INTRODUCTION

The focus of this project is to explore reinforcement learning by the means of a hands-on project; the basis of the code for developing the game is from python-engineer whose github is linked at the end of the report. I chose to use the game “snake” because of it’s simple but difficult style of play. My overall objective is to build an AI that will ultimately be able to play the game reasonably well; however, my primary learning goals focused on trying out different parameters to see how they would affect the way that the agent played the game. I wanted to be able to break down the problem and form predictions on what parameters would have the biggest impact on the performance and subsequently check to see if the behavior aligned with expectations.

BACKGROUND

Essentially, snake is a 2d survival game on a grid where the player controls a “snake” and chooses the direction that it moves in. The “snake” starts off as a block and once the game starts it moves in a linear motion at a predetermined speed. An important thing to note is that there’s no stopping at all once the game starts; the snake is constantly propelled forward, and the goal is to get the highest score possible without letting the snake run into the wall or run into itself. In order to increase the score, the snake needs to eat “food” which are the red blocks on the grid. As the snake moves around the grid, if its location overlaps with the piece of food, the block gets added on to the tail and the score increases by a set amount. In my case the score increased by 1 every time, and after the snake eats a piece of food, another one is randomly generated somewhere else on the grid.

The snake starts at a size of 1 block and gradually increases in length proportional to the number of food pieces that it eats. The size of the playing grid is also fixed so technically there is an upper limit for the highest score. It would be the length times the width – 1 because the snake starts off by occupying 1 spot with a score of 0. In my particular experiments, I used a grid of size 20 by 20 so the highest score that can be achieved is 399.

Figure 1. demonstrates that there are multiple different avenues that the snake can take in order to get the food. It’s almost impossible to calculate exactly how many different viable paths there are because the snake can always loop around infinitely without consequence. In the traditional game, as long as it doesn’t run into anything, it’s free to keep going. One thing that can be said for certain is that as the snake grows in size the number of viable paths decreases until eventually the snake either wins or encloses itself with no viable paths. If we take a conceptual examination of what happens as the snake grows, it becomes clear that each time a piece of food is eaten the number of valid positions on the grid decreases by 1. Another important consideration is that in my particular experiments I implemented a timer that would end the game if the snake would find itself in a long loop. This serves to remedy the issue of infinite loops since there were times in training that the snake would get stuck because there is not enough incentive to explore.

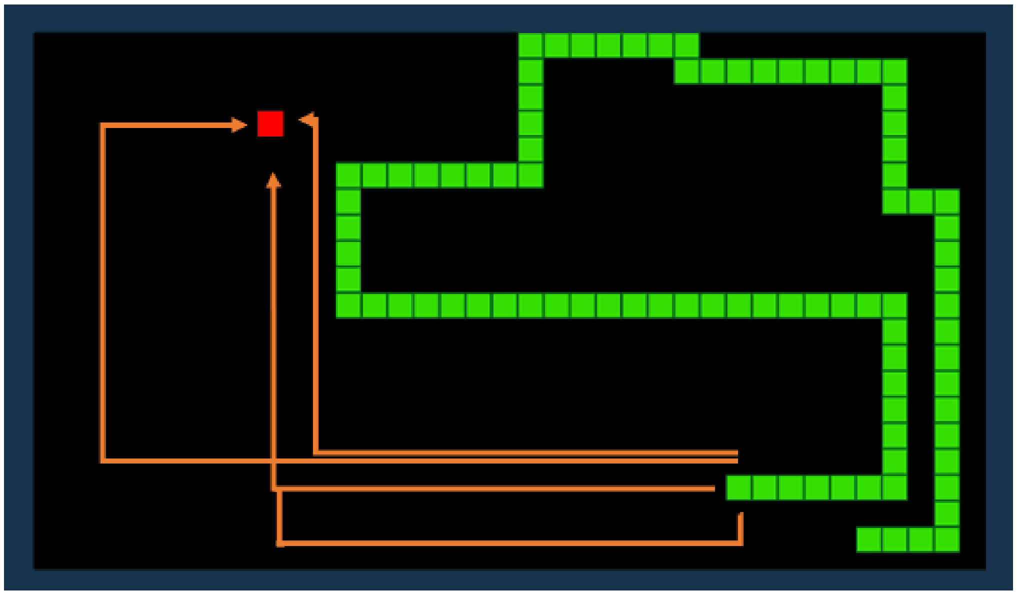


Figure 1. Demonstration of multiple paths

The reason why I decided to use deep reinforcement learning in order to tackle this problem is because I felt as though breaking down the problem into rewards for certain actions would be the most efficient way to go about learning as opposed to using something like supervised or unsupervised learning. Since it wouldn't really be feasible to have an "answer key" or have the model find patterns because of the nature of the problem. I was able to assign certain rewards for actions to encourage certain behaviors which I will discuss in the next slide.

REWARDS

When I was deciding how to allocate the rewards, I tried multiple different combinations to see which ones worked the best. In the case of eating the food the main two options that I was deciding between were whether to just reward the program when it successfully ate the food or whether to give it rewards for going towards the food and negative rewards for going farther away. The results, which can be seen in figure 2., clearly show that the superior option is simply rewarding the agent when it successfully ate the food. I think that this was because if I negatively rewarded it for moving away from the food then it would be reluctant to try more complex strategies. Sometimes moving away is advantageous because it positions the tail in a convenient location relative to the food and the direction that the snake is going. It's also advantageous sometimes to move away from the food because it allows for different attack angles; for example, if the southern side of the food is a wall it might be more effective to attack it from west to east instead of north to south. I think this was because of the reward being tied to actually getting the food and surviving for as long as possible after in order to continue the cycle. Again, this is not as much of a problem in the earlier stages but as the snake grows it gets enclosed in its tail much more often than if it would be able to move away from the food strategically without consequence.

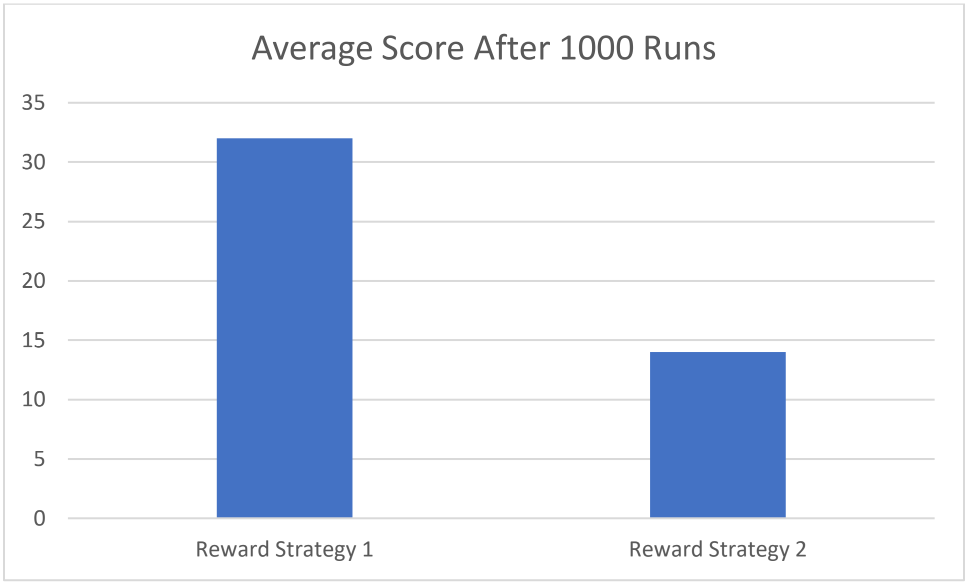


Figure 2. Comparing rewards structures

The next three rewards that were established were for hitting the wall, when the snake ran into its tail, and when the snake timed itself out. There wasn't much room for variation here – I just set a negative reward value for each action so the agent would try to avoid it as much as possible. Even though it was clear that those actions should yield a negative reward, I tested various values to determine how that value should scale to the rest of the rewards. The optimum value that was determined was -5 for each of them. When the value was a large negative number for hitting the wall and running into its tail, the snake did not try different routes and would end up in loops which would cause the game to end. Similarly, when I had a large negative number for ending the game by timing out, the snake had much more direct routes and ended up running into the wall a lot more.

Another factor that was considered was whether there should be a reward for the agent for staying alive. This implementation severely crippled the performance of the system. The snake would disproportionally try to loop around since it was rewarded for doing so. In my experimentation it seemed as though this detracted the snake from the immediate goal of getting the food. If it survived it was getting points so there was less of a reason it needed to risk itself and try different paths to get the food. This was obviously counterbalanced by the negative reward for the timer but nonetheless even with increasing the negative reward for the timer this reward did not prove to increase performance in any meaningful way.

INFORMATION ALLOCATION

In addition to deciding the rewards, I need to decide what information to give the agent such as where the food is. Between giving the general direction of the food relative to the head of the snake and giving the exact coordinates, my experimentation showed that giving the general direction actually performed better. The average score for the snake that was given the exact coordinates was 11, after 1000 runs, which is much less than the score of 32 for a snake that was just given the general direction.

This was definitely an unexpected outcome since I was under the impression that giving more information would yield better results. When the program had the exact coordinates it would often make it to the food faster but then get trapped in itself by going on more direct routes. When it had general directions the paths of the snake were a lot windier which ended up giving the head a better chance at survival. By having windier routes, the tail of the snake would fold and limit the space that it would take up. Even though technically the size of the snake is still the same, if the snake is laid out in an undesirable position, it can effectively reduce the size of the usable board by limiting the directions that the snake can move in without running into itself.

The other piece of information is the location of the wall relative to the head of the snake. Similar to the location of the food, I opted to limit the information given to the agent by only letting it know if there is an obstacle in any direction 1 block away from the head. This includes either the wall or the tail of the snake. My thought process behind this was to let the agent explore without having to consider the obstacles until it was necessary. If I gave it too much information at once I didn't want to unnecessarily complicate its decision process. Not only would this result in an overly complex process, but it would also waste computational resources and slow training down significantly. Since the walls were always at the edge of the grid instead of like a labyrinth, the 1 block away method worked reasonably well.

HYPER-PARAMTETERS

I decided to go for a single hidden layer with 512 neurons because of the relative simplicity of the problem. When I tried adding more hidden layers the performance didn't increase – I found that it capped out at around 32. Since it didn't increase performance, I decided that it would be a waste of resources and time to include additional layers.

For the input layer I used 11 neurons. I decided to use this number because of the 4 movement directions the snake is travelling in, 4 relative directions of the food to the head and 3 potential collision directions (either to the left, straight, or right).

For the output layer, I chose 3 neurons because the idea is to decide which direction the head should go in. The system outputs a probability for each direction and the agent chooses the highest value and travels in

For the activation function I used a ReLu activation function which is the popular choice.

I considered using a dropout rate; however, after thinking about it further I decided not to since there is no real concern for overfitting like in other types of problems.

I tested multiple batch sizes and found that smaller batch sizes worked terribly for this problem. Figure 3. depicts the learning curve for when I tried a batch size of 8. This is most likely because with such a small batch size the system was changing its parameters before there were any meaningful conclusions about strategies and thus was stuck with subpar performance. The breakdown of the scores makes it seem that the agent is performing no better than random movements. I ended up using a batch size of 1024. Any batch size larger than that did not performed similarly to a batch size of around 1024 but took much longer for the system to train on.

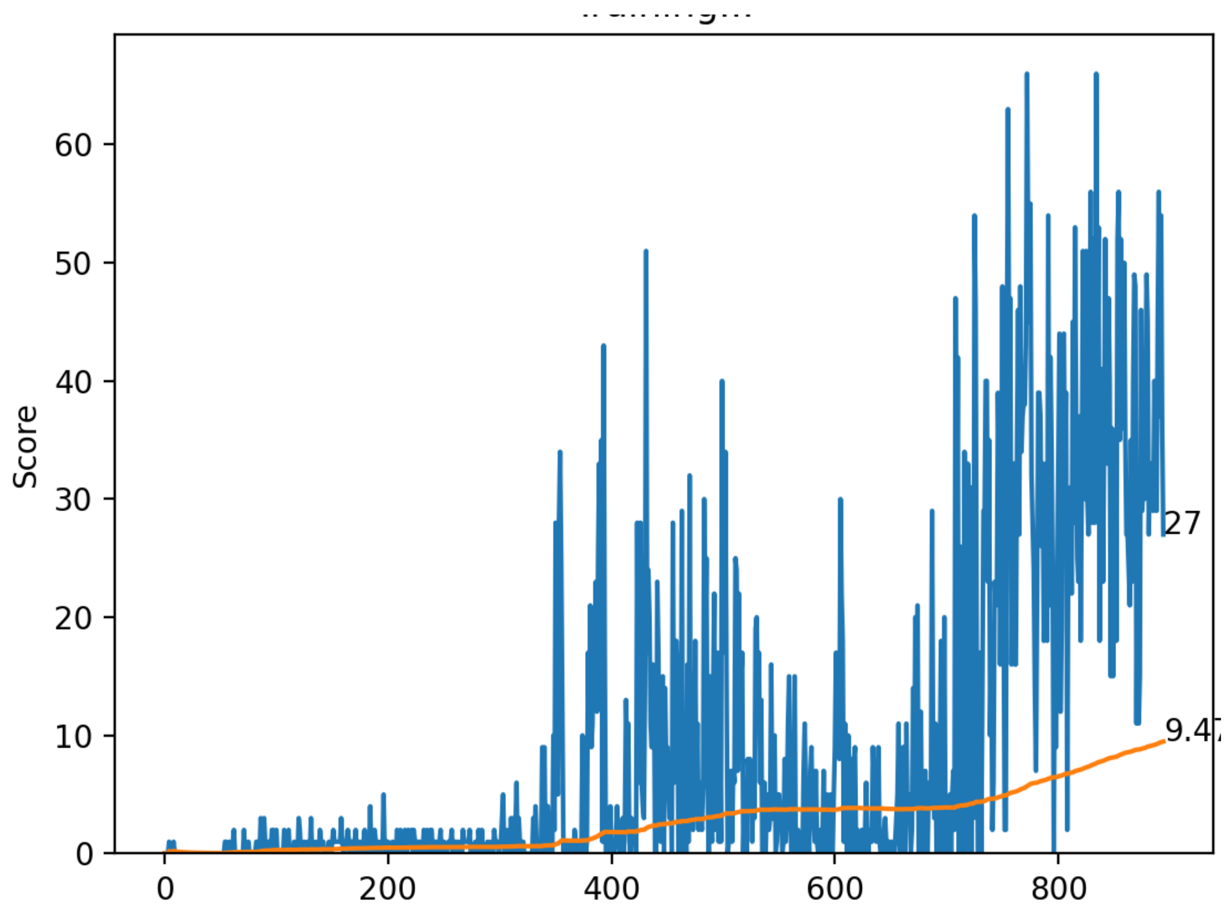
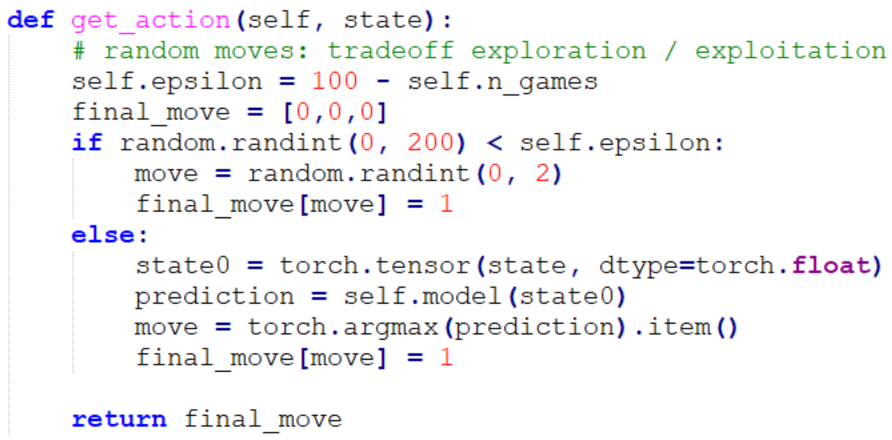


Figure 3. Batch size 8 example

EXPLOATION VS. EXPLOITATION

Because of the nature of the problem, I didn't have to deal with "overfitting" in the traditional sense. The training data is essentially the same as the test data because each level is random, and the agent is given the same information and quality of environment. One thing that I did need to account for however is the issue of exploration vs. exploitation. This is a common problem in reinforcement learning. Your agent wants to get rewards, so it wants to try new things to see which actions yield the most reward. But at the same time if it knows certain actions that will yield a reward it might not want to try new things in case those actions yield less reward than it could otherwise get. To combat this, I used a random variable epsilon to force a random move every so often. If we refer to figure 3., we can see that the epsilon value that I chose is 100 and the cutoff value is 200. This essentially means that when the game starts, there is a 50% chance to make a random move regardless of what the agent thinks is the best direction to travel in. As the games progress, the chance that the agent will be forced to choose a random move reduces to 0. After the epsilon value reduces to 0 or less, the chance that the random number generator chooses a number that is smaller than epsilon is 0. This would ensure that the agent would be forced to try new things regardless of what it already knows but after it has gained some experience it can start to rely more on its own decisions. The reason why I wanted to include this was to inhibit the agent from trying the same strategy too much in the beginning and getting stuck in certain patterns – hence fostering exploration. As mentioned earlier, this was not the only factor when considering the issue of exploration vs. exploitation. When deciding the different rewards distributions, this was definitely a major factor to consider. By not having a disproportionately high negative value for running into an obstacle, I tried fostering exploration of the environment. Similarly, by giving the agent a reward if it gets food, I was able to get it to take advantage of what it knows is effective to find food and continue to survive.



```python
def get_action(self, state):
    # random moves: tradeoff exploration / exploitation
    self.epsilon = 100 - self.n_games
    final_move = [0,0,0]
    if random.randint(0, 200) < self.epsilon:
        move = random.randint(0, 2)
        final_move[move] = 1
    else:
        state0 = torch.tensor(state, dtype=torch.float)
        prediction = self.model(state0)
        move = torch.argmax(prediction).item()
        final_move[move] = 1

    return final_move
```

Figure 4. Random moves

FINAL THOUGHTS

Overall, my system capped out at around an average score of 32. It showed that the agent was able to play the game reasonably well although a skilled player would be able to beat this score. It is interesting to see that even as time progresses and the agent has more experience playing the game, the range of scores that it gets remains relatively large. The highest score that it was able to achieve was 92, but even after that it scored less than 20 many times. Based on my observations on how the snake behaves as it plays the game, most of the time that it was losing was when it would enclose itself with its tail. This is much different than the beginning before any training where the snake would move erratically and either not be able to find the food until the time runs out or simply run into the wall. I think the agent isn't taking the location of the tail into account so when it gets longer it moves in such a way that it doesn't leave an escape route when it circles a food. A possible improvement on this system would be if I would give the agent more information about the relative location of the tail and how it's situated onto the graph so it can take it into account when taking actions. I could potentially do this with coordinates of each block on the tail.

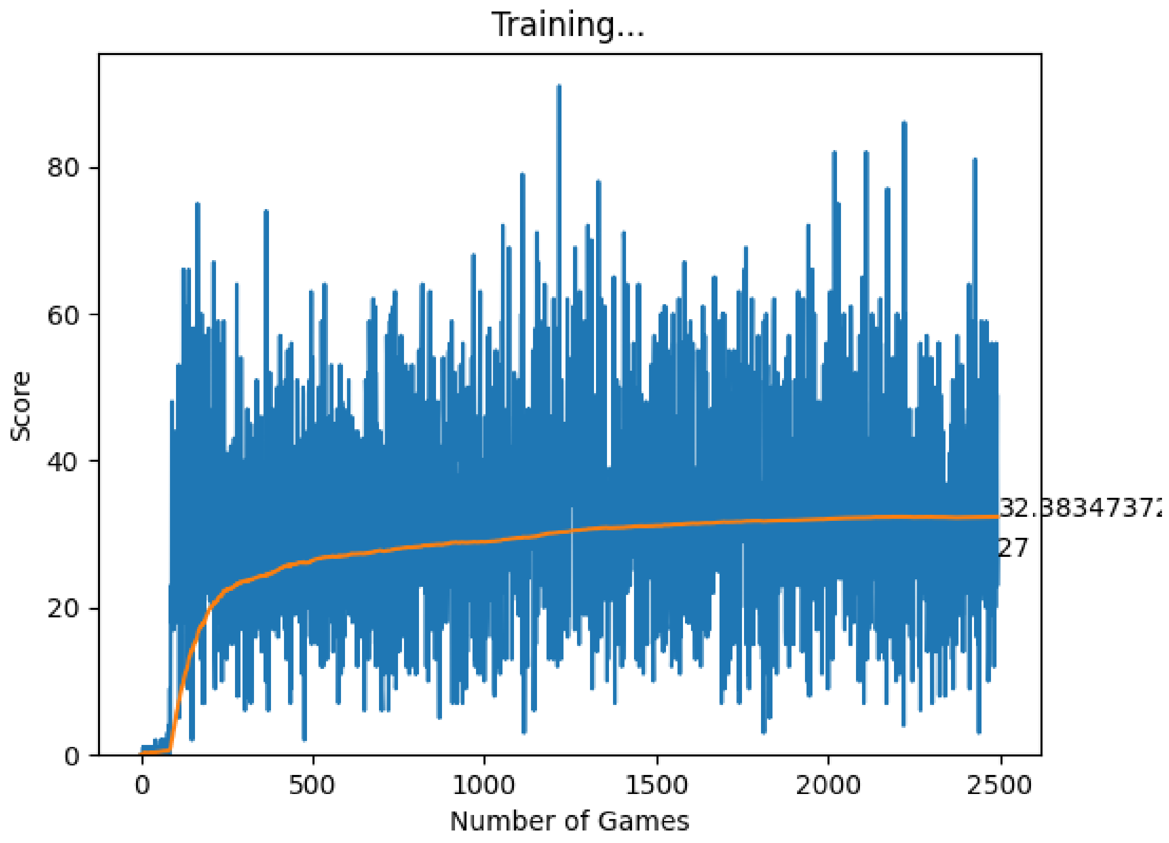


Figure 5. Optimal Performance

This project allowed me to explore the different intricacies of deep reinforcement learning. When I first started researching this topic, I was surprised to find that there weren't necessarily any benchmarks for hyper-parameters that scholars deemed appropriate for certain types of problems. After experimenting, I realized that this was because of the immense variability that each problem has from one another and how much the different combinations of hyper-parameter values actually changes the results. For example, when I changed the learning rate from 0.0001 to 0.00001, it completely changed the shape of the learning curve and decreased performance slightly, as can be seen in figure 4. However, for some other games, a learning rate of 0.00001 or even smaller might prove to work better than 0.0001. Another good example of why there is not a single standard for designing a neural network is when I was deciding how many hidden layers to include. As I mentioned earlier, I decided to have a single hidden layer and I stated that it was because the problem was relatively simple, but there actually isn't a metric that I can use to verify that or compare it to other problems that might be similar. I just needed to evaluate the game as a whole and after trial and error concluded that having only a single hidden layer was the best approach.

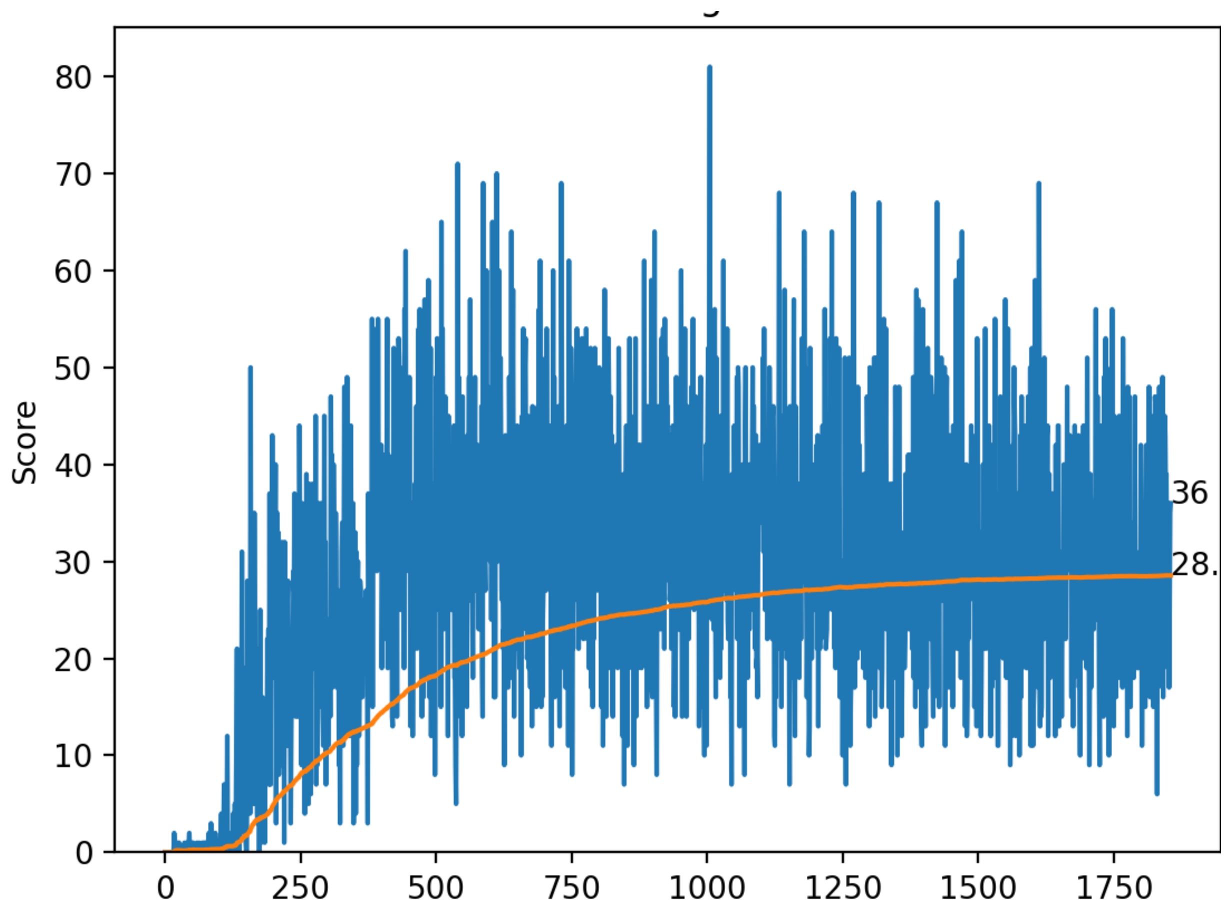


Figure 6. Learning rate reduced to 0.0001

Some future extensions to this project could be if I increased the size of the board after a certain score. It would give the snake more room to work with and since the main issue I saw was enclosing it would be interesting to see how the model would perform. I could also add obstacles to see how the model would avoid these.

Additional potential experiments would also be adding multiple pieces of food on the board at the same time to see if this could increase the high score. And finally adding a poison food that would give a negative score but not necessarily end the game. It would be interesting to see how heavily the model would try to avoid these blocks. If I would be able to implement these variations of the problem, then I think that would give valuable insight on how the model works and thus provide us direction on how to tweak the existing parameters to further improve performance.

References

https://www.mdpi.com/2079-9292/11/4/540/htm

https://www.baeldung.com/cs/reinforcement-learning-neural-network

https://neuro.bstu.by/ai/To-dom/My_research/Papers-2.1-done/RL/0/FinalReport.pdf

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.2894

https://towardsdatascience.com/snake-played-by-a-deep-reinforcement-learning-agent-53f2c4331d36

https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/#:~:text=ReLU%20Hidden%20Layer%20Activation%20Function,function%20used%20for%20hidden%20layers.

https://arxiv.org/pdf/1811.03378.pdf

https://github.com/python-engineer/snake-ai-pytorch