Run AwAI: StarCraftII Adversary Avoidance Using Neural Networks and Genetic Algorithms

Kyle R. Chickering **krchicke@ucdavis.edu**
Alex R. Mirov **armirov@ucdavis.edu**
Simon H. Wu **simwu@ucdavis.edu**
Xin Jin **jin@ucdavis.edu**

June 8, 2018

Abstract:
This project explores the integration of a neural network and an evolutionary genetic algorithm with the Star Craft II API. The goal of this project was to successfully control a StarCraft II agent via a neural network, and maximize its fitness through a genetic evolutionary algorithm. The agent uses the PySC2 API to connect with StarCraft II, and a neural network to control its actions. The fitness function is defined as the survival duration for an agent in a sandbox environment with hostile units, where surviving for longer is better. The action space of the agent was limited to movement in order to achieve a realistic scope for the project. The fitness of an individual is evaluated by running multiple simulation episodes and averaging the performance (survival time) for each episode. The neural network uses an explicit topologically unrestricted implementation. This implementation provides high degrees of freedom while minimizing the network size. The genetic algorithm uses tournament selection to select the most fit neural network (individuals) to breed. Each successive generation of neural networks is slightly better at surviving in the hostile sandbox environment. After 50 iterative generations, our resulting neural network had increased the average survival time of an agent by 288%.

Keywords: Machine Learning, Genetic Algorithms, Neural Networks, StarCraftII

1 Introduction

Since AlphaGo's defeat of the Go champion Ke Jie, the artificial intelligence community has been looking for a new problem domain to apply learning techniques to. A surprising candidate for artificial intelligence testing has emerged in the form of real time strategy games. Real time strategy games work by having a player make critical decisions in real time against an enemy player. The game StarCraftII, released by Blizzard entertainment in 2010, is a real time strategy game that has captured the interest of the artificial intelligence community. The game presents several challenges for artificial intelligence, the biggest of which is the magnitude of possible game states. The game technically has a finite number of game states, but the granularity of those states renders the game practically continuous.

We present a method of teaching an agent to avoid enemy players for as long as possibly by using genetic algorithms to evolve neural networks (neuroevolution). We interface our neural network agents using the PySC2 library and evolve an agent that shows a measurable increase in its ability to run away from adversarial opponents.

2 Background

This project combines two proven artificial intelligence paradigms: neural networks and genetic algorithms. These methods are traditionally orthogonal, but there has been a growing body of work in recent years with regards to evolving neural networks using genetic algorithms, most notably [6].

2.1 Neural Networks

Neural networks are a method of computing any computable function by using a network of nodes called **neurons** connected by a series of edges. This representation is inspired by the human brain, and has been proven over and over as a heavy hitter in the world of machine learning. These networks are differentiable, and by calculating error, they can be trained using a method called backpropogation that changes the strength of the connections between neurons. The backpropogation method is the status quo of training neural networks, however, alternative methods exist, such as neuroevolution. Neuroevolution is an efficient alternative, especially in spaces with continuous state spaces.

There are two main ways to represent neural networks in software, implicitly and explicitly. Implicit neural networks use linear algebra and a consistent layered structure to propagate inputs forward (called feeding forward). With numerical linear algebra libraries available for almost every major programming language, this method of representing neural networks allows for incredibly fast processing of neural networks. However, this representation's greatest strength, its speed, is the source of its greatest weakness. Implicitly defined neural networks suffer from the inflexibility of their layered structure. In the human brain, neurons are not limited to any specific structure, and are free to make connections that could not be possible in a layered model.

This leads us to explicitly defined neural networks. These networks allow the specification of new nodes that are disconnected from the traditional layered model. While this can be advantageous, it can also present problems for the implementer. Because these models are represented by graphs, we can no longer use linear algebra libraries to feed forward through the network. This causes an increase (often significant) in running time. For small networks this is not an issue, but for larger network this becomes (sometimes prohibitively) problematic.

2.2 Genetic Algorithms

In a similar line of though that led researchers to build neural networks by looking to nature for inspiration, genetic algorithms exploit natural selection to create optimal solutions to problems. Research into genetic algorithms started when researchers realized that there is nothing inherent in evolution that limits the process to nature [1]. In fact, by thinking of evolution as an algorithm in and of itself, we can extend the principles of evolution to digital systems. This is commonly done by defining what an "individual" and "fitness function" mean in the digital evolution landscape. The algorithm proceeds as evolution does in nature by picking the most well adapted individuals to "breed", and evaluating their "offspring" on the same problem. This technique has yielded several fascinating results, and often the algorithm generates novel solutions to difficult problems [4].

3 Methodology

Our approach involves combining genetic algorithms and neural networks to create an agent that learns through evolution how to evade antagonistic agents. We chose this method as the problem space closely resembles nature's predator/prey motif, and with such a large state space, neuroevolution is actually an efficient solution [6]. By modeling our agent as an evolving neural network we can get results that mimic generational evolution in nature. While evasion, especially in the StarCraftII space, is a fairly simple behavior, we chose this method as it is easily extendable to larger problem spaces by extending the input and output dimensions.

There are many techniques to accomplish running away from enemies in StarCraftII, for example, a simple expert system would be quite good at enemy avoidance, however we were concerned with the scalability of our approach. We chose the neural network and genetic algorithm approach because for dealing with more complex tasks, our approach is more robust than a simple expert system. Additionally, we could train various neural networks on different tasks (avoidance, attacking, resource mining, etc.) and splice the neural networks together to build an AI capable of multitasking during the game.

By introducing scalability, we could use our existing framework to train neural networks in more and more difficult simulated environments, eventually leading to an AI that could play the entire game outside of a sandbox mode. Neural networks generalize very well to a wide array of situations that are similar or close to the situations they were trained on, and they are a perfect tool for attacking a game that requires real time decision making and reasoning over a complex set of inputs.

3.1 Simulations

To compute the fitness for each neural network in the population, we run the network through a simulation. The simulation is run in StarCraftII, and our simulation controller is built on top of the PySCII interface, and simulates a "game" with our agent and a group of adversaries, which are controlled by the in game AI that come with the game.

Each simulation is called an episode, and we ran the networks through 5-10 episodes each so that we could get a representative sample of the amount of time that each neural network was able to survive.

We let the neural network choose the next action from our action space, which consists of a series of movement directions (eight cardinals and a stay command). The game runs a "continuous" time simulation (obviously continuity is limited by the computer's capabilities), but we chose to have our neural network make decisions in finite time intervals. We did this so that we would not fall victim to our network not making decisions fast enough, and chose a time step that would allow the network to make decisions quickly enough to function well.

Our whole system ran as outlined in 3.1. We decoupled the various aspects of the system as outlined as a good architecture practice in [3]. We run a general simulation controller, which interfaces with the neural networks through a module, and after creating and manipulating the networks, sends them to a simulation controller which interfaces with PySC2, which runs the simulations in StarCraftII.

We collect from each simulation the set $\boldsymbol{\mathcal{E}}$ of episodes, which we can use to collect data from the simulation. We also compute the number of steps taken by our agent in each simulation, denoted $\mathcal{S}(e \in \boldsymbol{\mathcal{E}})$. The steps are the number of actions that the agent made during the simulation, and we chose to measure this because it is a good indicator of success of the agent at running away.

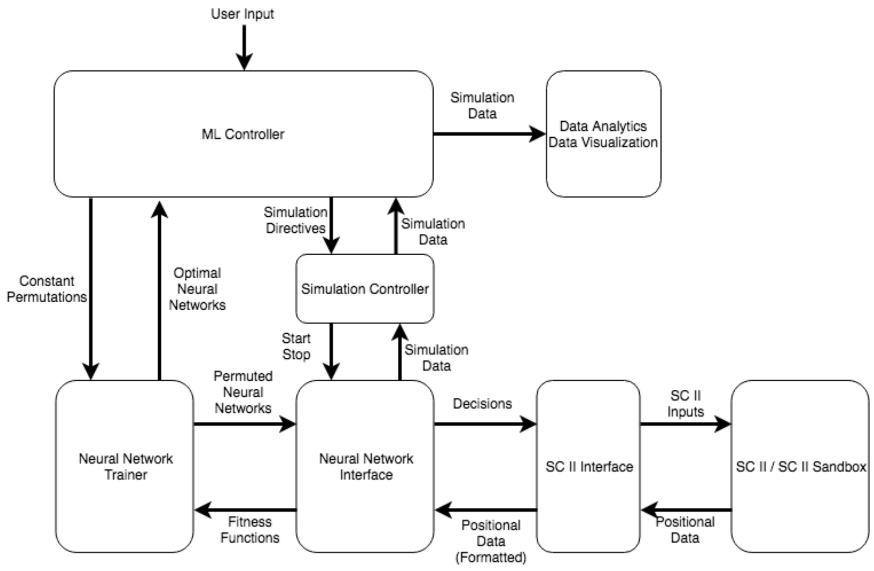

Figure 1: Control and data flow through the Run AwAI controller.

3.2 Neural Network

We started by using an explicit neural network implementation, which we chose because we were curious if the flexible topologies would lead to a more efficient neural network. We were inspired by the NEAT algorithm [6], which has shown promise in other video game domains. However we choose a simpler topologically explicit implementation to simplify and accelerate the development of our algorithm. We ultimately settled on an acyclic graph representation, which allowed us the freedom to add nodes and allowed simple manipulation of connection weights.

We chose to use our neural network to represent a function taking an input vector $\boldsymbol{I}$ consisting of relative agent position and adversary's relative position and return an output vector $\boldsymbol{V}$ of floating point values corresponding to the agent's belief that a particular action would lead to a successful outcome. The neural network is a function $\mathcal{N} : \dim(\boldsymbol{I}) \rightarrow \dim(\boldsymbol{V})$:

$$\boldsymbol{V} = \mathcal{N}(\boldsymbol{I}) \tag{1}$$

We choose the next action according to the simple maximization equation:

$$a_n = \max_{v \in \boldsymbol{V}} v \tag{2}$$

This input vector is flexible, and allows us to tweak the information that we feed to the neural network. Additionally, while the output space we used contains only movement in the cardinal directions, we could easily add other supported actions like attacking and mining resources.

Our neural network uses the standard sigmoid function ς as its activation function:

$$\varsigma(x) = \frac{1}{1 + e^{-x}}$$

We chose this function for a number of reasons, the least of which is its popularity as an activation function in neural networks [5]. Because of the shape of the sigmoid function, inputs to subsequent neurons stay in the nice range of $[-1, 1]$, which has the effect of normalizing the network. Since we are not using backpropogation, the nice properties of the sigmoid function were unimportant to us. However, if we were to do live backpropogation during simulations (See section 5.1), the sigmoid function would be a useful activation function.

When we initialize the network, we populate all of the connection weights from a random Gaussian distribution with $\sigma = 1$ and $\mu = 0$. This serves to make our input data "nice" for the sigmoid function, which ends up returning either 1 or -1 when the input data is amplified by non-gaussian data. This also ensures that most connections are "off" or close to off at the start, which allows the network to learn without any random pre-bias. Our first generation will have very weak suggestions on the effectiveness of the next move. This may seem counter-productive, but it allows us to do much slower mutation changes to the connection weights, which in turn increases the explorative power of our algorithm at the start of the evolutions. If the weights are too heavy in the beginning, the agent will favor a local maximum almost immediately, but with a Gaussian distribution, the algorithm explores more options before settling on a specific maximum.

3.3 Genetic Algorithm

For our implementation, we used a tournament style selection process to choose our most fit individuals. We use tournament selection over other selection methods (like probabilistic selection), to improve convergence speeds. Tournament selection is an exploitive selection method, and we choose exploitive behavior over exploratory behavior because our simulation was a substantial bottleneck in our iterative process. Because we choose an exploitive method, we limit the chances of finding a global maximum, but increase the chances of finding some local maximum, and finding it relatively quickly. This gave us a chance to better observe the convergence of our algorithm, and allowed us to demonstrate clear forward progress.

To get convergence, we spent a lot of time looking at fitness functions. We considered many different possibilities, but finally settled on a measure of the number of steps that our agent took during the simulation, with a penalty for inconsistency. We figured that this would be a good measure of our agent's ability to stay alive, and we thought that time alive was a good indicator of the agent's ability to accomplish the goal of running away. We computed the fitness F using the following formula:

$$F(\boldsymbol{i}) = \frac{1}{|\boldsymbol{\mathcal{E}}|}\sum_{e\in\boldsymbol{\mathcal{E}}}\mathcal{S}(e) - \eta\sqrt{\frac{1}{|\boldsymbol{\mathcal{E}}|}\sum_{e\in\boldsymbol{\mathcal{E}}}\left(\mathcal{S}(e) - \frac{1}{|\boldsymbol{\mathcal{E}}|}\sum_{e\in\boldsymbol{\mathcal{E}}}\mathcal{S}(e)\right)^2} \tag{3}$$

where $\boldsymbol{\mathcal{E}}$ is the set of all episodes run by the PySC2 simulation, $\mathcal{S}(e)$ is the number of steps taken by the agent in the episode, and $\boldsymbol{i}$ is a an individual of the population. η is a hyper-parameter we use to fine tune the convergence of our network. We are effectively calculating the average number of steps and penalizing the individuals based on the standard deviation from the mean of the episodes. This penalty serves to mitigate the effects presented by individuals that get "lucky" and have a few good trial runs that offset their average.

We seed the evolutionary controller using n random individuals, and in each evolutionary iteration, we breed nC2 individuals to compose the subsequent generation. After breeding, we randomly mutate some of the children, and then calculate the fitness for each individual in the population. To get the next generation, we choose the most fit n individuals from the population to breed, and repeat the cycle. In addition, we found that the algorithm converged more rapidly if we let the most fit individuals survive to the next generation, and after making this change we noticed a jump in convergence rates.

The genetic algorithm relies on several hyper-parameters. The most obvious is the population size. We find, unsurprisingly, that larger population sizes are more robust and promote faster convergence, this makes intuitive sense because with larger population sizes there is an increased chance of mutation, and mutations are largely responsible for exploratory behavior in learning algorithms. We have three mutation parameters, which are the rate at which nodes are added to the network, the rate at which new connections are added to the network, and the rate at which connection weights are changed.

We mutate connection rates based on a Gaussian distribution, which, combined with our Gaussian weight initialization, ensures that weights never change dramatically. We adjust weights according to the formula:

$$w_i \leftarrow w_i + \boldsymbol{\sigma} \tag{4}$$

where $\boldsymbol{\sigma}$ is a random variable from a Gaussian distribution. We found that changing the weights according to a Gaussian distribution promoted faster convergence than using weights taken from a uniform distribution.

3.4 Controller

The controller was the interface we used to communicate with StarCraftII and run the simulations need to commute the fitness function. Our controller has a function to step the game simulation, which is called several times per second. We implemented all of our agent's logic, consisting of feeding forward the inputs through our neural network, inside of this step function.

We spent some time deciding what inputs to send to our neural network during the simulation. Originally, we sent absolute coordinates, but we found that these coordinates made the network into essentially a binary decision network. Instead, we settled on giving the neural networks relative coordinates produced much better results. We also give the neural network the relative coordinates of the enemy units. The controller queries the PySC2 interface for these values every time we take a step in the simulation, and the values are pulled out of StarCraftII itself.

One problem we encountered that we had to correct was determining when units arrived at their target destination. We solved this by comparing the target destination and the unit's current location every time we step the simulation. Because the mini game uses multiple units (both friendly and hostile), the unit location coordinates are averaged over all selected units locations. This means that often units will never actually arrive at their precise destination, just get very close to it. To mitigate this issue, we decided to consider units within a certain radius of their target destination as at their destination. Once this condition is met, we could ask the simulation whether or not units had reached their destination. Once the units reach their destination, we can choose a new destination for them to move to.

The controller also handles the interpretation of the data from the PySC2 interface, and acts as a "human" playing the game. The controller is responsible for selecting movements, making decisions, and executing those decisions.

When the controller receives an arrived at target status, a new target destination is set. To generate a new target destination, the relevant aspects of the current state (current location of friendly and enemy units) are captured and fed into the neural network. The neural network returns a vector of the action space and a value between -1 and 1 for each action label key. The action with the highest number will be chosen, and then taken.

Our action space is movements that the friendly units can take. We decided to quantize the action space into discrete movements, as opposed to a continuous x-y coordinate plane. By only allowing our units to move in the 4 cardinal directions, diagonals, or stay put, we hoped our first implementation of the neural network and genetic algorithm would have an easier time converging.

To evaluate the fitness of the neural network, we count the number of steps that the units survived for. One problem we encountered and solved, was the initial series of steps taken before engaging the enemy units. For example, the DefeatRoaches map spawns the enemy and friendly units on opposite sides of the map, in a random selection of several predefined start locations. However, the distance between these start locations can vary between simulation episode. This means our neural network fitness (number of steps survived) could be slightly affected by an irrelevant condition. To mitigate this issue, we subtract the number of steps it takes until the friendly units reach their first destination (the location of the enemy units) and start their engagement. By only counting steps after enemy engagement has begun, our fitness function is more reflective of survival time.

In addition, the controller is responsible for calculating $\mathcal{S}(e)$ for each episode (simulation) that it runs, and returning the value to our genetic algorithm. This allows us to make modifications to our fitness function without having to make changes to the genetic algorithm. This also gives us the flexibility to extend the capabilities of the controller to run different style simulations in the future. For example we could run the agent through multiple types of maps, and compute the average fitness between multiple different episodes and maps.

The average fitness for the individual is then saved in a Python pickle file, along with the fitness of the top (most fit) individual. Because the Pysc2 simulation is invoked from the genetic algorithm code, the algorithm can then unpickle the file. After all individuals in a generation are tested and their fitnesses collected, the genetic algorithm can select the most fit individuals (neural networks) for breeding.

3.5 PySC2 Interface

Our simulation learning environment was based on an existing environment included in the PySC2 library[2]. We use the "Defeat Roaches" mini game as the baseline for our learning. Once we had the map, we used the StarCraftII map editor to tweak the mini game to better suit our needs. We created three maps that were used in testing. The default map is a simple square map which spawns a group of adversarial "roaches", and a group of "marines", who are controlled by our agent. The marines and roaches are spawned in the same place every time, which gives us a consistent platform to test our agent.

The second variation is similar to the first, except that there is only one "roach" and one "marine". This map type highlights the capabilities of our agent in a more intimate setting. The group of marines has the advantage of being able to get "lucky" and have some members survive, but the one-on-one variation forces the neural network to work well from the very beginning. With the one-on-one map, we were hoping to highlight more explicitly the capabilities of our trained neural networks.

The third map variation was a larger sized version of the original map, which we made in the hopes that we would see different convergence rates and survival strategies on a larger game space. We were hoping to see the agent take advantage of the larger space, but unfortunately did not have time to run extensive simulations on this map.

All three maps have a slight bit of non-determinism built in, which means subsequent simulations using the same neural network will not be identical. This gave us the opportunity to gather more realistic data about the performance of our agent, since we could accumulate an average over a number or trials.

4 Results

We ran several simulations with various choices of hyper-parameters, and our results were promising. We saw measurable improvement in performance after just a few iterations of the algorithm, which gave us a steep initial learning curve before the improvements started to taper off.

We see in Figure 4 an artifact from our choice of genetic algorithm. Because we keep the top individual from each generation in the gene pool, we see that the top individual has plateaued early, and no new individual is achieving a higher fitness. We see that the average is continually increasing though, which will increase the chances that an individual will eventually be bred that has a higher fitness than the individual that is dominating the maximum fitness.

We can also see that the average sometimes plateaus as well. This is due to a homogenization of the gene pool, where the children individuals are very similar to their parents. Because of low muta-

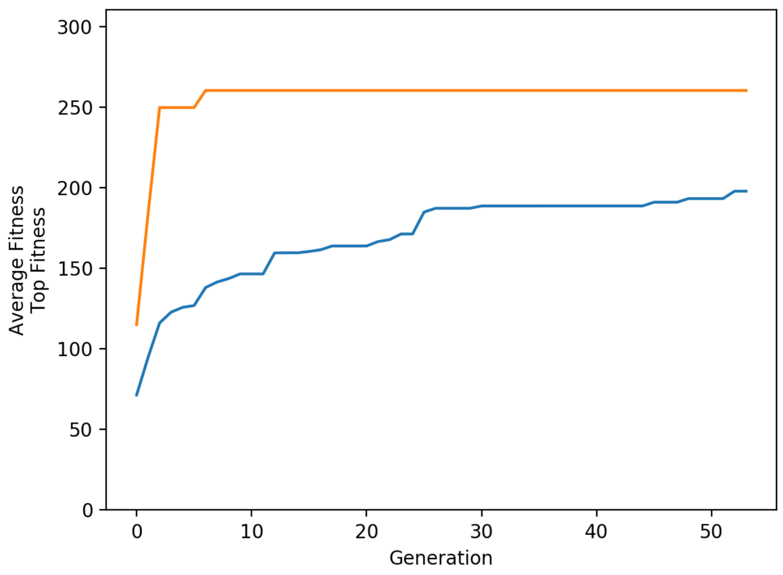

Figure 2: Graph of average fitness (blue) and top fitness (orange) for each generation in Trial 1. Top fitness is the fitness of the most fit individual at the end of each generation's simulations. Average fitness is calculated by averaging (3) over all individuals $\boldsymbol{i}$ in the population. The breeding constant is set to 15, node generation rate is set to 0.3, new connections at 0.2 and non structural mutations at a rate of 0.4

tion rates in this trial, we got "unlucky" and were not creating enough mutations to be exploratory, this is a case of finding a local maximum. Since we favored an exploitive algorithm, this is to be expected, however we see that the population can still achieve higher fitness measures which means that the algorithm still retains some amount of exploratory behavior.

In our second trial, we see that the average fitness increases much more regularly. However, despite having more generations, trial two did not reach the same average fitness or top fitness as trial one did. One possible reason for this behavior is that our mutation rates are higher, so our algorithm favored a more exploratory approach, leading to a more consistent increase in fitness. This exploratory behavior also accounts for the top fitness being lower than in trial one.

We did not have enough time to run extensive testing over a representative sample of hyper parameters, and expect to get better convergence by tweaking the parameters some more. We chose the parameters we did so that we could see convergence in a timely manner. Since our simulation times were the bottleneck in our pipeline, we choose a relatively small population size to run through our algorithm. We think that using higher population sizes would also lead to better convergence, as the networks would be put through more and more simulations, generating a more and more accurate picture of their fitness.

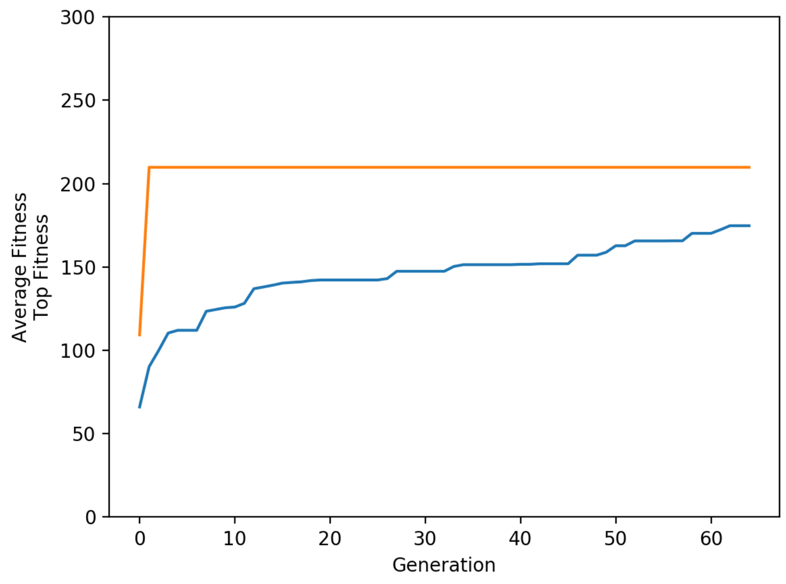

Figure 3: Graph of average fitness (blue) and top fitness (orange) for each generation in Trial 1. Top fitness is the fitness of the most fit individual at the end of each generation's simulations. Average fitness is calculated by averaging (3) over all individuals $\boldsymbol{i}$ in the population. The breeding constant is set to 15, node generation rate is set to 0.5, new connections at 0.5 and non structural mutations at a rate of 0.75

5 Conclusion

Based on the convergence we got during our trials, our method of training a neural network to run away is a feasable approach to the problem. Although we arguably did not achieve "optimal" results, given more time we are confident that we could significantly improve the convergence of our algorithm.

We have demonstrated that evolving neural networks using a genetic algorithm is a good approach to teaching agents to accomplish goals in countinuous or near countinuous state spaces.

5.1 Improvements

One of the foremost improvements we could do is to train the neural networks during the simulation as well as training them through evolution. In addition to improving the convergence, this strategy would mimic nature even more. Organisms learn during their lifetimes, and there is speculation that the knowledge obtained by an individual during its lifetime can be passed down through the genome.

We could accomplish this by using backpropogation during the simulations, and considering good measures of progress that we could calculate during the game. As an example, we could compute the Euclidean distance between the agent and the adversary and make decisions that minimize this value. The drawback to this method is that it may encourage sub-optimal behavior by biasing the agent towards known behavior. It is common for human experimenters to expect sub-optimal behavior because optimal behavior is so far unknown [4].

5.2 Future Work

This project can certainly be extended in many ways, for example, we could extend the output vectors to include offensive maneuvering by our agent, and train the agent to be aggressive by considering the number of adversaries killed by our agent during the simulation.

In the future, we could expand the output action space to a continuous x and y coordinate plane, as opposed to discrete directions. Although a larger action space can often require more iterations of the genetic algorithm to converge, the benefits would be more nuanced agent behavior, and more realistic agent movements. As it stands now, our agent moves in what may be considered a very unnatural way, and while this does not affect the agent's ability to preform, it could be seen as a disadvantage in certain situations where the fact that the game is being played by an AI may be undesirable.

We have also talked about putting more attention towards our fitness function. An interesting possibility is to take into account the number of episodes in $\boldsymbol{\mathcal{E}}$, and rewarding agents that have more simulations. We can then associate individuals with their simulation meta-data, and save time running simulations. We can start with a low number of simulations per individual, and as the generations progress, increase the number of simulations per individual. This would have the effect of starting with relatively low fitness, but increasing the "accuracy" of the fitness over time.

Another possibility that we did not explore, was the effect of removing neurons from the network. Neural networks can often be plagued by overfitting input data, and by removing neurons we hypothesize that any overfitting we encounter could be mitigated. We would remove these neurons in the same way that we add neurons, that is, non-deterministically as a possible mutation. In the study of biological creatures, we often see either total dropping of traits from the population genome or the transition to vestigial genetic structure [4]. In keeping with this, instead of outright removing the neuron, we could significantly decrease the strength of the connections to it, essentially removing it from the network. If the neuron in question was indeed problematic, we would see it continue to evolve very low weights, essentially pruning the neuron and rendering it vestigial.

We could also track the lifespan of certain individuals and kill them off after a certain number of generations. We found that the algorithm would sometimes plateau with a very fit individual for a period of time, and it would be interesting to see what effects a notion of "lifetime" would have on the algorithm as a whole.

References

[1] Kenneth A. De Jong. *Evolutionary Computation: A Unified Approach.* MIT Press, Feb 3, 2006.

[2] Google DeepMind pysc2. **https://github.com/deepmind/pysc2**.

[3] Andrew Hunt, David Thomas. *The Pragmatic Programmer: From Journeyman to Master* Addison-Wesley (Reading, Mass., 2000)

[4] Joel Lehman et al. *The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities.* **https://arxiv.org/pdf/1803.03453.pdf**.

[5] Stuart Russell, Peter Norvig. *Artificial Intelligence: A Modern Approach.* Upper Saddle River (New Jersey, 1995).

[6] Kenneth O. Stanley, Risto Miikkulainen. *Evolving Neural Networks through Augmenting Topologies*. The MIT Press Journals.