# CSCI4511W Project: AI for Agar.io using Evolving Neural Networks

Jacob Sampson
samps284@umn.edu

May 5, 2021

**Abstract**

Online multiplayer games offer an opportunity to test AI in dynamic environments, providing a specific challenge for parsing the field due to extraneous visuals obscuring the important information. This paper uses an online multiplayer game Agar.io as a testing ground for the NEAT algorithm which includes evolving arbitrary neural networks to control an autonomous agent. This study uses image processing techniques to identify circular agents in a manner highly independent of the environment, allowing its application in live servers and local clones. The paper concludes with a comparison of the developed AI with agents using the same input vector, parsed from the circle-identification algorithm, as basic decision trees.

## 1 Introduction

The following paper describes an AI for a single agent competing in an environment emulating the online game Agar.io. Agar.io was released in mid-2015 and found widespread popularity on web browsers and mobile devices. The game is a competitive environment where circular agents ingest food particles to grow in size. Larger agents move more slowly but can ingest other agents who are smaller in radius. Agents are controlled by a mouse pointed in the direction of intended travel.

The full game includes bombs, which destroy a player if they are larger, splitting of agents into two, and shooting mass to decrease size. We ignore these additional features with the hope of developing an agent that can effectively use food particles and compete against agents to increase mass. To reach and maintain a maximum size, an AI must balance searching for food with ingesting smaller agents, all while avoiding larger agents

My solution will consider a few approaches to developing AI in dynamic environments with an eventual solution built using Python and trained against other instances of the AI in a controlled server instance [1].

I chose to explore this project due to the intuitive mental application of an AI to agents in Agar.io. One can imagine how a simple circular agent could be taught to respond to food pellets and competitive agents based on ”sight” by moving toward food pellets within the agents' vicinity, away from larger agents, and toward smaller agents. I wanted to explore if an agent could develop its rule-set to perform well against agents that were following some of these simple rules. Despite the incredibly simple concept, players display a vast array of movement patterns to trap, outrun, and overpower players. Given sufficient time to train with quality feedback and inputs, an AI should be able to do the same.

## 2 Literature Review

### 2.1 Image Parsing

Without access to internal game logic, we find two common approaches to parsing video game content into features used for determining player actions: grid-based parsing of pixel data and object-based identification of relevant agents and environment components [2].

[3] parses screen captures of games into unique objects by separating agents and environment by gray-scale values. By blurring the image, they identify unique feature points using local maxima. They further extract information on each object by estimating the velocity of each feature using successive screenshots. Assuming nearer objects are more relevant to the player's next action, they sort the features by distance to the player using Euclidean distance. From there, the input is separated into a fixed-length input vector fed into a genetic algorithm with discrete outputs corresponding to movement directions and actions.

Similarly, the object-oriented approach from [4], rather than raw pixel data or a feature grid, allows for a new technique for reinforcement learning that relies on an object embedding network (OEN). The underlying algorithm the authors use was a convolutional neural network (CNN).

Neural networks are a machine learning method that combines weighted input vectors to produce output values–which may be fed into further "layers" of input/output vectors–that are then used to make decisions, such as a continuous output (movement angle) or discrete action (pushing a button).

Convolutional neural networks (CNNs) use multiple layers to derive useful features from complex inputs. The authors separate features into objects (such as the agent, competing agents, and obstacles) using this CNN and transform the dynamic vectors, representing all objects of one type, into a fixed-size input into the neural network. Compared to image- and feature-based grids, the OEN system performed similarly on two of the five and better on three of the five games.

The object-oriented methods from [4] and [3] easily apply to the dynamic nature of the competing agents with environments that affect performance. These agents and environments can be converted to feature vectors that represent the location, size, and type of the objects. However, the inherent weight given to nearby objects that arise from grid-based methods may be lost. Sorting based on location, such as in [3], helps to add value to potentially more relevant features However, these algorithms may misinterpret the playing field due to estimating object locations and/or velocities. Extraneous effects obscuring information about the field and other agents could affect uncontrolled systems.

The other primary approach to parsing images from video games uses the raw pixel data or a grid-based segmentation of the pixel data. Typically, this raw data is fed into a neural network for deciding the output action. CNN's have long been used in image recognition by identifying unique characteristics that differentiate output classes [5]. The network develops segments of the image more relevant to the function being tuned, which is often an output class. The approach can be adapted to video game AI by using the derived layer outputs taken from a video game screen to decide agent actions.

The OEN from [4] extends this by explicitly describing and separating potential agents, but a network using a CNN on the raw pixel data could derive similar patterns and structure through segmenting. An example of an AI agent using this method comes from [6] which parses raw pixel data using a CNN. [7] segments an image of the video game field into a grid, taking the average color value of each cell and using these cells as inputs into a CNN.

## 2.2 Algorithm Approaches

Monte Carlo Tree Search (MCTS) as a reinforcement learning tool applies well to discrete output spaces. By balancing exploring new actions with past successes, agents can develop a complex understanding of the search space, even with limited observability and in stochastic environments [8][9]. Furthermore, MCTS assumes nothing about the nature of the reward, allowing easy adaption to multiple possible objectives, as with [10]. Rather than training AI to optimize a single reward function, [10] treats objectives as a tree of possible rewards for an agent. Given agents that may balance multiple objectives, classic approaches fail to adapt.

Decision trees are used in a wide variety of ways for reinforcement learning; in the study [3], a separate tree determines the activated state for each output. These trees contain nodes representing elementary operations–such as adding subtracting, multiplying, and taking the square root–to convert all input nodes into a single output value. These networks can convert raw input data, from one of the methods above, to determine discrete output actions, such as button-presses or discrete mouse locations.

Reinforcement learning algorithms may learn from scalar rewards rather than label-based classification metrics. Q-Learning is a popular algorithm used with reinforcement learning for video game AI. Q-Learning uses a learning function that estimates future rewards and updates weights based on the actual result. The model selected actions from a set with the change in score (the calculation of which varies depending on the application) used as the reward. [4] uses Q-Learning with their object-oriented approach, padding features with inconsistent length.

Often, inputs are not independent for reinforcement learning. As seen with video games, successful game-play depends on a sequence of actions rather than a single action choice. Q-Learning uses the maximum return expected given *both* the current and prior states to determine the output. Deep Q-Learning extends the Q-Learning concept by sampling prior states and converting these into fixed-size input features [6] that can be used in a multi-layer neural network. This approach allows for learning useful sequences of inputs in highly dynamic environments. The use of a CNN directly enables its use in a wide variety of video games. However, Q-Learning limits the action-space to discrete outputs, making its application limited for certain video games that necessitate continuous outputs.

Often, reinforcement learning methods use a static neural network topology. In [7], the authors judge typical reinforcement learning algorithms, including Q-Learning and Continuous Actor-Critic Learning Automatan (CACLA), another method for continuous network updating with each action. The authors use the online game Agar.io as an environment while introducing a new continuous task choice model called Sampled Policy Gradient (SPG). The proposed task choice model, SPG, uses methods similar to Q-Learning by choosing from a sample of possible actions and updating the weights based on the expected values from other predictions. The evaluation of Q-Learning required a quantized output, forcing a set number of movement angles, while SPG and CACLA output continuous actions. The algorithms performed comparably in an environment without competition but were all out-performed by greedy-playing in a competitive field.

The widely used algorithms for reinforcement learning used in [7] each have benefits and drawbacks. The performance of Q-Learning is a huge draw while the quantized output parameter limits the behavior of the agent. CACLA provides similar performance without this limitation.

In [11], the authors propose a method for the dynamic evolution of both neural network weights and the underlying structure for the network, called NeuroEvolution of Augmenting Topologies (NEAT). NEAT starts with a minimized neural network with a single connection, building new generations through weighted additions to the structure. Mutations are tagged, allowing for several advantages: tags allow cross-over events to merge data in a relevant manner at the point of trait creation, species are naturally formed by analyzing the degree of similarity in the tags that make up a network, and new mutations are allowed time to develop useful weights as each species is separately judged by a different fitness payoff. NEAT can be applied to reinforcement learning problems, making the algorithm useful for general video game AI, requiring the video game input data to be parsed in a format understood by a neural network.

Genetic algorithms have become a popular tool for video game AI. The NEAT [11] method applies very generally, potentially allowing a large number of species to develop different strategies that combine to create complex behavior. For example, agents in Agar.io could tend to both forage for food and develop competitive reactions to other agents. As explained in the paper, the NEAT algorithm provided comparable solutions to other algorithms in its class with far fewer evaluations and fewer generations. Video games and other learning tasks that may be solved with incremental changes that open further avenues (such as platforming games) are well-suited to the NEAT algorithm.

If the AI is to be tested in a live setting, developing new agents as quickly as possible may be a useful metric. Certain implementations of genetic algorithms may not quickly converge, as rounds may be difficult to run quickly and new species with potentially valuable mutations develop randomly. Methods such as Q-Learning may more quickly converge due to continuous updates with each action.

## 3 Approach

I primarily used the method for developing a neural network-controlled AI in dynamic environments called NeuroEvolution of Augmenting Topologies (NEAT), outlined in [11], reference above.

I used an 'object-oriented' approach to parse video game information, described in [4]. Rather than using raw pixel data or a feature grid, they organized each object-type (such as the agent, competing agents, and obstacles) into variable-length feature vectors. I used a similar approach, parsing the field into enemies, food particles, and the player. I then transform these feature vectors into a fixed-length input vector for the NEAT neural network.

The output is two different continuous nodes with values ranging from -1 to 1, representing 'x' and 'y' coordinates. These coordinates represent the position of the mouse relative to the center of the screen, where the player is located.

For the NEAT algorithm, the linked code repository and Python package *neat-python* in [11] forms the body of the approach. The *neat-python* package contains modules for a forward-feeding neural network with associated configuration, including the size of populations, number of generations, activation functions, mutation rates, and other parameters necessary for designing a genetic algorithm backed by a neural network. I used the functions in the library to create neural networks for each instance of a species, test the agent in a training environment, extract the score, and pass the utility back into the network.

I developed tools to parse the playing field, including identifying food particles, enemies, and the player's score/size. The OpenCV [12] project includes utilities for adding filters to images and detecting circles. I was able to identify food based on the smaller size of the circles and assign all other circles as an enemy, assuming the circle in the center is the player.

## 4 Experimental Design and Results

### 4.1 Training Setup

For running the agent, I am using a modified Agar.io clone written for the Node.js platform [13]. This server is brought up in a local Docker container with agents connecting using a headless browser controlled by the Selenium client. A similar setup is used for web-scraping dynamic websites for testing dynamic pages. I ran multiple instances of the server and randomly assign groups of agents to each with every generation.

I developed a program that will take an image of the playing field on a set interval; runs image processing to identify food particles, enemies, and the player statistics; transforms the feature vectors representing each object-type into a fixed-length input vector; and feeds this into the neural network to choose the agent's movement (by controlling the agent's mouse cursor). On death or time-out, the agent records its current weight for use in updating the NEAT-controlled neural network

After the agent was trained across thousands of generations, I used checkpoints to run the highest-performing agents from each checkpoint generation against each of the benchmark agents. To score each generation, I perform 50 runs with one agent trained with NEAT and the other three benchmark agents in the field, taking the final weight of the agents after 30 seconds. From this, we can compare the general performance of the NEAT agent across generations.

### 4.2 Image Processing

The circle Hough Transform (CHT) [14] identifies circles in images based on parameters including size, density, and sharpness. My investigation included tuning these parameters using various screenshots from both the live Agar.io website and from the local, development server with states containing a variety of enemies and arrangements of food pellets. Parsing the field occurs in five steps after the agent takes a black-and-white screenshot:

1. The image is down-scaled to a fixed resolution.
2. The contrast and brightness are adjusted to differentiate the agents/food pellets and the background, removing the grid in the process.
3. A CHT identifies the smallest food pellets.
4. The identified food pellets are removed from the image, leaving the larger agents.
5. A CHT identifies the agent and enemies.

Below are the phases with distinct image outputs. The final stage highlights the player in green, the food pellets in blue, and the enemies in red.

I transform the absolute locations of each food pellet and enemy to coordinates relative to the player agent, assumed to be the center $(0, 0)$.

From there, I sort the enemies and food pellets by Euclidean distance to the player. To decrease the complexity of the neural network, I trim the list to a fixed-length vector (Figure 2) with the following: the player radius, the nearest enemy (including enemy radius and coordinates relative to the player), and the nearest food pellet (with the coordinates relative to the player). The list

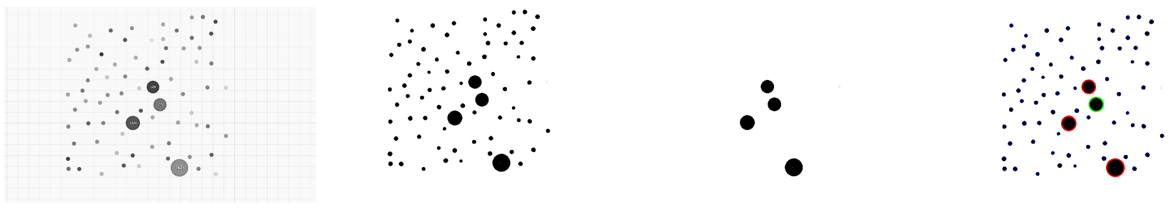
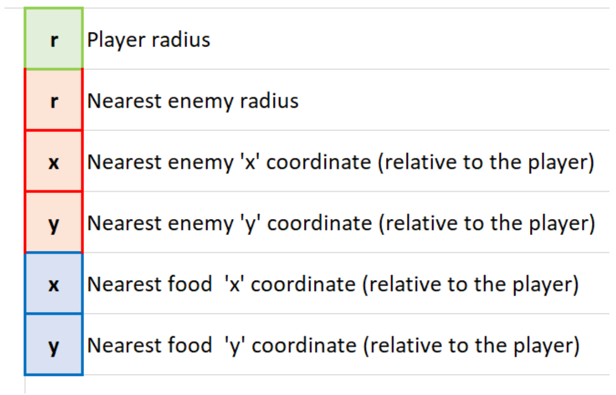


Figure 1: Field Parsing



| | |
|---|---|
| r | Player radius |
| r | Nearest enemy radius |
| x | Nearest enemy 'x' coordinate (relative to the player) |
| y | Nearest enemy 'y' coordinate (relative to the player) |
| x | Nearest food  'x' coordinate (relative to the player) |
| y | Nearest food  'y' coordinate (relative to the player) |

Figure 2: Input Vector

was sorted with the assumption the nearest food pellets and enemies are the most relevant for the agent's next action.

I chose to make this input vector fairly short due to later-described limitations with my hardware that limited the number of agents that were able to run at one time. A shorter input vectors limits the potential ability of the agent to parse the environment and make decisions but allows the network to converge more quickly. The code allows for a simple variable modification to increase the number of vector cells dedicated to the enemies and food.

## 4.3 Algorithm Design

The AI develops by running against other instances of NEAT agents (of the same and different species) and randomly-selected agents from a set of benchmark agents defined in the next section. They compete in a pool of locally-hosted servers. The agents are given a set amount of time (30 seconds) to move about the playing field before they are destroyed, their final measured mass being the output reward for the agent.

There is no explicit upper bound to the 'mass' an agent may attain. The maximum fitness a single agent could achieve is by ingesting all other agents and the maximum number of pellets on the field in the allotted time. If an agent is killed, its fitness is recorded as '0'. Other approaches to a death would be to take the final mass before death or a specific percentage of this mass. I chose this option to highly punish agents for not considering enemy players, but this may discourage developing competitive behavior as any competitive mis-steps would be costly. I monitored and display the average fitness of the whole population and individual species as the model trains.

As a baseline, I developed an additional set of three agents to compete against the agents trained using the NEAT algorithm and to prove the effectiveness of the image parsing:

1. A 'greedy' agent that heads toward the nearest food particle.
2. An 'aggressive' agent that will move toward any enemy in its feature vector list whose radius is smaller than its radius. If no enemies are matching this description, the agent follows 'greedy' behavior.
3. A 'defensive' agent that will away from toward any enemy in its feature vector list whose radius is larger than its radius and is within a set distance. If no enemies are matching this description, the agent follows 'greedy' behavior.

The decision process for the 'aggressive' agent.

```
# Find an enemies nearby the player
for enemy in enemies:
    if not ((enemy[0] == 0) and (enemy[2] == 0)) and player_radius > enemy[2]:
        return enemy[0:2]

# Closest food
return food[0]
```

The decision process for the 'defensive' agent.

```
if len(enemies) > 0:
    closest_enemy = enemies[0]
    dist_closest_enemy = (((closest_enemy[0]) ** 2) + ((closest_enemy[1]) ** 2)) ** (0.5)

    # Run from closest enemy, if within distance threshold
    if dist_closest_enemy < DefensiveAgent.CLOSEST_AGENT_THRESHOLD:
            return [-closest_enemy[0], -closest_enemy[1]]

# Closest food
return food[0]
```

As can be seen with the other agents, if no enemies affect the decision process, each reverts to the decision process for the 'greedy' agent:

```
# Closest food
return food[0]
```

Each of the above agents is implemented with just a few lines of code. While the functions explicitly split the different agents into separate arrays, their functionality essentially comprises basic decision trees that use the same feature vector fed into the AI's neural network. As the AI's neural network has a dynamic topology, the agent should be able to mimic any of these three agents, given proper feedback.

The neural networks used in the NEAT library have many nodes with different sizes of input and output vectors. Typically, an output value is a weighted linear combination of the input values passed through an activation function, a function that converts the single output value to the desired output range following some specific continuous function. The choice of activation function for the nodes of a neural network depends on the application. For example, classification problems favor sigmoid functions, defined as $S(x) = \frac{1}{1+e^{-x}}$, as they can convert a continuous range to a nearly binary output value.

The NEAT library allows specification for the activation function used for the hidden layers and the output layer. I decided to use a sigmoid activation function for the hidden layers to allow for clear decision boundaries to develop (similar to the aggressive and defensive agents). However, if the sigmoid function were to be used for the two output nodes (the 'x' and 'y' coordinates of the mouse cursor) the agent's movement could be biased toward specific movement vectors. I plot the output density given continuous input ranges for the 'x' and 'y' values of the below grid, representing a hypothetical neural network configuration where each of the two output nodes is connected to a single input node with the sigmoid function as the activation function, Figure 3. The graph displays a tendency for specific movement vectors: the diagonal directions in the case of the below graph. This specific case would result in an agent that heavily prefers diagonal movement. Such sensitivity to the inputs makes fine-grained control and unbiased input in any direction more difficult for the agent. I chose to use a linear activation function, instead, for the output layer as it does not have an inherent bias for specific movement directions nor magnitude of movement.

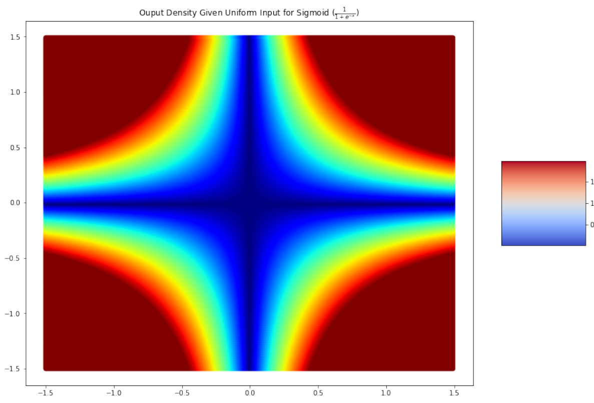


Figure 3: Mouse-Placement Density Given Single-Node, Continuous Inputs

I will measure the performance of each agent by taking the average final weight across a series of runs with each agent type (greedy, aggressive, defensive, and a trained NEAT agent) in competition. If the AI does not develop competitive tendencies by ingesting other agents, it should not perform better than the greedy agents. As both the aggressive and defensive agents include parameters that could be tuned–specifically the minimum distance to incite a reaction–the NEAT agent could find more optimal forms of each, even with identical decision-making patterns.

In addition to the effectiveness of the agent to attain a larger weight, I will measure the computation time for the greedy and NEAT-controlled agents for deciding a direction.

I am assuming the Agar.io game is simplified (excluding advanced features such as splitting of agents), the playing field is smaller than the live site (to force agent interaction), and the only competing agents are instances of the same and other species.

| Player | Average Score |
|---|---|
| NEAT | 35.418 |
| Aggressive | 34.106 |
| Greedy | 33.801 |
| Defensive | 33.390 |

Table 1: Average Score by Agent

### 4.4 Results

I trained the AI across 6,700 generations. Every 500 generations, I saved the best genome and tested the agent against greedy, aggressive, and defensive agents. Below I created visuals describing the performance of the NEAT agents and describe the performance of the agent across the generation sets, using the three agents as benchmarks.

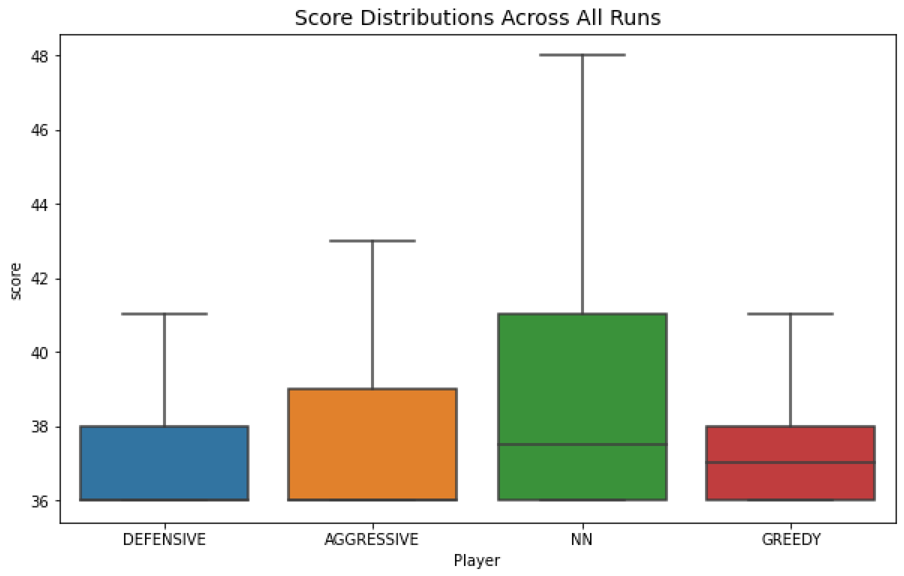


Figure 4: Distribution of Scores Across All Runs for Each Agent

## 5 Analysis

### 5.1 Performance

As shown in Table 1, the agent trained using NEAT generally performed better than the programmed agents. The average fitness is 4.8% higher than the mean of the other agents and 6.07% higher than the worst performing, 'defensive' agent.

The testing environment was set up to match the Agar.io production site. The agent design is highly successful at changing environments; the trained neural network and the other agent types can be run on the live site with a few lines of code, mostly different IDs for the website elements.

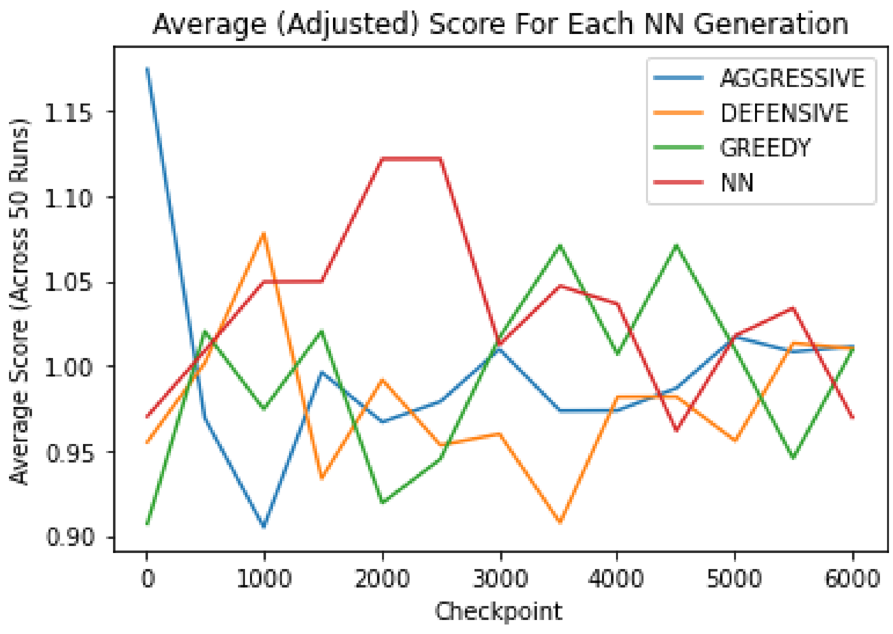


Figure 5: Average Adjusted Performance of Agents in a Competitive Environment

In terms of speed, the system is less successful. Extracting the image from the browser and running the image parsing to extract the player, enemy, and food information took an average of 0.082 seconds. Moving the player took an additional average of 0.293 seconds. This makes the total time for the agent to act–without the time to decide an action–0.375 seconds, an update rate of 2.66 times per second. Given the fast-paced nature of the live server, this update speed is not incredibly useful for a production AI. Further work would require implementing a much faster system, perhaps decoupling the browser from the control and image retrieval by using a more direct mouse controller and OS-controlled screenshots.

## 5.2 Comparison

Looking at Figure 5, we see no clear improvement in the performance of the neural network throughout multiple generations. Qualitative observation of the agents in the generations 1000-3000 shows erratic movement patterns that reach many food pellets early on, leading to lucky eating of other agents. However, as the NEAT agents beings to display obvious reactions to the presence of other agents later, the movement patterns become less erratic. The high penalty of death appears to have forced the agents to develop a slower approach to exploring the field.

Comparing the time to calculate the actual move, all of the agents take less than a millisecond. Due to the structure of neural networks, it is very quick to make a calculation when the structure is small, as in our example. The training time is the largest setback, requiring extensive computation and proper training setup. Similarly, the file size of the best genome is 270 KB. While the other agents do not rely on any sort of file storage for checkpoints or improvements, the file size to hold the usable neural network genome is small enough to not make a practical difference in the selection of the best agent.

## 6 Conclusion and Future Work

The image processing techniques used to separate the enemies, food pellets, and players allowed for generalized usage of the AI with both the testing and the live site. The 'greedy', 'aggressive', and 'defensive' agents were quickly implemented and performed adequately on the live site. With further testing and an expanded input feature column for the agent (including *multiple* nearby enemies and food pellets), a neural network could develop unique strategies for strafing and attacking other enemies.

Due to the high memory usage of browsers, running more than a few instances of the test server with connected agents is impossible without upgraded hardware. The NEAT library generally expects larger population sizes, allowing for a more diverse set of species and for a species to have a more accurate average fitness score. Future efforts would require a more performant solution for testing or a larger number of test machines, keeping the image processing intact.

## References

[1] Jacob Sampson. agar.aio, 2021. Available at https://github.com/JacobSampson/agar.aio.

[2] Baozhu Jia and Marc Ebner. A strongly typed gp-based video game player. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 299–305, 2015.

[3] Baozhu Jia, Marc Ebner, and Christian Schack. A gp-based video game player. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, page 1047–1053, New York, NY, USA, 2015. Association for Computing Machinery.

[4] William Woof and Ke Chen. Learning to play general video-games via an object embedding network. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, page 8490438, 2018.

[5] Daniël M. Pelt and James A. Sethian. A mixed-scale dense convolutional neural network for image analysis. *Proceedings of the National Academy of Sciences*, 115(2):254–259, 2018.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] Nil Stolt Ansó, Anton Orell Wiehe, Madalina M. Drugan, and Marco A. Wiering. Deep reinforcement learning for pellet eating in agar.io. In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence*, volume 2, pages 123–133, 2019.

[8] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[9] Peter Vrancx, Katja Verbeeck, and Ann Nowe. Decentralized learning in markov games. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 38(4):976–981, 2008.

[10] Diego Perez-Liebana, Sanaz Mostaghim, and Simon M. Lucas. Multi-objective tree search approaches for general video game playing. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 624–631, 2016.

[11] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[12] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[13] juslee. agar-io-clone, 2015. Available at https://github.com/juslee/agar-io-clone.

[14] Tim J. Atherton and Darren J. Kerbyson. Size invariant circle detection. *Image and Vision Computing*, 17(11):795–803, 1999.