Snake A.I. Documentation

What is Snake A.I. ?

The Snake A.I. is an agent that learns to play the game of snake. This agent implements the idea of the Genetic Algorithm (GA), adapting its environment and attempting to maximize its own score. The environment in question is a simple grid with adjustable size (50x50 by default), with the only variable factor being the apple that spawns randomly at the start of the game and respawn once it is eaten. The score is determined by the fitness of the snake, which, in turn, is determined by how well it performs in regards to eating apples.

What do I need to know before starting ?

Basic understanding of neural networks is required to make sense of the scripts and features of the asset. I highly recommend watching the basic neural network series by “3Blue1Brown” on youtube to get a general understanding behind the principles of neural networks. Additionally, there are multiple blog posts online describing the idea behind the genetic algorithm and its implementations.

Everything within the asset is written in c# from scratch. Normally, when making a neural network you would probably use libraries like Tensorflow or PyTorch, while using Python as your primary language. This would be because Python is very widely used for A.I. research. The libraries that I just mentioned have a lot of built in functions that will make life much easier for you, while letting you focus on your research. These are mostly focused on array manipulation, liberating you from the need to write your own methods from scratch. For example, most of the methods you will see within “CoreFunctions.cs” are things that can be achieved by calling a simple line of code when using these libraries. Thus, you can ignore the principles of most functions within CoreFunctions.cs if you feel like you want to focus on the general features rather than the inner-workings of a neural network (although I wouldn’t really recommend that).

All in all, you will need a basic understanding of neural networks, a basic level of knowledge in c# or any similar languages, and at least a vague idea of genetic algorithms.

How does the snake navigate the map, and how is the fitness calculated ?

The fitness value is calculated in a very straightforward manner. However, before we cover the way it is calculated, we must first understand what the snake “sees”.

The Snake only has six “senses”, or in more appropriate terms, inputs. It knows whether there is a wall or tail to its front, left, and right, as well as whether there is a fruit in any of these directions. For each one of these factors, the snakes receives inputs of 1s and 0s (1 means yes, 0 means no). The snake then propagates these inputs through the network, using the network’s weights to get an array of outputs. The index of the output with the

highest value is then used as the direction in which the snake has decided to move. Now that we understand how and “what” the snake sees, let’s take a look at what the fitness represents.

Note: You can browse through the code if you wish to understand the inner workings of this system.

Note2: The way that the snake observes the map is crucial for how well it performs. You are free to try and experiment with different approaches. For example, you could try and feed the snake the distance from the apple instead of whether there is an apple in a specific direction.

Note3: The front, left, and right directions are relative to the snake’s current directions. For example, if the snake is facing down this means that the “front” of the snake is the downwards direction, its left is “right”, and its right is “left”. In the code that governs the snake’s decisions local directions are converted into global directions.

The fitness is essentially the performance of the snake. The better the fitness, the better the snake is in comparison to other snakes. To decide how to calculate the fitness, we must first decide what our goal is. In this case, our goal is to make the snake eat as many apples as possible before eventually dying. Thus, for each apple eaten we will be giving a reward. Secondly, we must also ensure that our snake is not aimlessly wandering around, and that it is instead focused on chasing after the apples. For example, a snake could get 100 apples after 50 minutes of randomly wandering around the map (and not dying in the process), or it could get 100 apples after 10 minutes by actively chasing after the apples. The difference is essentially that of a blind-folded person versus a seeing person, both looking for needles in haystacks. It is not even worth mentioning who will finish faster.

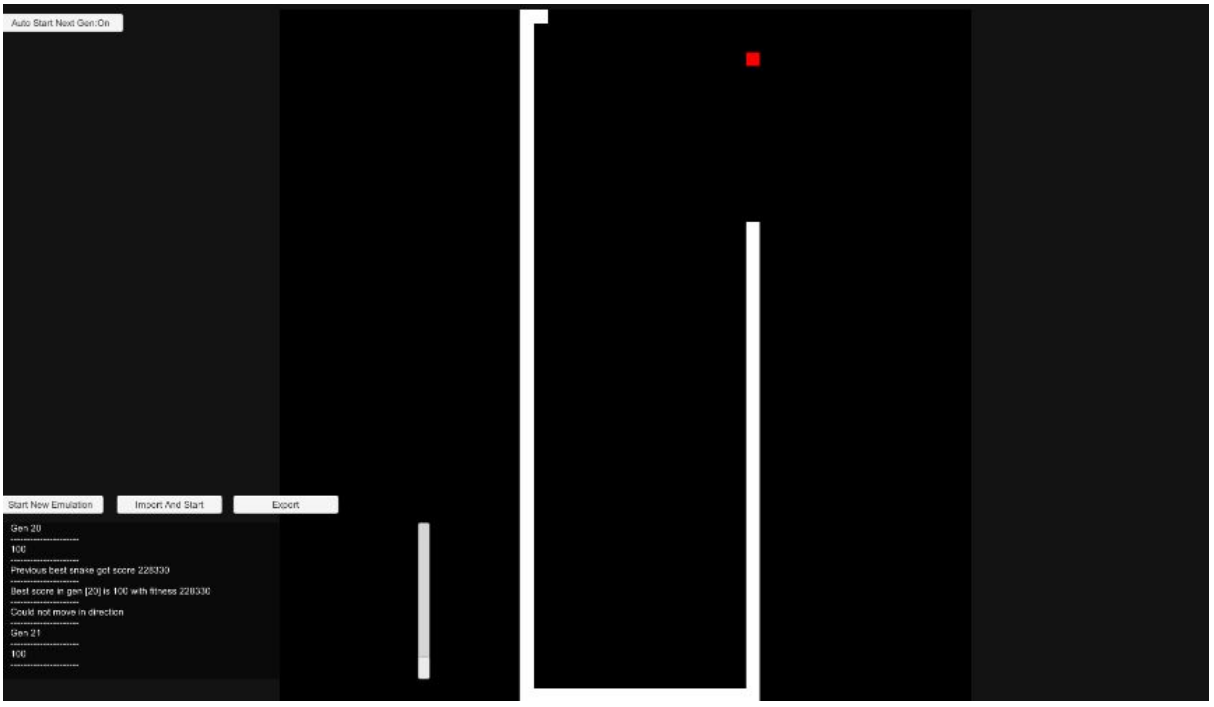
Based on the above, we need to incentivise the snake to chase after the apple. This can be done by giving it small rewards after each step taken closer to an apple. On the contrary, we will be subtracting an equal amount if the snake is actually distancing itself from the apple. This is done to prevent snakes from going back and forth and obtaining high scores this way. Additionally, we must ensure that the snake is not stuck in a loop without doing anything. Thus, we also implement a safeguard in the face of a limited amount of steps that can be taken without eating an apple. For example, if a snake moves 250 times without eating one, it will die. On the other hand, eating an apple will award it with a reward, and will reset the amount of available steps.

Lastly, we also penalize the snake for bumping into things. This step can be arguably avoided, but I personally reduce half of the amount of what the snake gets from an apple, when it dies. This helps eliminate the snakes that die after eating a single apple in the early stages of the emulation.

In-Game UI and Inspector

The inspector will be your main way of configuring the agent. Most settings can be found by accessing the “SnakeManager” script that is attached to the “GameManager” object. You

can find a brief explanation of each of the configuration elements by viewing the script itself. It is highly recommended that you have at least a basic understanding of what each of those settings represent before you start tweaking them.



The In-Game UI on the other hand only allows for basic actions. These are as follows:

Emulation mode (auto/manual): The emulation mode is essentially the way in which the algorithm operates. The mode can be toggled by using the “Auto Start Next Gen” button in the top left corner. If the button displays an “On” states, this means that after each generation has ended, a new one will automatically start. This option is useful when you don’t want to manually start each new generation and simply wish to leave the agent to learn on its own until all the generations have finished training. The second mode, namely “Off”, allows you to manually start each generation after the previous one has ended. This mode could be useful in scenarios where you want to “pause” the process and return to it later, or in case you wish to view the full playback of each snake’s actions.

Note: The playback that you see on the screen represents the actions that the best snake in the last generation took. It will automatically reset once the currently running generation has finished their emulations, and the playback of the best snake of the latter generation will start instead.

Start new game: This option will reset any progress made so far and start a new set of emulations.

Import / Export: At any given time you can freely import and export the current weights of all neural networks. You can configure the export/import paths and file names via the Snake Manager script.

Clicking on export will generate one file for each snake in the population. Such a file will contain all the weights of a snake's neural network. The name of these files will be in the form of "yourChosenName" + prefix1 + prefix2. Prefix 1 will only be generated in the case there the "overwrite" option is disabled and a file with a similar name already exists. For example, if you chose the name "Snake" for your export, the engine will try and generate a snake called "Snake_0_prefix2" (where prefix 2 is just the index of the generated snake, because as we mentioned we will be generating a file for each snake), however, if the "overwrite" option is not enabled and such a name already exists, the engine will increase the value of the prefix by 1 and will try again.

Clicking on import, will cause the engine to attempt importing all the files with a matching name. This action will also restart the game with the imported weights for each snake in the population.

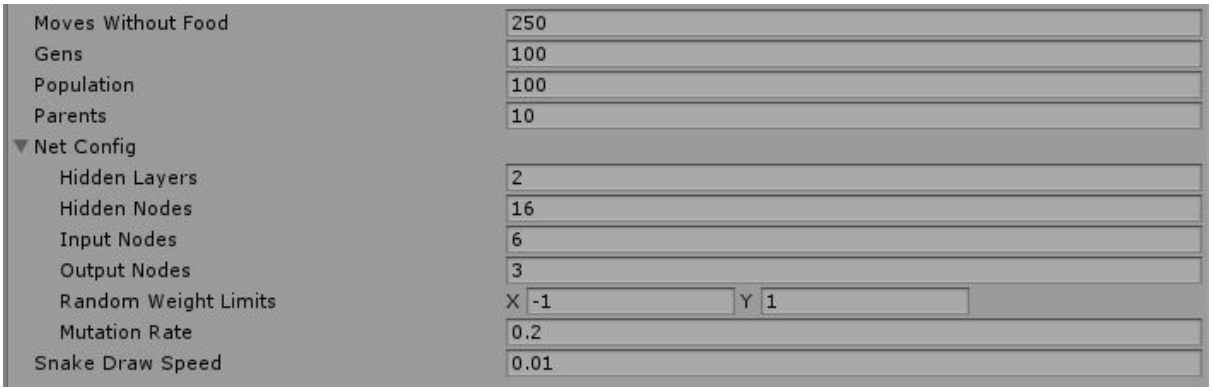
Note: Please make sure that if you exported the weights of a 100 snake population, you only import these weights when you have a population of exactly 100 snakes. This is required because the engine will automatically assign a set of weights to each snake in the population, and if the number of files is not matching the number of snakes, this will cause issues.

Log: The log is the equivalent of the default unity's debug log. Whenever a value is debugged, it is done via a special "debug" function, instead of the usual Debug.Log method. This allows results to appear both in the original console and on-screen.

How do I configure the neural network ?

Note: As stated before, it is highly recommended that you have a basic understanding of NNs before you proceed to the configuration of this network.

The network can be configured via the SnakeManager script, attached to the GameManager gameObject.



Moves without food: The amount of moves that a snake can make without eating. After eating this amount will reset.

Gens: The number of generations for which the training will continue.

Population: The number of snakes in each generation. A number between 500-2000 is recommended for decent PCs, while a number around 100 is recommended for weaker machines.

Parents: This is the number of the top snakes that will be chosen for breeding. For example, if the number of parents is 200 for a population of 2000, only the top 200 snakes will be chosen for breeding.

Net Config: The configuration of the network itself.

Snake Draw Speed: The lower this number is, the faster will the snake move.

The **input nodes** value is the amount of input nodes that each neural network will have. For example, if the value is 4, this means that the snake can only have 4 input nodes. You can view the SnakeManager and CoreFunction scripts if you wish to take a closer look at how the input nodes are assigned.

Output nodes, likewise, is the number of nodes that the network will return after propagation. Each of these nodes will have a value between 0 and 1, and the node with the highest value is used to determine the direction in which the snake will move.

Random weights limits and the maximum and minimum values that a weight can be randomly assigned. All weights are randomized in the beginning.

The **mutation rate** is the chance of a snake's genes (weights) to be altered. For example, if the mutation rate is 0.3, then there is a 30% chance on any of its weights to be randomized.

Hidden layers & nodes: Assuming that you understand their function, the hidden layers and the number of nodes per layers can be decided after running trials with various values. Although there are algorithms that automatically adjust these, usually they require a lot of trial and error to be discovered.

Setting up the project

First of all, please take a look at the source code of the project, by inspecting the relative scripts within the script folder. Every script is well commented and should give you plenty of insights in regards to how the system works.

To set up the project, navigate to the "GameManager" object and take a look at the "SnakeManager" script. As was previously mentioned, this allows you to set up your agents (snakes) and network (brains). Please take a look at the source code of the script, or scroll up to the previous section to familiarize yourself with all the available configuration.

After the “SnakeManager” script has been set up, the last thing you need to do is set up the “ExternalFunctions” script. This script contains functions that are not directly related to any “A.I.” operations.

Snake A.I. allows to export and import neural network (weight) data as xml files. Such an approach allows to view and manually edit these files. Furthermore, each snake in the population will have a seperate file. Thus, please be careful when exporting data as your chosen directory will be filled with files.

Note 1: When importing data from a directory it is very important that the directory contains the exact number of files as the number of snakes in your population. This will not be an issue if you export and then import your data without changing the number of snakes of the population. However, if you export your data, change the number of snakes in the population (by increasing it) and then try to import the old data, the system will not be able to properly import the data. All in all, please make sure that you do not change the number of snakes in the population after you have exported the data, otherwise importing it will cause errors.

Note 2: Just as a reminder, you can change the number of snakes in a population by editing the **Population** variable under the “SnakeManager” script.

The script is configured as follows:

Export Path: Here you should specify a path to a folder where you wish your network data to be exported to. For example, if you specify C:\Users**YOUR USER NAME**\Desktop\New Folder (given that you are using windows), a file will be generated for each snake when exporting the data inside of “New Folder”. Please see “**Note 1**” to ensure that you are exporting your data properly.

Import Path: The folder from which the data will be imported. Usually, the import path and the export path will be identical, however you may want to change the import path if you have your reasons. Please see “**Note 1**” to ensure that you are importing your data properly.

Export File Name: The group name of the exported files. Since more than one file will be created at a time, each of these file-names will have a prefix in the form of this string. Each of the latter files will also have a postfix in the form of its index value inside the folder, and a second postfix in the form of its index value among the snakes. For example, if your export file name is “Snake” and the folder is empty, the first file will be called “Snake_0_0”, the second one “Snake_0_1” and so on. However, if you already have a bunch of files called “Snake_0_0”, “Snake_0_1”, etc., if your “Overwrite” option is disbaled, your new snake filles will have their first postfix incremented. Thus, your first file will be called “Snake_1_0”, your send one “Snake_1_1” and so on.

Import File Name: The name of the files to be imported. This is essentially the export file name plus the first postfix. For instance, if you wish to import files “Snake_1_0”, “Snake_1_1”, …, “Snake_1_N” (where N is the last snake’s index), your import file name

would be “Snake_1”. The indexes at the end are added automatically, thus you only need to specify the group name.

Overwrite: This option lets the engine know whether you want to overwrite the files with the same group name in your specified export direction when exporting data. For example, if you already have files with the group name “Snake_71” in the export folder (given that this is the export file name that you’ve chosen), if this option is enabled and you click “export” all of the files with the name “Snake_71” will be overwritten by your new files. If this option is disabled, a new batch of files will be created with an incremented postfix (In this case it will be “Snake_72”), leaving the previous batch intact.

Is this agent considered good? Can it be improved?

This agent is very simple and not optimized at all. The purpose of this asset is to assist in your understanding of A.I. and Neural Networks, not to break any records. My hard assumption would be that you will have modified the agent quite a bit by the time you have finished playing with it. It can be improved a numerous ways, such as changing the rewards system, changing the rules of the game itself, feeding different inputs to the snakes, modifying the number of hidden layers and nodes etc. Overall, this agent is designed for learning purposes and will not yield incredible results in its default state.

It is recommended to run an A.I. in unity?

No. Currently, the most popular language/platform for developing A.I. is Python. This is due to the large amount of libraries and support that is available for Python developers in the area. Additionally, most environments designed for testing A.I. are also focused on Python. However, this does not mean that Python is plainly superior and that you should abandon whatever you are doing and start using Python. Although the resources available for other platforms are fewer, other platforms and languages may suit your needs better. As an A.I. developer, your main focus should be on the efficiency and speed of your training, not the tools you use. Certainly, decent tools will make things much easier, but in the end, it all comes down to what you are creating.

Note: It is also worth mentioning that if you plan to run complicated networks, using your personal computer may not be the best idea. Ideally, you would need to utilize technologies like CUDA, getting a dedicated GPU and enabling it to do the processing instead of your CPU. These are many reasons behind this (and you are free to look them up online), but the gist of it is that your CPU may not be as efficient. However, it is not guaranteed that your GPU will be better for training than your CPU. I would recommend making a quick google search to find out whether utilizing your GPU will indeed yield better results. You can also use cloud services for this task, paying a fixed fee for processing power. However, all of this should only concern you if you have big plans for your research. If you are simply learning A.I., or developing a simple A.I. for your projects, you can very well ignore all of this info.

I am making a game and I want to implement an A.I. in it, how do I approach this issue?

Ideally, you need to simply pre-train an A.I. to achieve your goal. For example, if you wanted to make a game of snake where the user observes the game play on its own, you would simply need to train a network beforehand, only using your trained neural network for your game.

In a similar manner, you could pre-train an agent to play chess, and only include the resultant network in your final project. I would also recommend making a theoretical network (not an actual term) in order to achieve this. Just like we do in this project, you would need to simply train your network using numerical values, without actually making the network feed the results into an actual game session during training. Imagine having to wait while 2000 snakes finish their game sessions for just one generation to end, this would take an incomparably long time to achieve in contrast with simply “running the numbers” for 2000 snakes and then displaying the results of the best snake amongst them.

Overall, I would recommend pre-training our networks using numerical values only, without any visual representation of the training itself. I would also recommend using a different combination of tools for training your network (such as Python in combination with tensorflow), while enabling GPU processing on your system for faster data processing (if your GPU is suitable for training and your network has tons of connections). If you can afford a cloud service and leave your network to run there without taxing your pc, even better.

Feedback and questions

If you have encountered any bugs or issues, please feel free to contact me:

Unity Connect: Link (Preferable way)
Email: cinationproject@gmail.com (Faster response)