Snake Game: A genetic neural network approach

Shen Hau Hor
School of Computing
Asia Pacific University of Technology & Innovation (APU)
Kuala Lumpur, Malaysia
tp061524@mail.apu.edu.my

Mun Kye Yan
School of Computing
Asia Pacific University of Technology & Innovation (APU)
Kuala Lumpur, Malaysia
tp056066@mail.apu.edu.my

Yoke Shin Sim
School of Computing
Asia Pacific University of Technology & Innovation (APU)
Kuala Lumpur, Malaysia
tp059851@mail.apu.edu.my

Sheng Jeh Tan
School of Computing
Asia Pacific University of Technology & Innovation (APU)
Kuala Lumpur, Malaysia
tp056267@mail.apu.edu.my

Zailan Arabee bin Abdul Salam
School of Computing
Asia Pacific University of Technology and Innovation (APU)
Kuala Lumpur, Malaysia
zailan@apu.edu.my

***Abstract*— There have been multiple attempts at playing the snake game with AI approaches which include Genetic Algorithm and Neural Network. This paper aim to research on how tuning several parameters of genetic algorithm and neural network will affect a snake agent in its performance. The parameters changed in this experiment are the mutation percent, percentage of best/worst performing, mutation intensity and arena size to test the effects of each parameter on the performance of the snake agent. The consistency as well as performance of the snake agent are both observed closely in this study. We have found out that each parameter has its own degree of effect on the performance of the snake agent.**

Keywords—snake game, genetic algorithm, neural network, parameter tuning

I. Introduction

The snake game was originally created as a mobile game by Nokia back in 1997. It quickly gained popularity as it was a simple yet addictive game. This game involves the user controlling a single block known as the snake within a fixed-size arena and only able to move it in four 90-degree directions, forward, backward, left and right to search for food. Upon the collection of each food, the snake grows an extra body segment which will result in gradually increasing the difficulty of the game. As the snake collects more food the snake dies when the snake collides with the arena wall and its own body. The scores of the snake will be higher as it eats more food. The main objective for the player playing the game is to collect as many food and surviving for as long as possible to obtain a high score.

Snake is a computer action game, and many researchers demonstrate it using various classical algorithms. Up to now, a number of studies have indicated to use Genetic Algorithms (GA) [3], Deep-Q-Network (DQN) [16], Double-DQN, Proximal Policy Optimization (PPO) [5], Q learning Algorithm, SARSA [2], A* Search, random move, almighty move respectively, best first search[11] and Evolutionary algorithms (EA) [19] as the algorithms for playing the snake game. In previous studies on snake game, the type of input of genetic neural networks approach has been found that it can affect the snake to perform a strategy better from basic input sensors [9]. Another research has clearly explained that a slow algorithm can help the snake to achieve the highest score possible while a fast algorithm can help to achieve a score faster but also get the game ended faster than slow algorithm [11]. Surveys such as that conducted by Yamini & Jain [18] and Bialas [3] have shown that an accurate result should have an adjustment of parameters like higher runs and higher generations will affect the scores become higher.

One of the algorithms involved in the study is called Genetic Algorithms which belong to the category of optimization algorithms and they are usually used to discover the best optimal solution for an issue that decreases or increases a function. Genetic algorithms also came from a field of study that is known as evolutionary computation in which the algorithm was used to copy the reproduction process and select the fittest solutions. This function enables the algorithm to discover solutions to issues that other methods cannot pick up due to a lack of features. The basic structures of the algorithm are fitness function for optimization, a population for chromosomes, chromosomes to be selected, crossover for next generation chromosomes, and also mutation of chromosomes in the latest generation. Hence it can be said that Genetic Algorithm is suitable to be applied in sectors that involve finding the optimal solution for the problems [8].

On the other hand, another algorithm that is involved in this study is known as Artificial Neural Network is an emulation of the human brain. It belongs to the Artificial field alongside Expert System and Fuzzy Logic. Besides, Artificial Neural Network also consists of interconnections of artificial neurons that carry signals in the same way of how a human brain does. The artificial neuron will be trying to reproduce the structure of the natural neuron which consists of input and output and also the function of activation. The simplest form of an Artificial Neural Network architecture is known as the Perceptron which is made up of one neuron with two inputs and one output. The way of training neural networks can be undergone through using the back propagation algorithm which uses supervised learning and also an architecture known as feed-forward as it is one of the most used methods for categorization and prediction [12]. Therefore, it can be said that an artificial neural network is certainly a great addition to the large algorithms group that can be used to solve problems.

The use of Neural Network and Genetic Algorithm has been used in playing games. Many researchers had done experiments that are able to play games successfully using some version of neural networks, while some even manage to match or even surpass human performance [9][17]. Typically, Neural Networks are not utilized as a standalone solution, but is bundled together with other algorithms like Evolutionary Neural Network [3][4][7][9][13][14][15][19] or deep Q network [5][17]. There are several parameters that can be tuned in Artificial Neural Networks, which include number of hidden layers and number of hidden nodes [3]. In the study of Mishra et al, they found that models with Neural Network that have only a single hidden node may sometimes give high performance, however it was very unpredictable, hence it is unreliable

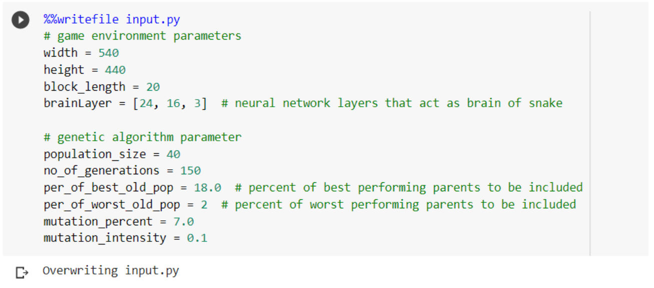
Studies on the use of Genetic Algorithm to solve problems has also been successful on problems like Balancing Biped Characters in Games [7], Playing the Snake Game [3][9], Solving Cross-matching Puzzle [10], and also Enhancing the Computational Performance and Quality of Search Engine Results [15]. However, there are also some shortcomings when we implement a solution using Genetic Algorithms. Although the Genetic Algorithm is able to generate a good set of controllers, it is only can be applied to a specific game environment. The trained controller does not perform good enough for unknown game environments compared to using Heuristics controllers [9]. Genetic Algorithms also generally take a lengthy time to run and to achieve the convergence of a problem [7][13][9].

II. Materials and Methods

A) *Materials and Methods*

In order to run the snake game bundled with Genetic algorithms and Neural networks, we used python 3.7 as a tool to run the game. The source code that we used was originally developed by Ali Akbar[1] allowed us to modify the parameters of the genetic algorithm and neural network. This source code allowed us to change the parameters, train the snake using Genetic algorithm, save the model as a pickle file which is used to sequence and deserialize Python objects such as lists and dictionaries into byte streams of 0 and 1, and display the results of the trained snake by running the snake game. In this case, we realized that the code could not record the results of each run. Therefore we made some modifications to game.py in the source code and pushed it to our fork in GitHub [6].

After modifying the code, we chose to use Google Colaboratory to train the snake to ensure that we are in a consistent environment when running the code. This platform has a fixed model such as Intel(R) Xeon(R) CPU @ 2.00GHz and 13G of space to run the code.

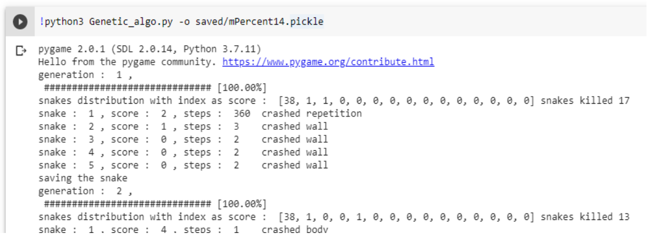


```
%%writefile input.py
# game environment parameters
width = 540
height = 440
block_length = 20
brainLayer = [24, 16, 3]  # neural network layers that act as brain of snake

# genetic algorithm parameter
population_size = 40
no_of_generations = 150
per_of_best_old_pop = 18.0  # percent of best performing parents to be included
per_of_worst_old_pop = 2  # percent of worst performing parents to be included
mutation_percent = 7.0
mutation_intensity = 0.1

Overwriting input.py
```

Fig. 1. Baseline parameters

The trained snake will save as a pickle file, and we can download the trained model and run it on our local devices. The recorded performance, such as number of generations, crash causes and scores will be saved as a csv file in the save folder and all the record will be used for demonstrating the results of this paper.

There is a file that was created to store the input of the snake game which is “input.py”. The purpose of that file is to control all the parameters of the snake in the game which are the game environment parameters and genetic algorithm parameters. There is a baseline set for the parameters so that the results can be compared later on. The snake will get trained from the first generation until the last generation which is 150. Since we also have to test how the change in parameters affected the snake performance, we changed each of the parameters in “input.py” by overwriting the parameters using the “%%writefile” magic function. For instance, in order to test how the changes in mutation intensity will affect the snake performance, the mutation intensity parameter in “input.py” needs to be overwritten.



```
!python3 Genetic_algo.py -o saved/mPercent14.pickle

pygame 2.0.1 (SDL 2.0.14, Python 3.7.11)
Hello from the pygame community. https://www.pygame.org/contribute.html
generation :  1 ,
 ############################## [100.00%]
snakes distribution with index as score :  [38, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] snakes killed 17
snake :  1 , score :  2 , steps :  360  crashed repetition
snake :  2 , score :  1 , steps :  3    crashed wall
snake :  3 , score :  0 , steps :  2    crashed wall
snake :  4 , score :  0 , steps :  2    crashed wall
snake :  5 , score :  0 , steps :  2    crashed wall
saving the snake
generation :  2 ,
 ############################## [100.00%]
snakes distribution with index as score :  [38, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] snakes killed 13
snake :  1 , score :  4 , steps :  1    crashed body
```

Fig. 2. Training the agent

The code block above demonstrates how to train a model using the genetic algorithm and saving the pickle file as any specified name. The trained model is then downloaded onto our local device. The snake was then trained by using each variation of the particular parameter once and each trained model was saved in a separate pickle file. Once the pickle files were obtained, the next method was to run the snake using Python with the specific pickle file. Before the snake game starts, the program will prompt for the input of the file directory where we would like to save the result csv file. Once we specified the filename, the snake game would begin. After all generations of the game is finished, the results csv file can be obtained from the file directory that was inputted before the game started. Each variation of the parameter is being tested to run for a total of 3 times to get an average result. The process is then repeated with the other pickle files.

The results were then exported into Microsoft Excel and the line graph was plotted using the Insert Graph function in Excel. Once each variation of the parameters has completed the run, the results were recorded and analyzed in the Results and Discussion section.

B) *Algorithm Implementation*

The Artificial Neural Network is served as the main decision-making part of the snake agent and Genetic Algorithm is used to optimize the Neural Network of the snake game model. The Artificial Neural Network allows the agent to determine what steps should be taken at each point of the game. Meanwhile, the Genetic Algorithm is responsible for applying the idea of natural selection to filter the best set of weights for the Neural Network for playing the snake game. At the end of the experiment, we aim to find out what are the factors that affect the performance of the snake agent.

There are several sets of parameters that we have changed to study the performance of the snake. Firstly, the mutation intensity of the genetic algorithm. The mutation intensity defines how much any weights in the neural network can change if it undergoes mutation when producing the next generation. Given mutation intensity is set to x, the actual value change will be a random value between -x and x. Secondly, mutation percentage is dictating how many weights in the neural network will undergo mutation for the next generation. Given mutation percentage p, the number of weights that will undergo mutation is between 1 and p, which will be determined randomly. As for the arena size, we have modified the block length parameter which will define the width and height of each block in the game screen. The smaller the block length, the bigger the arena. Lastly, proportion of best performing parent vs worst performing parent for breeding defines the ratio of best performing agent to worst performing agent to be used for breeding the next generation of agents in the genetic algorithm. We have kept the proportion of parent agents to be included in the next generation at a constant of 20%. The remaining 80% will be the offspring of the said parent agents.

We have defined a set of constant parameters as our baseline agent as shown in Fig 1.We have studied the change of 4 sets parameters on the performance of the snake agent. The first experiment, we have investigated how the mutation intensity of the genetic algorithm with mutation intensity of 0.01, 0.1 [baseline] and 0.5 respectively. On the second experiment, we trained the agent with mutation percentage with 3.5%, 7% [baseline] and 14% respectively. For the third experiment, we ran the baseline agent by changing the block length parameter, which directly corresponds to the arena size. The block length that was used was 20 (27*22 [baseline]), 10 (54*44) and 5 (108*88) respectively. As for the last experiment, the proportion of best performing parent vs worst performing parent for breeding is set at the ratio of 18%:2% [baseline], 10%:10% and 2%:18% and is trained respectively

III. Results and Discussion

A) *Mutation Intensity*

This experiment's purpose is to investigate the effect of changing mutation intensity on the snake game. For the experiment, we had studied 3 variations of mutation intensity which are 0.1, 0.5 and 0.01 respectively. Each mutation intensity is being trained and run for 3 times in the snake game and the results are being recorded. The performance of the snake in each generation is being averaged based on the 3 times runs. The baseline of the mutation intensity is being set at 0.1 and we also tweaked the intensity to 0.5 and 0.01 for the test. The reason that we decided to select 0.5 and 0.01 mutation intensity for the test is because we want to observe how such differentiation will impact the snake performance in the game.

Based on the results, we can observe that when the snake that is trained with baseline mutation intensity of 0.1 has achieved a highest score of 60.00 on average. The results show that the performance suffered a big drop on generation 43 before it bounced back and started to fluctuate until the last generation.

For mutation intensity of 0.5, the snake managed to achieve a highest score of 81.33 on average. The results graph fluctuated between generation 43 and 146 until the performance is getting better and hits the peak at generation 92 with the highest average score.

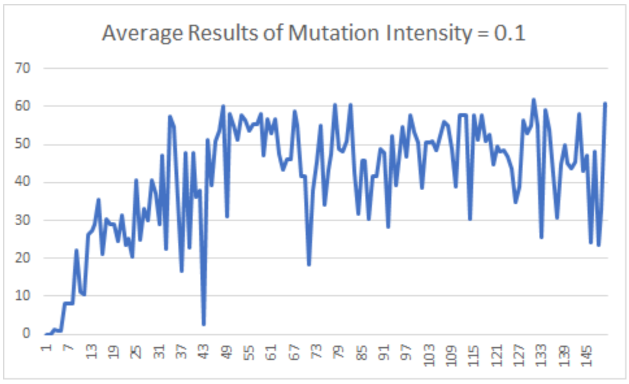


Fig. 3. Mutation intensity 0.1

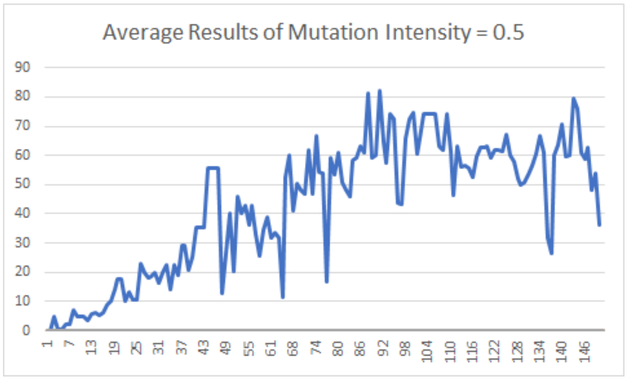


Fig. 4. Mutation intensity 0.5

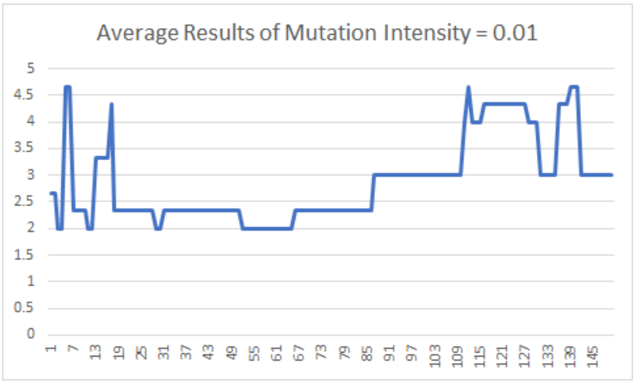


Fig. 5. Mutation intensity 0.01

Furthermore, after the mutation intensity has been tweaked to 0.01, the performance of the snake is getting worse as the snake managed to achieve the highest score of 4.50 on average. The performance suffered a great drop starting from generation 19 until reaching generation 67. Although the snake was getting better scores on average starting in generation 67 , it is very low when compared to other mutation intensity. Overall, the performance of the snake at this mutation intensity was the worst when compared to the other intensity.

After comparing the results, we can find out that the snake's performance is the most consistent when it is being trained at 0.01 mutation intensity. One of the reasons is because lower mutation intensity would not affect each generation of snake too much as all of them are almost similar. Although sometimes the performance might fluctuate, the results are proved to be the most consistent among all the mutation intensity.

However, having consistent results does not guarantee that the snake is getting good scores. We can observe that higher mutation intensity snakes are getting much higher scores when compared to lower mutation intensity snakes. This is because at a higher rate of mutation intensity, some of the snakes tend to mutate much greater than others. Thus, this will cause inconsistency in results but at the same time higher scores might be achieved.

To summarize, the mutation intensity to be chosen should be in medium level as too low mutation intensity will cause the snake not to be performing at its best while high mutation intensity will cause the results of the snake to be inconsistent.

B) Mutation Percentage

We had studied 3 different mutation percentage, which is 3.5%, 7% and 14% respectively to study how it affects the model's average performance. Each variant is trained, and the trained model is used to play the snake game for 3 times. The performance of each generation is then averaged out across the 3 separate runs. The baseline of 7% mutation has been set. The reason for choosing 3.5% and 14% a 2-fold increase and decrease will show how the performance is affected with different mutation percentages, but not on an extreme scale.

As we can see, the agent with a mutation percentage of 3.5% only has a maximum score of 22 on average. Except for the first few generations, the line graph increases and decreases consistently around the value between 5 and 15. It does have performance spikes and dips especially around generations 65 to 100. However, the difference between the fluctuations is around 15 to 20.

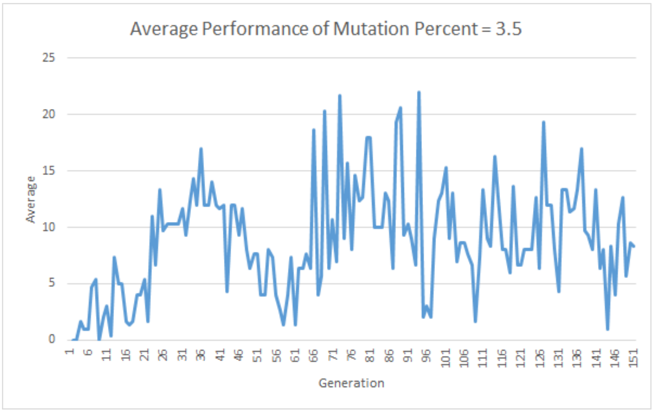


Fig. 6. Mutation percent 3.5

As for mutation percentage 7%, the performance of the agent goes on an upward trend until generation 70. Then it drops slightly for a few generations and at generation 83, it takes a deep dive from average performance of 33 to 10.33. A similar trend can also be seen in generation 110 where it took a dip to 9.67 and quickly rises again to average performance of 56.67 at generation 116. Overall, the maximum average performance that is achieved by this model is 57.67, and the performance seem to fluctuate between the values of 10 to 50 after generation 70.

For a mutation percentage of 14%, the performance also has an upward trend. However, the trend is not as prominent as it is in 7% mutation as the performance fluctuates a lot between the value of 10 and 50 for generations before generation 61. The fluctuations of performance is even more noticeable around generation 123 where the agent can score a maximum average score of 61 and the next generations it plummets to average score of 2.

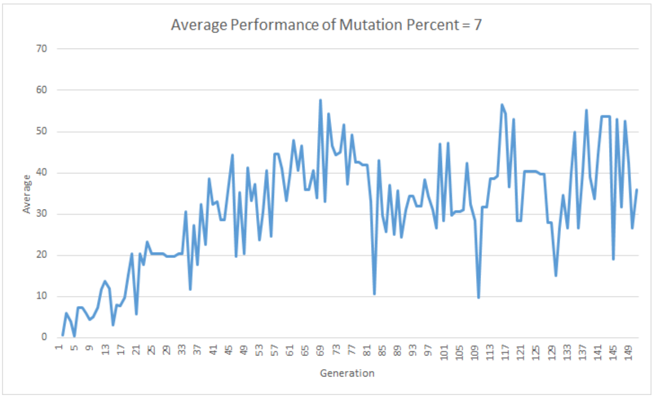


Fig. 7. Mutation percent 7

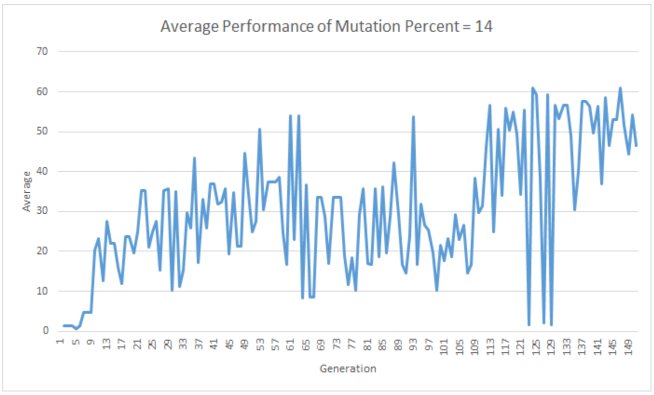


Fig. 8. Mutation percent 14

Overall, we can see the performance of the snake agent is more consistent across generations with a lower percentage of mutation. This is explained because the number of weights that are changed is lower. This caused the agent of the next generation not being too different from the previous generation, thus it will perform similarly with the previous generations. However, there surely will be some form of fluctuations in performance in the agent as each game at each generation is generated randomly.

It is also noticed that with a low mutation percentage, the model is not able to perform as well as those with higher mutation rate. This phenomenon occurs because with a lower number of genes being mutated each generation, there are lesser chances for the weights in the neural network to be mutated. This means that those weights that correspond to the performance of the snake will have a lower chance of getting changed or improved through the process of mutation. However, with a high mutation percentage, there are too many genes that undergo mutation. This caused some generations to gain a very good performance, but the next generation the performance plummet. This is because too many genes undergo mutation, making the parent and child differ too much in traits. These big fluctuations in performance cause the genetic algorithm to not be able to tune a good set of weights even after many generations of training.

To conclude, the mutation percentage of each generation should be chosen at a moderate percentage as too low mutation percentage takes many generations to converge and too high mutation percentage will cause the performance to fluctuate a lot between generations.

C) Arena Size

The parameter that is being altered here is the block length which is corresponded with the arena size in the "input.py" file which is responsible for the size per pixel in the agent's arena. The aim here is to observe the effects on the scores if the snake agent is trained under a specified environment and then put in a different environment. The size of the arena will increase as the value of block length decreases. The agent is initially trained using arena size 27*22 as the baseline and stored under the "baseline.pickle" file.

The block length parameter is tuned to values of 5, 10 and 20 accordingly in the "input.py" file. Each of these values then undergo three runs each, scores recorded and the average across the three runs is calculated. The baseline of 20 is selected and the justification for 5 and 10 block length is arbitrary as we want to reflect the effects of manipulating the parameter on the agent with modesty.

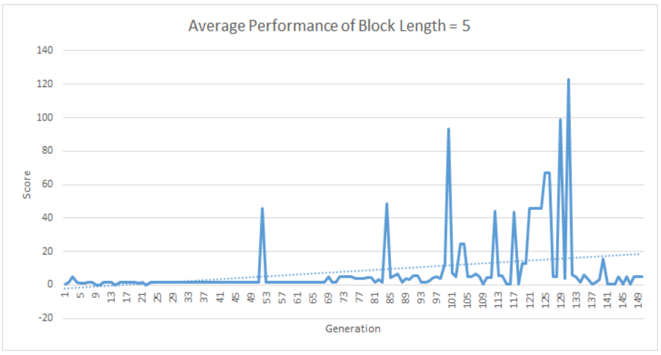


Fig. 9. Block length 5

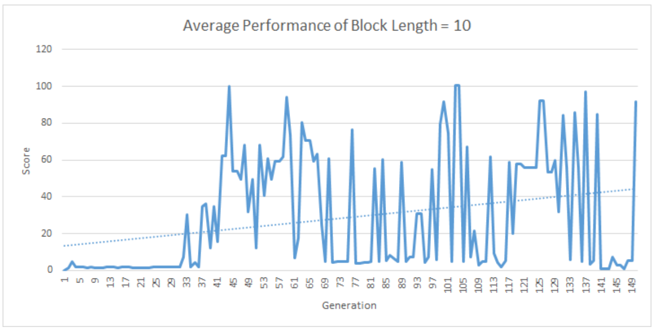


Fig. 10. Block length 10

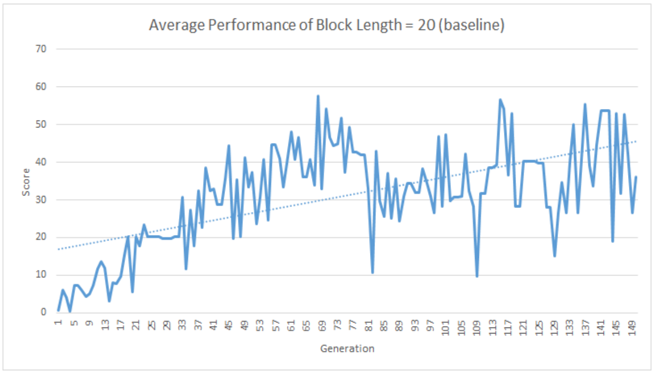


Fig. 11. Block length 20 (baseline)

TABLE I. RESULT OF 20 BLOCK LENGTH RUN

Game End Method	***20 block run***			
	Run 1	Run 2	Run 3	Average
Killed	23	12	22	19
Body	88	128	107	107.6667
Wall	39	10	21	23.33333

TABLE II. RESULT OF 10 BLOCK LENGTH RUN

Game End Method	***10 block run***			
	Run 1	Run 2	Run 3	Average
Killed	112	104	98	104.6667
Body	35	41	51	42.33333
Wall	3	5	1	3

TABLE III. RESULT OF 5 BLOCK LENGTH RUN

Game End Method	***5 block run***			
	Run 1	Run 2	Run 3	Average
Killed	144	143	144	143.6667
Body	5	6	5	5.333333
Wall	1	1	1	1

The agent tested with baseline parameters inclusive of block length 20 achieved a maximum average score of 57.67 and on average a score of 31.11. It can be observed that the agent has a steady increase in score until it reaches a short plateau and starts to have fluctuating scores from generation 37 onwards. These fluctuations continue to have its peaks and troughs on the line graph but have an overall increasing average score. The sharpest dip in average score was observed around generation 80 to 82 where the score dropped from 42 to 10.67.

The agent with its block length parameter changed to 10 has a peak average score of 100.33 and on average a score of 28.82. The agent has a plateau of single digit scores at the beginning but has a steep climb in scores around generation 33. There are drastic fluctuations in scores from generation 59 onwards to 150. These fluctuations are very noticeable as represented on the line graph with dipping as low as 4.66 from 100.33 at generation 103.

As for the agent with block length set at 5, it had the highest achieved peak average score amongst all the other agents of 122.67 and despite having inconsistent scores with spikes around generations 52, 84, 100 and 131. There are also significant drops in scores which are very noticeable around generation 131 and 132 where the average score dropped from 122.67 to 6. It can be observed that there are overall low scores but there is still an overall increasing trend in average score.

It can be seen that the snake agent is more consistent when put in the environment and parameters it was trained in but It lagged far behind in maximum average scores when compared to its tests using 5 and 10 block length. As the arena gets bigger when the block length value gets smaller, the snake agent had more room to navigate as well as maneuver, reducing the instances where the snake collides with its own body or arena walls.

This can be seen in the data where it is shown that the number of times the snake agent collides with its own body significantly decreases as the block length value decreases.

It is also observed that the snake agent put in lower block length has far lower scores and this is exceptionally noticeable in block length 5 where the snake agent has a consistent average score of lower than 2 between generations 1 to 81 where it's a completely different story when compared to the results of snake agent in block length 20 where the scores are consistently above within 25 and 57.67 from generation 53 to 150. There are some peculiar behaviors that are observed mainly in snake agents of block length 5 and 10 where the snake agent will just get stuck in an endless loop which forces us to kill it manually or it wouldn't progress to the next generation. This can be reflected in the number of "killed" count in results of block length 5 and 10. It is also visible that as the arena size increases, the average number of agents dying to wall collisions decrease.

In conclusion, we should train the agent in the environment and with the parameters that we intend to execute it in to obtain the best, consistent and performing results as can be observed in the baseline experiment of arena size 27*22 results which the agent was originally trained in.

D) Percent of Best/Worst Performing

In this experiment, we will identify and discuss the effect of 3 different percentage ratio of best to worst performing parents for breeding which are 18:2, 2:18 and 10:10. The percentages of best and worst performance were selected from the current generation and used to breed the next generation. These images show the average score for each variable over the 3 runs.

Fig 12. is about the average score of percent of best/ worst performing of 18:2. It has been set as the baseline for observing how the change of percent will affect the score. It has shown the score is increasing in a progressive manner, with a maximum of 60 points. The biggest reason for this performance is that its present best performance is set as 18%. This means that the 20% of the current generation which used to breed the next generation has a high probability of being the top performing agents. This also indicates that the good set of weights will have a higher probability to get inherited to the next generation.

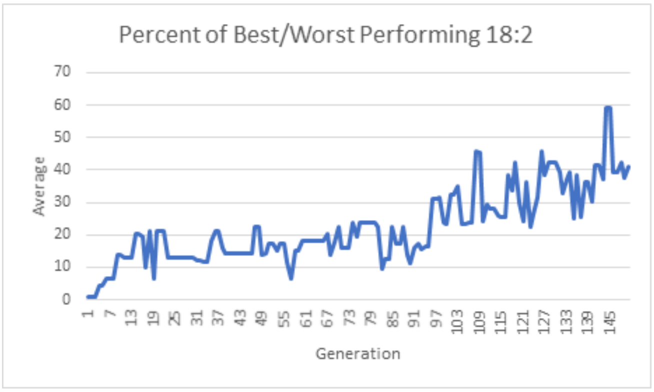


Fig. 12. Percent of Best/Worst Performing 18:2

Fig 13. is about changing the percent of best/worst performing to 2:18. We can see that the score of this graph has the lowest score among three graphs and its score rises gradually to about 2 and gets a 0 score after the 10th generation. The main reason for this performance is it has used 18% of worst score performance with 2% of best score performance to breed the next generation, the new generation will definitely have a high probability to get a badly performing parent, thus inheriting the bad trait and reducing the performance of the next generation.

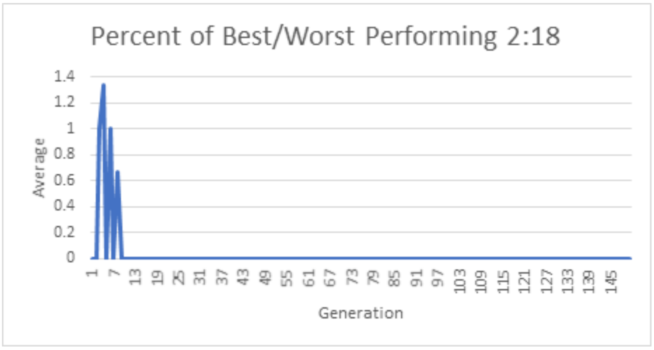


Fig. 13. Percent of Best/Worst Performing 2:18

Fig 14. has shown the performance of percent of best/worst performing of 10:10. As stated in the results, it got low scores in the first 50 generations, after which the scores increased but were very unstable. Also, the highest score of this graph, which has scored 55, is located at the 90th generation, even if this generation has scored the highest score, it will fluctuate and only get score of 2.

This is because half of the set of generations used to feed the next generation are chosen from the best and worst performance. Therefore, there is an equal probability of getting a good generation set that can lead it to food, or a bad generation set that will cause it to be killed by the wall or repetition. That is why the performance of the snake is not consistent throughout the 150 generations.

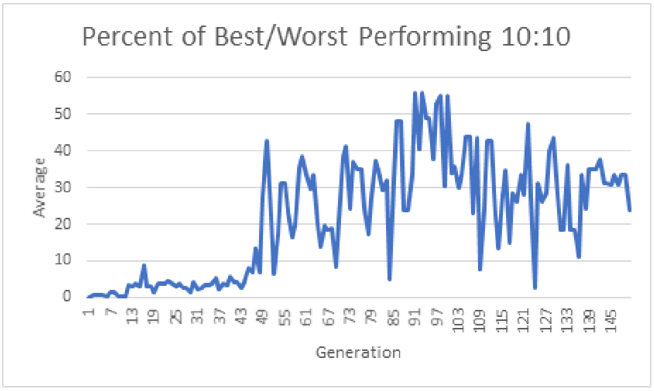


Fig. 14. Percent of Best/Worst Performing 10:10

After comparing the overall result, we can notice that the higher the percentage of best performing, the higher score we can get. The score of percent of best/worst performing of 18:2 has performed the best result of the score among the three variables as its score is increasing without instability. However, the performance of best/worst performing of 10:10 also has an increasing score, but it's score is unstable and can't get a higher score than the score of percent of best/worst performing of 18:2.

In a nutshell, referring to these three graphs, we can see that higher best/worst performing parent ratio can lead to a consistent and good performing agent. Thus, we should choose the highest percentage of best performing parents to ensure the trained model is competent.

IV. Conclusion

In conclusion, the performance of the snake agent under different circumstances of parameter changes is closely studied and analyzed. Each parameter have a certain degree of effect on the performance of the snake agent. It is observed that the agent performs well when parameters like block length, mutation percentage and mutation intensity are set at moderate values. As for the percentage of best/worst performing parameter, choosing the highest best/worst performing ratio of 18:2 will give us the best performing scores of snake agents. In future research, we could increase the number of generations for snake agent to observe if the snake agent performs worse or better if given more time and generations.

References

[1] A. Akbar. "aliakbar09a/AI_plays_snake: AI trained using Genetic Algorithm and Deep Learning to play the game of snake." *GitHub*. https://github.com/aliakbar09a/AI_plays_snake. [accessed Dec. 06, 2021]

[2] A. J. Almalki & P. Wocjan (2019). "Exploration of Reinforcement Learning to Play Snake Game". *In International Conference on Computational Science and Computational Intelligence, 2019 (CSCI)* pp. 377-381. [accessed Dec. 06, 2021]

[3] P. Białas (2019). "Implementation of artificial intelligence in Snake game using genetic algorithm and neural networks." *CEUR Workshop Proceedings*, http://ceur-ws.org/Vol-2468/p9.pdf [accessed Dec. 06, 2021]

[4] T. Boris and Š. Goran, 2016, November. "Evolving neural network to play game 2048". *In 24th Telecommunications Forum (TELFOR), 2016,* pp. 1-3. [accessed Dec. 06, 2021]

[5] R. Cai and C. Zhang, 2020. "Train a snake with reinforcement learning algorithms". *Open Review*. https://openreview.net/forum?id=iu2XOJ45cxo [accessed Dec. 06, 2021]

[6] C.Yan (2021). "mkcarl/AI_plays_snake: AI trained using Genetic Algorithm and Deep Learning to play the game of snake," GitHub. https://github.com/mkcarl/AI_plays_snake [accessed Dec. 06, 2021].

[7] C. S. Carlsen and G. Palamas (2019). "Evolving Balancing Controllers for Biped Characters in Games," *Advances in Computational Intelligence* (pp. 869–880). doi: 10.1007/978-3-030-20518-8_72. [accessed Dec. 06, 2021].

[8] J. Carr (2014). "An Introduction to Genetic Algorithms," [Online]. Available: https://www.whitman.edu/Documents/Academics/Mathematics/2014/carrjk.pdf. [accessed Dec. 06, 2021].

[9] B. Halmosi and C. Sik-Lanyi (2019). "Learning to play snake using genetic neural networks," *Pannonian Conference on Advances in Information Technology (PCIT 2019),* vol. 4, no. 5, pp. 126-132. [accessed Dec. 06, 2021].

[10] O. Kesemen and E. Özkul (2016). "Solving cross-matching puzzles using intelligent genetic algorithms," *Artificial Intelligence Review*, vol. 49, no. 2, pp. 211–225, doi: 10.1007/s10462-016-9522-6. [accessed Dec. 06, 2021].

[11] S. Kong, & J.A. Mayans (2014). "Automated Snake Game Solvers via AI Search Algorithms." *COMPSCI 271 INTRO ARTIFCL INTEL.* [accessed Dec. 6, 2021]

[12] H. Kukreja, N. Bharath, C.S., Siddesh and S., Kuldeep (2016). "An introduction to artificial neural network." *Int J Adv Res Innov Ideas Educ, 1, pp.27-30.* [accessed Dec. 6, 2021]

[13] M. Miller, M. Washburn, and F., Khosmood, 2019, August. "Evolving unsupervised neural networks for Slither. io." *In Proceedings of the 14th International Conference on the Foundations of Digital Games (pp. 1-5).* [accessed Dec. 6, 2021]

[14] Y. Mishra, V. Kumawat and K., Selvakumar, 2019, May. "Performance Analysis of Flappy Bird Playing Agent Using Neural Network and Genetic Algorithm." *In International Conference on Information, Communication and Computing Technology (pp. 253-265). Springer, Singapore.* [accessed Dec. 6, 2021]

[15] N. Nezamoddini & A. Gholami (2019). "Integrated Genetic Algorithm and Artificial Neural Network." *In 2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC) (pp. 260-262). IEEE.* [accessed Dec. 6, 2021]

[16] Z.Wei., D.Wang, M.Zhang, A.H,Tan, C.Miao, and Y.Zhou, 2018, July. "Autonomous agents in Snake game via deep reinforcement learning." *In 2018 IEEE International Conference on Agents (ICA)* (pp. 20-25). IEEE. [accessed Dec. 6 , 2021].

[17] M.Xu, H.Shi, and Y.Wang, 2018, June. "Play games using reinforcement learning and artificial neural networks with experience replay." *In 2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)* (pp. 855-859). IEEE. [accessed Dec. 6 , 2021].

[18] R.Yamini, and A.Jain, (2020). "GENERAL ARTIFICIAL INTELLIGENCE MODEL FOR DEVELOPING ADAPTIVE NON PLAYABLE CHARACTERS IN COMPUTER GAMES." *Journal of critical reviews*, 7(06). [accessed Dec. 6 , 2021].

[19] J.F.Yeh, P.H.Su, S.H.Huang, and T.C.Chiang, Snake game AI: Movement rating functions and evolutionary algorithm-based optimization. *In 2016 Conference on Technologies and Applications of Artificial Intelligence (TAAI)* (pp. 256-261). IEEE. [accessed Dec. 6 , 2021].