StarCraft 2 Neural Network Trained With Evolutionary Algorithm

An AI Learns How To Macro

Code found at: github.com/Turtdle/Starcraft2AI

A Brief Introduction

This project was an adaptation of a previous project I have been working on. I have been playing SC2 (StarCraft2) for quite some time and have developed a fascination with creating a bot that plays SC2. I have previously developed bots in the past that have been hard-wired to respond to stimuli from the game in exact ways. These bots have had limited success, and after taking COMP 151, I started thinking about how I could incorporate the methods we learned in COMP 151 into a SC bot.

StarCraft is a real-time strategy game where you start out the game on a planet with minerals scattered around. You have a starting base equipped with 12 miners. You use the money the miners gather to create more workers, bases, and eventually combat units. The goal of the game is to destroy all the enemy buildings with your combat units.

For this project, due to the limited time, the goal of the project is to make a bot to automate the economical part of SC2 (making bases and making workers). The bot is fed the following information: The in-game time, the amount of workers we have, how much supply we have, our supply maximum, and how much money we have. The bot can use this information which is fed to its neural network to make 3 decisions. These decisions are: Make a base, make workers, and make a supply depot.

After the neural network is created, we then test 10 random neural networks by letting them play the game for 4 minutes, and then tallying how they did after the game. Depending on how each neural network performs, we then decide which neural networks can make it to the next generation and breed by combining their attributes to create more neural networks. The better the neural network performs, the higher the chance it has of passing on its traits to the next generation.

To make this project possible, several libraries were necessary. The most important library was the SC2 library for Python. Tensorflow was very important for the neural network component. I also used Numpy and random for their utility. Special thanks to Sentdex's AI Learns to Play SC2 video.

- GitHub - BurnySc2/python-sc2: A StarCraft II bot api client library for Python 3
- TensorFlow
- NumPy
- A. I. Learns to Play Starcraft 2 (Reinforcement Learning)

Deep Dive On the Code

The first step is creating a bot to interact with SC. The bot has 3 main functions found in Bot.py:

```
async def build_supply_depot(self):
async def build_worker(self):
async def build_base(self):
```

The bot then has 3 helper functions to automatically handle tasks:

```
async def sature_base(self):
async def return_idle_workers(self):
async def lower_all_depots(self):
```

Saturate bases is to manage the workers. In SC, each mineral field can only simultaneously be mined by 2 workers at once. Each base has 8 mineral fields, so if there are more than 16 workers at one base, we will be mining inefficiently. Saturate bases automatically check if each base has 16 workers, and if it doesn't, it searches to see if there is any base with less than 16 workers, and if it finds any, it sends the worker to the new base. Return idle workers is used to return any idle workers back to mining. After building a structure, the worker who builds the structure is idle after building the structure and doesn't do anything until it is told to. This function continuously checks if there are any idle workers, and if there are, it sends each idle worker back to mining. Lower all depots is used to lower every supply depot that we own. In SC, units cannot usually walk through buildings, but supply depots are special. You can lower a supply depot into the ground which allows any unit to pass over the supply depot. This function is necessary because if a worker makes a supply depot that is in the way of another worker, this will be very disruptive.

In the game tick function, we first check our prediction using our neural network with

```
prediction = self.net.predict(self.get_inputs())
```

Get inputs is a helper function that gathers the inputs mentioned earlier in an array for the neural net to use.

After this, we check the output and then do the function above according to the output. Then, we calculate the reward with:

```
self.reward["money"] = self.minerals
self.reward["supply"] = self.supply_used
self.reward["income_rate"] = self.calculate_income()
```

Then, we use our helper functions:

```
await self.return_idle_workers()
await self.lower_all_depots()
await self.sature_base()
```

After the bot is finished running, we save the neural network along with its score as a file to access later.

After running each bot in the epoch, we move on to the breeding phase. We have 4 functions in Breeder.py:

```
def breed(model1 : NeuralNet, model2 : NeuralNet):
def mutate(neural : NeuralNet, mutation_rate=0.05):
def get_best_networks(epoch):
def breed_networks(epoch):
```

Breed combines two networks into a hopefully better version of the two. Mutate has a chance to randomize the weight of a neural net. The get best networks function gets the top 40 networks. The breed network takes the top 40 networks and breeds them randomly. Then it takes one network from the top 40 and 1 network from the bottom 60 and breeds those 60 times to create 100 new networks. After this, we save each network and then move back to the testing phase to generate more data on how each network performed.

How to run

Running this program is quite complicated as it requires many different components. First, we need SC2, which can be downloaded from the Battle.net client. Then you need to install the map the bot plays on, which is “AcropolisLE.” This can be downloaded from github.com/Blizzard/s2client-proto. After dragging this map to the maps folder in your StarCraftII directory, we need to install our Python dependencies. These can be installed with pip install <dependency>. These dependencies are:

- sc2
- numpy
- keras
- tensorflow

After this, you would have to run CreatePopulation.py to create our random initial 100 neural nets. After it does this, you can run Main.py. After you run Main.py, you can run CompileData.py to find out which one of your nets performed the best. Out of all my testing, this is the network that did the best for me. In the video, the red player is the bot and the blue player is a bot that does nothing. At the end of the video, the bot has about 2.5x the income of the player who has done nothing. The video can be found at this link: https://www.youtube.com/watch?v=ydgX7ikJlQE

My neural network that performed in this video can be found at this link: Copy of MakeMoney35.h5

You can run this neural network by using the Python function in Bot.py:

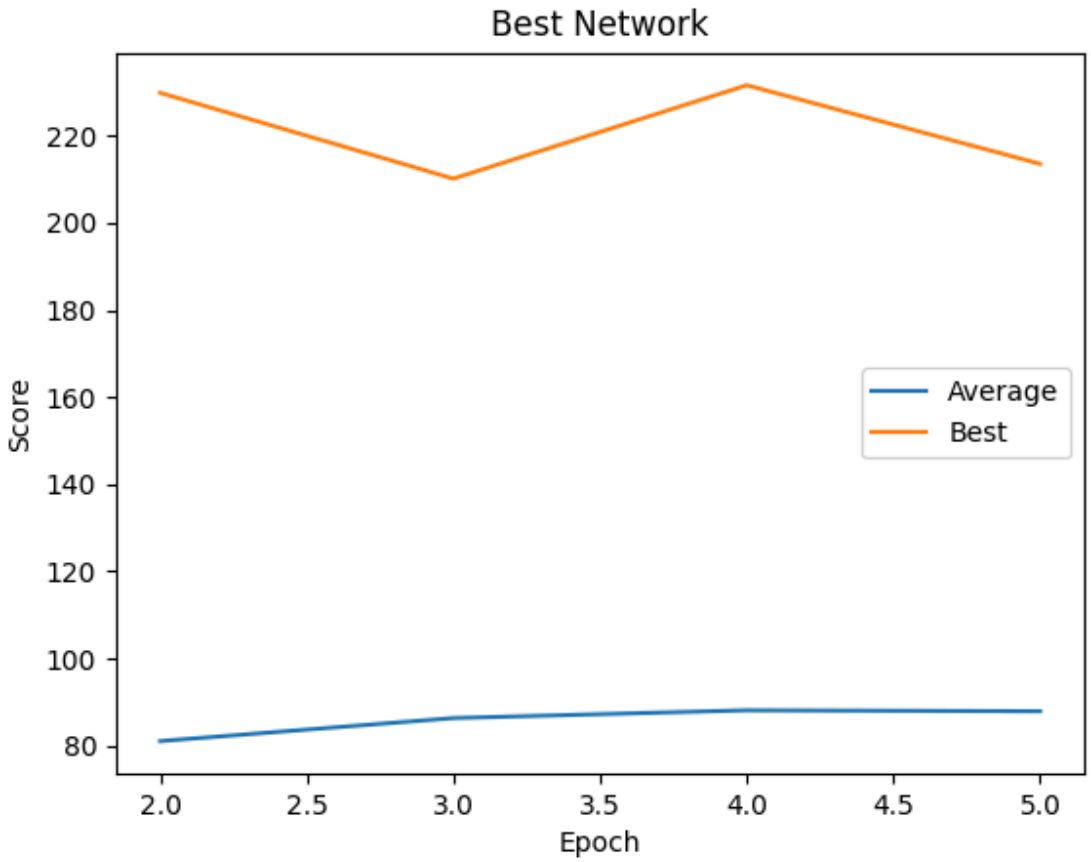
```
def run_bot_from_file(Filename : str, id : int, epoch : str):
```

Id and epoch are used to tell the bot how to save the file in this line:

```
save_directory = os.path.join(script_directory, 'saves' + epoch)
with open(os.path.join(save_directory, "MakeMoney" + str(id) + ".txt"),
"w") as f:
        f.write(str(reward))
```

Results and Conclusion

After 400 4-minute games of SC2, this are my bots performance:



As you can see, the average slowly grows while the best network remains erratic. I believe that if I could run the algorithm through 50+ generations, the bot would be able to optimize the game and possibly gain more income than a human player like I would be able to do.

I definitely believe that the breeding algorithm and the neural network could be better, but as this has been my first time using TensorFlow, it took a lot of experimenting to get a neural network to even work properly. If I had more time, I would have definitely liked to improve this project. For the future, my idea was to train a neural network to perform tasks for each of your units. For example, if I told my soldier to attack, he would query his own neural network to decide where to attack, how long to attack, and when to retreat.