Slither.io Deep Learning Bot

James Caudill

Computer Engineering
California Polytechnic State University, San Luis Obispo

Senior Project Report
Advisor: Dr. Foaad Khosmood
12 June 2017

Abstract

Recent advances in deep learning and computer vision techniques and algorithms have inspired me to create a model application. The game environment used is Slither.io. The system has no previous understanding of the game and is able to learn its surroundings through feature detection and deep learning. Contrary to other agents, my bot is able to dynamically learn and react to its environment. It operates extremely well in early game, with little enemy encounters. It has difficulty transitioning to middle and late game due to limited training time. I will continue to develop this algorithm.

1 Introduction

Since the dawn of artificial intelligence at Dartmouth College in 1956, games, and more recently video games, have greatly aided the development and testing of intelligent algorithms. There is no question that without the help of games, the science would have never taken off. First it was tic-tac-toe and checkers that machines dominated and many games were deemed to be 'solved'. These early programs were all similar in the fact that they all followed an instruction set, no algorithm could play any other game. These basic games graduated into more complex ones with victories or world champions in games like chess by IBM and now the game of go by Google. However, these programs benefit from advanced search techniques that help with the problem of dimensionality. But since the last year some incredible breakthroughs have been made by researchers in the field. Most notably google deepmind created a bot to play numerous Atari games at better than human expert level. This achievement, as impressive as it is, derives its success from one main re-invention, the neural network (specifically deep recurrent neural nets). This, along with many other advances in computing power and open sourced learning frameworks, has undoubtedly moved the science forward and now promises widespread application. While industry leaders are moving fast to implement deep learning in their businesses, whether it transportation, trading, or medicine, we built another video game bot.

The game Slither.IO celebrated its first birthday on March 25 this year. With that it also celebrates a year of highly positive reviews. It's popularity was only increased as it became publicised by many famous streamers and YouTubers. It rode the curtails of a previous popular game called agar.io, but the overall complexity of slither is much lower. The game is basic, a player uses the mouse to control a snake that is trying to stay alive in a large circular 2D map littered with food and other snakes. Anyone can play it online at http://Slither.io or on any device as a mobile app. (Though playing on a computer is recommended.

The program can be broken down into two major parts, feature detection and machine learning. They will both have their own sections with figures and code explanation below. One thing to keep in mind: Nobody has ever done this before for this game and there are significant differences between more simple games, specifically the fact that it is mouse controlled, that created many challenges.

2 Background

As mentioned before there is no deep learning slither bot out there to reference, but there are other algorithms designed for different games. Two major papers were used as reference and inspiration for this project. The first: Playing Atari with Deep Reinforcement Learning by seven researchers at Google DeepMind is the pivotal paper showcasing the power of Deep Q learning on a number of different Atari games. [1] This groundbreaking work shotput the neural net approach onto the scene of reinforcement learning. Previous to this paper most advances in machine learning were with supervised and unsupervised learning, leaving little market share to reinforcement. But that all changed once DeepMind showed how good a single algorithm was at learning to play so many Atari games at a superhuman level. The second: Deep Reinforcement Learning: Pong from Pixels by Andrej Karpathy is an exploration into policy gradients and the classic game of pong. [2] As mentioned before, pong was one of the first games to be attacked and beaten by earlier artificial intelligence programmers. Andrej's approach is a novel one utilizing discrete game states and a markov decision process. This blog post/paper is a follow up to the Google paper showcasing the tradeoffs between deep Q learning and policy gradients. The reason this pong from pixels paper was particularly useful was because it utilized the same environment toolset, OpenAI Gym.

3 System

The system is split up into front end and back end processing. This project focused on the development of the front end which is also comprised of two main parts, feature detection/environmental labeling and machine learning. The back end is handled with the framework provided by OpenAI. The framework consists of two major components, Gym and Universe. Below I discuss how everything connects and operates.

3.1 OpenAI Back End

In April 2016 OpenAI, a nonprofit research institute, released an open beta for Gym. Gym was the foundation for Universe, which was released eight months later in December. Gym was built to be a training ground for user programmed artificially intelligent agents. Some games were built into the framework for basic reinforcement learning. Universe is what blew the whole operation wide

open. It added thousands of games to the programmer's accessibility and sped up the main workings of Gym to account for more training. What these frameworks provide the AI programmer is a capability to train a bot remotely. Instead of always hosting the game on a local machine, it adds a layer of abstraction to allow the program to not worry about the game and focus on training. How it does this is not so simple, it's a mesh of server side game processing and websockets. The bot sets up a TCP VNC connection to receive the pixel data and send the action data and an auxiliary reward connection to receive reward and latency information. This allows the program to interact with the server with two simple function calls: *make* and *step*. Make creates the necessary processes both locally and on the game server (which can be remote or local). The step function takes as input an action for the bot to take, which can be arrow keys or mouse inputs. It returns a tuple consisting of the current game pixels, the reward received since the last step, a boolean to check if the bot has died, and latency information.

```
env = gym.make('internet.SlitherIO-v0')
observation_n, reward_n, done_n, info = env.step(action_n)
```

Universe has many unique environments for the bot to train on, such as flash games, atari games, web games like slither, and even browser tasks like booking flights. Since Slither.io provided the most dynamic yet simple environment it was chosen for base training.

3.2 OpenCV Feature Detection and Labeling

Due to lack of extreme computing resources simple mimicry of other deep learning researchers won't be possible. Instead of training the model just purely on the raw pixel data it was necessary to add some feature detection into the equation. This mechanism would take the raw pixel input and execute computer vision algorithms to produce a feature dictionary. In order to differentiate the in-game elements a myriad of blurs, thresholds, color conversions and high level OpenCV functions were used. Since the pixel array is set at a native 1024x786 resolution and the actual game only takes up a portion of the screen it was necessary to carve out only the game pixels, the game screen was a round 500x300 pixels. After carving, the image was blurred to curb any noise from the connection and round the surfaces. The BGR image was transformed to grayscale and thresholded to eliminate the background hexagons as seen in **Figure 1**. In order for the game to not misunderstand the map and score located at the bottom of the screen for something interactable they are masked to black. A duplicate image is created to apply a higher threshold catching only very bright objects such as dead snake mass.

After the vision processing is done there are three images, two for OpenCV operations and one for user output to check if the functions found any mass/snakes. To recognize all elements on the screen three helper functions were created that flow in this order: findMass, findSnakes,

findDeadMass. Each is pretty self explainable, but has a unique attribute and OpenCV function call from them. To track the findings all observations are stored in a vision dictionary with keys representing pixel location and values representing what the item found was (‘S’ for snake, ‘Y’ for your snake, and ‘M’ for mass). An important note is that the dictionary keys are floor divided by 5 for the x and y. This was done to simplify processing for the neural network since a matrix of 100x60 is far more manageable than 500x300. So, if a mass was located at (323, 107) it would be marked as such in the vision dict at (64, 21). This vision dictionary would be fed to the neural network as pseudo-labeled features.

FindMass is the simplest function, taking in a grayscaled image and calling cv2.HoughCircles to return a list of wholly circular objections on screen. The only filled circular objects being the regular mass that is scattered throughout the server. From there the algorithm loops through each of them and label them in the vision dictionary and draw a green circle around them for the user to see.

FindSnakes is more complicated than findMass taking in a black and white image and calling cv2.findContours (findContours only takes images that are black background and white foreground whereas HoughCircles takes grayscaled). FindContours can be executed to return all points where there is an edge on an object, or it can return a simplified list of points. For speed purposes, the simple approximation was ran. This returned a list of observed objects. The program calculates the area of the contour with a built-in function called cv2.contourArea and only loops through objects with an area of 170 or higher. (This was tested as the minimum size of a snake with mass of 10) All points in the contour are labeled as ‘S’ in the vision dictionary which means enemy snake. To differentiate your snake from the enemies the moments of each contour were taken using cv2.moments. As the algorithm loops through snake contours it calculates the distance of the center of mass of each snake from the center of the screen. The snake with the closest center of mass is yours and re-labeled in the vision dictionary as ‘Y’.

FindDeadMass also calls cv2.findContours but on the higher threshold image to return only the contours that are mass left behind by a dead snake (which add much more mass to your total if retrieved). Each contour is marked as a ‘M’ in the vision dictionary if the area is less than 130. This is implemented because it sometimes would pick up the heads of very large snakes as dead mass. So through testing 130 was chosen as the max size the algorithm would consider the object as mass.

Another way of looking at these labels are (+) for mass, (0) for you, and (-) for snakes.

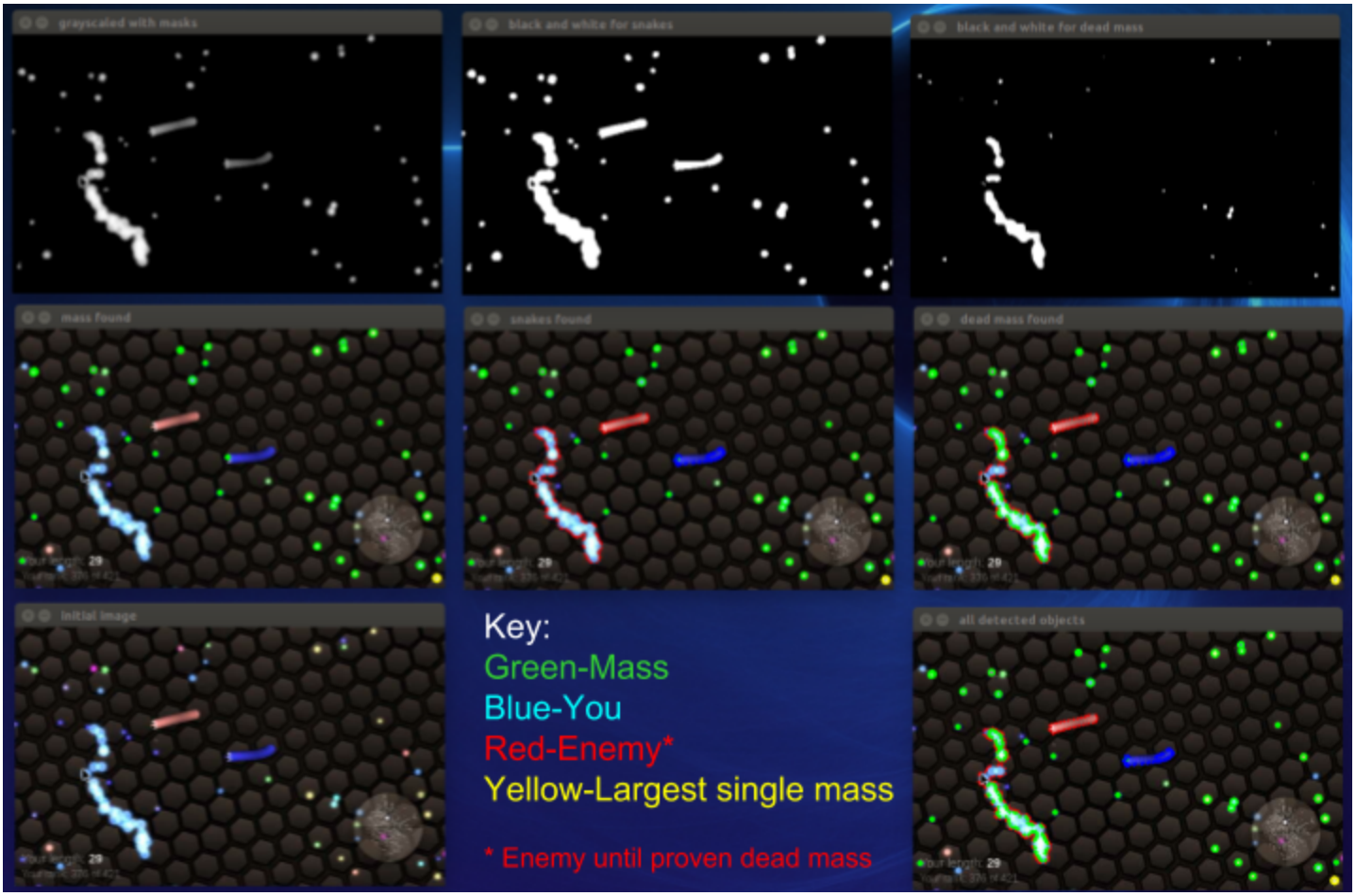

Figure 1: All Stages of Element Attribution and Labeling (Zoom In)
Top: Grayscaled, Black and White, High Threshold
Middle: Mass Labeled, Snakes Labeled, Dead Mass Re-Labeled
Bottom: Initial Image, Key, All Labels

Overall, the correctness of the feature labeling is somewhere between 85-90% for all objects. There are only two unfortunate negatives. The first is that boosting snakes are recognized slightly larger than they actually are because a light aura is emitted that throws off the findContours. This issue is not as big of a deal, but does through off the accuracy. The second is in the fact that dead mass is initially labeled as a snake then re-written as mass. This wouldn't be a problem if it was using the same image to derive the contours from then re-writing over the vision dictionary like what was done to find the bot's snake. However, the function takes a higher thresholded image which has its area decreased slightly. This means that when down sampling and floor dividing some points may not be correctly overwritten. This is difficult to measure since it doesn't happen much.

3.3 Neural Network

Building neural network architectures, much like computer architecture in the early days of computing, is a tough job. There are many tools to aide the process, but it really comes down to what the model will be built for. With previous deep reinforcement learning algorithms taking advantage of two major techniques, and with little time to implement and test the model myself, it was necessary to modify their approaches. Between policy gradients and Deep-Q learning, the one that was better applicable to slither was Deep-Q. Where policy gradients do well for games with a small number of actions, Deep-Q is capable of choosing the best action among far more choices. This highlights the biggest difference between my game and other applications, options. Atari Pong and Breakout have only about 5-10 actions mostly moving the paddle in a linear fashion (up/down or left/right). Slither, uses the mouse, thus, having thousands of potential moves one per pixel. Since I shrunk the game down to 60x100, it has to make a decision to mouse over one 5x5 box and decide whether to boost. That's 6,002 possible actions since inaction is not an option. (The two are from activate and deactivate boost) With all that in mind, and little formal neural network education other than reading these papers and watching/reading hours worth of informational content, the architecture was pieced together.

Taking after Google Deepmind, the architecture would be as follows in **Figure 2.** There are three main components to the learning network each labeled with a different color.

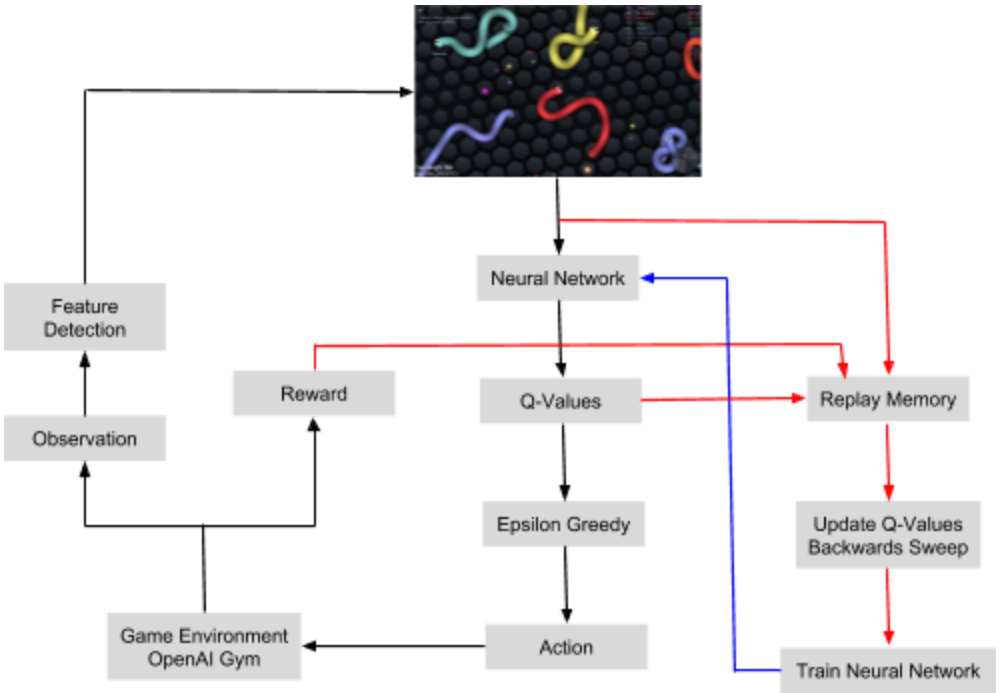

Figure 2: Deep Learning System Flow

The main game loop is in black starting at Observation on the left. The algorithm does image preprocessing talked about in the previous section. It then consults the neural network to create Q-Values from the frame. This is a list of probabilities of gaining a reward for each square it could move the mouse to. Instead of always picking just the max from that list it factors in randomness with the Epsilon Greedy algorithm. What epsilon greedy does is choose the max Q-value or a random point based on a small percentage. It's pseudo random because I already have labeled data to pull a value from. So if the network doesn't know what to chose if defaults to my original naive computer vision based decisions. Once an action is selected it updates the game. An important note to make again is that this is processing 30 frames a second, each time updating the neural network.

Highlighted in red is the replay memory network training. The agent needs to recognize awards occurring from a string of events rather than just one frame. With a little bit of python code we can save the top q values from previous states. Then we can update all of the values based on the next step and so forth. The frame is fed into the neural network as training data and it updates the model. After the model is updated it then creates Q-values for the current state.

In blue is the learning model being upgraded on the final step of the replay memory network. This is where the model gets tuned to the game. The model is 6,000 inputs nodes, four hidden layers of 1024 fully connected nodes and 6,002 end nodes representing the possible actions. The input to the algorithm is both the downsampled game pixels and the labeled game data. All labels are added to pixels in the area. This provides helpful distinction for the neural network to learn from.

The model I used was a generic model previously built for atari breakout by Hvaas-Labs in a string of tutorials they have on github for machine learning. [3] Similar to this project's implementation the tutorial adds helpful labels to the raw data. The model is still in training at the point of writing, but preliminary testing runs show progress similar to the atari tutorial. There is still more fine tuning necessary to achieve better performance.

4 Evaluation

There are a pair of Slither.io bots on the internet written in JavaScript originally written by Ermiya Eskandary. [4] These bots are given all environmental elements through the browser, and they make defensive and offensive decisions based on a radius around your snakes head. I wanted to build a system that could reach an average above their averages. Without the neural net implemented my bot would average 250 mass and the other bot would average around 300. With the maxes around 1900 and 3300 respectively. This is not too surprising since without the neural net my bot wouldn't take into effect the negative consequences of navigating into a snake.

With the neural net implemented, the agent slows at making decisions, due to computational expense. Training the network takes many many hours until any positive return is seen. With the help of the labels the network knows what is and isn't a good selection and with the help of the replay memory past moves can be associated with a reward. Overall, for the algorithm to truly exceed human levels of play the learning algorithm would need to play the game for much longer. Initial training on 10 hours shows some improvement.

5 Future Work

Without a doubt I will continue to build out this project more. The automatic labeler is near perfect, so not much work is needed on that. I do, however, want to keep on designing and altering the parameters for the machine learning portion of the system. The approach taken to accomplish the project was different than the way both the papers approached the problem, which proved to be challenging and indeterminate. Over the next summer I will modify the machine learning model provided by Hvass to better fit my data and environment. The idea of building programs that can solve many games with little code alteration is enticing. After this game, I will try altering the algorithm to fit other games as well.

6 Conclusion

Overall the project was a lot of fun to work on. I thoroughly enjoyed taking the time to understand the underlying system that OpenAI is offering, and I think that if more people knew about it many advances would be made in AI. Their mission is to create an agent that can learn to play them all, my mission was to create an agent to play one very well. I imagine that if there are good features labeled on all the games the problem would be much easier. Completing this senior project has provided me with more experience with Python, OpenCV and Tensorflow. The system operated fairly well, maybe not top ten of the leaderboard every game, but even the best human players can't accomplish this feat. With further work and tinkering I believe this bot could beat expert human averages.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Playing Atari with Deep Reinforcement Learning
https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[2] Andrej Karpathy. Deep Reinforcement Learning: Pong from Pixels
http://karpathy.github.io/2016/05/31/rl/

[3] Hvass-Labs. Reinforcement Learning Tutorial
https://github.com/Hvass-Labs/TensorFlow-Tutorials/blob/master/16_Reinforcement_Learning.ipynb

[4] Ermiya Eskandary. Slither.io JavaScript Bot
https://github.com/ErmiyaEskandary/Slither.io-bot