Matthew Gardner

CS570

December 5th, 2024

Tayyaba Shaheen

Artificial Intelligence Implemented into Snake

This project implements three different versions of the classic game “Snake.” The first version is the classical manual approach where the player controls the snake directly. The next two approaches are automated and require no player input. The second version uses an implementation of the A* search algorithm, and the third employs a Deep Q-Learning reinforcement learning technique to have the AI learn to play the game through iterations of storing to memory. The learning model is rewarded for actions such as eating the fruit (increasing score) and penalized for collisions with obstacles or the snake’s own body.

Through implementation, the two automation styles cannot match a skilled human player manually controlling the game. Both automation techniques can play the game at much faster speeds than a human, but when comparing average scores and highest scores, the human player wins by a large margin. In testing thousands of games, the A* implementation achieved an average score of 69.98 and a high score of 137. The learning model, with an initial training of 1,000 episodes when tested with an epsilon (exploration rate) of zero over another 10,000 runs averaged a score of around 8 and a high score of 44. Researching similar implementations online, I struggled to find a learning approach that consistently compared to the A* algorithm. Published data often involved only 100 runs or so, making direct comparisons difficult.

This data suggests that while the A* algorithm could outperform an average snake player, it would likely struggle against a skilled player familiar with the game's intricacies and strategies.

The paper will be broken up into sections; beginning with a brief Introduction on the field as a whole and the importance of the project, and a brief statement about each method implemented and its importance. Following that will be a section on the research done before implementation and the methodology for the project. Then an analysis on the current state of my project and the data collected, ending with a conclusion and future work needed.

Introduction:

As the field of artificial intelligence grows within the technology industry, as well as industries it might not seem immediately applicable, I wanted to explore this growing advancement by creating and testing my own AI models.

With how rapidly artificial intelligence is progressing many people are worried about AI growing to a level beyond human comprehension and intelligence. To get an estimate right now on the strength of AI, we can compare data between two different levels of Artificial Intelligence against a human in the game of Snake. Comparison between computer and human abilities can show that even as the field grows it is rather difficult to beat the human mind in computing power and ingenuity.

If you're unfamiliar with Snake, it is a very early videogame where you play as a snake that is constantly moving and only has the single goal of eating as much fruit as possible. The game is fielded in a certain NxN square with each piece of the snake taking up one square (N size). Once a fruit has been eaten the snake grows in length, so as the game goes on it gets harder to maneuver around the playing field due to the increasingly filled squares. The game ends if the player hits a wall or any part of its own body, setting the score for that round to however many pieces of fruit the snake had eaten.

Through this comparison of scores in the game Snake we can see an accurate measure of ability between a human mind and a computer. Even though the game may seem simple enough, travelling from one goal node to another over and over, eventually planning is needed and some thought is required to make it to a high score. Through this complication we can see just how much the AI is able to handle in both methods, compared to the human which might not be able to play the game as fast as the AI, but can see ahead and plan moves.

The first Artificial Intelligence approach implemented into the project is the A* search algorithm that will lead the snake down the fastest path to the goal. This approach is very basic and struggles to deal with more complex situation such as changing states. For example, in Snake the changing length of the snake and the risk of surrounding oneself in the walls or within its own body. The second implementation is a machine learning approach, specifically a Deep Q-Learning technique that involves rewarding the snake for taking certain actions like getting closer to the goal and heavily rewarding the model for reaching the goal. The same approach is used as punishment for actions you want to have the model avoid such as colliding with the wall, or its own body. This method requires a lot of time and effort, as to train the model it can takes thousands of episodes to achieve even a small improvement, and in a game where the environment is constantly changing due to the growth of the snake it is difficult to train the model in conditions where the snakes' length is both short, and then when the snake is longer.

Research and Methodology:

Test Bed:

The programs were created and ran for initial testing on an Apple MacBook Pro M2 Max. The MacBook has a total of 32 GB of ram. For the main testing of the program and all data discussed and shown below the program was ran on a Desktop PC with a Ryzen 7950x3D CPU, a NVIDIA 4090 GPU, and 64GB of ram to collect the most accurate and fair data. When tested and ran on the desktop the program was created to use the GPU for the heavy lifting. All code was written and ran through VSCode.

From my initial research, the A* algorithm is most efficient in scenarios requiring the shortest and safest path to a goal. It combines Dijkstra's Algorithm, which explores vertices closest to the starting point, and Greedy Best-First Search, which prioritizes vertices closest to the goal using heuristic estimates. By combining these methods, A* achieves both efficiency and accuracy.

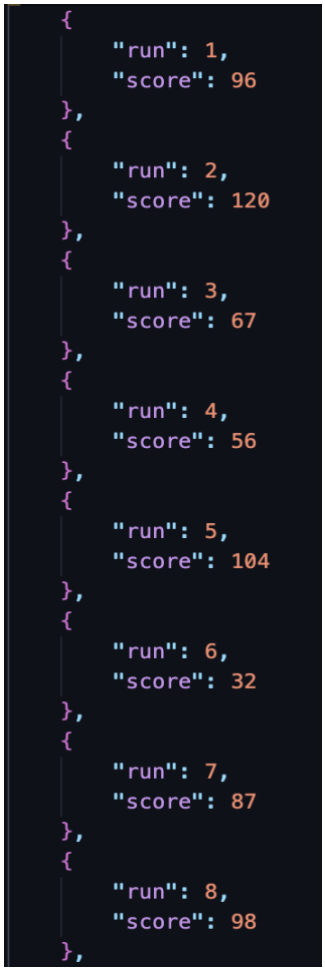
For the Snake implementation, the A* algorithm successfully directs the snake to the fruit until it becomes trapped. I found that adding a "coiling" behavior allowed the snake to wait safely for a new path to emerge, improving its performance. The methodology behind this coiling feature will be discussed more in depth in the Methodology section. However, the algorithm remains unable to anticipate future consequences of its actions, often resulting in the snake getting trapped in a 1x1 section or a corner. Testing over thousands of runs showed that while A* achieved high scores, it faltered in later stages of the game. Throughout the research section, not many readings were found that implemented A* into Snake as it is just a basic search algorithm it should have no issue playing the beginning stages of the game when the field is mostly empty.

In my literature review section of the project, many of the readings on learning models were specifically CNN (Convolutional Neural Network) models, but when starting the actual implementation, I found myself doing a lot of research and instead implementing the above mentioned Deep Q- Learning technique, as it seemed far more efficient and easier to implement specifically in python due to the torch library, which allowed for a much easier time building out the learning model. Even though the technique changed, the main idea behind the AI is the same. The AI has all the same information that a normal person playing the game would have and can do all the same things a person could; the main difference is the time it takes for the AI to learn how to play the game and achieve a higher score. A quick note on what differs between this type of learning AI and normal AI, like what was discussed in the previous section: "The world is consumed with the machine learning revolution, and particularly the search for a functional artificial general intelligence, or AGI. Not to be confused with a conscious AI, AGI is a broader definition of machine intelligence that seeks to apply generalized methods of learning and knowledge to a broad range of tasks, much like the ability we have with our brains." (Lanham, p.8). The main difference is that the A* method is a much more methodical, mathematical approach, with heuristics and routing involved to find the fastest path. Whereas with a learning-based method, the AI just does what it wants every round, learning more and more about how to score higher points. From the reading *Hands-On Reinforcement Learning for Games*, there are four main elements to reward-based learning: the policy (representing the decisions and planning process of the agent), the reward function (the amount of reward an agent receives after completing a series of actions or an action), the value function (determining the value of a state over the long term), and finally the model (representing the environment in full, all game states).

The programs were fully written in the Python coding language, with key libraries being:

PyGame, Torch, pandas, NumPy, and json. PyGame was used for all the visualization in the game, and basic game mechanics such as rendering, updating, and initializing the screen and game parts. The Torch library was used fully for the learning model, handling and creating the neural net and handling all Q-values etc. Pandas and NumPy were used together to handle the plotting and graphing for the data of the project, creating all the graphs shown later in the paper. JSON was the main method of saving data from all the runs into specific files, to than be loaded into the function used to create the graph and file the information onto the parts of the screen that display things such as total runs, last run, high score etc.

As said above the data is stored in JSON files depending on the set mode of that run. Each method is connected to two files one being the file that stores a collection of every run, and another storing the statistics of every run, shown below (Structure does not differ between methods):



Through the statistics file we can calculate things like average score etc. The information is stored this way to allow easy read and write especially for the creation of graphs after runs etc. The only different file for storage is a file called “current_automation.json” that is constantly reset whenever a new automation is started, and filled specifically with information from that specific run for graphing of a specific amount of turns that does not include the conglomerate of data stored in the normal files.

Information stored for the learning model is all handled by the Torch library and is in a file titled “model.pth” This file alone stores the memory and history for the learning model to learn

from. The A* method has no need for storage as its pathfinding is done fully during the game, this information however is logged to the terminal and if needed can be logged to a file for storage.

For this specific Snake implementation, the A* implantation finds the best path to the goal and the snake follows that path, updating the path every time the snake moves. This path is generated using the Dijkstra's Algorithm combined with a Heuristic function, for this implementation the Heuristic function will consist of a Manhattan distance from the snakes' head to the goal node. As I could not find much on a basic path finding implementation into Snake, this creates the main goal for this part of the project to compare this algorithm to a normal player and see how it reacts to the changing environments.

The learning model, however, is more complicated. For the rewarding of the model, like discussed before, we give the model a reward whenever it reaches and goal and a negative reward whenever it has a collision. The main goal of this section of the project, like the A* implementation, is to compare the learning model to not only a human player, but to the A* algorithm as well.

Both models will also be put into different environments than just the normal state of the Snake game but an environment with preplaced obstacles, to see how each handles the changed environment. The prediction being that the A* shouldn't be affected as it is programmed to avoid all obstacles. The main testing for this section will be the Learning model that was trained in an empty playing field, and it will have to react with the knowledge it already has to the preplaced walls.

Current Work and Analysis:

The first step of the project was the game and the visuals both having been fully implemented. The game can be played normally by a person using the keyboard and can be seen working on the screen of the device. This is done fully through the python library PyGame, rendering everything to its own application screen on the computer. PyGame also handles things like updating the screen every frame, player controls/movement, and all the rendering for both the log section and game section. When you launch the program through its main driver you will first be entered into a menu where you have three options, being Normal, A* mode, and learning mode. Selecting an option loads the game into those specific mode to be run one time or multiple by manually letting the game reset each time. Through the code however the game can be run in an automation mode that is mostly used for training the model and collecting data, letting the program run thousands of times without requiring any input.

In each run the metrics collected where, the number of runs, and the score achieved. This data as discussed before was than stored in a JSON file that allowed for calculations of averages and plotting onto graph for analysis. Again, individual runs were saved when set in automation modes to achieve graphs of specific amounts or methods such as the fully trained learning model.

The first Artificial Intelligence implementation, the A* algorithm, using the same logic for the base game we just create a function that is constantly being called in the Update function provided by PyGame that is constantly being called every frame of gameplay, allowing for anything inside of it to be called every frame as well. The function constantly being called allows the snake to find the best path, continue for a grid slot and then recalculate another path to the goal node. This may introduce some redundancy as the snake is constantly calling the search function every frame, but this also allows the snake to have the massive ability to react to a changing

environment almost immediately. The heuristic for the algorithm is a Manhattan distance to the goal, which is calculated and measures the sum of the absolute differences between the coordinates of the points, in this situation being the goal node and the position of the snake's head. Obstacles for the A* method is the snakes' body, and the walls, whether that being the normal wall border of the playable area or any preplaced walls to change the environment for the snakes.

After implementing this I did however find that the main issue of the algorithm was that the snake would often find no path to the fruit due to specific situations such as the snake's body blocking the fruit, or the snake has entered a loop of its own body. To solve this issue, I implemented another function that would have the snake coil itself whenever it was unable to find a path to hopefully allow the snakes bodies time to clear and the A* algorithm to find a safe path to the fruit.

This idea is implemented in two parts, the first being a function that is called whenever the A* algorithm returns "No Path Found" meaning that either the fruit is positions somewhere within the snake's body, or the snake is stuck within its own body. If the snake did not have any function to combat this situation it would simply keep going in the direction it was and inevitably hit either itself or a wall. This function than collected all the empty squares around the snake and saved them to a list. The second part of the function is another function titled "flood_fill" that uses these empty spaces in the area around the snake and makes moves to "coil" the snake, filling all available space in a way to maximize the remaining space. During all this the function is still calling the A* algorithm in the update section of the gameplay loop and the moment it finds a path reverts to that new path to follow.

This implementation did improve the A* algorithms chances, but the algorithm still however could not see the consequences of its actions, leaving the snake to just coil up in a

situation it would not be able to escape from. On average the way this algorithm had a collision the most was sending the head of the snake into a situation where nothing it does can result in it continuing, such as running into a 1x1 section, running into a corner etc. After testing this algorithm 1000's of time it did achieve a reasonable high score of 124 and an average of 70, a score that many normal human players may struggle to achieve. This data however does show the weakness of this algorithm being later, higher score portions of the game.

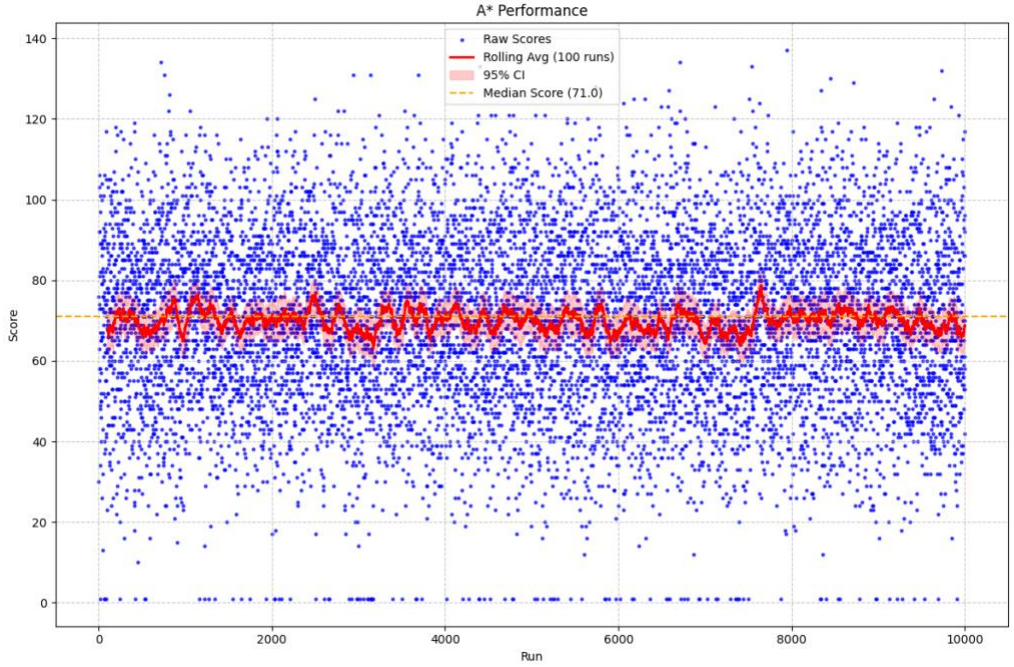


Figure 1 A Algorithm Run Data.*

From figure one we can see that the A* algorithm is rather consistent with the average staying mostly the same over the 10,000 runs. The average score for all 10,000 runs was 69.98 staying consistent with the median shown in the graph. Looking at the graph though, there is some inconsistency with a lot of 0 scores appearing through the episodes. This error or bug can be assumed to be from a glitch in the code where the A* algorithm doesn't start right away if the snake speed is set to a high number leading the snake to just collide right away with the wall. Another bug in this implementation is that if the model, algorithm, or player inputs two different

movement commands too quickly the snake will not move fast enough and collide with itself causing the end of that episode.

The Deep Q learning model is implemented using the python library Torch. The implementation is broken up into a couple parts being the Q network, and training modules, and the state of the board. To give the model the best chance both a long-term memory and short-term memory have been implemented with the table being updated every 100 runs. The information sent to the snake is very similar to what a human player would be able to receive just playing the game normally, being: If the snake is in danger of a collision in each direction, the relative food position, and the current direction the snake is moving. The reward function for the model is simple, if the Snake reaches the goal it is rewarded with a value of 20, if it has any collision being either with a wall or its own body it is penalized 10 points. The model is also rewarded with a value of 2 if while moving it is getting closer to the goal to help the snake learn to get to the fruit as quickly as possible, it is similarly penalized a value of 1 if the model is moving farther away from the goal. I tested multiple different reward values and state representation and still think there is room for improvements as in Fig 2 shown below the learning stagnates rather quickly. I also found that if the reward function penalizes a death more than collecting the goal the snake would no longer want to collect the fruit and would rather just spin in circles in the middle of the board.

The values for the models' settings were tweaked and are still up for change to try and get the model to achieve higher scores, in the current implementation and for the results seen below they are as follows. The learning rate was set to a consistent 0.002 I attempted to implement a decaying learning rate to facilitate the snake having explosive learning at the beginning when most valuable and then use the lower learning rate towards the end of the episodes to refine its memory. The max memory and batch size were set to 100,000 and 1000 respectably as the testing was done

through a RTX 4090 GPU these numbers probably could have been increased, but as the game isn't the most complex, I did not want to break anything. The epsilon was set to an initial value of 1, with a minimum of 0. The initial value decayed over the course of the episodes by a decay rate that was calculated by the number of runs being tested. This decay rate would lead the epsilon to zero by the end of the session with a couple runs left to get a measure of the model at zero exploration. Data from the learning model with these settings is displayed below:

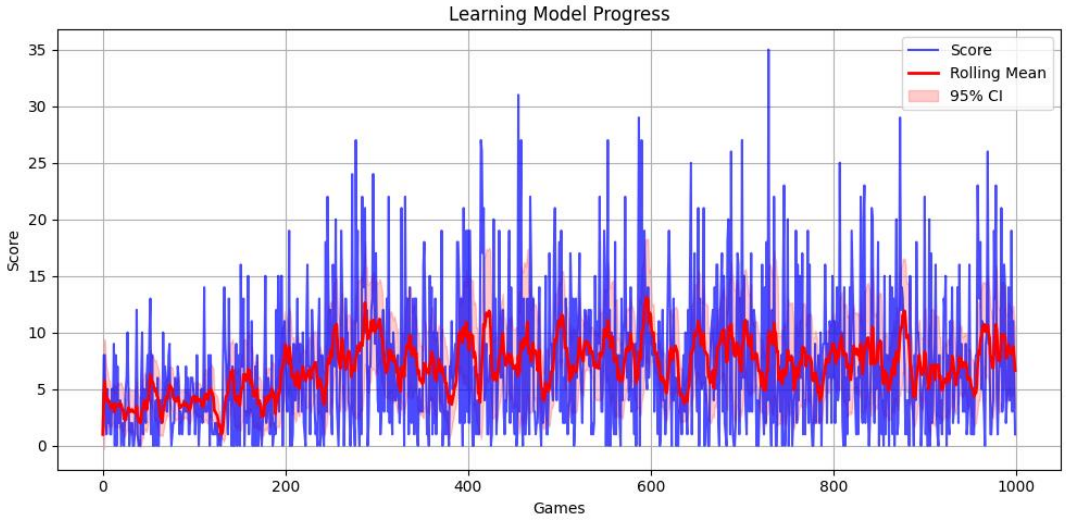


Figure 2 Learning model training data.

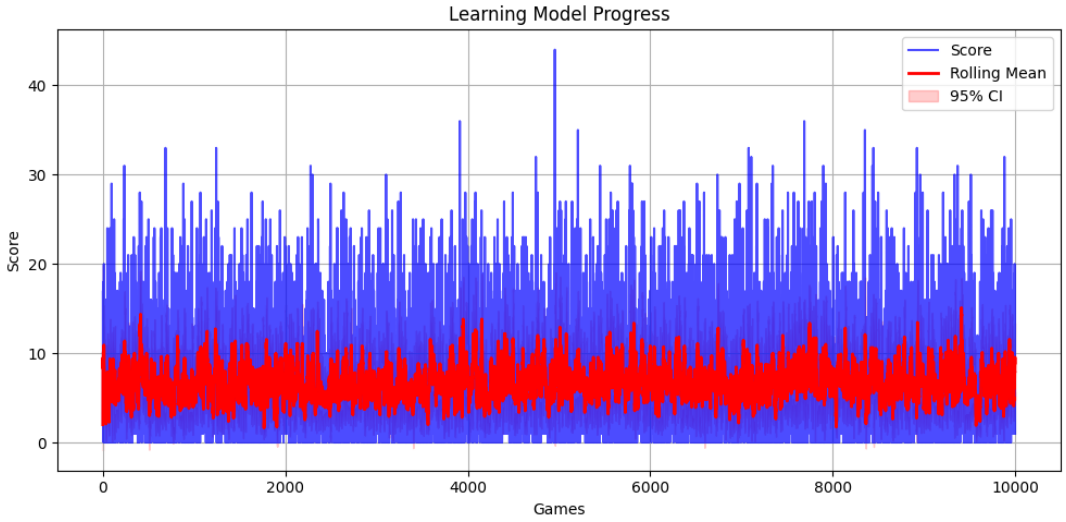


Figure 3 Learning model with epsilon 0.

Throughout all testing for each of the AI modes, an automatic mode was used that didn't require any input from the user to continue onto the next run, this was done through starting the program with special parameters were the user could set the mode to be tested, and the number of runs to be run. For human scores however, the data was collected manually by having a human play the game. The human data was only a couple hundred of runs to avoid having to manually play the game thousands of times, the A* approach was tested a total of 10,000 times to collect ample data, and finally the Learning model was allotted 10,000 runs to collect data and train itself and then to get an average score at that point was set to have a epsilon score of 0 (The amount of exploration allotted) and allowed another 10,000 runs. Each model was then ran another 1000 times in an environment including 15 additional walls scattered randomly through the playable area to test how the models react to different changes in the environment.

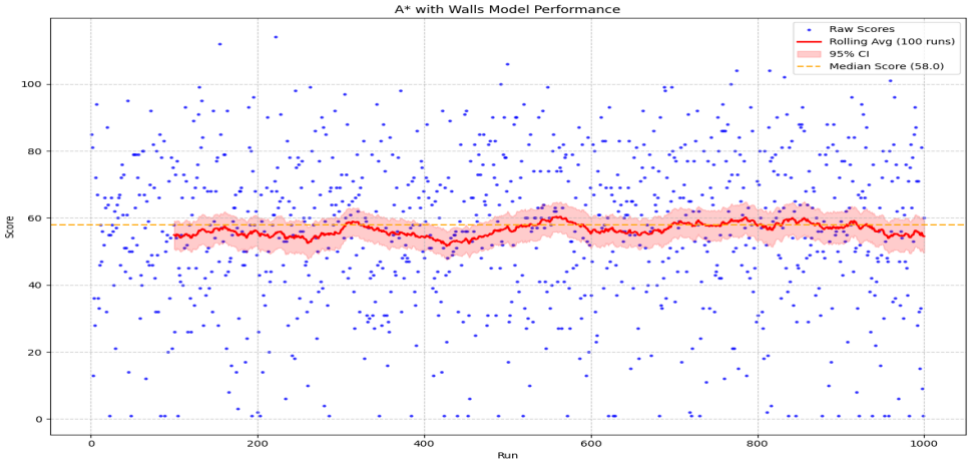


Figure 4

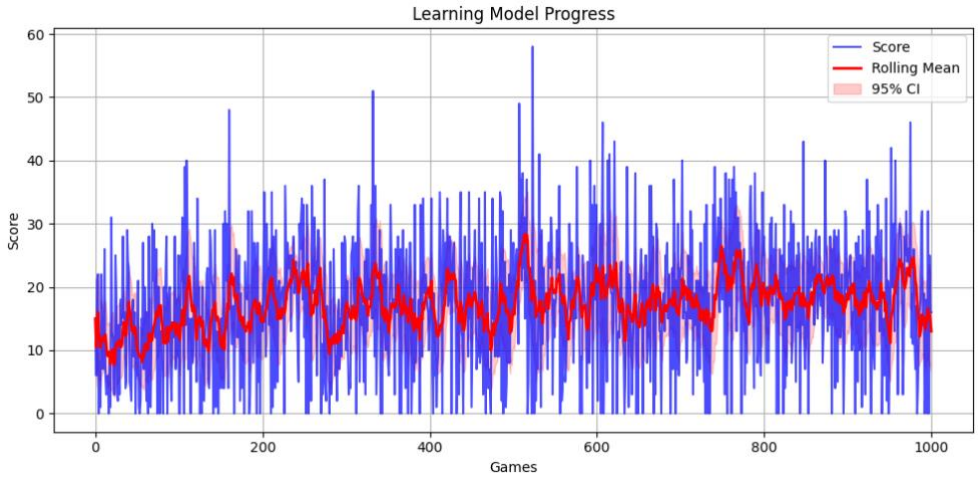


Figure 5

From Fig 4 the A* algorithms averaged a score of 56.21 dropping 13.77 points or 19.68% a fairly large change all due to the more complicated field. Whereas in the Learning model Fig 5, we see an increase in the average score from around an average score of 8 to 12, showing the adaptability of the learning model even from just 1000 additional runs in the environment.

Conclusion:

In conclusion it can be difficult to compare the two different intelligence methods as the Learning model takes immense effort to train and compete at a high level, whereas the A* method is a more rudimentary drop and play style. Comparing the methods to a human player is simpler, both methods can play the game at a much faster speed. Both methods were run at almost 50x the normal gameplay rate, showing how computers can make decisions must faster than a human brain.

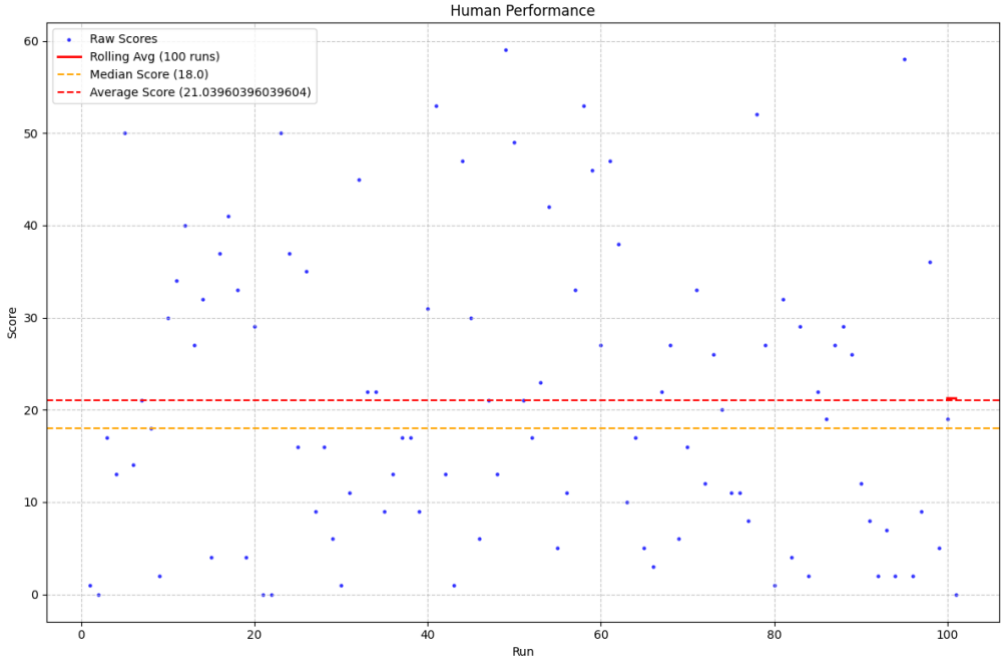


Figure 6

Though Fig 6 we can see that the human average isn't very consistent, especially when compared to the models, and from the same idea the human struggles with high score consistency as it takes a large amount of time to get to a high-score and even then, a human error means the end of the run. The average score being 21 also is much less than the A* average, and like the later average of the learning model. This was only 100 runs so the data isn't the most complete and can be seen

as subjective as the data is only from me personally playing the game, not accounting for the average human performance.

The project still requires some work, mostly with the learning model and its training. Currently the learning model has only done 10,000 training runs and isn't achieving a very high score compared to the A* method. While doing some more testing, I also found that the snake speed may have been detrimental to the learning model as the high speed may be lowering its average, so to test this theory another training run will need to be made on a slow speed, which would take a very long time. Overall, the code base could be refined more as it is a little messy in parts, but those are the main issues still left over.

Through this project I have been able to learn a lot about artificial intelligence, even though rudimentarily it was great practice and will help further my advancement in my career and learning. The project was also a great learning advancement in my graduate studies as I have never done a project as large of scale of this one by myself.