Comparing the Performance of an Adaptive and Non-Adaptive System to play the game of Snake.

Abstract

This report presents a comparison of the performance of two artificial intelligence (AI) programs that control the movements of a snake in the classic Snake game. The goal of this report is to understand and demonstrate the significance of adaptability in AI systems by comparing the performance results achieved by an adaptive AI model designed using reinforcement learning against a non-adaptive AI model. There were significant differences in performance in gameplay, strategic decision-making, and adaptability to the game environments observed between the two AI implementations. The results emphasize just how important adaptability is in the world of AI development.

1. Introduction

Throughout the course of the Adaptive system module, two concepts have remained consistent in association with an adaptive system: learning and adaptation. A general understanding of the concept of learning is the ability of a biological being or system or a machine to gather information either through direct knowledge transfer or gathering it through lived experience from its environment. In machines or software, learning means the capability of a system to enhance its performance on a specific task by utilizing data or feedback. Adaptation is the process by which organisms or systems adjust their behavior or characteristics to better suit their environment based on the information they learned from it When it comes to machines, adaptation involves changing the parameters or strategies of the system to enhance and maximize system performance. Any system that is capable of learning and then adapting according to that learning will be referred to as an adaptive system in this report. Furthermore, any machine or software system that is referred to as a self-adaptive system must be able to learn on its own with minimal human intervention. This definition is a much simpler version of one of the definitions given in the coursework – “An adaptive system is a system which is provided with a means of continuously monitoring its performance about a given figure of merit or optimal condition and a means of modifying its parameters by a closed loop action to approach this optimum.” [1] Although it might be considered debatable by many, artificial intelligence is not necessarily always a self-adaptive system. An AI can learn and output results based on learning but may not be able to adapt based on new data on its own without outside intervention. Reinforcement learning is a type of learning where an agent learns to make decisions by interacting with its environment. The agent receives feedback in the form of rewards based on its actions, allowing it to learn optimal strategies to maximize its rewards over time. The concept of deep reinforcement learning can be used to implement adaptability into an AI. Especially we use Q-learning in the program. In Richard S. Sutton’s book about reinforcement learning, Q-learning is explained as a type of reinforcement learning algorithm that enables an agent to learn the best action-selection policy directly from interactions with its environment. [2]

The classic snake game can be used to perfectly simulate a simple agent-environment interaction which will be helpful in the demonstration of both the adaptive and non-adaptive systems. The primary rule of the snake game is for the snake(agent) to eat the fruit and grow and avoid hitting the grid walls or its own body to survive. Two programs are implemented to control the movement of the snake (agent). The first program used a basic heuristic-based approach. It manages the snake in the game by following a preprogrammed set of rules. This algorithm did not involve any complex learning or optimization methods like neural networks. It makes decisions based on the relative positions of the snake's head and the fruit. It determines which way to go by computing the difference in coordinates between the fruit and the snake's head. Since it's not learning from experience or training data, this approach does not fall under the definition of an adaptive system. It is more of a reactive system where it only gives appropriate reactions to the environment. The second program uses a simple neural network implemented with the Q-Learning algorithm for training, which is a reinforcement learning technique to train the snake agent to play the game optimally by learning from its interactions with the environment. The second program initially randomly explores its game environment and based on these explorations the training data is generated. The later games are played based on the predictions made from the initial explorations. The learning and adaptation align with our understanding of the adaptive system as the program initially begins with random moves but begins to strategize its gameplay in later games based on the learning from interacting with the game environment.

2. Methods

2.1 Snake_1.py – Heuristic-based Snake

In the file program Snake_1.py, three main classes are defined. Snake class initializes the snake with its initial body, direction at random, and growth state. It has a method to control the turning of the head of the snake based on its current direction. It also contains a method to check if the snake has collided with walls or itself. The fruit class generates a new random position for the fruit.

The bot class initializes the bot with references to the snake and fruit objects. It contains two important methods, move() and move_2(). The move() method determines the direction the snake should move based on the relative positions of the snake's head and the fruit. If no valid direction is found towards the fruit, it defaults to a random direction. The move_2() is similar to move(), but it also considers avoiding walls and the snake's body. It prioritizes directions that lead to the fruit while avoiding collisions. While doing experiments, any one of the methods can be called to check performance. The play_game() handles the gameplay of the snake game.

2.2 Snake_2.py – Deep Reinforcement learning

This code is majorly referenced from Patrick Loeber's implementation of the model [4] and the parameters are adjusted to different requirements of experimentation.

This code defines the Snake class, which serves as the agent for playing the Snake game using a reinforcement learning approach. The class initializes various parameters such as the number of games played, epsilon (for epsilon-greedy exploration), memory for experience

replay, the neural network model for learning, and a trainer to handle model training. The method get_state() computes the current state of the game environment, including the snake's position, direction, distances to walls and body segments, and the fruit. The memorize() method stores the agent's experience tuple (state, action, reward, next_state, game_over) into memory for learning. The training() method samples experience from memory and perform training on the neural network model. The learn() method allows for direct learning from a single experience. Finally, the get_action() method selects an action based on the epsilon-greedy policy, either randomly exploring or exploiting the learned policy from the neural network model.

The epsilon-greedy exploration is useful in maintaining a balance between exploration and exploitation of the model. During exploration, the agent takes random actions with a probability determined by epsilon. This randomness allows the agent to discover new states and actions, which helps to get better training and better strategies. Other methods like UCB (Upper Confidence Bounds) tend to work well in balancing exploration and exploitation, especially as the problems get more complex [3] but for this report, the epsilon-greedy method gives optimal results.

In Trainer.py, we have two classes: Net and Trainer. The Net class creates a neural network, and the Trainer class trains the network to make better predictions based on the input data and rewards received. The Trainer class calculates the target Q-values based on the reward and next state, depending on whether the game is over. After that, it computes the loss between the predicted and target Q-values using the Mean Squared Error (MSE) loss function. Then it performs backpropagation to update the model parameters using gradient descent. The Net class defines a neural network model using pytorch with an input layer, a hidden layer, and an output layer. It applies a ReLU activation function to the input-hidden layer, and then passes the result to the hidden-output layer.

There is only one hidden layer used here that consists of a high number of neurons. Although it is unadvisable to use too many neurons due to the possibility of overfitting the program here is fairly simple and the results seem to function much better with a single hidden layer with a high number of neurons than adding more layers with fewer neurons. This was also checked during experimenting but the results of multiple hidden layer neural nets were too poor to be included.

To follow up with the experiments, change the state parameters and the neural net function parameters accordingly to check the results. But the program must run for at least 800+ iterations to see any meaningful output.

3. Results

3.1 Heuristic-based Approach

In the program file, Snake_1.py calculates the difference in coordinates between the snake's head and the fruit and then chooses a direction based on this difference. With this predefined strategy, the program is run for 500 – 700 iterations for multiple runs. Given below are the

graph plots from 2 different runs. Fig-1.1 shows the graphical plot of the scores and the average scores after each of the 500 game iterations. From another run, fig-1.2 shows only the average scores of 700 game iterations.

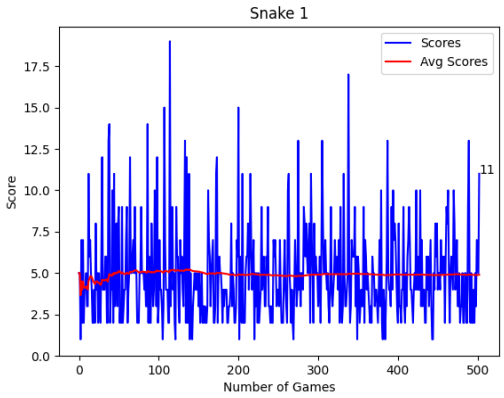

Figure 1.1: Snake_1.py (500 game iterations)

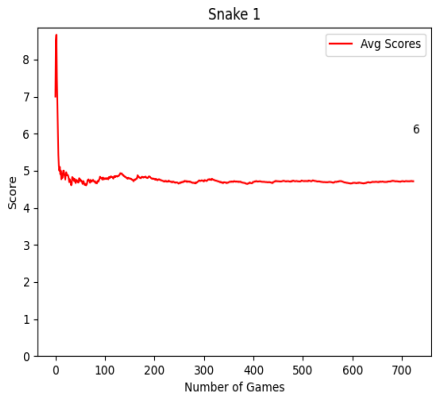

Figure 2.2: Snake_1.py (700 game iterations)

In most of the runs, the highest score, i.e. number of fruits eaten without collision with walls or itself, is about 15-19 which is fairly a good score. The snake in the game rarely ever received a score of zero even in the initial games. However, the average of the cumulative scores achieved in all the game iterations remains the same (~5) throughout the run and can be seen in the graph. Here the snake is only programmed to look for the fruit and not to avoid the walls. The next run is taken after programming the snake further to avoid collisions. The program was run for 200-350 game iterations each time. Again, Fig-2.1 shows the graphical plot of the scores and the average scores after each of the 350 game iterations and fig-2.2 shows only the average scores from a run of 200 game iterations.

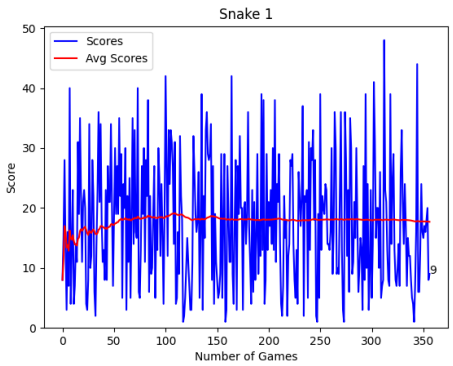

Figure 2.1 Snake_1.py (program avoids collisions)

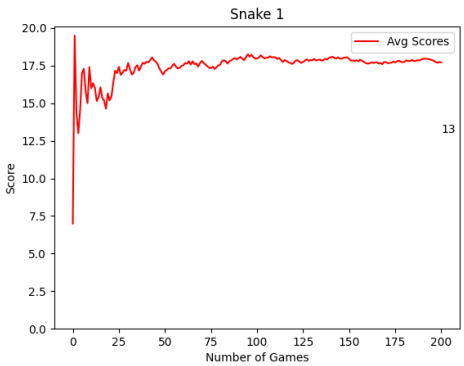

Figure 2.2 Snake_1.py (average scores)

After manually improving the heuristics the highest score the snake was able to score was close 40-48, which is a significant improvement from previous runs. The average score, although improved after the heuristic changes, remains relatively the same throughout the run and shows no improvement on its own.

3.2 Reinforcement Learning Approach

Here we run the program Snake_2.py for about 1000-2000 game iterations. For each run, we change some parameters such as the number of game state parameters given as the number of neurons as input in the Neural network, size of hidden layers, number of hidden layers, and rewards system. The first set of runs was with a neural network of input size 13, 1 hidden layer of size 256, and an output size of 3. Fig-3 shows 2 runs one with 1000 games and another 2000 games.

In each run, the initial 100 to 150 runs perform quite poorly when compared with the heuristic approach results. This is an expected outcome as we try to maintain some balance between exploration of the environment and exploitation of the training policy. These initial poor performances help the model figure out how to achieve bigger rewards later. It's important to gather plenty of information to ensure the best move decisions overall [3]. After about 250-350 games, the scores slowly start to improve and we see a slow incline in average scores. However, after 1000-1500+ iterations the improvement slows down further. Here average score may not be the best way to assess overall performance because a lot of games initially score 0 due to exploration.

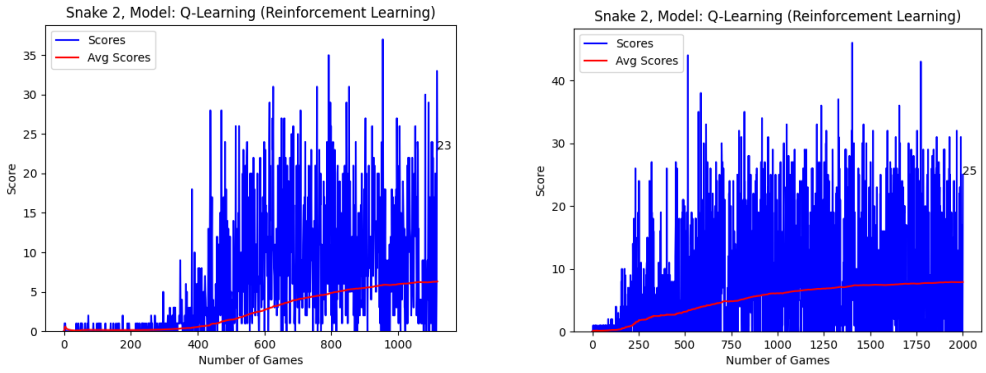

Figure 3 Snake_3.py results

The next set of runs was with a neural network of input size 11, 1 hidden layer of size 256, and an output size of 3. Fig-4 shows an instance of a run with 1000 games. The inputs/states removed were information on the distance between snake body and head and the distance between snake head and fruit.

Here the expectation was to check how changing the input size might affect the results of the games. Again, the initial 100 to 150 runs perform quite poorly when compared with the heuristic approach results and the high scores achieved are slightly lower than the previous 13 input one. After about 250-700 games, the scores show a significant improvement which can be visible in the average score plot. However, similar to the previous results, after 1000+ iterations the improvement slows down and starts to level out.

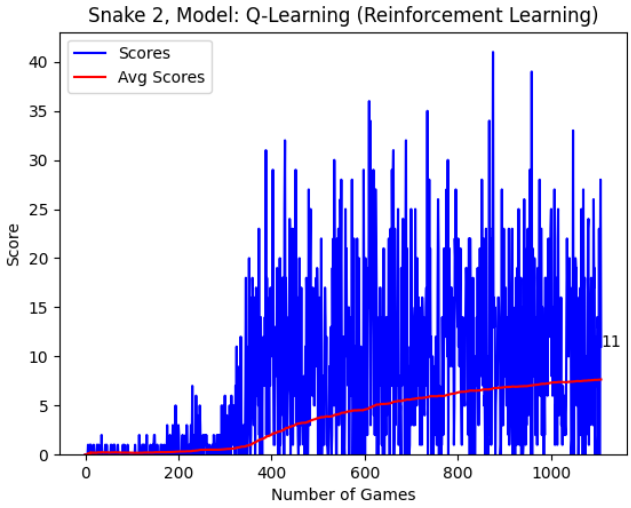

Figure 4 Snake_2.py results with 11 inputs

The next set of runs was performed with even lesser input/states. This time 7 inputs were given with the rest of the information same. The information was regarding the current direction of movement. However, the results were too poor, even after 200-300 runs, to be considered further, hence the results are not mentioned.

4. Analysis

The first approach has fairly good performance from the beginning. Just the information for finding and moving towards the fruit and changing its direction of movement accordingly seems a fair strategy. But this is not an acquired knowledge. However, the snake does not learn to do so, it has only been programmed that way. The snake does not learn to avoid collisions or to develop strategies to avoid its body while moving towards the fruit. Although snake performance is good, it remains throughout and never improves on its own. It only improves when a strategy to avoid collision is programmed in, that is an outside intervention is required.

According to the definition given in the introduction, this approach creates a non-adaptive system. It does not learn on its own and has no provisions to adapt based on it. It only reacts to its environment based on predefined strategies.

In the second approach, the performance is expectedly poor in the beginning, unlike the first approach. Most of the games played initially have a score of 0. This is due to the greedy epsilon method used to explore its environment. As more games were played the performance improved on its own without having to re-program to code. The snake gathers information based on its state in the game, that is information regarding the position of the fruit, distance from the fruit, its current direction of motion, danger of collision, and distance of its head from its body. Using these states, the snake learns how to navigate its environment based on the rewards it receives. With these information, it adapts its strategies to play the game. The more states it has, it gets a better

understanding of its system and creates a better strategy which is clear with the results of 13 states when compared to 11 states, and with 7 states, it has the poorest result.

According to the definition given in the introduction, the heuristic-based approach creates a non-adaptive system. The snake does not learn on its own and has no provisions to adapt based on it. It only reacts to its environment based on predefined strategies. The reinforcement learning snake, however, has the ability to learn and adapt based on its learnings, therefore is an adaptive system.

Although the heuristic-based snake does have some intelligence it is not an acquired one. Of course, it might be debatable if it is really 'intelligent' if it needs to be instructed to do everything and not figure it out on its own. It depends on one's definition of 'intelligence'. In my opinion, the heuristic-based snake should be considered intelligent as it is aware of its environment and has the ability to react to it. However, incorporating adaptive behaviours into artificial intelligence would enhance the intelligence as not only does it react to its environment but it has the ability to learn to do so, which is demonstrated through the reinforcement learning snake. This is evident in the performance of both the snakes. The performance of the first approach remains constant whereas the performance of the second snake not only improves but surpasses the best performance of the first snake scoring as high as 46.

However, the learning and adapting ability does slow down after a certain number of game iterations. This could be solved using better algorithms. But for this report, we continue with the existing algorithms.

5. Discussion

Another approach in addition to Deep-Learning that was unsuccessfully attempted was to build a genetic algorithm on top of the deep-learning results. The program file is included and named as Snake_3.py. The implementation of the algorithm was too simplistic and did not give results as expected. In this approach, a population of 50 neural network snakes was initialized. Each snake was meant to learn and adapt by reinforcement by playing the game for 800 iterations and the fitness score was the highest score achieved. The decision of 800 iterations was based on the performance results of the reinforcement snake since after about 1000 iterations there was not much of an improvement. The idea behind adding the genetic algorithm on top was to tackle the slowing performance from the deep-learning. As mentioned before, the performance did not improve much in later games.

Prospective further work in this experiment, one could find a method to tackle the slowing performance with algorithms like genetic algorithms or combine different algorithms to further create a more intelligent snake.

As per the definition given in the introduction, the snake possesses the ability to learn and adapt from its interactions with its environment. Similar to a human's approach in the snake game, the system adjusts its tactics based on past experiences. Through reinforcement learning, it continually enhances its gameplay strategies, aiming for improved scores. Thus, instead of being confined to a static predefined method, it develops a process of gradual improvement. Therefore, it can be referred to as an Adaptive system based on the definition.

As mentioned before in the report, artificial intelligence is not always necessarily adaptive in nature. AI software like ChatGPT, Alexa, and Siri are obviously artificial intelligence but they don't adapt with

time and new emerging information. It needs human intervention to adapt to current information. Incorporating an adaptive ability can significantly improve the quality of artificial intelligence and the goal of this report was to demonstrate this idea.

However, from the results of the reinforcement learning snake, it is clear that machine adaptability also has its limits. Of course, with better algorithmic implementation this can be resolved but an important question that emerges from this report is how much adaptive a system can be designed. The popular idea among the general public is artificial intelligence, one day might be able to replace human intelligence and it does have the potential, but it is unlikely to replace it completely.

6. Conclusion

In this report, we experimented with two different approaches to building a program to play the game of snake. One is a basic heuristic-based method and another is a reinforcement learning method. The heuristic-based approach demonstrated good performance from the beginning, using predefined rules to navigate the game environment. However, it lacked the ability to adapt and ability to learn on its own, relying completely on programmed strategies. Although it showed some level of intelligence in reacting to its environment, it did not possess the ability to improve its performance. Whereas, the reinforcement learning approach initially had poor results but after a certain point by training from its initial losses, the snake gradually enhanced its strategies and performance by gathering information about its state in the game and adapting its tactics accordingly, the reinforcement learning snake demonstrated adaptability and autonomous learning. This enabled it to surpass the heuristic-based approach and achieve higher scores over time.

In conclusion, while both approaches showed intelligence, but the reinforcement learning method showcased the potential for adaptability and self-learning. As we continue to further in the field of creating better and smarter artificial intelligence, incorporating adaptive behaviors will be important. Even with challenges and limitations, the goal of creating truly intelligent AI can be successfully achieved by merging it with adaptive system concepts.

7. References

1. Dracopoulos, D. C. (1997). Evolutionary learning algorithms for neural adaptive control. Springer.
2. Sutton, R. S., & Barto, A. G. (1998). "Reinforcement Learning: An Introduction." MIT Press.
3. Uzair, Muhammad, and Noreen Jamil. "Effects of hidden layers on the efficiency of neural networks." 2020 IEEE 23rd International Multitopic Conference (INMIC). IEEE, 2020.
4. Patrick Loeber's code - https://github.com/patrickloeber/snake-ai-pytorch