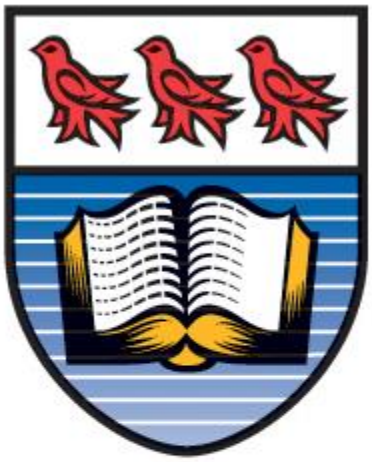

University of Victoria

CENG 420:

Final Project Report

Applying Artificial Intelligence and Machine Learning Algorithms to the Game Agar.io

Group Name: Deus Ex Machina

Group Members:

Yukun Huang	V00804445	kennethhyk@gmail.com
Cameron Mulgrew-MacFarlane	V00775582	Cammac93@gmail.com
Willy Su Yep	V00795480	Op4977377832@gmail.com
Sam Wheating	V00816465	SamWheating@yahoo.ca
Matt Johnson	V00757521	matthewj@uvic.ca

Contents

Abstract 3
Introduction 4
Methodology 5
Game Environment 5
Game-AI Interaction 5
Encoding and Situational Awareness 5
Algorithm Design and Implementation 7
Results 8
Real-Time Performance 8
Effect of Training Data Quality on Algorithm Performance 8
Ability to Train Neural Network with Human Input 9
Comparison of AI and Human Abilities 9
Conclusions 9
References 10

List of Figures:

Figure 1: The Agar.io game as available at https://agar.io 4
Figure 2: Illustration of the search radius and sector model using N=8 6
Figure 3: Cell radius as a function of time given different sizes of training dataset 8

List of Tables:

Table 1: The encoded food and threat values corresponding to Figure 1, represented in a 2xN array. 6
Table 2: Evaluation of different SciKit-Learn Algorithms 7

Abstract

Agar.io is a massively multiplayer online game in which players aim to grow their cell by eating small food particles and smaller players, while avoiding larger players. This project aimed to use machine learning to create an artificially intelligent player which can navigate the map and avoid enemies whilst growing as fast as possible.

A local instance of the game was set up with a custom API in order to interface between the game and python scripts using simple API requests. All implementations were written in Python 3 with libraries including SciKit-Learn, Numpy and Requests as well as many other standard python libraries.

Several algorithms of varying complexities were implemented for this project, and a controlled test environment was set up in such a way that the different methods could be reliably compared. Algorithms included a greedy algorithm and weighted heuristic function as well as a neural-net-based classifier which could be trained on either bot-generated or human generated data.

Results were promising as it was demonstrated that the neural net could successfully learn to play the game, with ability being dependent on the size and quality of the training data set. Further developments could include the implementation and comparison of other algorithms or a comparison of different neural-network implementations used with the existing algorithm.

Introduction

Agar.io is a popular online game released in 2015 by the Brazilian Developer Matheus Valadares. Players assume the role of a cell in a petri dish and must navigate around the map collecting food and absorbing smaller players whilst avoiding larger players.

As this game involves many similar situations and constant situational processing and decision making, it was decided that it would be an interesting application of artificial intelligence and machine learning algorithms. The game is simple enough to break down into a relatively small, numerically represented encoding but complex enough that there are many possible ways of doing so. Additionally, as it is a multiplayer game we will have a chance to face different models and algorithms against each other as well as against human players, thus allowing further assessment and comparison.

Furthermore, this game is very fast-paced so it challenges the real-time performance of the algorithms and their ability to process inputs and make decisions in short timeframes. Different models can have widely different run times and often there is a trade-off between model accuracy and decision time. By running tests in a closed, consistent environment it will be possible to isolate variables and relate model parameters to both accuracy and runtime.

The main purpose of this project was to compare the performance of the chosen algorithms to both each other and the abilities of a human player. Performance can be gauged on a variety of metrics including cell growth rate or navigational accuracy as well as traditional machine learning metrics such as mean error.

Figure 1: The Agar.io game as available at https://agar.io

Methodology

The requirements for implementing an intelligent agent can be separated into three main areas: extracting the state information from the game, parsing the state information into a format suitable for machine learning, and applying the machine learning algorithms to the formatted data and evaluating their performance.

Game Environment

As the game was recently updated to a much more secure and less accessible platform, it was decided to run a local clone of this game built using node.js, which allowed both offline play and near-instant communication between scripts and the game itself. A custom API was built which let scripts interact with the game using simple commands such as createPlayer, movePlayer, getBoardState and others. With this custom version of the game it was possible to provide an environment similar to the original game but more responsive and easily controlled.

Game-AI Interaction

Main programming language used to develop and train artificial intelligences is Python3, artificial intelligences can communicate with the game server by calling APIs through HTTP requests. The game server will be hosted locally for all testing and evaluations of this project, and API calls will take at most 5ms when communicating with local game server.

Encoding and Situational Awareness

A significant component of this project was the encoding of the game state into a form which could be more easily understood and manipulated by the algorithms. Several possible methods were discussed, including a constantly-generated list of all items on screen and their properties (implemented within the game source) or a computer-vision based method which would run outside of the game and read objects off of the screen. It was eventually decided on to implement an API function which could return a list of objects within a certain radius of a specified player. This allowed flexibility in the adjustment of the search radius / field of view while not providing more data than necessary and thus complicating the model.

From this list of nearby items an array was built, which summarized the presence of both food and opponents in the surrounding area, divided into a series of symmetrical sectors. At this point, the game state was reduced into a series of 2*N numbers, with N being the number of sectors (typically set to 16). This was an easy structure to work with and facilitated sufficient situational awareness for our machine learning models.

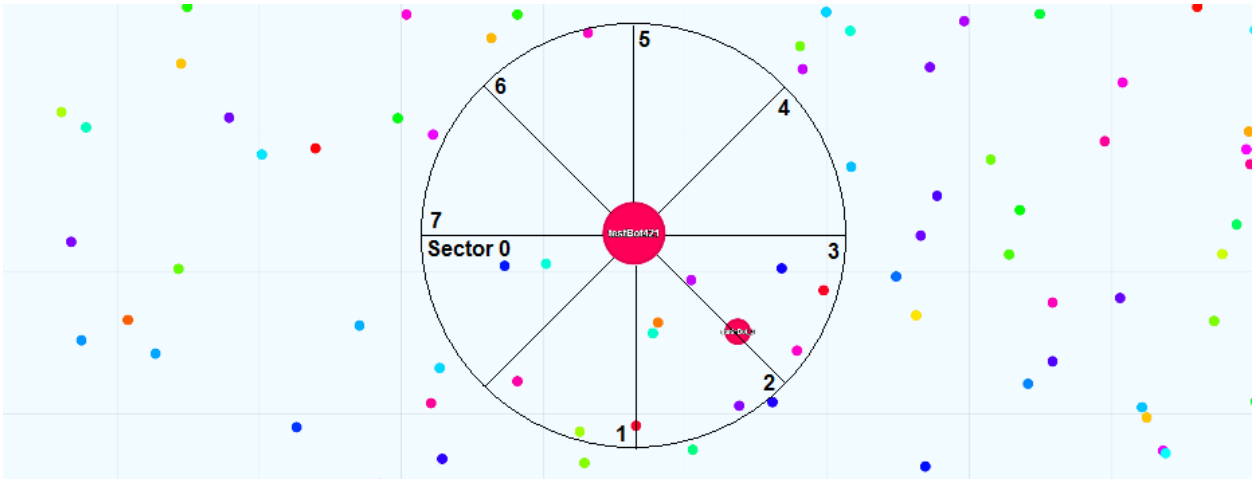

Figure 2: Illustration of the search radius and sector model using N=8

The threat score for enemies located within the search radius is calculated using the formula where $threat_i$ is the threat for a given object, and x_i and y_i are the realtive coordinates of the threat.

$$threat_i = \frac{Mass_i}{\sqrt{{x_i}^2 + {y_i}^2}}$$

The threat score and food score for a sector are found using the formulas

$$sector\ threat = ceiling\left(\text{MAX\_THREAT\_LEVEL} \times \frac{\sum threats\ in\ sector}{\sum all\ visiblet\ hreats}\right)$$

and

$$sector\ food = floor\left(\text{MAX\_FOOD\_LEVEL} \times \frac{\sum food\ in\ sector}{\sum all\ visible\ food}\right)$$

where enemies that have a mass less than a certain threshold are counted as food instead of threats. Thus, according to the above formulas, the game state in **Figure** 2 is processed into the 2N numbers listed in
Table 1 below.

Sector	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Food	2	2	4	> 4	0	0	1	0
Threat	0	0	0	0	0	0	0	0

Table 1: The encoded food and threat values corresponding to Figure 1, represented in a 2xN array.

Algorithm Design and Implementation

Multiple algorithms were implemented using the same API and framework for situational awareness.

The hungry algorithm was a simple test which caused the player to constantly move towards food and thus continue growing. While it worked, it had no defensive mechanism and thus was prone to being absorbed by a larger player.

The Heuristic algorithm used a weighted function of both food and threats to decide which sector to move into. This was an improvement over the hungry algorithm as it included a defensive mechanism and acknowledged the presence of multiple pieces of food (it would move into an area of high density food rather than towards a single, closer piece of food.)

$$reward = (\sum all\ visible\ food) - (\sum all\ visible\ threats) + \text{REWARD\_FOR\_EATING} \times \Delta m$$

The Neural-Network based classifier used the Multi-layer-perceptron neural net implementation from SciKit-Learn to learn to play the game based on a large set of sample moves. The input data was given as a set of 2*N numbers representing the game state as well as a label indicating the move taken. Two sets of training data were assembled: one which recorded the state and subsequent decisions made by the heuristic algorithm and another which recorded the state and subsequent decisions made by a human player. With a long enough training period this algorithm would be able to interpret the playing style of a human and make similar moves. Once the neural network was constructed and batch-optimized, inputting a single state would return a number from 0 to N-1 indicating which sector should be moved into.

Several algorithms from Scikit learn were evaluated using the data format described above. The results of this evaluation are sumrised in Table 2 below.

Algorithm	Accuracy (%)	Standard Deviation (%)
Logical Regression	92.05	1.78
Linear Discriminant Analysis	88.58	1.87
KNN	75.26	2.12
Multilayer Perceptron	91.23	1.83
Gaussian Naive Bayes	19.36	1.8
Support Vector Machine	90.47	1.91

Table 2**:** Evaluation of different SciKit-Learn Algorithms

Results

The performance of the machine learning algorithms is evaluated with regard to several criteria.

Real-Time Performance

It was decided that in order to play the game effectively, the program should decide on a new move ten times per second. All of the algorithms used in this project were capable of making decisions significantly faster than this and thus there were no issues with real-time performance. A delay was added in the code to standardize moves to 0.1 seconds. In other applications where a higher response frequency is required there may be more emphasis on algorithmic simplicity, but there were no issues in this implementation.

Effect of Training Data Quality on Algorithm Performance

A test was conducted to illustrate the effect of training data size on the algorithm performance. A controlled environment was set up with ten non-moving, small opponents and the Neural net classifier was run with training the training data set constrained to 250, 500, 1000, 2000 or 3000 samples from the Heuristic algorithm training data set. Size over time was plotted as a measure of algorithm effectiveness as a better-trained algorithm would be expected to grow at a higher rate. Each test was run for 2000 moves, at 0.1 seconds / move for a total of 3 minutes and 20 seconds per test.

Using smaller training datasets, the algorithm was more likely to get 'confused', or become stuck in a corner or oscillating position. This was likely due to a poorly-tuned neural network giving different directions for similar states, thus leading to a repeating loop. Possible solutions for this issue include increasing the size of the training data set, increasing the field of view or randomizing the delay between moves to limit the possibility of consistently repeating patterns.

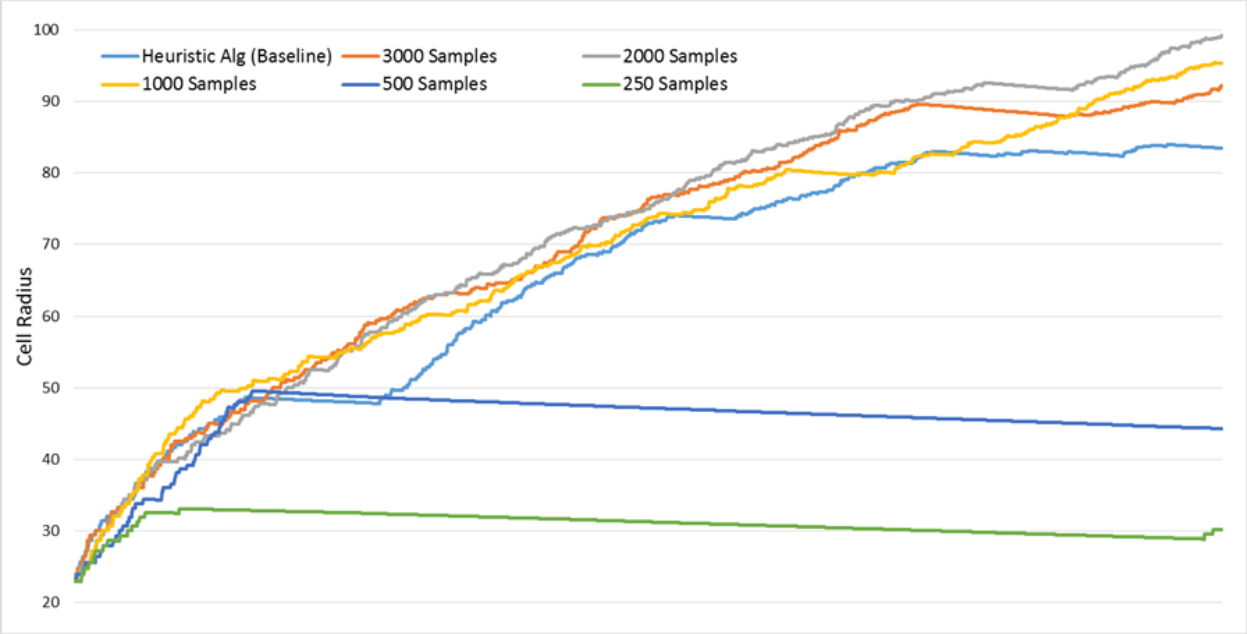

Figure 3**:** Cell radius as a function of time given different sizes of training dataset

It was also observed that players trained on a smaller training set followed less direct paths towards targets and had a tendency to 'zig-zag' rather than following straight or smoothly curved paths.

The Heuristic Algorithm is prone to 'local starvations' as it can consume food in a pattern such that there is no food in its field of view. In this case the algorithm will move in a random direction and thus will eventually self-correct, though it can sometimes take quite a while. The Neural-network algorithm is less prone to this as an empty input (no players or food in field of view) will return a constant value and thus the player will move in a straight line and reach food faster than if it were moving randomly. These periods of local starvation can be seen as flat portions on the above graph, and they are more pronounced on the baseline plot.

Ability to Train Neural Network with Human Input

A script was written which would operate at the same frequency as the running model and record the state and subsequent decision based on a difference in player position. This enabled the creation of training data sets identical in format to the heuristically-generated training data to be 'recorded' from a human player's actions.

When the Neural-Network based classifier was trained on human-generated data its performance was inferior to that which was trained on the heuristic function dataset. This is likely due to inconsistencies in human decision making and thus irregular patterns in the training data, whereas the heuristic function would always make the same decision given identical situations and thus produced a more consistent training dataset with more prominent patterns.

Comparison of AI and Human Abilities

When trained on the heuristic function dataset the neural network classifier was proficient at playing the game. When playing against human players it can successfully avoid larger opponents and chase smaller prey whilst maximizing food intake. With multiple identical bots playing in a single instance they can be observed chasing each other, though there is still visible evidence of the limited field of view (bots won't react to objects until they enter the search radius).

A video of multiple bots playing against each other can be found at the address below. Each bot is labelled with the algorithm which is determining its actions and the smaller, stationary bots are simply added as additional food.

A video demonstration of the trained agents is available at: *https://www.youtube.com/watch?v=7WaBukyt_l8*

Conclusions

In conclusion, this project was overall successful as we were able to implement a neural-net capable of deducing playing styles and applying them to the game. Further development could include the implementation of different algorithms (such as the q-learn algorithm) for comparison as well as the application of these programs to the full-scale online game, which would provide a much less controlled environment.

References

[1] M. Harvey, "Using reinforcement learning in Python to teach a virtual car to avoid obstacles," 6 February 2016. [Online]. Available: https://blog.coast.ai/using-reinforcement-learning-in-python-to-teach-a-virtual-car-to-avoid-obstacles-6e782cc7d4c6. [Accessed July 2017].

[2] T. DeWolf, "Reinforcement learning part 1: Q-learning and exploration," 25 November 2012. [Online]. Available: https://studywolf.wordpress.com/2012/11/25/reinforcement-learning-q-learning-and-exploration/. [Accessed July 2017].

[3] J. McCullock, "A Painless Q-Learning Tutorial," Mnemosyne Studio, [Online]. Available: http://mnemstudio.org/path-finding-q-learning-tutorial.htm. [Accessed July 2017].

[4] A. Juliani, "Simple Reinforcement Learning with Tensorflow Part 0: Q-Learning with Tables and Neural Networks," 25 August 2016. [Online]. Available: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0. [Accessed July 2017].

[5] Outlace, "Reinforcement Learning: Part 1 - Action-Value Methods and n-armed bandit problems," 19 October 2015. [Online]. Available: http://outlace.com/rlpart1.html. [Accessed July 2017].

[6] H. Tr, "Agar.io Clone," 2015. [Online]. Available: https://github.com/huytd/agar.io-clone. [Accessed July 2017].