Snake Game AI: A Comparative Study of A-Star Algorithm and Deep-Q Neural Network Approaches

Alex Iliarski
Isaac Berlin

Abstract

Creating automated intelligent agents to complete various tasks is a significant focus of developing research in the field of Computer Science, both in academia and industrial applications. By taking the simple example of the classic video game *Snake*, analysis can be done by comparing different approaches to creating intelligent agents capable of playing the game. The objective is to create an agent capable of achieving consistently high scores by navigating the snake through the grid to collect apples while avoiding self-intersection. We explore and compare two distinct approaches: a Deep-Q neural network utilizing reinforcement learning and an A-star pathfinding algorithm with the Manhattan heuristic. By conducting experimentation with these differing implementations in Python, we evaluate the performance of both algorithms across many iterations. The findings from this study advance strategies in game-solving as well as offering insights that apply to broader real-world applications, such as pathfinding and autonomous vehicles. We determined that, after multiple thousands of iterations, the deterministic A-Star algorithm performs better than the nondeterministic Deep-Q neural network when comparing both the average score and maximum score.

Introduction

Snake is a popular and well-known video game, first originating from the 1976 game *Blockade* [3]. Since then, many variations have been released by various groups. Most notably, the game gained notoriety after its release on the Nokia 3310. The Nokia version of the game, interestingly enough, constituted a “pioneer base” for future mobile games [7]. In this game, a player controls a “snake” which grows in length with each item it collects. This item is displayed as an apple in many variants of the game and will thus be referred to as such. To collect a randomly placed apple, the head of the snake must touch the square that the apple lays on top of. The challenge of the game is to navigate the snake through the grid of squares as the snake automatically moves forward, with only the ability to turn in perpendicular directions, without the snake doubling onto itself. Once the head of the snake contacts a part of its own tail, the game ends. Of course, as the snake grows longer, it becomes more challenging to maneuver the snake in such a way as to continue collecting apples while avoiding contact with the snake’s tail. Additionally, in some variations of the game, there are obstacles throughout the grid or a wall at the edges of the grid. When these obstacles are met, the game will also end. For simplicity, the game referenced in this paper will have no obstacles within the grid and a wall surrounding the grid. In some variants of the game, the snake will move through the grid faster after more time has elapsed or more apples are collected, as a mechanism to increase the difficulty of the game. Once again, for simplicity, we

assume there is no such mechanism in our implementation of the snake game. The player's final score is determined by the amount of apples that the player collected throughout the course of the game.

We analyze various ways to create an automated computer agent that successfully plays the game of Snake. Our goal is to create an agent that is able to achieve a high score in the game consistently. There are many possible implementations of an automated Snake-playing agent. Thus, by comparing the performance of different agents, we can learn what the most successful approach to automating Snake gameplay is. The different methods of playing the game can be categorized into two different ways of approaching the problem: deterministic and non-deterministic approaches. Deterministic approaches typically involve the use of pathfinding algorithms to determine a path for the snake to take from its current state to the location of the apple. This includes commonly used search algorithms, such as Breadth-First Search (BFS), Depth-First Search (DFS), and A-Star [7]. These algorithms employ a particular method or heuristic to determine the snake's next move, given its current location and the location of the apple. A-star is among the most popular and widely used pathfinding algorithms, and will thus be examined further in this paper. Non-deterministic algorithms that are used to play Snake typically involve the use of machine learning and neural networks. These approaches involve playing the game many times, with the machine "learning" which moves and strategies work and which do not. Theoretically, after many games are played, a machine learning agent will be much improved over its initial game playing, which is essentially a set of random moves. There are many different variations of machine learning algorithms that can be applied to playing Snake. The method that is explored in this paper is the method of Deep-Q Neural Networks (DQN), as it is a recently developed and interesting approach to machine learning [2].

This simple game is a smaller example of techniques that are applicable to a variety of larger-scale real-world applications, such as autonomous vehicles and pathfinding.

Related Work

Artificial Intelligence has been used to play and attempt to solve games for many years. AI has been used to play games from Chess to Checkers to Backgammon [8]. As time goes by, more advanced games are being played by AI including even some more modern video and computer games. Mnih et al [6] have even implemented artificial intelligence agents to play Pong, Breakout, Space Invaders, Seaquest, and Beam Rider. The snake game is a popular candidate for many articles focusing on artificial intelligence game playing because of how simple or complex the game can be played. As previously described, the game can have a simple or highly complex environment with items such as game board size variation, the inclusion of extra obstacles, or the possibility of the snake jumping over itself once or twice per game. This variation allows a lot of work to be done on how best to implement this game and which algorithm or model provides the best results in terms of game score or number of moves taken.

One approach to playing Snake is to use a deterministic algorithm to determine how the snake should move. Many pathfinding approaches exist in the literature, such as BFS, DFS, and A-star. These algorithms take the given game state, including the location of the snake and the location of the apple, to determine a path for the snake to follow in order to reach the apple. Sharma, Mishra, Deodhar, Katageri, and Sagar, for example, make use of the Best First Search algorithm [7]. This algorithm is explained as a "combination of BFS and DFS." It makes extensive use of the Manhattan distance heuristic function to approximate the distance between the head of the snake and the apple. If the location of the snake is (x1, y1) and the location of the apple is (x2, y2), then the Manhattan heuristic calculates the distance between these points as

$$ManhattanDistance = |x1 - x2| + |y1 - y2|.$$

Another approach is to use the A-Star algorithm. A-star combines consideration of the cost to reach a given location and a heuristic, or estimate, of the cost to reach the goal state from that location. This is expressed mathematically as

$$f(n) = g(n) + h(n),$$

where *f(n)* is the estimate of total cost from start to goal through node *n*, *h(n)* is the heuristic estimating cost from node *n* to the goal node, and *g(n)* is the actual cost from the start node to node *n*. With a grid-based game like Snake, the Manhattan Distance heuristic described previously is an extremely useful and accurate heuristic to estimate cost. Additionally, A-star can employ forward-checking techniques, which investigate the state of the game after the goal node (apple) has been reached. For example, with forward-checking, the algorithm may be able to find a way to an apple such that the snake doesn't "corner itself in," thus avoiding a self-collision [7]. For simplicity, we further investigate a simple A-star algorithm, with the addition of basic checking to avoid walls and snake collisions when possible.

Additionally, it is worthwhile to note that there is a path the snake can take that guarantees it will get an apple and stay alive as long as possible. This is referred to by Sharma, et al, as the "Almighty Move" [7]. Although this method may take more moves to reach an apple, it is expected to stay alive for as long as possible. See *Figure 1* below for a diagram of the Almighty Move. For our consideration, this technique is extraneous. We are searching for a method to play the game of Snake, notwithstanding an existing deterministic loophole to guarantee success. Thus, we do not give further consideration to the Almighty Move.

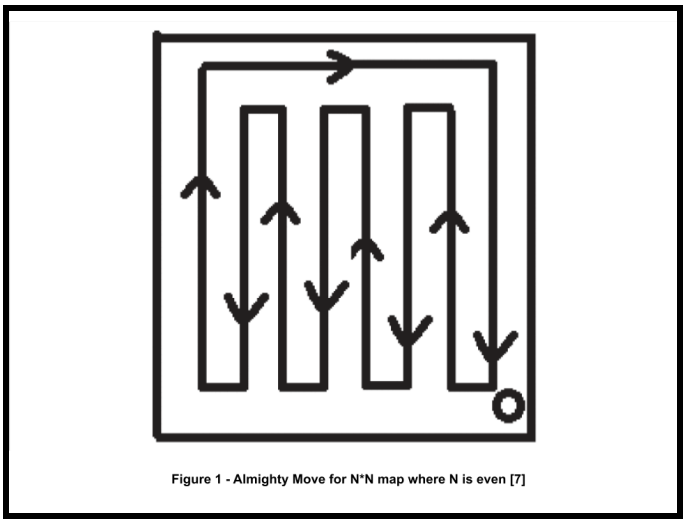

Figure 1 - Almighty Move for N*N map where N is even [7]

Another method to play the game of snake is to use machine learning. By far the most popular and applicable type of algorithm for this problem is a Deep-Q Neural Network (DQN). This algorithm, first used by Mnih et al [6] to play Atari games, abstracts from Q-Learning.

Q-Learning is a Reinforcement learning policy that will find the next best action, given a current state. It chooses the next action based on previous actions set out in a Q table. Q-Learning calculates the value of the current state and the value of possible moves. Q-Learning uses the Bellman Equation [10] to find this as an exact value.

$$NewQ(S,\ A)\ =\ Q(S,\ A)\ +\ \alpha\,[\,R(S,\ A)\ +\ \gamma\, MaxQ'(S',\ A')\ -\ Q(S,\ A)\,]$$

In this equation, S represents the current state and A represents a given action in that state. Q(S, A) represents the current Q value for a state and action and R(S, A) represents the reward value for our current Q value. α denotes the learning rate and γ denotes the discount rate. MaxQ'(S', A') represents the maximum expected future reward for an action.

In normal Q-Learning these computed Q values are stored in a table for later use. This can lead to problems with larger environments with many possible states and actions. To remedy this, we use a Deep-Q neural network. Instead of storing calculated Q values in a table, a Deep-Q neural network uses these action Q value pairs as inputs and outputs of a neural network. Essentially, it uses a neural network to estimate the Q value for any given move. Figures 2 and 3 represent how Q learning and Deep-Q calculate their Q values [9].

Example Q-table	
State-Action	Q-Value
(S1, A1)	5
(S3, A0)	9
(S5, A1)	1

Figure 2 - Q-Learning Q value evaluation [9]

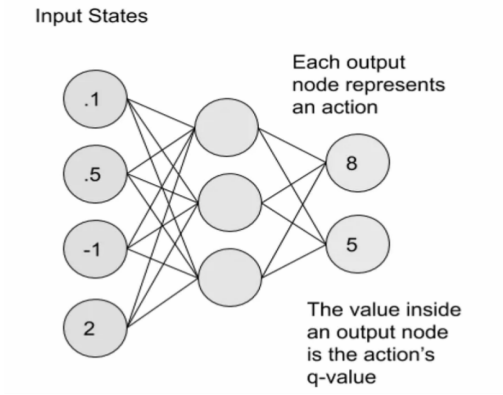

Figure 3 - Deep Q-Learning Q value evaluation [9]

We will use a DQN because it has been shown to be previously successful at learning the snake game [1,3,11]. A DQN has been shown to be able to handle things like a “poison” apple [1] or variable sizes in the game environment. It has also been shown to score higher than humans on average [11]. Instead of testing to see if a DQN will work, we will compare its results to a more deterministic method.

Approach

To solve the problem of playing the game of Snake correctly and efficiently we will implement a few algorithms and compare the results. Working in Python, we create a Deep-Q neural network that learns as a genetic algorithm and compare its results against an A-star pathfinding algorithm to find the next item to collect. We compare average scores after running both algorithms 10, 100, 1000, and 10000 times and see how much the Deep-Q neural network improves and also at what point, if ever, it surpasses the A-star algorithm implementation. We make use of the A-star algorithm with the Manhattan heuristic, as described further later. We choose to examine the average scores after these differing amounts of iterations have been run since a small sample size may skew results for the A-star algorithm, as well as demonstrate the anticipated learning that is expected from the DQN agent.

To implement the actual game of Snake we use the openly available project called by Patrick Loeber. This code is licensed to be freely available on the GitHub account by the name of *patrickloeber*, and an explanation of the code is available on YouTube under the channel with the name *patloeber* [5]. This code base uses the open-source Python package PyGame to implement the game logic and graphical interface. We slightly edit the source code to allow for our algorithms to run the game themselves, as opposed to a human player.

The A-star pathfinding algorithm is a widely used algorithm applicable to a wide variety of fields [4]. The algorithm involves the use of a heuristic function in order to

choose the next step from a pool of neighbors in a traversal from an initial state to a goal state. By investigating the cost to reach a given node in conjunction with the estimated distance between that node and the goal node, also known as the heuristic function, the algorithm can efficiently determine a useful solution to the pathfinding problem. A commonly used heuristic is the Manhattan heuristic, so-called since it models the heuristic on the block-based structure of the Manhattan street grid. Since the Snake game environment is modeled as a grid, both to players and on the back end, the Manhattan heuristic is highly applicable to this problem. That, along with the simplicity of the heuristic, are the fundamental reasons why we choose to employ this heuristic in our analysis.

Additionally, we make minor modifications to the A-star algorithm to avoid collisions. Simple checks are added to the A-star algorithm which prevents it from immediately colliding with the wall surrounding the grid or itself. That is, if the head of the snake is immediately next to one of these obstacles, it will not choose to continue into the obstacle and lose. However, A-star will still often “corner itself in,” and cause itself to have no valid choice except to collide with an obstacle since we do not implement forward-checking. This is an extremely simple and useful extension to the algorithm which greatly improves its performance since it will find paths that do not collide with its own body or the wall.

For our Deep-Q neural network, we used Pytorch. Pytorch is a Python-accessible library developed by Facebook to work with the open-source machine learning library torch. We used PyTorch built-in functions for neural networks and optimizers. To estimate the Q value we used a neural network with an input size of 11, a hidden layer size of 256, and an output of 3 (for the 3 possible moves). We used linear regression for our layers and the rectified linear unit as our activation function for the hidden layer.

To implement that Deep-Q model we also used PyTorch. We used a learning rate of 0.001 and a discount rate of 0.9. We used the Mean Squared Error function to calculate the loss and Adam optimization as our optimizer. We weighed eating an apple and gaining a point as a +10 reward and dying as a -10 reward. We also automatically terminated/died if the snake was unable to reach the apple within 100 moves. This was done to prevent the snake from learning how to not die and not learning how to increase points.

Experimental Design and Results

The A-Star algorithm with the Manhattan heuristic is used to play the game of Snake. Graphs are created of the performance of this algorithm over its first 10, 100, 1000, and 10000 games. The results are seen in *Figures 4, 5, 6, and 7*, respectively. The blue line represents the score of each game, while the orange line represents the average score attained by the A-star algorithm up to that point.

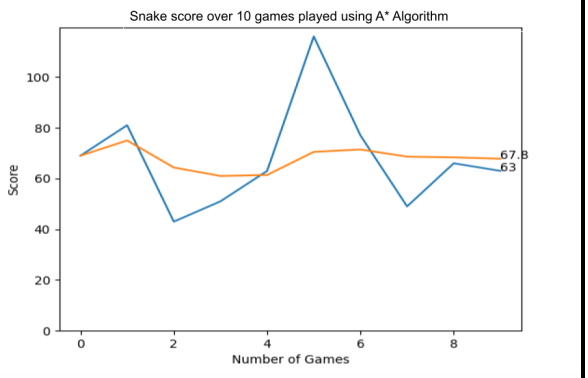

Figure 4 - Experimental results of A-Star algorithm playing 10 games of Snake

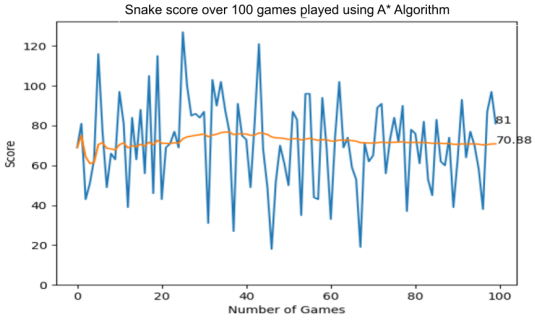

Figure 5 - Experimental results of A-Star algorithm playing 100 games of Snake

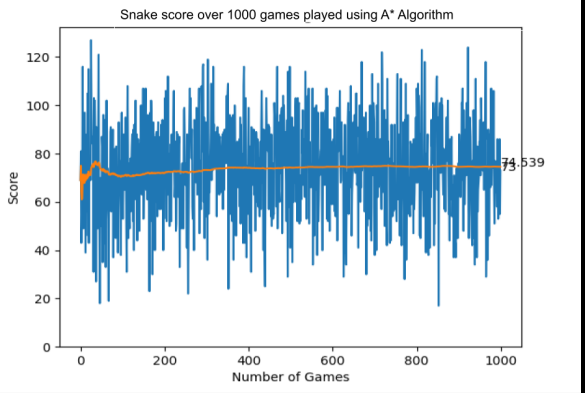

Figure 6 - Experimental results of A-Star algorithm playing 1000 games of Snake

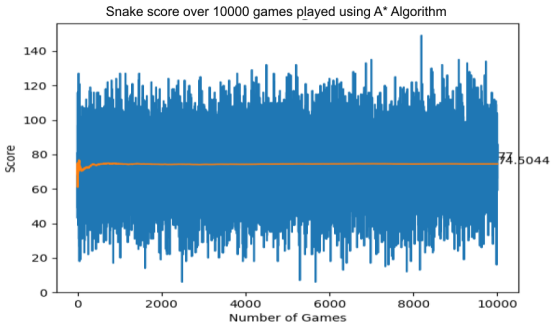

Figure 7 - Experimental results of A-Star algorithm playing 10000 games of Snake

The average score the A-star algorithm achieves in a game of snake after 10000 games played is *74.5044,* as observed in *Figure 7*. The best score achieved in a single game by the A-star algorithm over these 10000 iterations is *149*. We observe in *Figure 6* that the average score, represented by the orange line, is fairly stable after about 400 games are played. Additionally, the average score over the first 1000 iterations is *74.539*, which is only slightly larger than the average score after 10000 games are played.

The DQN is also used to play the game of Snake. Graphs are created of the performance of this neural network over its first 10, 100, 1000, and 10000 games. The results are seen in *Figures 8, 9, 10, and 11*, respectively.

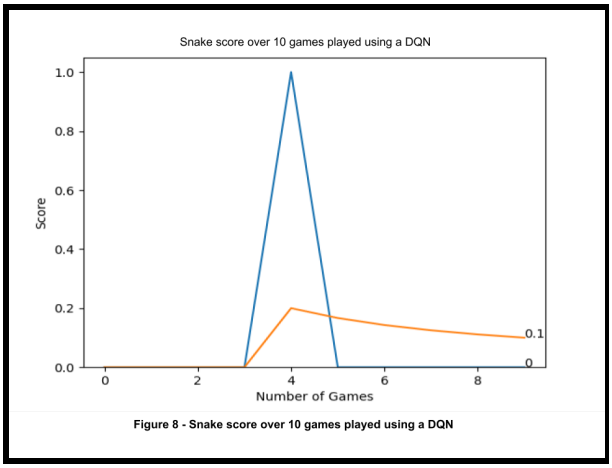

Figure 8 - Snake score over 10 games played using a DQN

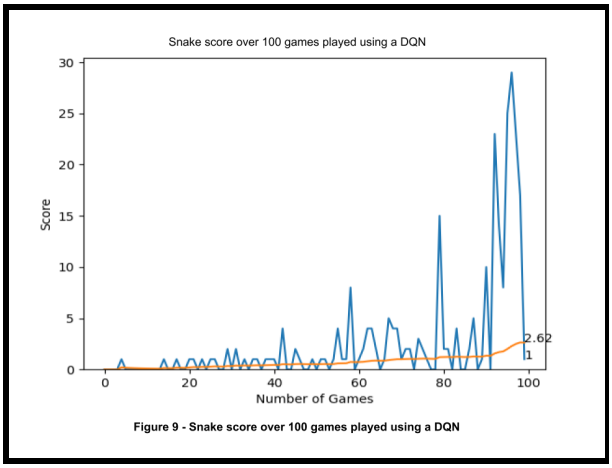

Figure 9 - Snake score over 100 games played using a DQN

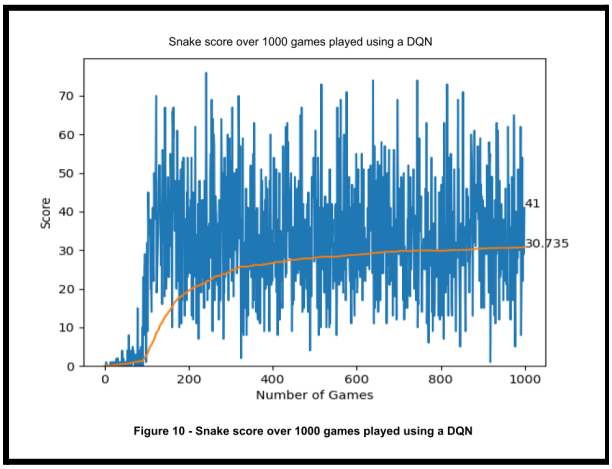

Figure 10 - Snake score over 1000 games played using a DQN

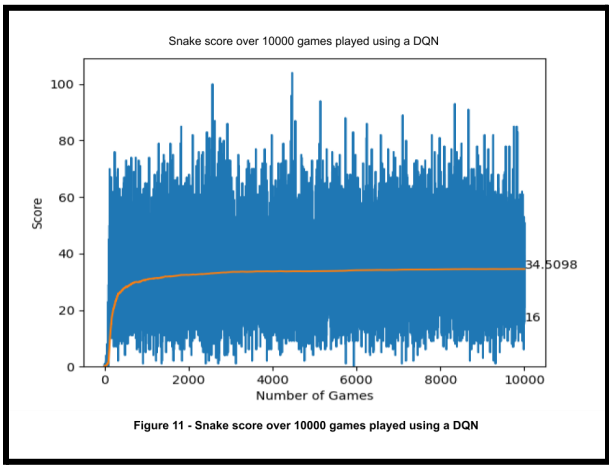

Figure 11 - Snake score over 10000 games played using a DQN

The average score the DQN algorithm achieves in a game of snake after 10000 games played is *34.5098*. The best score achieved in a single game by the DQN over these 10000 iterations is *105*. We observe in *Figures 10 and 11* that the average score, represented by the orange line, rises quickly over the first 400 iterations. After this, the average score continues to increase, albeit very slowly. This is also seen by looking at the average score after 100 iterations, which is *2.62*, as seen in *Figure 9*. This is much lower than the average score after 1000 iterations of *30.735*. Additionally, we observe that for the 100 or so iterations the DQN improves linearly compared to after the first 100 iterations where it improves logarithmically.

Analysis

The experimental results of the A-star algorithm are consistent with expectations. While there is much variability in the scores achieved in each game played by the agent, over many iterations we can determine an average expected score. Even going from 1000 iterations to 10000 iterations, the average score only decreased by *0.0346* points. This is consistent with a growing sample size of a non-learning, deterministic, algorithm. We do not expect much change in the average score once the sample size

becomes sufficiently large. The only randomness that is introduced is the location of apples, which can affect the performance of the algorithm. However, over many iterations, there will be both “lucky” and “unlucky” placements of apples. Thus, these will balance out in the long run and the average score will be a fairly useful indicator of how well the algorithm works. We thus determine that an average game of Snake played by the A-star agent is expected to score approximately *75* points, on average.

The experimental results of the DQN show that we can, in fact, learn the game of snake using a Deep-Q neural network. In *Figure 8*, we notice that for the first 10 iterations, the network only manages to get a single point across all 10 games. This is because, at the start of any neural network, the weights are not well-tuned and the inputs are more random than actual prediction. We can see a similar phenomenon in *Figure 9* where the model is starting to learn but is still highly variable in its scores.

Figure 10 shows how it takes around 100 to 150 games for the network to start consistently achieving nonzero scores. This is because the Q value network has started to get more accurate and thus we are getting better at predicting Q scores based on our current position. *Figure 11* also shows that after it reaches 10,000 iterations the average score is around *35*. This signifies that the algorithm is continuing to grow and improve as more and more iterations are run.

Additionally, after the initial 10000 iterations, we ran the DQN for another 15,000 iterations (a total of 25,000) to examine whether the DQN simply needed more training to approach A-star’s performance level. On the contrary, we observed that the average score ended up dropping to *33.5766* points per game. This is a textbook example of overfitting (training a network too much) and shows us that the neural network performs best when only trained for around 10,000 iterations.

Even from a quick glance, the results are fairly obvious to see. The A-star algorithm simply outperforms the Deep-Q neural network when comparing both maximum scores and average scores. The A-star algorithm’s pure ability to seek out apples immediately, while maintaining a simple ability to avoid collisions, outperforms a DQN without such explicit instructions. It appears that the DQN simply “tops out” at a certain point, and is unable to progress much farther in its performance. It is likely that with modification of the layer sizes and weights of rewards and penalties, the DQN could certainly improve its performance. With the correct modifications, it could possibly reach A-star’s capabilities, or even outperform them. However, the DQN in its current state that we implemented is quite clearly not capable of playing the game at the same level as the A-star algorithm.

Conclusion

Snake is a simple game, but attempting to get artificial intelligence to play the game itself is a challenge that yields fascinating results. Approaches to designing artificially intelligent agents to play Snake fall into two main categories: deterministic and non-deterministic. Of the many deterministic algorithms commonly used for similar

problems, A-star stands out as a particularly applicable choice for playing Snake, since its use of the Manhattan heuristic applies perfectly to the grid-based setting of Snake. Most non-deterministic approaches involve the use of machine learning and training of neural networks to create an agent that specializes in playing Snake. Of the many different varieties of neural networks that could be used to play Snake, a Deep-Q Neural Network presents itself as a fascinating choice, due to its recent emergence in popularity, its unique mathematical structures, and its success in similar studies. An experimental comparison is done to compare the success of these two approaches over 10000 iterations. We observe that although the DQN is able to learn to play Snake, after struggling during the first 200 iterations (as would be expected from a neural network), it is not able to reach the same level of performance as the A-star algorithm. This is likely due to the extremely strong and reliable capability of the A-star algorithm to direct itself towards apples and avoid some types of collisions.

On the other hand, it is also likely that the method we implemented to train the DQN could be faulty. With a modification of the reward and penalty weights, it is likely that a “sweet spot” could be found with weights that maximize the DQN’s long-term performance. Future work could be done in this area to determine exactly what weights are the most successful and whether there is any combination of these weights that can catapult the DQN above the A-star algorithm in terms of the average score while playing Snake. Additionally, the structure of the DQN could be modified to potentially add or subtract the number of layers and the number of nodes in each layer. This research signifies that although there is a tendency to use machine learning algorithms to approach modern problems, there is still a use for more reliable deterministic algorithms in certain situations. Additionally, this study highlights the importance of further research to determine the optimal way to construct and train a DQN.

References

Works Cited

[1] Almalki, Ali Jaber, and Pawel Wocjan. “Exploration of Reinforcement Learning to Play Snake Game | IEEE Conference Publication | IEEE Xplore.” *Ieeexplore.ieee.org*, 5 Dec. 2019, ieeexplore.ieee.org/document/9070827. Accessed 15 Dec. 2023.

[2] Ausin, Markel Sanz. “Introduction to Reinforcement Learning. Part 3: Q-Learning with Neural Networks, Algorithm DQN.” *Medium*, 25 Nov. 2020, markelsanz14.medium.com/introduction-to-reinforcement-learning-part-3-q-learning-with-neural-networks-algorithm-dqn-1e22ee928ecd#:~:text=The%20Deep%20Q%2DNetworks%20. Accessed 21 Nov. 2023.

[3] Bonnici, Russell Sammut, et al. “Exploring Reinforcement Learning: A Case Study Applied to the Popular Snake Game.” *Disruptive Technologies in Media, Arts and Design*, vol. 382, 2022, pp. 169–192, https://doi.org/10.1007/978-3-030-93780-5_12. Accessed 12 Mar. 2023.

[4] Candra, Ade, et al. “Application of A-Star Algorithm on Pathfinding Game.” *Journal of Physics: Conference Series*, vol. 1898, no. 1, 1 June 2021, p. 012047, https://doi.org/10.1088/1742-6596/1898/1/012047.

[5] Loeber, Patrick. “Patrickloeber/Python-Fun.” *GitHub*, github.com/patrickloeber/python-fun/. Accessed 15 Dec. 2023.

[6] Mnih, Volodymyr, et al. “Human-Level Control through Deep Reinforcement Learning.” *Nature*, vol. 518, no. 7540, Feb. 2015, pp. 529–533,

web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf, https://doi.org/10.1038/nature14236.

[7] Sharma, Shubham, et al. “Solving the Classic Snake Game Using AI.” IEEE, 18 Dec. 2019.

[8] Tesauro, Gerald. “Temporal Difference Learning and TD-Gammon.” *Communications of the ACM*, vol. 38, no. 3, 1 Mar. 1995, pp. 58–68, https://doi.org/10.1145/203330.203343. Accessed 20 July 2020.

[9] Wang, Mike. “Deep Q-Learning Tutorial: MinDQN.” *Medium*, 1 Feb. 2021, towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc.

[10] Watkins, Christopher J. C. H., and Peter Dayan. “Q-Learning.” *Machine Learning*, vol. 8, no. 3-4, May 1992, pp. 279–292, https://doi.org/10.1007/bf00992698.

[11] Wei, Zhepei, et al. *Autonomous Agents in Snake Game via Deep Reinforcement Learning*. IEEE, 31 July 2018.