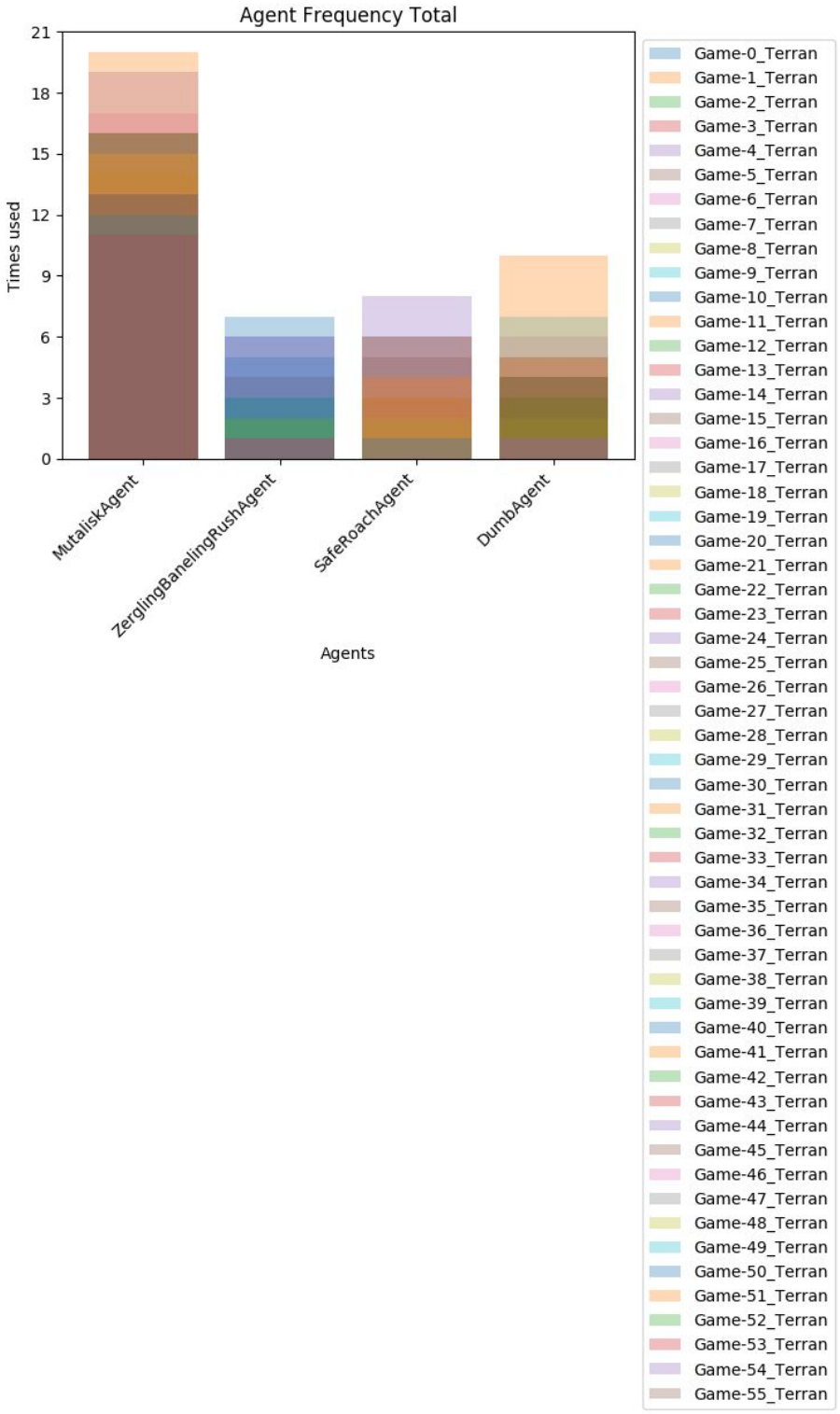


Hard Terran Opponent Results:

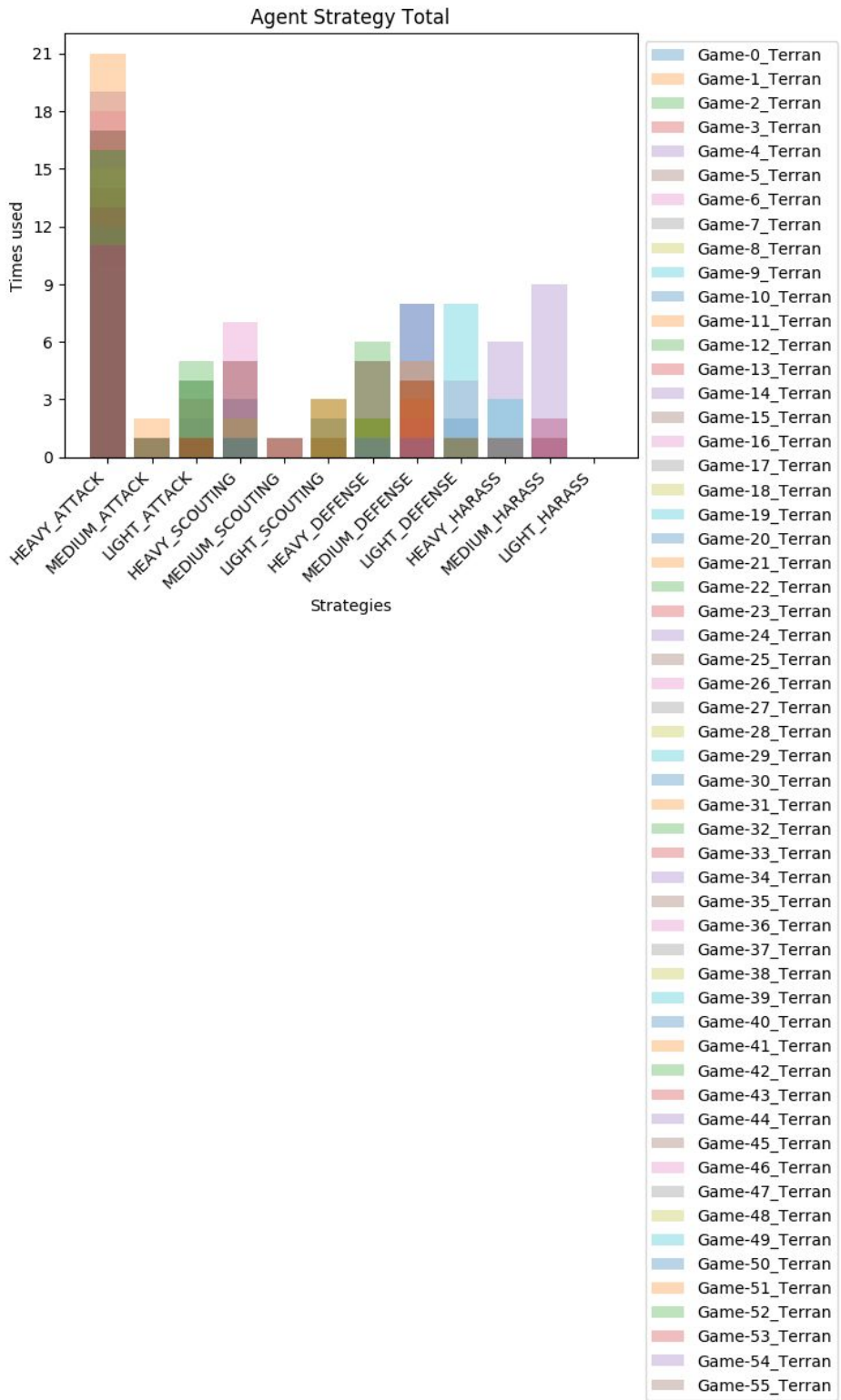
Fitness:



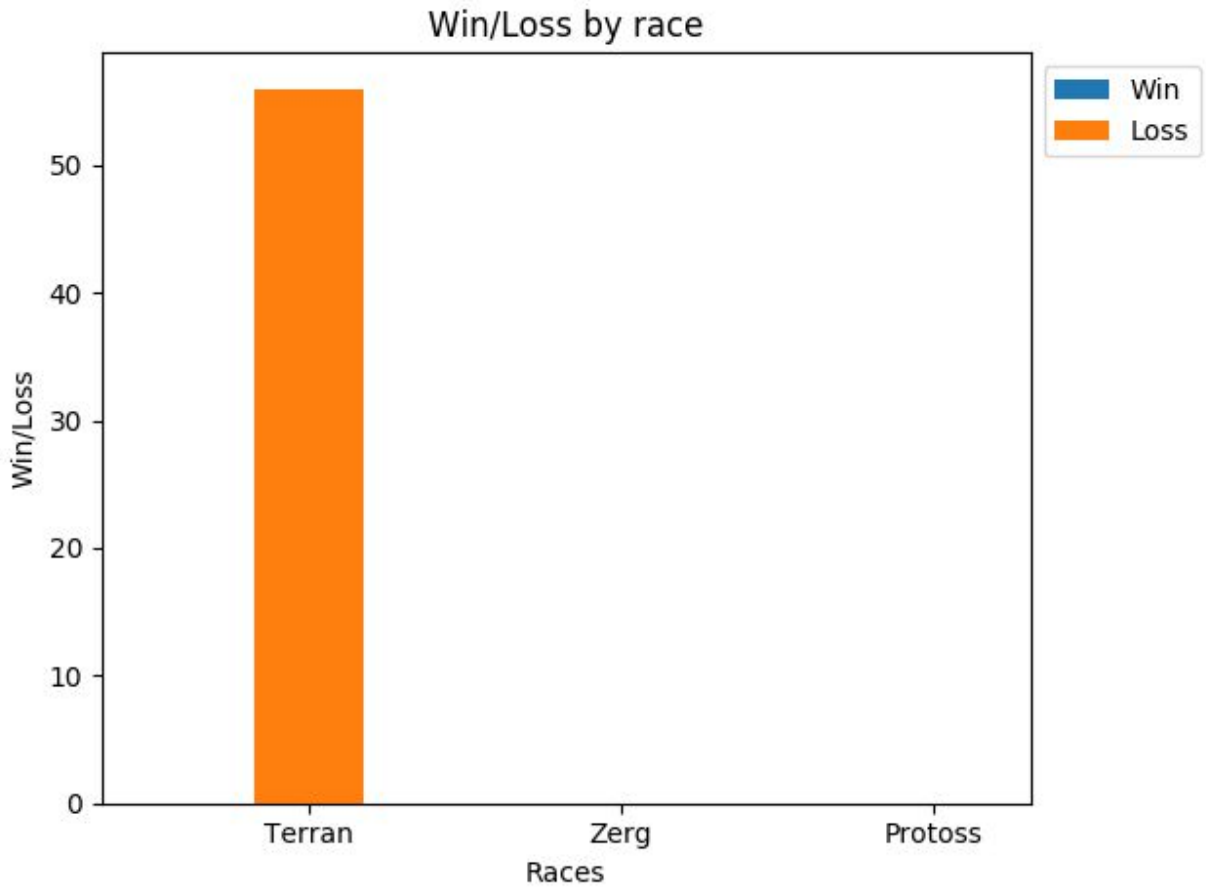
Agent Frequency:



Strategy Frequency:



Wins and Losses:



From this, we found that the bot heavily favors the mutalisk build, and the heavy attack strategy. Similar trends occurred when training the bot against other races as well. Given more time, we would improve on having the bot not stick with one strategy and build all the time.

This issue of the bot deciding to stick to one build was most likely also impacted by both the nature of the game of SC2 and also the way that our build orders worked together. SC2 does not usually favor switching strategies completely mid-way through the game due to the necessity of building additional buildings and upgrades to support the new units. Our build orders also needed to be synchronized to work together better, in terms of smoothing out the transitioning between each build and making sure we didn't build repeats of the same units or buildings. Overall, these issues made it so that the bot seems to have favored a simpler build order and strategy over something more complex like roach/hydralisk or zergling/baneling.

Plan for Deliverables

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

Github repo
Github pages

Separation of Tasks for Team

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

Out separation of tasks mostly stayed the same, but once we decided to split strategies into build orders and strategy, Davey began working on the strategies, while Anjo and Josh worked on the build orders. Thomas worked on creating the graph data instead of Josh, due to the graph data fitting with Thomas's previous experience better. Matthew ended up creating the fitness calculation while Thomas implemented when the fitness changed by -10% it would penalize the bot for selecting a bad strategy. Davey ended up taking over the scouting portion in return.

Personal Contributions

Matthew:

I worked on establishing the inputs for both the neural network and fitness. The neural network input is defined as all our units, what is currently known about the enemy, worker breakdown (minerals, vespene, idle, other) and amount of resources we have. We did not include creep since it muddles the performance against Zerg since the game considers all creep as creep and not selectively to one player or the other. Fitness was implemented in a similar fashion; however, the unit breakdowns were slightly different since the fitness was used as a quick performance check in midgame to determine if we would need to alter our performance. I also helped debug a lot of miscellaneous issues with compatibility between the scripted strategies and agent_selector as well as went through code and cleaned up deprecated code.

I had a hard time determining how to weigh army units for each race because I do not play the game. Therefore, I do not know the differences between each of the individual units. I was able to break down the buildings a little bit better since the buildings had cleaner divisions between uses. The fitness was then defined as the difference between our total unit break down and what was known about the enemy at that state.

Joshua:

I nearly completely only worked on the SafeRoachAgent, which is an economy-focused safe build that expands to three hatcheries early-ish and creates roaches until about 60-70 supply and then works on hydralisks. Most of my work was put into figuring out some of the logic for my agent that made it more than a simple "follow the build order and rush the enemy" bot.

To that end, I had my agent do a couple of "smart" things. On the simpler side, I had it do things like keep the workers mining, only harvest vespene when the bot was running low on vespene, have it mine vespene for a crucial point in the build order, and constantly keep the workers distributed evenly between the bases.

The creation of the ability for the agent to have queens put down a set amount of creep tumors (and inject larvae constantly after doing so), have the creep tumors self-spread, and have the queen replenish the creep tumors when they were unable to spread further due to the destruction of the tumors at the leading edge (aka destruction of the tumors with the ability to create another creep tumor) was stymied slightly by the obtuse unit and ability IDs. However, I managed to get that working eventually, with the end result of the agent spreading a carpet of creep tumors that would eventually arrive at the enemy base while still covering a good portion of the map and would be replenished to start anew if they were cleared.

For the creation of units, such as drones and the army, I added in logic that had the bot replenish a set amount of workers and constantly have a set amount of roaches that were replenished if depleted. This ensured that the bot would always have workers mining at the efficiency dictated by the number of expansions and not sacrifice economy for army and vice-versa. The agent also replenishes queens if they're destroyed, allowing for the continuation of larvae injection. There is also some logic in there for

when it would be best to build units based upon existing numbers of units, and what the maximum number of drones should be based upon expansion count and a hard limit.

The agent also has some logic for when it would be best for it to build additional structures, such as a hydralisk den upon reaching a certain supply and excess minerals, or an evolution chamber to make further upgrades if there are excess minerals. The agent currently also takes into account if there are excess drones and excess minerals, then builds spore crawlers and spine crawlers with those units in order to free up supply and also improve upon static defenses. I would like to improve on this in the future by having the upgrades be based upon the number of units (that will actually use the upgrades) that exist or will exist, and also have some code for the replacement of buildings if important structures that allow for the creation of the units are destroyed.

The agent also self-expands and attempts to maintain a base/expansion count of three, which is approximately the most efficient number of expansions to have for most of the game. In the future, I'd like to be able to determine the difference between hatcheries/bases and expansions by looking at their distance from mineral fields, as building additional hatcheries for the supply and larvae production and blocking off enemy expansions with in-progress regenerating hatcheries (that are canceled before death) are legitimate strategies. The bot also allows for the creation of around 6 total expansions once excess minerals are available, as that is approximately the maximum number of expansions one should have for the late game, as the maximum drones that one should have in the late game should be around 70-80 (which is also the hard limit for the agent's drone count) or so in order to avoid taking up precious supply that could be used for a large army.

Overall, SafeRoachAgent can hold up in the late game for a decent amount of time and even win against hard bots due to its ability to constantly create units at a very fast rate and it's very general army composition of roach/hydralisks that can attack both air and ground units. However, I'd like to improve upon this in the future by going through with building all of the upgrades for all the units possible and having it be a little more discerning about things such as bases versus expansions. There are also a lot of potential work to be done on this bot too, in terms of transitioning to building lurkers and ravagers with the roach/hydralisks for a better late game and adding in some vipers.

For the scripted build order agents overall, I'd like to have had more time to add in simpler "add-on" build orders that allow the bot to switch more effectively for adapting to new circumstances with new buildings and units. For example, perhaps adding a script that would tech toward ultralisks and broodlords in the late-game for superior ground combat, or adding in scripts that would actually be able to create and use vipers to counter things such as siege tanks, which are kind of the bane of our bot right now versus terran.

Davey:

I set up the repository and worked on created the base LoserAgent that all other agents derive from. This is so that all agents have access to a set of utility functions to make creating a bot easier. This class also contains the strategies that all other agents use. The 4 main strategies are attack, defend, harass, and scout. Attack has varying levels that allow you to group a certain pentange of your army together before advancing to the enemy. Defend returns all army units to the main base, build spine crawlers, and builds spore crawlers. Scout sends a percentage of army units to random locations on the map. Harass attacks with army units and returns a unit to base if it gets below a certain percentage of health. Harass also has mutalisks fly in from the side of the enemy base. I also forked Dentosal's

python-sc2 repo and integrated pysc2's viewer with it, so that we could easily view what was going on in the game on a linux machine and be able to affect the game while viewing it.

Anjo:

I wrote two of the agents used, MutaliskAgent and ZerglingBanelingRushAgent. These two agents were essentially following different build orders found online, one focusing on Mutalisks with Zergling support, and the other building a hybrid rush with Zerglings and Banelings. The agents expand to a second base early on, creates Queens to inject larva, and purchase attack and defense upgrades for all units when available. I had hoped that the bot would learn to switch between mutalisks/banelings as needed, but it seems as is that it just picks a single strategy and runs with it. One main issue I never ended up solving was the fact that I couldn't find a way to have the agents share the variables found in the init() functions. When trying to only initialize variables in LoserAgent (the base class), the other Agents files were not able to recognize them when called. This is likely the cause behind the bot's aversion to switching strategies, as it thinks that it does not have any buildings built.

Thomas:

I worked on setting up functions for the fitness and other bug fixes. One of the larger bug fixes I made was proper game exiting which before would hang indefinitely. This is done by implementing an interrupt function that listens when the user presses Ctrl-C which then it will exit the looping game session. I also got the game to use optional terminal commands which includes the type of race, type of difficulty, and the number of games. The type of race can be randomized which sets a new random race to be played against on every new game match within the game session. I implemented the graphs for the overall fitness, agent, and strategy. Each is categorized by the results from every game or by race. Furthermore there are graphs for each individual game for more fine details. The total agent graph and strategy graphs have overlapping bar graphs to compare their frequency usage while the total fitness graph compares all the fitness performance from every game.

Alex:

I created the base agent selector which facilitated switching between agents. I also determined how agents were going to share data, since python-sc2 only actually gave game/unit data to the agent that you started the game with. I solved that by making a single global main agent that all other agents would reference whenever they tried to make changes or decisions in the game. I then created the basic Neural Network class that implemented a Keras neural network. This class had sufficient flexibility to allow simple set up of input/output size, depth and width of the hidden nodes. I also set up a simple save/load system that could save and load the weights of the neural network. I then partially implemented the neural network in the agent selector, setting up most of what was needed to train the Neural Network, and predict agents and strategies.

References

Open AI's RL algorithms

https://github.com/openai/baselines

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

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

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

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

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