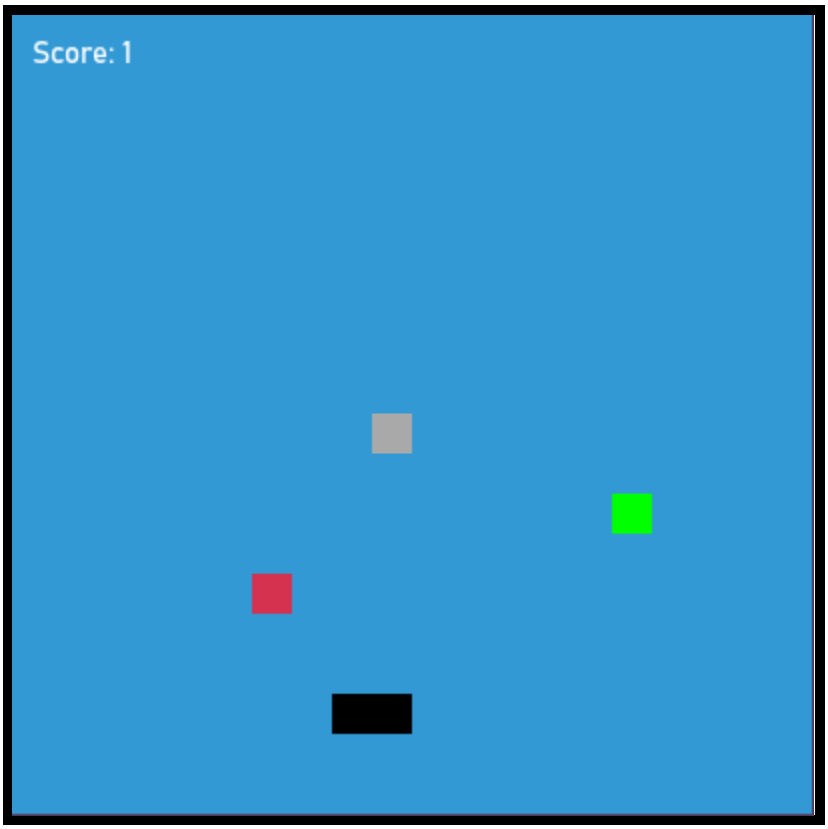
Implementing Q-Learning for Snake

CS 4100 - Artificial Intelligence



Team Members: Pavan Hirpara, Gary Hu, Ben Newfield
Professor: Kevin Gold / Nate Derbinsky
TA: Gail Renee Pinto

Introduction

Since the dawn of arcades, Snake has been a famous game concept with many different versions and adaptations. In this project, we were tasked with implementing the various AI concepts we learned this semester to play the Snake Game. Specifically, we utilized Q-learning to have our agent learn as they played.

In addition to the base concept of Snake, we introduced new elements such as bad fruit and walls to see how the agent would handle these new obstacles. The bad fruit randomly moves whenever the player eats a good fruit, and would result in a -2 score penalty, or death if the player's score was less than 2. The walls spawn in a random location whenever the player eats a good fruit and result in a death (similar to running off the screen or self-collision). These new elements allowed us to experiment with parameters of the learning, explore a unique version of a classic game, and consider the question: how well can Q-learning navigate a dynamically growing "maze"?

Throughout the process of training, we encountered many challenges, including identifying the optimal parameters for Q learning, and adapting existing code to work with these new elements. Furthermore, the spawning of walls and bad fruit led to an element of randomness for the agent that we attempted to work around, since sometimes the agent would get lucky with a run, while other times, the agent would die within the first few steps. Taking into consideration all these challenges, we choose to represent our state space as the direction the food is in relation to the snake, and the direct surroundings of the head of the snake, in order to reduce the state size, improve performance, and simplify the problem at hand.

Experimental Results

The first alteration we made was connected to the state space of the problem. Originally, the snake only had knowledge of whether a space had a wall or bad fruit in it. In our state representation, we have a 4 long string of 0's and 1's that correspond to the direct surroundings of the snake head (in the four cardinal directions). If one of the four spaces around the snake's head had a wall, bad fruit, or snake in it, or was off screen, then that character would be a 1, and if it was a free space, the character would be a 0. For example a snake completely surrounded by itself would have the state representation of 1111, while a snake that had 2 walls around it would be something like 1010.

What we wanted to investigate was if we introduced a third character "2" into our state representations that stood for there being a bad fruit in the tile, how would this impact the performance of our agent. To our surprise, trials with the snake being able to tell the difference between walls and bad fruit performed worse than those that treated them the same. Shown below is the performance of each of those implementations.

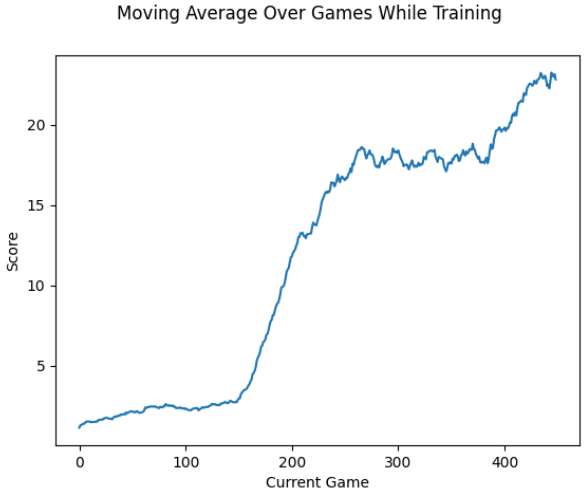
Performance of 0, 1 State Representation

Performance of 0, 1, 2 State Representation

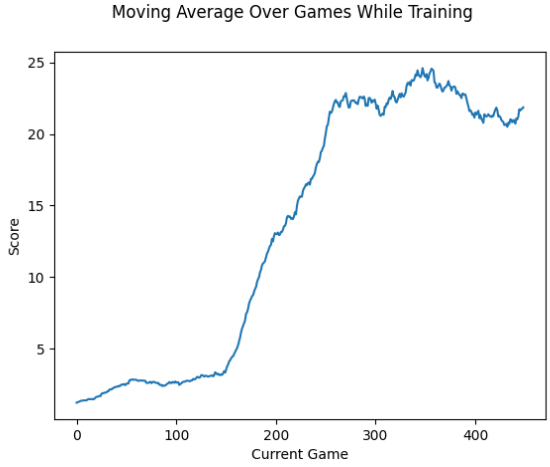
As we can see, introducing a third symbol to represent bad food into our states had the opposite effect. Instead of improving our snake's performance, it reduced the average score by a noticeable amount. This is likely due to the fact that the snake is now taking bad fruit more often in the 0, 1, 2 representation in order to avoid walls and tiles with a larger negative reward. However, since the snake takes the bad food more often, the score will go down, even if the snake survives longer. The preferred method for the snake in the first representation seems to be avoiding bad food all together since it will see it as a wall/snake, thus increasing the average score.

The second experiment we performed was altering the learning rate of the agent. We tried three different rates (0.5, 0.75, 0.9), as shown below and concluded that 0.75 was the best learning rate for the snake, but there was only minimal improvement from the other rates. The importance of the state representation and other parameters seemed to outweigh the learning rate.

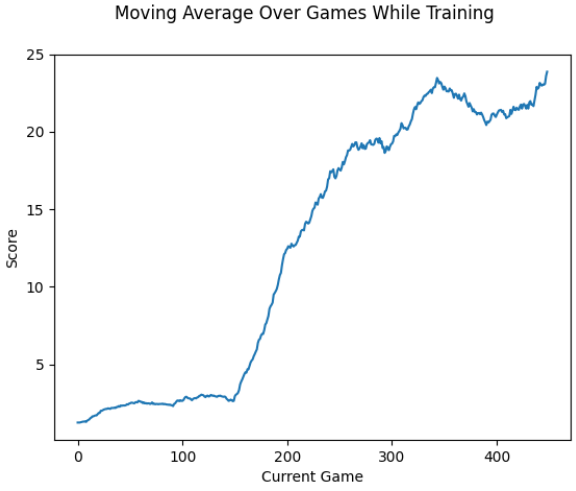
- Performance for Different Learning Rates
  - Learning Rate of 0.5



  - Learning Rate of 0.75

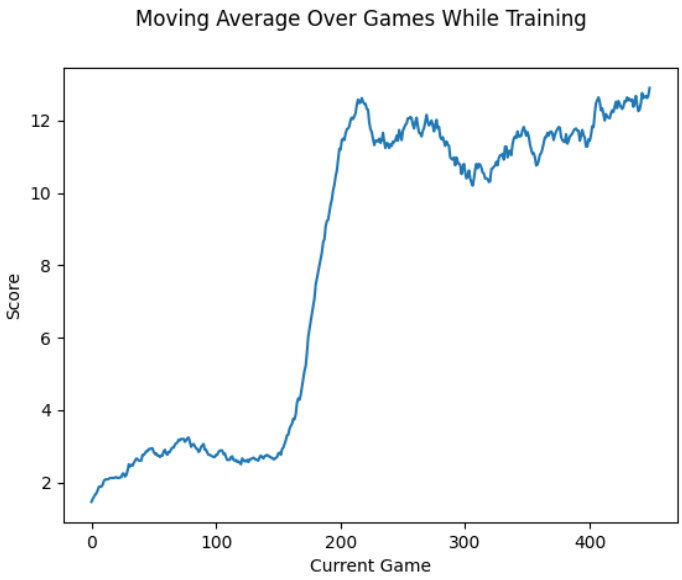


  - Learning Rate of 0.9

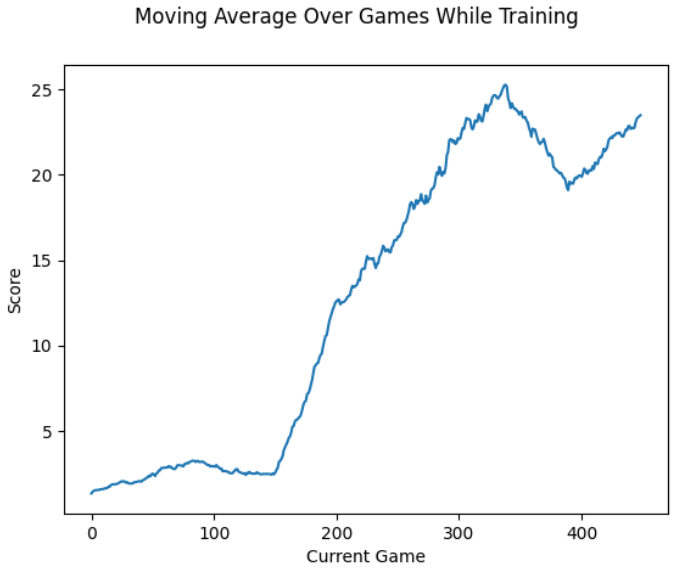


The final experiment that we conducted was altering the reward values to see what would perform the best. We tried three main options: all rewards were of the same magnitude, but with different signs; slightly different reward magnitudes; vastly different reward magnitudes. Shown below are the results of each of the cases, with specific reward values.

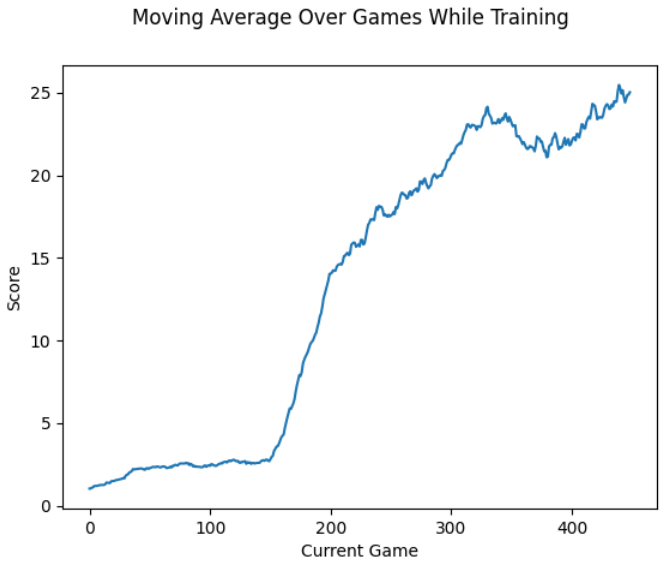
- Case 1
    - Death : -1
    - Eating Good Food : 1
    - Eating Bad Food : -1
    - Moving away from Good Food : -1
    - Moving towards Good Food: 1



- Case 2
    - Death : -3
    - Eating Good Food : 2
    - Eating Bad Food : -1.5
    - Moving away from Good Food : -1
    - Moving towards Good Food: 1

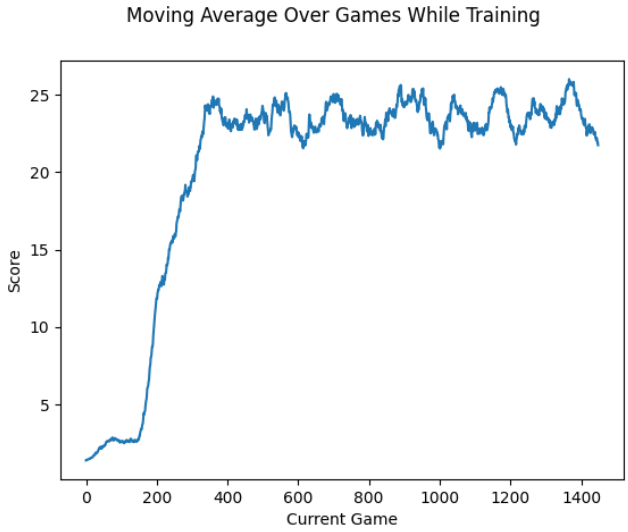


- Case 3
    - Death : -100
    - Eating Good Food : 50
    - Eating Bad Food : -25
    - Moving away from Good Food : -15
    - Moving towards Good Food: 15



As shown above, case 3 had the best performance, just beating case 2. This is likely due to the fact that the reward values correctly reflected what the snake would want. Specifically, eating bad food was a negative reward but not as bad as death, and moving away from the good food was not as bad as eating the bad food. By choosing reward values that made sense for what we wanted the snake to do, the snake was able to perform much better than if the reward values were all of the same magnitude.

After altering the three prior-mentioned aspects of our agent, we were able to reach the performance shown below.



Conclusions

From this process, we learned that the state representation and reward values had a substantial impact on the performance of the snake, maybe even more so than the individual parameters. Q-learning is able to choose the correct action to take, as long as the structure of the states and rewards are thoroughly defined, and match what we would expect the snake to do. However, as we increase the state space, we risk sacrificing simplicity and more importantly performance, so it is important to think of what exactly is necessary for the agent to know before creating our state representatives.

In the future, we can consider using machine learning methods (such as gradient descent) to determine the optimal parameters (such as learning rate, epsilon, discount, etc.) and reward values, so that the only thing we have to worry about is the state representation. Utilizing ML allows us to automatically choose the best options without having to manually go through them all. Overall, learning how to implement Q-learning with the Snake game has taught us the importance of correctly defining your state space, and how different representations will have a noticeable effect on the behavior of the agent.

Appendix

All code presented in this project was done in Python 3 and code that was adapted from others has been cited in the code. The codebase consists of 3 Python files, “InitializeQVals.py”, “AI_Snake.py”, and “Agent.py” and also a JSON file “qvalues.json”. Most packages we utilized are already present in the base Python version, but packages such as “pygame” or “matplotlib”

needed to be installed prior to running any code. If there is a module missing from your Python version, simply use “python -m pip install -U pygame --user” in the command line to install the package. Any other packages follow this same command (but substitute pygame with the package name), so one can either use this method, or any other preferred method for installing packages.

Once all the required packages are installed, you will be able to run the code with the following process. If you would like to start learning from scratch, use “python InitializeQVals.py” to reset the JSON file containing the Q values. This is not necessary if you would like to continue learning from where you left off. Once you have your Q values initialized, you can run the snake game using the following command “python AI_Snake.py”, at which point you should see the snake game playing. Any changes to parameters for learning must be changed directly in the code (usually “Agent.py”). Listed below are important variables for testing and experimenting:

AI_Snake.py:

- q_vals_n : Determines how many trials before Q values are updated
- snake_speed : Determines the speed of the snake
- max_move_count : How many moves the snake is allowed in each run, preventing infinite loops
- total_batches: How many times the Q values will be updated before displaying results from trials
- agent.epsilon: This is the epsilon for the agent, and unlike the other variables, this can change over time, or remain stationary (see Experimental Results)
- draw_display : Determines whether to draw the snake game or not while training

Agent.py:

- self.epsilon : Same as the prior epsilon variable, but we do not alter this in the agent
- self.lr : Learning rate of agent
- self.discount : Discount over time
- self.reward_array : Array of rewards for actions (specific description present in code)