EVAC Assessment 1

Exam No: Y3868718

1 Methods

1.1 Changes to Provided Code

While the snake game discussed in this report is functionally the same as the initially supplied code provided with the assessment, some large changes have been made to the structure. Most notably, the snake game and neural network has been split into multiple separate scripts for ease of development. These files are detailed as follows:

1. `code/displayGame.py` - This file contains the `DisplayGame` class, the only change to this class is changing the snake body colour to be only green to remove any confusion on which way the snake is facing.

2. `code/runGame.py` - This file contains the `runGame` function, other than the code added to facilitate AI snake control into this, other changes have been made. A guard was added to the check of if the snake has eaten food to see if the snake has completed the game. Game completion is defined as there being no available spaces on the board to place food as the snake is taking up all available grid positions. There are also changes to the clean-up of the Python turtle to prevent Python kernel crash.

3. `code/snake.py` - This file contains the `snake` class. While sensing functions have been added to this class, the only change to initially provided functions has been to use `self.XSIZE` and `self.YSIZE` instead of the global `XSIZE`, `YSIZE`.

1.2 Chosen Evolutionary Method

A Genetic Algorithm (GA) was used to evolve weights in a Neural Network (NN). Input nodes to this neural network were the snake sensing functions, with four output nodes representing snake direction; left, right, up and down. Heaton [1] discusses that most problems to not require more than one hidden layer therefore a single hidden layer was used for the neural network with the number of hidden layer nodes corresponding to the sum of the input and output layer nodes. Using a single hidden layer has a benefit of reducing overfitting compared to a multiple hidden layer network, along with decreasing time taken to train the network. A softmax function is used at the output layer to obtain the direction the snake should take, where left=0, right=1, up=2 and down=3.

Two selection methods from generation to generation were created and will be evaluated in this report, one method being a standard tournament selection while the second method utilizes an n+n method

1.3 Development of Different Snakes

Four differing snakes were developed for the snake game and will be referred to in this report as:

1. `local-standard`

2. `space_aware-standard`

3. `local-n+n`

4. `space_aware-n+n`

Two directions of investigation were performed on the snake game, where each name is comprised of these two components separated by a hyphen. The first direction investigated the effect of differing collision sensing functions (`local` against `space-aware`) while the second investigated changing the generation selection method (`standard` against `n+n`).

1.4 Sensing Functions

Sensing functions were created for the snake to use to learn about its environment, these can be classified into two groups; food sensing functions and collision sensing functions. All sensing assumes grid format of `[y,x]` and directions are relative to the grid, not the current direction of the snake.

1.4.1 Food Sensing

Food sensing was constant throughout all developed snakes. Multiple methods of food sensing were explored during development including passing the exact position of the food, calculating the manhattan distance between food and snake, and sensing the quadrant of the grid the food was in relative to the head of the snake. The chosen method instead simply checks if food is in the direction specified, for left and right the x axis of the snake is checked while up and down checks the right axis. The sensing function returns a boolean value; True if food is present and False if not.

If the Snake is directly in line with food in that direction, both directions on the axis will return False. For example, a snake head at position [4,4] and food at [6,6] would return True from `downFood` and `rightFood`, and False from `upFood` and `leftFood`. A snake head at position [4,4] and food at [6,4] would return True from `downFood` and False from all other sense directions.

1.4.2 Collision Sensing

Two methods of collision sensing were developed, the first method of collision sensing is referred to as `local` and was used by both `local-standard` and `local-n+n`. Each direction has two collision sensing functions that both check based on a given position; one used to sense if the position is within a wall, and the other used to sense if the position is within the snake body. Both of these sensing functions return True if a collision is detected, False otherwise. These positions are given for each direction by calculating the position of the snake head if the snake moved in the given direction. For example, if the snake head is at position [4,4], the position given to `senseWall` and `senseTail` would be [4,3] when `leftWall` and `leftTail` are called.

These wall and tail functions could be combined into a single function to detect if a collision has occurred, however this was not implemented as providing both collision detection components separately allows a snake to learn separate behaviors depending on collision type, for example chasing it's tail or circling the grid perimeter.

The second method of collision sensing is referred to as `space_aware` and was used by both `space_aware-standard` and `space_aware-n+n`. This method was created to attempt to prevent the snake being trapped with no space to go. To create this sensing function, a recursive search was performed for a given position that counted all available spaces from that position. This search is performed at each of the four possible positions that the snake can move to (up, down, left and right).

2 Results

2.1 Algorithm Testing

Multiple different algorithms have been investigated for this report; standard evolution against n+n evolution, and local sensing against space aware sensing. These two factors combine to provide four statistical comparisons between algorithms:

1. `local-standard` vs `local-n+n`
2. `space_aware-standard` vs `space_aware-n+n`
3. `local-standard` vs `space_aware-standard`
4. `local-n+n` vs `space_aware-n+n`

Each algorithm was ran 10 times with a population of 200 for 500 generations, an average of these runs was then computed to provide a fair representation of the performance of each algorithm.

2.1.1 standard evolution compared to n+n evolution

As seen in figure 1 comparing standard evolution to n+n for local collision sensing, while `local-standard` was the prevailing algorithm after 500 generations, `local-n+n` is consistently performs better from generation 0 until a crossover point at approximately 50 score. Performing a Mann-Whitney-U test for the two average datasets gives

$$U = 107778.5, p = 0.00016$$

Figure 2 compares standard to n+n evolution for space aware collision sensing. As shown in the graph, n+n performed significantly better than standard evolution, though standard deviation was large. Standard evolution struggled to make any progress Throughout generations, with a consistently average score throughout generations of 0.

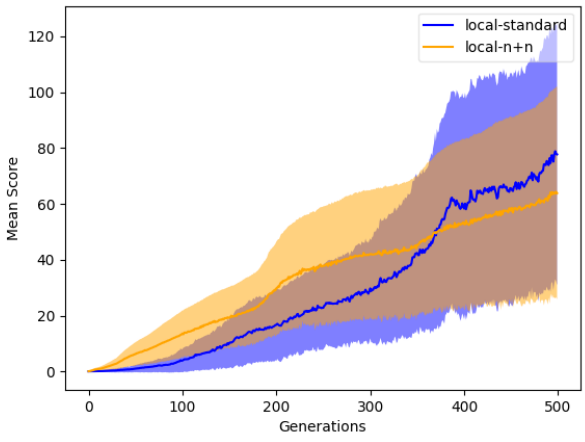

Figure 1: `local-standard` compared to `local-n+n` over 500 generations

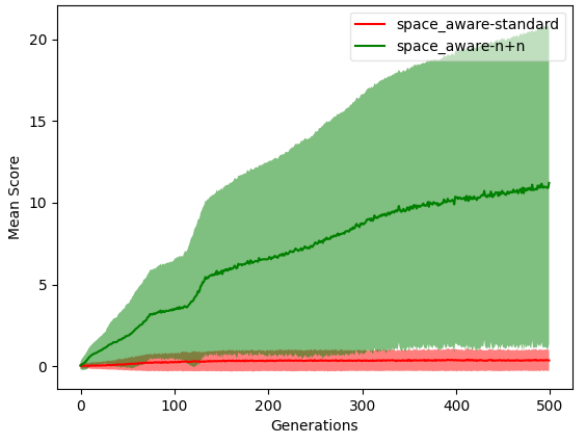

Figure 2: `space_aware-standard` compared to `space_aware-n+n` over 500 generations

3 Conclusions

3.1

References

[1] J. Heaton, *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.