Slither.io Approximate Q Learning Agent

Mingyu Lu, Chihying Deng, Hsu Wei-Hung

January 3, 2019

1 Introduction

Reinforcement learning has been proved to be successful in many games e.g.Starcraft II,flappy bird. That inspired us to build a Q learning agent in an interactive environment. The game is Slither.io, the goal of which is to become the longest worm in the game.

One can achieve this by eating the food/energy that randomly appears on the board and by eating the glowing remains of your opponents. Like the classic game Snake, if you run into an opponent snake, you die. 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.

Considering the large state space of the snake world, several problems rises 1) The table of Q-value estimates can get extremely large. 2) Q-value updates can be slow to propagate. 3). High-reward states can be hard to find. State space grows exponentially with feature dimension e.g. the state space of Pacman with 107 possible location could be larger than $107^3 * 4^2$. A conservative estimation of the location within the world of slither.io is more than $10^6 * 10^6$. accordingly, it's inefficient and unrealistic to create a state-action Q table for it. Reward Shaping[3], a method giving some small intermediate rewards that help the agent learn, might be another approach. However, it requires intervention by the designer to add domain-specific knowledge. Besides, If reward/discount are not balanced right, the agent might prefer accumulating the small rewards to actually solving the problem. Most important of all, it does not reduce the size of the Q-table.

Therefore, we adopt function approximation, an agent learning a reward function as a linear combination of features, as a solution to slither.io. For each state encountered, we determine its representation in terms of features, performing a Q-learning update on each feature. The value estimate is a sum over the states features. We extract the "closest-food", the distance in slither steps to the closest food pellet, and "closest-enemy", the closest distance to the slithers ((regardless of whether they are safe or dangerous) that appears in the same screen. By function approximation, we are able to deal with continuous states and reduce the size of Q-table due the fact that many states share many features[3]. Similar to Q-learning agent, the agent has no previous understanding of the game. It is able to learn dynamically and react with each condition by approximate Q learning.

2 Background and Related Work

The system is split up into 1) image processing and 2) leaning agent implementation. This project focused on the development of the function approximation which is also comprised of two main parts, feature detection and feature processing. 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.

2.1 OpenAI Gym & Universe

In April 2016 OpenAI, a nonprofit research institute, released an open beta for Gym, a toolkit for developing and comparing reinforcement learning algorithms. OpenAI makes no assumptions about the structure of your agent, and is compatible with any numerical computation library, such as TensorFlow or Theano. Released in December 2016, OpenAI universe is an extension to OpenAI Gym. It provides an ability to train and evaluate agents on a wide range of simple to real-time complex environments. It also added thousands of games including Flash games, browser tasks, and games like slither.io and GTA V, to the programmers accessibility and speed up the main workings of Gym to account for more training without access to program internals, source code, or bot APIs. What these frameworks provide the AI programmer is a capability to train a bot remotely. Instead of hosting the game on a local machine, it adds a layer of abstraction to allow the program to focus on algorithm and training by 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. That is to say, this enables the program to interact with the server with two simple function calls: make and step .

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

Make creates the necessary processes both locally and on the game server (which can be remote or local). One could create a customized environment wrapper e.g. image filtering and rewarding shaping to process the pixel output. The step function takes an action as an input for the agent 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 value to check whether the game has done, and latency information.

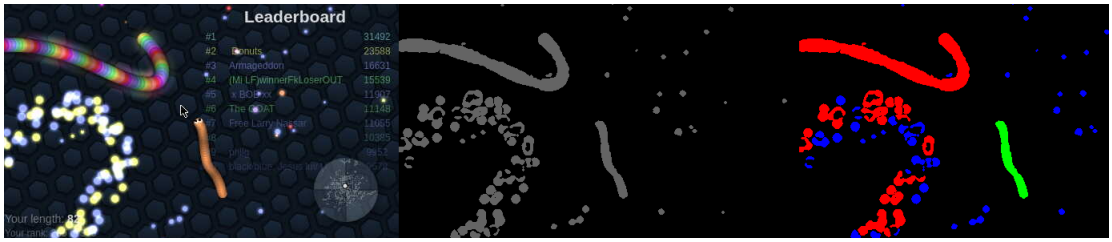

Figure 1: All stages of Recognizing Components
From left to right. 1. the original gaming image of slither 2. Filtered image of features. 3. Food, enemy, and user snake are stratified into 3 different color

2.2 Slither Processor

2.2.1 Image pre-processing

Since the data outputs of OpenAI Universe are screen pixels, a preprocessing is necessary for the implementation of our approximate Q learning agent. 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 around 500x300 pixels. We then implemented a processor to generate more representative states/observations by incorporating the previous work of Zach Barnes et al[6] who stratified the pixel output into 3 different layers of pellets and the player's snake by removing the background, recognizing the connected components, and processing the shapes of the components. The RGB image was transformed to grayscale and thresholded to eliminate the background hexagons as seen in **figure 1** above.

- **Player's snake**: as we mentioned above, the game screen width and height are (500, 300) so the connected components located at the center of the screen (145:155, 245:255) are identified as player's snake and labelled as green.
- **Pellets**: The groups of connected components with size are smaller than 235 are identified as the enemy snake and labelled as blue.
- **Enemy's snake**: the rest of the groups of connected components of which size are larger than 235 are identified as opponent and labelled into red.

2.2.2 Feature extraction

After the vision processing is done, we further implement an extra layer as feature processor which generates all the features according to the current observations in each direction as described in the approach section with the following helpler functions: extract_features, get_closest_loc, dodge_snake, and dis_per_in_area.

3 Problem Specification

We aim to implement a reinforcement learning agent to play the Slither.io game and optimize the performance by dynamic learning and interaction with each condition.

4 Approach

4.1 Rule-Based Agent

In this approach, we implement a rule-based agent that would react to the environment as following 1) dodging the enemy snake while there is a snake nearby trying to survive. 2) If there are no opponents in the area, searching the nearest food instead. The helpler functions used in this rule-based agent are as following:

- **get_closest_loc**: This a simple function that takes an list of coordination of components as input, calculate the euclidean distance according to the me_snake location, center of the screen (270. 235), and return the tuple of closest coordination as an output.

- **dodge_snake**: This function takes two list of closest location of the food and the enemy snake obtained from the **get_closest_loc** function. If there's enemy snake in the screen, we return the opposite direction of the closest enemy snake. If not, we head for the closest food.

4.2 Approximate Q learning Agent

Algorithm 1 Approximate Q learning

for each action **do**
$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + ... + w_m f_m(s,a)$
end for

procedure UPDATE Q VALUES
$difference = [reward + \gamma MaxQ(s',a')] - Q(s,a)$
$Q(s,a) \leftarrow \alpha[difference]$
end procedure

4.2.1 State/Observation

Approximate Q-Learning is an off-policy, model free algorithm which refines the policy greedily by the maximum Q values of the action. However, Slither.io is a game considered as a partially observable Markov decision process (POMDP), were not able to get information of the true state. Therefore, we extract the feature value of each action directly.

4.2.2 Action

The head of the player's snake is controlled by the mouse which has numerous direction in 360 degree. To simplified the model, We limited the action into eight directions (right, up_right, up, up_left, left, down_left, down, and down_right), twelve directions and thirty-six directions for different experiments.

4.2.3 Features

Two main features that can be extracted from the current settings are the food and the snake pixel output. All the features we used are extended from these two values and we will describe the details in the following helpler functions used in the approximate Q learning approach:

- **dis_per_in_area**: This function first divide the screen into 8, 12, 36 parts depends on how many action directions we have in the experiments. The center of the screen (270, 235) served as the center of the circle. Due to the rectangle shape of the screen, each part doesn't have the same area but each arc has the same degree 45, 30, 10 obtained by 360 degree divided by the number of the part. Then, we calculate the closet distance to the food_pix and snake_pix the by euclidean distance in each part. We also sum up the food_pix and snake_pix in each part as the percentage of food and snake in these parts. We return four numpy array of minimum distance to food and snake and percentage of food and snake in each part as output.

- **extract_features**: This function takes the processed frame, a numpy array, generated by the connected components and further transformed all features. Basically, there are three layers being processed here, **food_pix**, **me_pix**, and **snake_pix**. Based on the coordination within the frame, we convert it into representative features as an output for each observation for the updating. The area of each zone is corresponding to the direction of the action in the screen divided area. The neighbor area is defined by the average value of the two adjacent area of the action area (as shown in Fig 2). The output contains up to ten features (ten numpy array) including the the closest distance to snake in the area, closest distance to food in the area, percentage of snake in the area, percentage of food in the area, closet distance to snake in the area less than 50 (dummy variable), and closet distance to snake in the area less than 100 (dummy variable), closest distance to snake in the neighbor area, closest distance to food in the neighbor area, percentage of snake in the neighbor area, percentage of food in the neighbor area. The first 6 features are used in the first two experiments, and all the ten features are used in the last experiment.

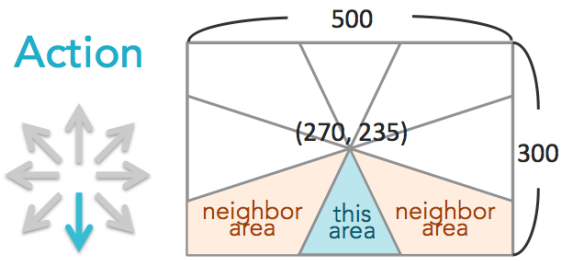

Figure 2: Demonstration of one action and the associated features in the area

5 Experiment

5.1 CropScreen/Features

Initially, we consider one single observation, the 500 x 300 game screen, per timestep, that is to say, the features of closest enemy does not contain information about the direction. There is no difference that the enemy snake appears on the top right or the top left since we only return the closet distance to the opponent. So does the feature of food. Accordingly, without knowing the direction of the closet enemy or food, the agent performs no better than random exploration since the Q-value for each direction is the same. The result is shown as **figure1**. In order to solve the issue, we implement a fan-out method as described in **Approach** to crop the game screen according to the possible directions of our agent. With this transition, the agent would be able to choose the action with the best Q-value among all the possible directions. The survival time and total reward improve significantly after more representative information is given to the agent.

5.2 Reward shaping

As reward shaping has been proven to be an effective technique when applying Reinforcement Learning in a complicated environment [9], we modify the gaming reward for each episode. In

the gaming environment of Slither.io, rewards are sparse and delayed so we apply immediate rewards in addition to basic rewards earned by consuming pellets on the map. The shaped reward are listed as below

Condition	Reward
$r > 0$	$r* = 2$
$r = 0$	$r = -1$
action != action'	$r+ = 1$
Done/Die	$r = -10$

The above reward shaping is applied on every training time step. If the agent receives positive rewards, the Q learning agent would further double it. To encourage eating pellets rather than simply avoiding enemy , We penalize every move it takes without reward. Exploration are rewarded as well, if the next move that the agent takes is not identical to the last action recorded, it would get additional bonus. The application of reward shaping enhances exploration and eating pellets and penalizes repeated actions, which helps the agent learn that it is applicable to choose different actions. Lastly, the agent receives a -10 penalty once the observation is done(reaching the border, killed by enemy , internet connection lost)

5.3 Parameter Tunning

As we know, learning rate α controls how much of the old estimate we keep. $Q(s,a) = Q(s,a) + \alpha(R(s) + (s,a)Q(s,a))$. The converge time are also dependent to learning rate[2]. We therefore test different learning rate by specifying different setting, 0.2, 0.5, 0.7, 1(greedy). The result are described as **table1** below.

6 Result

6.1 Different learning rate

First, we train the agent by 8 directions and 6 features (closest distance to food and snake, percentage of food and snake, and two dummy features) with different learning rate from 0.2 to 1.0. We evaluate the performance by two parameters: the reward and the number of time steps. The reward means the maximum reward of each episode return by the universe environment when the player's snake are dead; the number of time steps represents the longest survival time of each episode. Since Slither.io is an online multi-player's game, it is hard to predict other players' (enemy snakes) next movement or strategy, which makes the environment extremely unstable and complicated. This explained to the spiking curve of rewards and time steps (Fig 3). Due to the unstable environment and high possibility of erroneous image pre-processing, we excluded the episode with zero reward. Using learning rate 1.0 (greedy) yield the best result of mean rewards (63.91), whereas learning rate 0.5 has the best result of mean and maximum survival time steps (186 and 894). More details are shown in Table 1.

	Mean Reward	Max Reward	Mean Time Steps	Max Time Steps
Learning rate α				
0.2	53.04	435	127.41	660
0.5	52.71	256	186.00	894
0.7	47.09	217	154.00	846
1	63.91	266	160.19	812

Table 1: Result of 75 episodes

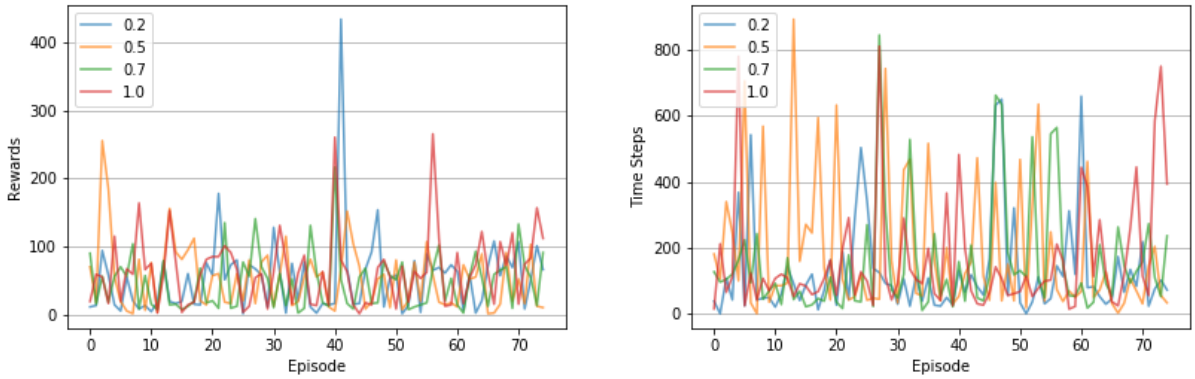

Figure 3: 75 episodes of rewards and time steps with different number of directions

6.2 Different directions

Next, we increase the number of directions, in order to control the player's snake more precisely. Regardless the striking curve of rewards, we can notice that the local peak of the rewards are gradually increased, indicating the performance are gradually improved throughout the training course. The mean The survival time steps for 8, 12, 36 directions are 273.00, 283.06 and 251.56 respectively and are all better than the rule-based approach. Another interesting finding is that after increasing the number of directions to 36, the mean and maximum rewards decreased, we will discuss this in the discussion section. More details are shown in Table 2 and Fig 4.

6.3 Different features

We also try to improve the performance of Q learning by adding four addition features (the closest distance to food and snake in neighbor area, and the percentage of food and snake in neighbor area). As the result shown above, 8 direction has the best rewards, therefore, we use 8 direction in this experiments. The mean rewards and time steps of 6 features and 10 features are 60.12, 48.34, 273.00, 267.82 respectively. However, after adding four additional features, the performance are decreased. This could be explained by using the neighbor area might result in similar features of different action. More details are shown in Table 3 and Fig 5.

	Mean Reward	Max Reward	Mean Time Steps	Max Time Steps
Rule-Based	52.54	160	219.10	506
Approximate Q learning				
8 directions, 6 features	60.12	225	273.00	691
12 directions, 6 features	57.16	554	283.06	708
36 directions, 6 features	50.58	129	251.56	952

Table 2: Results of 50 episodes with different number of direction

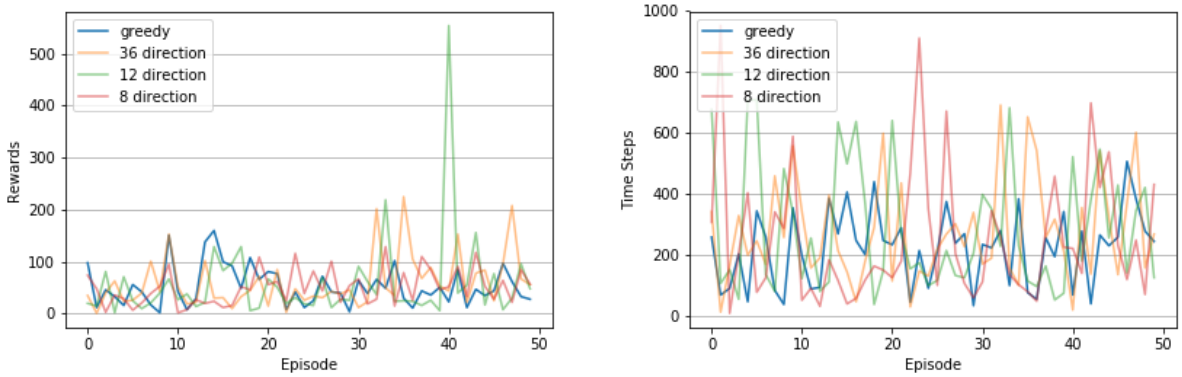

Figure 4: 50 episodes of rewards and time steps with different number of directions

	Mean Reward	Max Reward	Mean Time Steps	Max Time Steps
Rule-Based	52.54	160	219.10	506
Approximate Q learning				
8 directions, 6 features	60.12	225	273.00	691
8 directions, 10 features	48.34	225	267.82	998

Table 3: Results of 50 episodes with different number of features

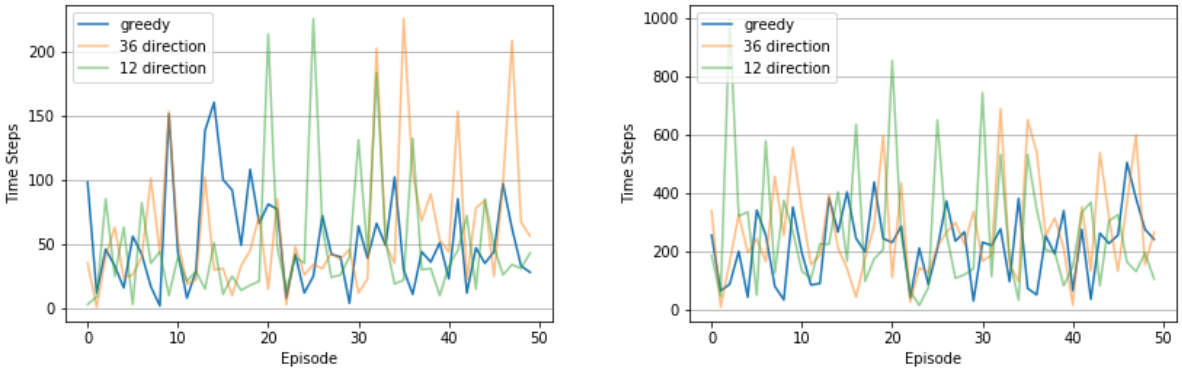

Figure 5: 50 episodes of rewards and time steps with different number of features

7 Discussion

The report had an objective to "win", eat as much as pellets as one can and survive as long as possible, in the game of slither.io. using OpenAI universe and approximate Q learning agent. After modifying features and tuning parameters, the best method is using 8 directions and 6 features. However, as the result shown above, it's not as competitive as human beings or tampermonkey bot[1].

Interestingly, during the process of increment features and directions, the performance gradually decreases as we increase the number of directions from 8 to 36. One of the explanation is that When the number of actions increased, the area of each part decreased as well. Also, considering the fact that features of the adjacent $\pm$ 45°areas are included when using 8 directions and 10 features, nearby actions e.g. action (0°- 45°) & action (45°- 90°) would have the similar features and Q values. After several episode/iteration, the Q values for each action become all the same, and the agent chooses the first action which is head to the right. To overcome this issue, we abandon the approach that adding additional four features in the neighbor area and avoid dividing the screen into more than 12 directions. In terms of feature selection, all the features that we used are based on the food and the snake pixel. As a consequence, when we increase the number of features, it decreases the diversity of features of different action and thus reduces the performance of the model. However, due to the environmental limitations, we can't obtain more information from the observation.

Compared to an agent always knows with certainty the environment state in MDPs, our slither agent does not directly observe the environment's state, the agent has to make decisions under uncertainty of the true environment state. This might explain why our result doesn't show significant improvement after episodes of training, we're updating the weights based on the observation emitted by the true gaming state. That being said, even though our learning agent performs better than rule-based one, the result, weights, might be biased. Accordingly, an implementation of POMDPs might improve the result but solving a POMDP is often intractable due to their complexity[4].

In addition to features selection and parameter tuning, one of the major limitations of our model is that the rule-based image vision processor sometimes, 5%-15%, might not be able to recognize the components correctly. 1) some of the enemy snakes with customized skins have the same colors as the dark blue background. 2) A giant floating pellet would be recognized as enemies due to the size that large than 235. 3) The enemy snake that moves to the nearby area of the center would be misrecognized as player's snake. 4) Part of the body of the enemy would be classified as food. As a consequence, the efficiency and correctness of the learning process would be affected. However, it's extremely difficult to find the perfect parameters for the rule-based processor due to the possible combinations of the skin of the enemy snakes. One approach is to implement another machine learning identifier beforehand but it equally requires a considerable amount of time. Another constraint is that instability of the gaming environment. Since it's a multi-player game, internet latency and lost of connection hinder the learning efficiency as well. We have to check the gaming status and reconnect manually every time if OpenAI VNC connection is lost.

One method potential method to improve the performance is to combine these two approaches we used **rule-based and approximate Q learning**, applying the on-policy SARSA algorithm to find the optimal policy with the updated rules. Instead of updating by the maximum Q values

of the action, we can use this "greedy selection policy" to update the Q values. After sufficient training episodes, SARSA and Q learning might converge to a similar result. Besides, one of the advantages of using the updating method is that the total training time would possibly be reduced.

As we know, deep reinforcement learning has dominated over controllable environments, where agents are trained against manipulable and predictable computer opponents in games like Atari Games[5]. A combination of DQN and convolutional neural networks which shows remarkable performance on image recognition might also be a solution to the game like slither.io, even though a considerable amount of training time and high computational power are required.

Additionally, our agent is still a visual responsive bot that lacks tactic and aggressiveness. Ideally, the agent should learn how to attack other players to earn more potential reward from others' death, rather than just surviving, however, we shall leave this part to future works.

In conclusion, we successfully implement an approximate Q learning agent based on the Partial Observed Markov Decision Process (POMDP) environment. We demonstrated that the proposed Q learning agent could evolve and adapt to each timestep. Even though the performance of the approximate Q-learning approach is not as competitive as human beings so far, we believe that the architecture of our learning agent can be expanded and applied to different environments and that it is suitable in the sense of training more intelligent agents that are capable of interacting with humans in an intellectual way.

A System Description - Installation and Testing

- Install Conda
- Create and activate Conda Env

```
conda create --name slither python=3.5
source activate slither
```

- Install needed packages(meant for ubuntu VM):

```
sudo apt-get update
sudo apt-get install -y tmux htop cmake golang libjpeg-dev libgtk2.0-0
    ffmpeg
```

- Install universe, universe installation dependencies

```
pip install numpy
pip install universe
```

- Install codebase and packages

```
git clone https://github.com/q8888620002/slitherRL.git
cd slitherRL
pip install -r requirements.txt
```

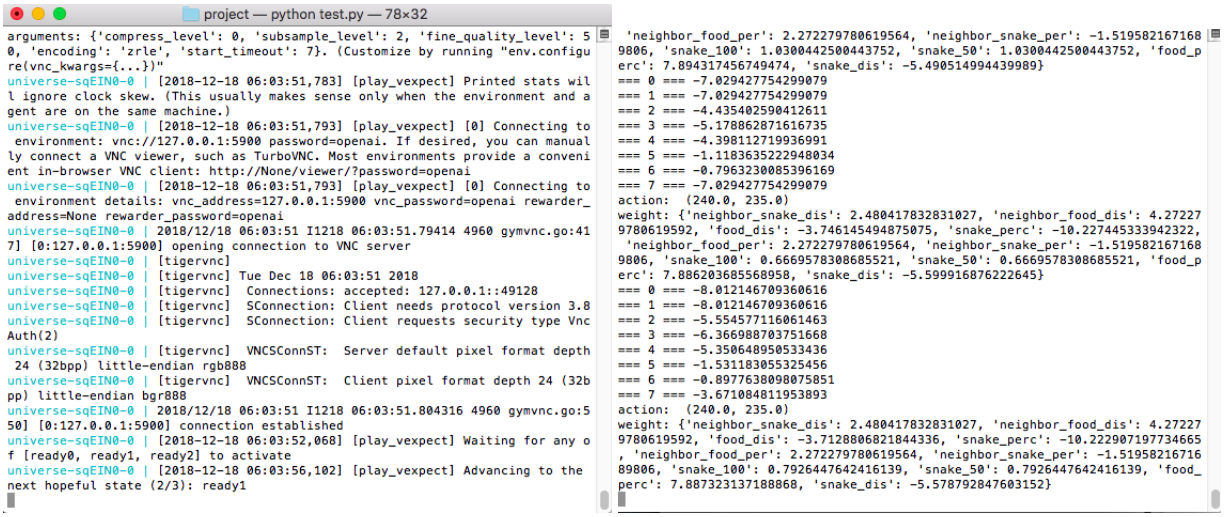

Figure 6: OpenCV connection & Features output

- Install docker
- Restart VM
- Test installation
- Run the test agent script

```
python test.py
```

- After the first episode, trained weights woulbd be saved as a pickle file. If one wants to use the trained weight saved, just uncomment the code is test.py.

```
learning_agent.weights = pickle.load(open('weights.pickle', 'rb'))
```

Once the installation completes, the a list of features and weights would show up in command lineas as **figure 6**. A screen of gameplay of slither agent would show up as well.

B Group Makeup

Appendix 2 A list of each project participant and that participants contributions to the project. If the division of work varies significantly from the project proposal, provide a brief explanation. Your code should be clearly documented.

	Mingyu Lu	Chihying Deng	Hsu Wei-Hung
Project Proposal	33%	33%	33%
Code Implementation	50%	25%	25%
	Learning agent implementation	Image Cropping	Code refactoring
	OpenAI Universe Backend	Features extraction	Features modifying
	SlitherImageProcessor		
	Features modifying		
Training Model	30%	30%	40%
Final Report	60%	35%	5%

References

[1] Ermiya Eskandary Thophile Cailliau. https://github.com/ermiyaeskandary/slither.io-bot. 2016.

[2] Yishay Mansour Eyal Even-Dar. Learning rates for q-learning. *Journal of Machine Learning Research*, 9(8):1735–1780, 2003.

[3] Marek Grzes. Reward shaping in episodic reinforcement learning. *Neural computation*, 9(8):1735–1780, 2017.

[4] Sbastien Paquet Brahim Chaib-draa Stphane Ross, Joelle Pineau. Online planning algorithms for pomdps. *Journal Of Artificial Intelligence Research*, 9(8):1735–1780, 2014.

[5] David Silver1* Andrei A. Rusu1 et al. Volodymyr Mnih1*, Koray Kavukcuoglu1*. Human-level control through deep reinforcement learning. *Nature*, doi:10.1038/nature14236, 2015.

[6] Frank Cipollone. Zach Barnes, Tyler Romero. Rattle: a slither.io reinforcement learning agent. 06 2017.