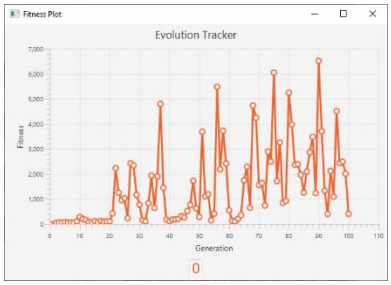
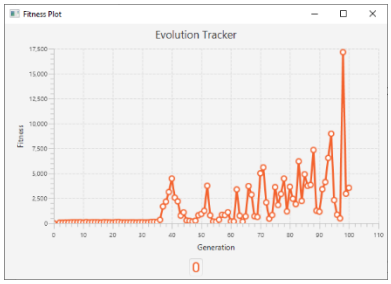
Evolution Write-up: Report

I. Optimal Configuration

Sample trials:



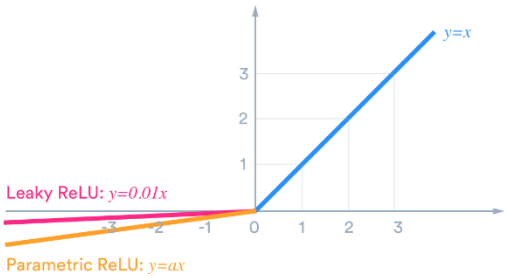


All following optimization was done under the environment that

PIPE_DISTANCE = 250px // **important**, I didn't test other distances
PIPE_WIDTH = 50px
GRAVITY = 1100px
REBOUND_VELOCITY = -400px
PIPE_GAP = 130px
NUMBER_OF_HIDDEN_LAYER = 5
NUMBER_OF_INPUT_NODES = 4
DECISION_TO_JUMP < 0.1
BIRD_MAX_LIFE = 30000 //kill Switch
NUMBER_OF_DIRECT_INHERIT = 25
LARGE_MUTATION_RANGE: +1 to -1
SMALL_MUTATION_RANGE: +0.125 to -0.125
DIRECT_INHERIT_BOUNDARY = 3000 (score)
SMALL_MUTATION_BOUNDARY = 2000 (score)

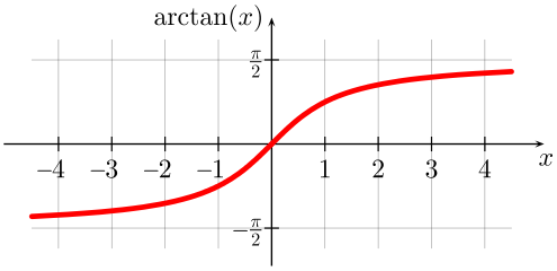
Activation function used

Input nodes to hidden layer:
f(x) = max(0.1*x, x)



Reasoning: Though I don't really understand the reason behind this activation function, but from my online research I found out that ReLU is one of the most popular machine learning activation functions, and Leaky ReLU can effectively solve the "zero-slope" parts in ReLu, which causes loss of data. Usually, Leaky ReLU learns faster than plain ReLU.

hidden layer to final output:
f(x) = abs(arctan(x/500) / (pi/2))



Reasoning: Arctan is a sigmoid function that has the range of positive half pi to negative half pi, also, it is less easy to converge to the asymptotes compared with other sigmoid functions. I divided the original output x by 500 because I would like to suppress their range—when x is larger than pi, f(x) will tend to be very close to the function's horizontal asymptote, making the output either pi/2 or negative pi/2. The pi/2 in the end is just an experimentation that squeezes the range of the function to 1 to negative 1. It worked well, so I kept it, though I believe it doesn't have real contribution to the program. I finally tried to add a absolute value in front of this equation, and changed the decision value to 0.1. It also worked better than before, so I kept it.

II. User Guide

Please run the game, click the AI Game, select 250px distance. This is the game that I tested on.

III. Summary of Design

The Part I of the game is largely similar to the DoodleJump we coded for CS15. The key design was that the bird will only move vertically, and the pipe will scroll to the left instead to make the visual effect of birds moving. Once the bird hits one of the pipe, it will die and the game will be restarted. The keyhandler for the human game included two functions, one is the space bar, which makes the bird jump, and another one is a pause button. The main data structure that I used in part 1 is ArrayList, which dynamically controls the number of pipes in the game.

After coding the initial single-player game, I have Evolution, Birds, and Pipe classes ready for me to code the second part. I went to the TA hour and Zack the TA recommended me to have a Population class, which controls the life and death of the birds. Of course, I will also need a NeuralNetwork class for each bird to think (my spellings in the code are all wrong... sorry).

I created the Population class and instantiated a 1-dimensional array (`_birdArray`) that contains 50 bird instances. I used a for loop to create 50 birds. I also allocated two multi-dimensional arrays, `_birdSyn0[][][]` and `_birdSyn1[][]` to be their permanent weight keeping array, so that they will reach a decision based on these two weights arrays. In this game, the bird and their weights are independent from each other. However, their index in the Arrays are always the same, so they match each other. That is why I could use the same index of the best bird in selecting the best weight.

My algorithm for selecting the best birds is rather straightforward: whenever a bird hit the pipe and dies, I will add the bird to an ArrayList called `_currentGenBirdsArrayList.` This ArrayList will keep adding the dead birds, and intuitively, the last bird added to the ArrayList is the best bird that survived the longest (fittest). Meanwhile, I will set the bird in the `_birdArray` to null. When I am updating the `currentGenBirdsArrayList`, I am also updating bird's best weight to two arrays, `_tempSmartSyn0`, and `_tempSmartSyn1`. These two arrays are 1 dimension less than the `_birdSyn0` and `_birdSyn1`, so that they only hold the last killed bird's weights. When all birds die, they will hold the best bird's weight combination.

In Evolution class, I detect the time when all birds are dead by counting the empty array blocks in the `_birdArray`. When that happens, I will execute the regeneration method, which will regenerate the bird population by calling the overloaded `Population(int overloadingIdentifier)` constructor (designed for generating birds that is not the 1st generation). However, before generating the bird instances, I will first calculate the new weight array for them, so that they can jump accordingly. This is achieved by the `constrNextGenWeight()` method.

The `constrNextGenWeight` method is located in the Population class. It first instantiate two Weight ArrayLists as temporary weight value holders for the next generation. If the best score from the last generation is good, I will allow 25 birds to directly inherit. By that, I mean directly adding the best weight 25 times to the `_nextGenSyn0ArrayList/_nextGenSyn1ArrayList.` Otherwise, if the best score from last generation is not ideal, I will only add one to the ArrayLists that temporarily hold birds' weight information.

After that, I used a while loop to make sure that, at the end of the method, every time exactly 50 weights are being generated, not more and not less. In the while loop, I have a `_evoWeightSyn0/Syn1`, which holds the mutated weights for a single bird. Once they are mutated, they are being added into the `_nextGenSyn0/Syn1ArrayList`, which holds weight information for 50 birds. An optimization is being applied here that, when the result from last generation was ideal, the

birds will mutate in the range of +0.125 to -0.125. Otherwise, they are going to receive a larger mutation with the range of +0.5 to -0.5. This helps to preserve the good genes in birds.

After I filled the `_nextGenSyn0ArrayList` with weight information, now I transfer/copy those information into the `_birdSyn0/Syn1`. Therefore, the `_birdSyn0/Syn1` is being updated every time before the next generation birds are generated.

The above methods, together, helped the birds to evolve and learn overtime. Finally, I have to change the single bird instance `_bird` to `_birdArray` in Evolution class and use for loops to point to each bird to make them jump/die.

I still have a few more comments to include here in the summary of design.

1. My App class instantiate the top-level object of StartScreen, rather than PaneOrganizer. In PaneOrganizer, I instantiate different Evolution games based on the user's input.
2. I have used constructor overloading in PaneOrganizer, Evolution, Pipes and Population classes.
3. I frequently used static variables in order to ease the process of transferring the variables to other classes.
4. I have 4 input nodes, which includes the y-position of the bird, the horizontal distance to the next pipe, the vertical distance to the next upper pipe, and the distance to the next lower pipe.
5. Also, for GUI improvements, I chose to allow users to switch between human/AI games, and allow them to select the pipe distance of their choice.

IV. Optimizations that were not properly documented

Though I will include a detailed optimization process in the following section, in this section I would like to write about the optimization I have tried before I finished the whole program. I didn't document them because I wasn't able to get my birds learn properly, so I didn't know I was finished and can start the next section of the project. Here is a few things I have tried that was not documented:

1. Activation function

I used the logistic function at the beginning, but soon I realized that when x is large, the outputs became either 0 or 1. At that time, I thought about changing to another activation function. After some googling, I found out that the arctan(x) has a larger range and can thus produce more different outputs. Also, arctan(x) is centered at 0, rather than 0.5. Therefore, I tried to use arctan(x) for both Syn0 and Syn1, and it started learning. Then, I found out that currently, the ReLU function is widely employed in the industry. Though I don't understand why the function is better, I applied that to my Syn0 calculation, and it worked better. Therefore, I kept using ReLU for Syn0, and arctan for Syn1.

2. The decision value

I believe this is a very very important parameter in the program. I experimented a few other decision values with small deviations to 0.1, for example, 0.15, and the birds learns significantly slower.

3. Fifth input node

I tried to add the fifth input node: the bird's distance to the midpoint of the gap. However, I realized that I would have to change the decision value many times as the output becomes different, therefore, I deleted the input nodes in the future optimization process.

Evolution Write-up: Optimization

I. Environment Setting:

All the graphs represent the average fitness of the group
All following optimization was done under the environment that

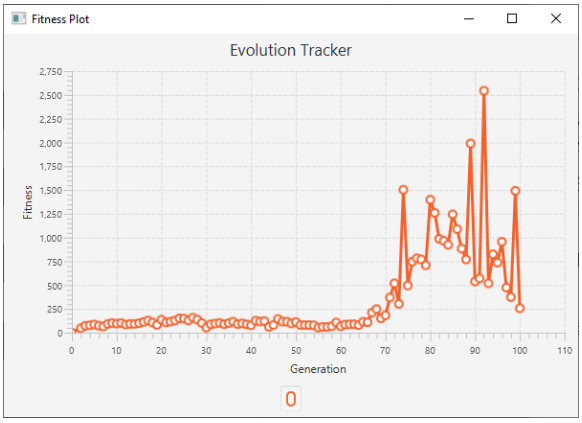
PIPE_DISTANCE = 250px
PIPE_WIDTH = 50px
GRAVITY = 1100px
REBOUND_VELOCITY = -400px
PIPE_GAP = 130px

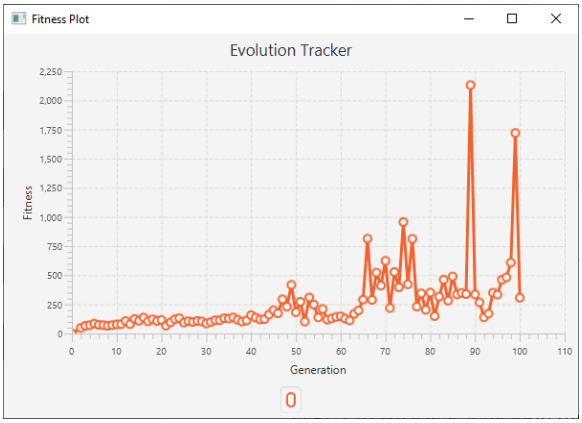
II. Trial 1: Unoptimized Program

Key settings:

INPUT_NODES: 4
HIDDEN_LAYER: 5
INPUT to HIDDEN activation function: Math.max(0.01 * input, input)
HIDDEN to DECISION activation function: Math.atan(input)
NUMBER OF BIRDS: 50

```
for (int i = 0; i < Constants.NUMBER_HIDDEN_LAYER; i++) {
        for (int j = 0; j < Constants.NUMBER_INPUT_NODES; j++) {
                _evoWeightSyn0[i][j] = _tempSmartSyn0[i][j] + (Math.random() * 0.25 - 0.125);
        }
}
```





Evaluation:

The curve did show that the birds do learn overtime, but the convergence is very slow. Graphs have shown that the birds only start learning after 60 generation, and the first trial have shown decline after 90 generations. I believe it is just a one-time phenomenon, but I could try to improve it by decreasing mutation for good birds.

II. Trial 2: Have different mutation scale depending on Best Birds

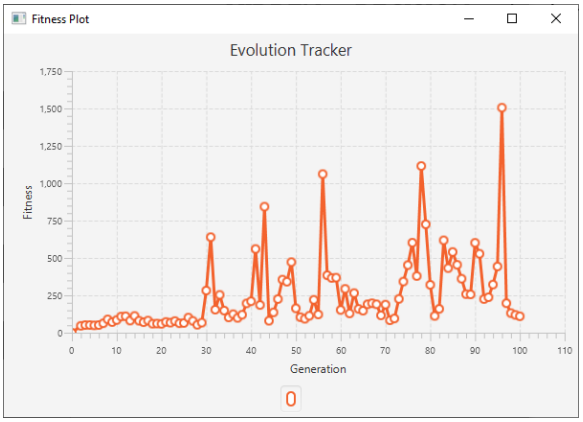
Key settings:

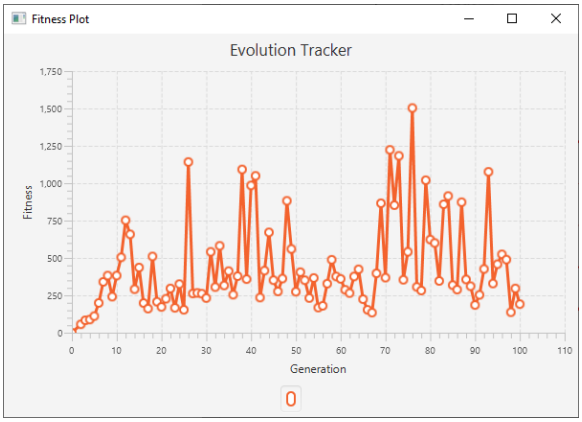
INPUT_NODES: 4
HIDDEN_LAYER: 5
INPUT to HIDDEN activation function: Math.max(0.01 * input, input)
HIDDEN to DECISION activation function: Math.atan(input)
NUMBER OF BIRDS: 50
DECISION_VALUE < 0.1

Key changes:

If the best bird's score is smaller than 1000, the mutation will be in range of 1 to -1; else, the mutation for Syn0 and Syn1 will be in range of 0.125 to -0.125

```
for (int i = 0; i < Constants.NUMBER_HIDDEN_LAYER; i++) {
        for (int j = 0; j < Constants.NUMBER_INPUT_NODES; j++) {
                if (_bestScore < 1000) {
                        _evoWeightSyn0[i][j] = _tempSmartSyn0[i][j] + (Math.random() * 2 - 1);
                } else {
                        _evoWeightSyn0[i][j] = _tempSmartSyn0[i][j] + (Math.random() * 0.25 - 0.125);
                }
        }
}
```





Evaluation:

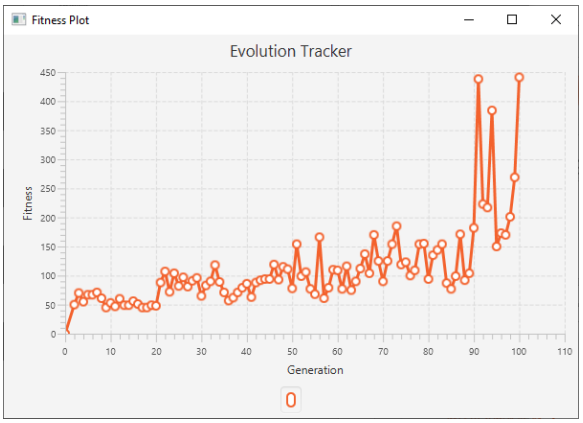
After the change, we could see significant reduction in the converging time. In the first graph, the birds started learning after 30 generation, and in second trial birds almost immediately started learning. However, it seems that their fitness decreased a little. The change was effective, but I am still not satisfied with the relatively slow learning curve at the beginning. I would like to change that by regenerating the Syn0 and Syn1 when the birds are not smart enough, rather than inheriting from their non-smart ancestors. Also, since this attempt made the curve converge quicker, I would keep this change for the future optimization process.

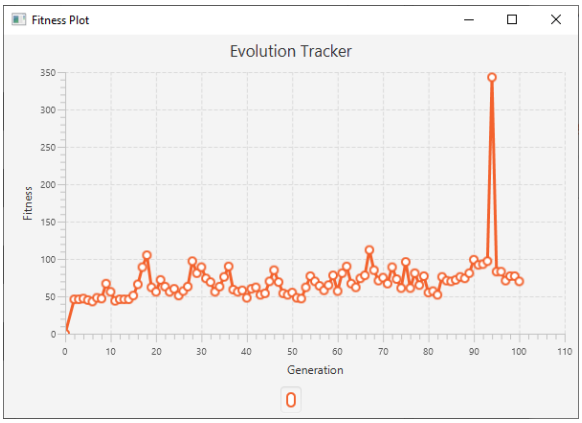
III. Trial 3: If all the birds are relatively non-smart, regenerate rather than inherit

Key changes:

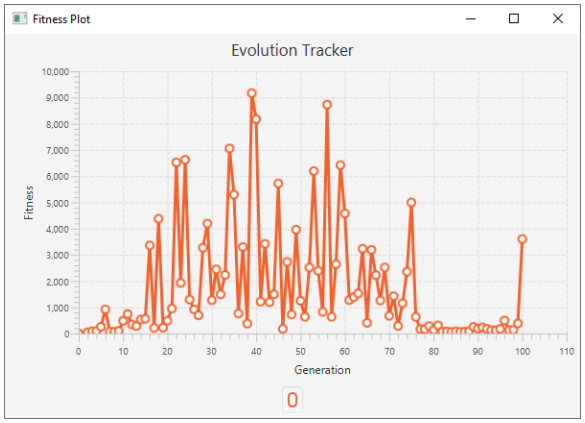
If the best bird's score is smaller than 100, which means all the birds didn't pass through the pipes, I will regenerate the Syn0 and Syn1 rather than choosing the best bird to inherit. I hope this change could allow the algorithm to converge quicker.

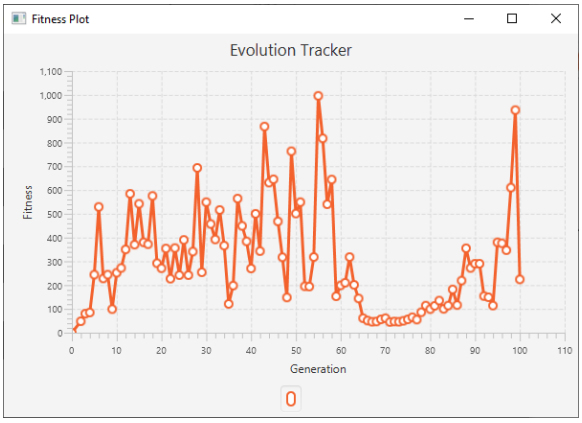
Evaluation:





The graph above represents the bad scenario when I applied this change to the code. Although the graph looks like none of the birds are very incapable, but the Y axis scale was significantly reduced. This means generally the birds are more incapable than before. The potential explanation for this is that as the number of inherit/mutation decreased, birds weren't given enough chance to evolve and learn.





The graphs above represents a better scenario, where the change effectively made the convergence shorter. However, when one of the mutations goes wrong / unlucky case happened, the change is likely to cause a number of generation maintain that mode as they are not evolving. This is not the worst scenario, but it is still not ideal.

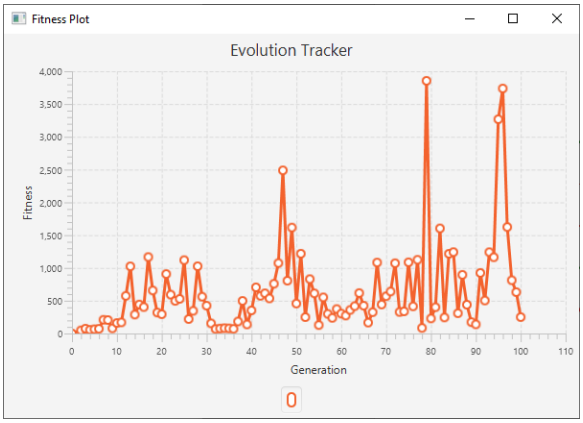
In conclusion, I believe this change is not providing stable enhancement to the performance of the program. Thus, I decided that I am not going to keep this change in the future optimization process.

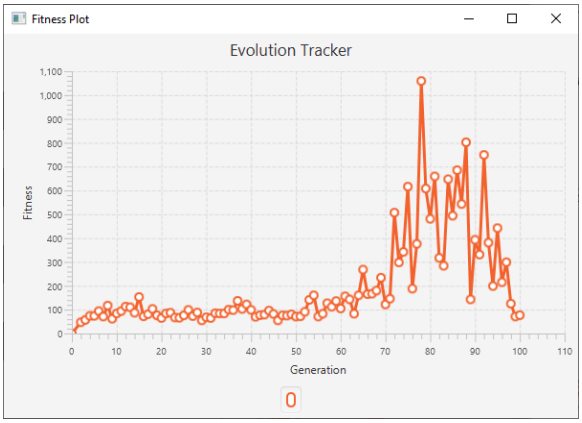
IV. Trial 4: Adding the best birds 5 times directly into the next generation

Key changes:

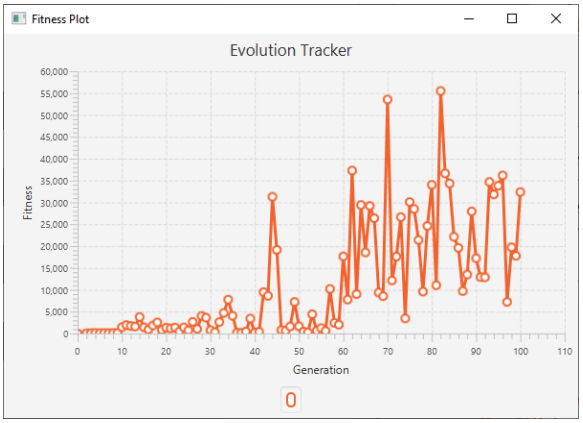
In previous graphs, I saw a lot of zig zags. I believe this is because sometimes the best birds get unlucky (eg. Started at a very bad position), and wasn't able to show its best performance. Therefore, I am wondering if adding the best birds 5 times directly into the next generation could solve this problem.

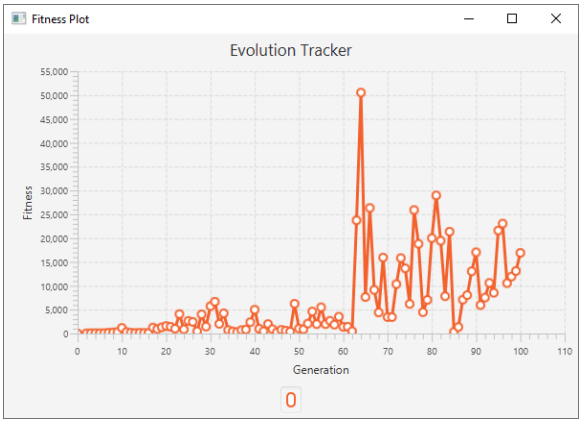
```
for (int i = 0; i < 5; i++) {
        _nextGenSyn0ArrayList.add(_tempSmartSyn0);
        _nextGenSyn1ArrayList.add(_tempSmartSyn1);
        }
while (_nextGenSyn0ArrayList.size() < Constants.POPULATION_SIZE) {…}
```





This change is doing fairly well, as we can see that the different between generations are (generally) smaller. However, from the graph on the right side, we can clearly see that this method is preventing birds to evolve when the birds are not-so-smart at the beginning. That is why I modified the code a little bit, by adding a constraint on when it is going to add 5 times instead of 1.



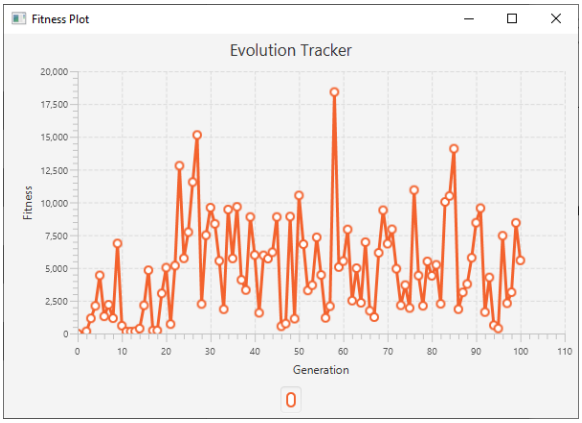


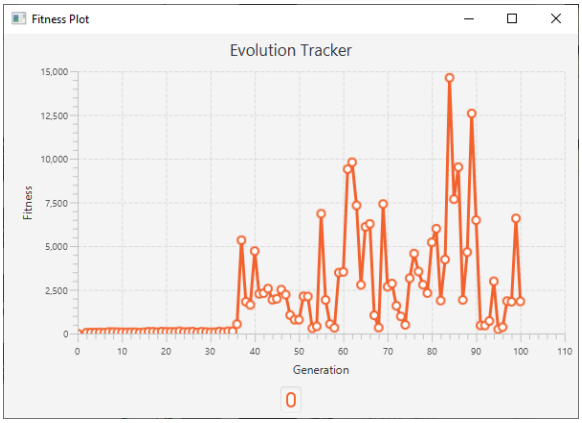
I tried the code; it improved the performance for a bit. Then, I thought about why don't I make it 25, thus half of the population will definitely have the good breed, and we can significantly improve their performance. The results are very good, and they are exhibited above. The y-axis value has skyrocketed. Even though it looks like the birds at the beginning are not learning, I believe it is affected by the y-axis scale. They can all be considered well-trained birds now.

V. Trial 5: Changing the number of Hidden Layer

Key changes:

I decided to change the number of Hidden Layer to see whether that would improve the performance of program. I simply changed the constant NUMBER_HIDDEN_LAYER = 5 to 20 to see what would happen.





Evaluation:

Though I expected the program to improve, I did not see apparent changes when I changed the number of hidden layers to 20. Also, this significantly increased the computations required, as I could see program frozen from time to time when the computer has to deal with many alive birds under the 1000x timeline rate. Therefore, I decided to change it back to 5 to reduce calculation steps for computer.