Snake

Sophie Horwitz, Arinah Karim, Matthew Compton, Sabrina Blumberg

April 2021

1 Project Description

For our project, we decided to create the Snake game that will teach itself to pass through as many game levels as possible with reinforcement learning. The rules of the Snake game are to get the head of the "snake" to overlap the targeted square. The "snake" begins as a single square and gets one square longer for each targeted unit (the 'food') it overlaps, and the game ends when the snake hits a wall or into any part of itself. Once the targeted square is 'eaten', a new food unit will be randomly placed on the board while the snake stays in the same position, and the game will continue this process until it meets the ending game requirements. The main challenge is making sure the snake does not run into itself or the wall since more space will be occupied by the snake as it eats more food units which will leave less room for it to maneuver around the game board. The way that the AI chooses to move directly influences the future state of the game; for example, the board (represented by pixels/blocks), becomes occupied by the body of the snake over time. The path that the snake will have to take will be different depending on its size and the snake will have to recognize itself continuously. Additionally, since the food is generally randomly generated across the board, paths will have to be different depending on the randomization of the reward as well. Pixels do not support diagonal paths and rely on finding linear routes through corner turns so finding the corners to turn that minimize the risk of impact is important. We recreated the single player version of the game where the machine learns to play by utilizing Python and some of its libraries such as PyGame, numpy, random, pickle, queue and used a board of 200 x 250 pixels.

2 Algorithms and Design

We are using an algorithm called QLearning which is a type of reinforcement learning. Because it is a form of reinforcement learning, there is an AI agent, states, rewards, and actions. But unlike various other reinforcement learning algorithms, QLearning uses model-free environments so the agent does not try to learn or understand mathematical formulae or statistics. Instead, the agent interacts with the environment directly. The agent explores the environment

with different approaches and learns through trial and error and updates its understanding of the environment with what it has learned through those trials and errors. QLearning has additional characteristics on top of the characteristics it has as a reinforcement learning algorithm. One characteristic is that the number of possible states is finite so the AI agent will be in one of the many possible, fixed situations. Another characteristic is that the number of possible actions is finite so the AI agent has to make choices on what action to take based on a set of fixed possible actions (left, right, up, down). The agent will begin at any state because it will appear on the board in any location. The agent will not know initially what the goal state(s) is/are, but it learns how to through QValues. QValues are represented in pairs where the first item is an action and the second item is the quality of the state with the given action. This is our heuristic as the QValues will give an estimate of the sum of future rewards. The agent will eventually be able to identify the highest quality action in any given state. The QValues will increase in correlation of the agent getting closer to achieving the highest reward (whether positive or negative rewards to minimize its total punishments) which is 0. QValues are usually stored in a QTable, which is acting like a policy for what actions should be taken by the agent, (but for our program, we are using a .txt file), which has one row for each possible state and a row for each possible action. In order to figure out what the Qvalue for an action taken in the previous state should be changed for the current state's action, QLearning uses temporal differences. This just means that the previous QValue will be updated after each step. For instance, if the current state provides the agent with a good reward, the previous QValue will increase; the opposite is true. The Bellman Equation calculates what new value should be used as the QValue for the action taken in the previous state. It uses the old QValue for the action of the previous state as well as what the agent has learned after going to the next state. The programmer can choose what the learning rate parameter can be to adjust how quickly the QValues are refined.

The algorithm starts with initializing the QTable. Then an action is chosen from the QTable for the current state, which in most cases is whatever state has the highest value. This is where we use an epsilon where the agent will choose the action with the highest QValue a certain percentage of the time and randomly for 1 - the percentage mentioned before. By doing this, the agent will explore less promising paths initially, but in the long term it will learn a better course of action. After the agent has decided what action to take, it will move on to the next state. The agent will receive a reward (again, either positive or negative) for the action previously taken and use that reward with the knowledge of our new state to compute the temporal difference for the previous action. Then, the Bellman equation is used to update the QValue for the most recent action. This process repeats with the agent choosing an action for the current state until a terminating state is reached. After the agent has reached a terminating state, the agent is moved to an initial state and the process starts again with the updated QValues. Eventually, the model will be fully trained and the QValues will not be updated. The agent will take the actions with the

highest QValues. Because the number of states the agent can be in and based on how restrictive the environment is, the time and space complexity will be relatively related to the number of spaces on the board. The lower bound on these complexities are:

$$O(n)$$

. This takes into account the need to store the reward.

In the description in above, it was mentioned that there were two additional characteristics of QLearning on top of its characteristics as a reinforcement learning algorithm. These were the finite states and the number of actions taken. These are both a limitation as this hinders the exploration the agent can do in the environment. Another limitation is the QTable as it assumes that all states and actions can be represented in a matrix which can cause issues when the number of actions and states become very large. Some alternatives we were considering include, but are not limited to, are Deep QLearning and BFS (which would be used for comparing the algorithms). We plan on exploring Deep QLearning if the time permits. There is also another variant of QLearning such as double QLearning and maybe we could compare this to QLearning and Deep QLearning.

3 Solution as a Human Model

Our solution imitates the way a human would think through negative and positive reinforcement. The AI is rewarded with points based on their actions. If the system hits the target it gains 10 points while it will lose 1 point for landing on empty space and lose 5 points for ending the game. The goal is for the game to get as many points as possible or the smallest negative number possible. Similar to a person, the AI tries to win by receiving a reward for what others deem as good actions, which in a human would initiate a positive response. Both the AI and a person are motivated by the reward so they would attempt to repeat the actions that earned them the points. Losing points would cause a similar reaction as well. Both the AI and a human will try to prevent losing points since they are punished for wrong actions. In doing so, both the computer and person would want to avoid the actions that would reduce their points or subtract from what they previously earned.

4 Empirical Analysis

Trial	QLearning Average Score
1	0.1
5	0.3
10	0.5
15	0.9

These results were formed from a QLearning algorithm with the snake game. Since this is one of the most basic algorithms, it is expected for the agent to

learn very slowly. Each trial ran 10 times to collect a reasonable amount of data on how the snake performed. In this reward system, eating an apple gives 10 points while dying is -5, and if the snake is moving in blank spaces it awards -1 to encourage it to find the food. Seeing that the first trial had an average score of 0.1 food eaten and the last had 0.9, one can see a gradual increase in learning. Similar to QLearning, a better algorithm to implement would be Deep QLearning. Instead of using a QTable, txt file in our implementation, Deep QLearning replaces this with a neural network.

Trial	BFS (Score per Trial)
1	68
2	53
3	71

As one can see, BFS initially is much more efficient than QLearning. These results were completely expected. This search algorithm will look much better in the beginning since it does not have to train the data. Theoretically, QLearning will run as good as, if not better, than BFS search once it has learned how the game works. This brings about a difficult question for the problem space. Since QLearning (without a neural network) is much less efficient in the beginning, is it better to implement a search algorithm? BFS is good in this situation, but A* search algorithms have been proven to consistently perform better.

5 External Resources

In our program, pickle is a library similar to JSON that is used to mainy keep persistent data. It converts objects into a byte stream that then can be stored. Pickle is very easy to use and can quickly be imported into Python. The code below simply opens the created txt file and then uses "rb", opens the binary final in read mode, to then be able to easily have pickle open it.

```
infile = open("LearnedData.txt", "rb") #read binary
self.Q = pickle.load(infile)
```

Below is the pickle.dump() command. The first argument it takes is what we want to convert to binary data and the second is where we want the converted data to be saved. In this case the Q needs to be saved in file "f" which is the "LearnedData.txt" file. Of course after this process is finished, the file needs to be closed so no errors occur. The last line below is performing this action.

```
f = open("LearnedData.txt", "wb") #write binary
pickle.dump(self.Q, f)
f.close()
```

https://www.datacamp.com/community/tutorials/pickle-python-tutorial

BFS Search is another algorithm implemented in this project. In order to make this work, BFS takes advantage of a queue. In Python there is a queue module that can make this very simple. A queue works on the idea of FIFO, or First in First out. These inbuilt Python functions make BFS much easier to

implement. Below, the code shows a constructor for the FIFO queue. Maxsize simply sets the upper bound limit for the number of items that can be inserted into the queue. 0 in this situation means infinity.

```
Q = queue.Queue(maxsize = 0)
```

https://www.tutorialspoint.com/stack-and-queue-in-python-using-queue-module

Below is a link to a page describing the BFS algorithm. For this project, a lot of inspiration for BFS was used from this source. It helped describe how to implement it correctly and efficiently. Between the source below and Python's queue module, they both helped create this algorithm.

https://pythoninwonderland.wordpress.com/2017/03/18/how-to-implement-breadth-first-search-in-python/

6 Conclusion and Additional Considerations

All in all, the AI does adapt to its environment fairly quickly and does eventually learn how to get the food. However, trying to surpass 5 or 6 points would get more and more difficult almost exponentially as time goes on- the reason being that the snake's body would continue to grow and the way the algorithm adapts would be slow to avoid self collision, especially when near completion. In order to help alleviate the burden a higher level would cause, giving reward to having a larger amount of non-enclosed space would be a good way to ensure better future results. Non-enclosed space refers to the space that the snake does not cover with its body- by minimizing the space (by going in looping or closely knit patterns), the AI will be more efficient in solving the problems and eliminate some common defeats in the late stage game. All in all, we learned that while writing the initial game and learning algorithm was not difficult, getting the algorithm to give the snake the optimal chance of winning was difficult to fine-tune, and required forethought to the design of the game and the way that the AI would eventually learn to handle the board.