Maheel Luqman
Final Project Report: Using NEAT to Beat Flappy Bird
14 December 2024

Introduction

When computer scientists create traditional neural networks, they typically presuppose the best neural network architecture. However, the Neuroevolution of Augmenting Topologies (NEAT) challenges this belief. Usually, neural networks learn under a fixed architecture, preset by its creator. They only refine the connection weights and biases in order to minimize loss. However, NEAT prioritizes learning differently by evolving both the connection space (weights and biases) and topology of neural networks. As a neuroevolutionary algorithm, NEAT combines the genetic algorithm and neural networks. Unconventionally, it managed to crack the code in allowing neural networks to "build" themselves by overcoming three hurdles. The first hurdle was the competing conventions problem. This problem states that if neural networks with different structures "mate" to form offspring networks, they are likely to produce damaged offspring. Thus, NEAT created a standard method to transform (or encode) neural networks with varying structures into "gene lists" (called genomes). Each neural network is represented by a single genome. This genome consists of two types of genes: node (neuron) genes and connection genes. To solve the competing conventions problem, the connection gene contains an innovation number. This number dictates the first time that this connection has ever come into existence. Thus, every gene has a historical marking that allows NEAT to properly crossover neural networks for viable offspring. The second hurdle comes from the nature of genetic algorithms: the survival of the fittest. When new topologies arise from NEAT, they may initially not be as fit as the veteran networks that have already undergone multiple generations. Thus, to protect innovation, NEAT divides the network population into species where only members of the same species can compete with each other rather than the whole population. The third and final piece is finding the most optimal structure for the problem at hand. We want a neural network that solves its task without an excessive topology. In other words, we want the most minimal and efficient structure, but we have to make sure that NEAT's random evolution doesn't create topologies that are too complex or excessive. NEAT, fortunately, keeps topologies minimized throughout evolution. Every initial NEAT population starts out with the most minimal networks (simply an input layer and output layer). Then, NEAT performs structural and value mutations incrementally (single additions or alterations).

By fending off these dangerous hurdles, NEAT makes self-building neural networks possible. I intend to showcase this wonder in a simple game environment. Specifically, I will utilize NEAT to train an AI agent to beat the game Flappy Bird. My goal is to evolve an AI bird capable of navigating pipes indefinitely without dying, achieving a theoretically infinite score. Starting with minimal networks of three input nodes and one output node, I will allow NEAT to determine the optimal structure through evolution. The optimal structure will be showcased in the experiment analysis. I intend to show that I don't always have to presuppose the structure or behavior of a neural network. Sometimes, I may only need to give them the right information to play with.

Method Description

Game Development

In order to let my AI beat Flappy Bird, I first have to make Flappy Bird itself. To build Flappy Bird, I used pygame. I identified the different objects comprising the game. The objects are the bird, pipes, floor, and background. Having represented each object using a class or functions, I, then, programmatically translated their behavior and interactions into proper game mechanics. For example, the bird's and pipe's velocities were the perfect speed for both a human or AI to play the game. Furthermore, since Pygame represents all entities using rectangles regardless of their shape, pixel perfect collisions were necessary between the bird and pipes. Thus, I used Pygame's mask feature to make the bird have a more "circular" hitbox, so its contact with the environment was believable to the user. The most important aspect of the game (with respect to NEAT) was the main game loop.

NEAT Configuration and Set Up

The NEAT algorithm runs the game loop for every network population. Each neural network in the population controls a bird created by the loop. However, before I could give the game to NEAT, I first needed to perform some proper setup. The library that I used to implement NEAT is called neat-python. This library requires me to make a configuration file for the code to reference as parameters for my neural networks and their evolutionary behavior. First, I must come up with the input and output layers of the neural networks. The input layer must contain enough game information for the neural networks to properly do their job. I made the input layer contain three nodes: the bird's vertical position, the bird's vertical distance from the top pipe, and the bird's vertical distance from the bottom pipe. The output layer represents the bird controls. Thus, the output was one node that dictated whether the bird should jump. Now, the neural network has a starting topology to grow from. Next, I had to decide on the activation function between layers. What makes NEAT even more interesting is that neural networks can choose their own activation functions through mutations. However, I still had to give them a starting one. I chose the tanh function because it is zero centered and keeps outputs between a small range of -1 to 1. The range is preferable since I want the networks to make a binary decision (to jump or to not jump). To explore more possibilities, the networks also had a 10% chance to mutate this activation function into either sigmoid, ReLU, or tanh (once more). For the genetic algorithm set up, I had to choose the population size and fitness function. I selected 50 neural networks to run each generation. I landed on this number due to the simplicity of the task, network diversity, and feasibility to learn the game. However, this number was very flexible. Meanwhile, the fitness function was deemed crucial and required careful finetuning. NEAT provides a list of genome objects that represent the networks. These objects come with a fitness attribute. Thus, if a network performed a certain action, I would either add or subtract value to its genome's fitness attribute. I performed this alteration in 3 locations resulting in this fitness function:

fitness = 0.1 * frames_survived + 5 * pipes_passed - 1 * pipe_collided

For every frame that the network controlled a surviving bird, I would reward the network with 0.1 fitness. Here is what the code looked like:

```
            bird.move()
            # if the bird has survived a frame from moving, add a little
fitness
            genome_list[index_3].fitness += 0.1
```

For every pipe that the network managed to get its respective bird through, I rewarded the network with +5 fitness. I really wanted to encourage the network to survive the game and go through pipes.

```
        # if we are ready to add a new pipe to the game
        if add_pipe:
            # add 1 point to game score
            score += 1
            # for each genome still in the game
            for surviving_genome in genome_list:
                # give them 5 points for getting through said pipe
                surviving_genome.fitness += 5
```

Finally, if a neural network killed their bird by colliding it with a pipe, I took away 1 fitness point from their genome. Here is the last piece of the fitness function in code:

```
                # if the bird collided with the pipe
                if pipe.collide(bird):
                    # we want to discourage the bird from hitting the pipe
                    genome_list[index].fitness -= 1
```

The last piece of setup was the ending condition for the NEAT algorithm. I simply stated that if a bird reached a score of 50, the NEAT algorithm does not need to run anymore. The reason is because by that time, I knew that the neural network controlling said bird is very much capable of playing forever without dying. The environment was too simple to surprise a neural network of that caliber.

Implementing NEAT

Now that NEAT is properly configured, I had to implement it into my main game loop. I created 3 arrays: neural_networks, genome_list, and birds. The neural networks list holds every neural network going through the game loop. The genome list holds each neural network's corresponding genome. The birds list holds each bird that a neural network controls. Each index in every list corresponds to the same individual in the population. Using the configurations, NEAT generates the genome objects for me (the first generation of genome objects is the initial random population of minimal neural networks). I just pack them up in my genome list and then create neural networks out of them. Then, I give each neural network a bird in the same index within the birds list. Here is the code for more context:

```
    # NEAT actually gives us a list of genome objects (genomes). NEAT
starts
    # with these genomes which are encodings of potential neural networks
(node
    # genes and connection genes).
    # for each genome given by NEAT
    for genome_id, genome in genomes:
        # construct the neural network from the genome
        neural_network = neat.nn.FeedForwardNetwork.create(genome, config)
        # add it to the list of neural networks
        neural_networks.append(neural_network)
        # give it a corresponding Bird to control
        birds.append(bird_class.Bird(230, 350))
        # set its initial fitness to 0
        genome.fitness = 0
        # add this genome to the genome list for later usage (keep track of
its
        # fitness and change it as we desire)
        genome_list.append(genome)
```

To allow neural networks to control their bird, I have to feed 3 input values to the network which will pass this information forward and produce an output (Therefore, these neural networks are actually called feedforward networks). If the output is above 0.5, then, my pygame code will make the bird jump. The output value could be anything between 0 and 1 since I can have either sigmoid and tanh at play here. 0.5 felt like the perfect, “middle-ground,” threshold value for my neural network to make a decision. Here is the code for more context:

```
        # for every bird still playing the game
        for index_3, bird in enumerate(birds):
            # move the bird
            bird.move()
            # if the bird has survived a frame from moving, add a little
fitness
            genome_list[index_3].fitness += 0.1

            # holds the value of the output of the neural network
            # the arguments are the inputs that we feed the neural
networks:
            # the bird's y position, the top pipe's vertical distance from
```

```
            # the bird's position, and the bottom pipe's vertical distance
from
            # the bird's position. output is actually a list
            output = neural_networks[index_3].activate([bird.y,
                        abs(bird.y - pipes[pipe_index].top_pipe_end),
                        abs(bird.y - pipes[pipe_index].bottom_pipe_start)])
            # if the output reached the jump threshold which is 0.5
            if output[0] > 0.5:
                # make the bird jump
                bird.jump()
```

A neural network could fail in one of three ways. First, it could send the bird flying above the game window or let the bird fall into the ground. Otherwise, it could send a bird smashing into a pipe. When any of these events occur, I simply pop out the individual from the three lists.

Running the NEAT Algorithm

With all of these configurations and code finally completed, the NEAT algorithm can finally take place. The neat-python library performs most of the heavy-lifting in this regard. First, the library generates a population of 50 neural networks. Then, passing in the population as 50 genomes, it runs my main game loop. In the loop, the entire population plays one game of flappy bird. During this play session, each network's fitness value is calculated as mentioned before. Once every bird in the population has died, neat-python splits the population into species using their genes, innovation numbers, and a formula for determining gene similarity. Within each species, every individual gets ranked by their calculated fitness. From the configuration file, neat-python only keeps the top 2 individuals plus the remaining 20% of the species while eliminating the rest. Then, neat-python calculates the ranking of species based on the sum of fitness scores of their members. This ranking determines the proportion of offspring that a species is allowed to reproduce. Now that each species knows how many children it can reproduce, the members reproduce with each other in two ways. Either a member simply mutates into a new offspring, or it crosses over with another member of its species and forms a new individual. This individual also mutates like the first asexual method. The mutation occurs in one of three ways. First, the weight value of a connection can be randomly changed. Second, a new connection can be randomly generated. Finally, a new neuron can be generated in place of a connection, splitting the connection into two (From: node1 → node2, To: node1 → node3 → node2). From this reproduction, neat-python creates 50 new individuals (a new generation) to go through my game loop once more. We hope that these 50 individuals can give their birds a longer lifespan than the previous generation did. This algorithm runs until either a bird gets 35 points, or neat-python has made 50 generations. Here is the function that passes my code to neat-python's algorithm:

```
def run(configuration_file_path):
    # connects the config file to our code, by connecting all of our
subheadings
```

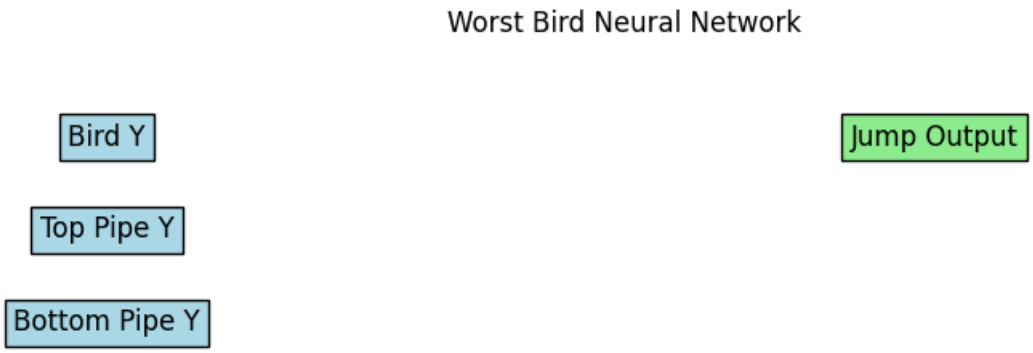
```
    # in our text file, we're telling neat all the properties that we're
setting
    config = neat.config.Config(neat.DefaultGenome,
neat.DefaultReproduction,
                        neat.DefaultSpeciesSet, neat.DefaultStagnation,
                        configuration_file_path)
    # generate the genome population
    population = neat.Population(config)
    # stats reporters, gives us the terminal output that details the NEAT
run
    # with information such as generation statistics
    population.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    population.add_reporter(stats)

    # 50 means how many generations we are gonna run.
    # it also calls the main function 50 times and pass it all of the
genomes
    # and config file. Main will generate a game based on all of the birds
    # (in reality, genomes) that were given.
    # put the population through my main game loop and if we get a winner,
    # return him
    winner = population.run(main, 50)
```

Experiment Output

Finally, once the algorithm has beaten Flappy Bird, we grab the best bird from the run and store it into a pickle file. With this pickle file, we can draw the neural network of this amazing bird using matplotlib. The drawn neural networks will be shown in the next section.

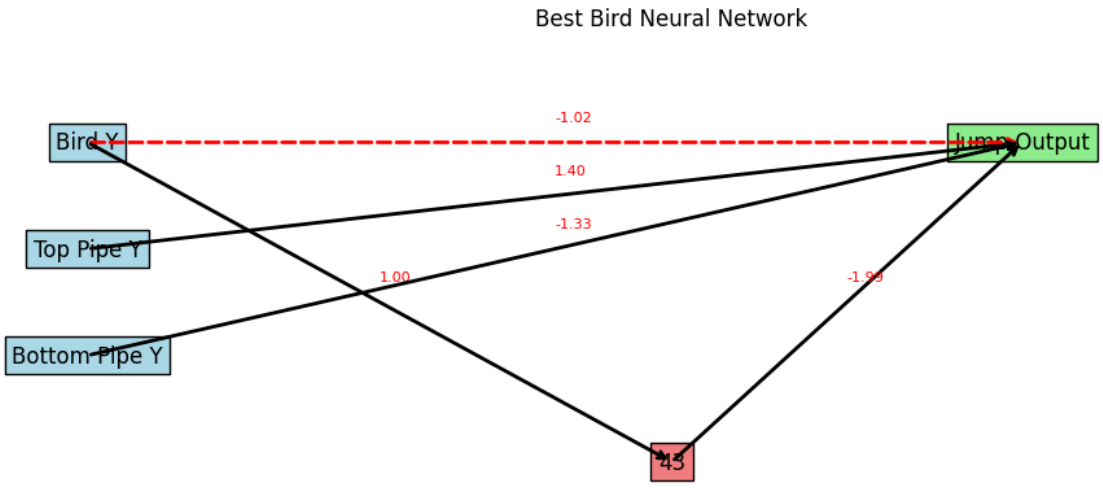
Experiments Analysis

I ran through the experiment twice. Therefore, I acquired two birds that successfully beat the game. To showcase the evolution from beginning to end, I will show the initial and final structures of the birds. Both birds start with the same structure. I call this structure the worst bird structure as shown below:

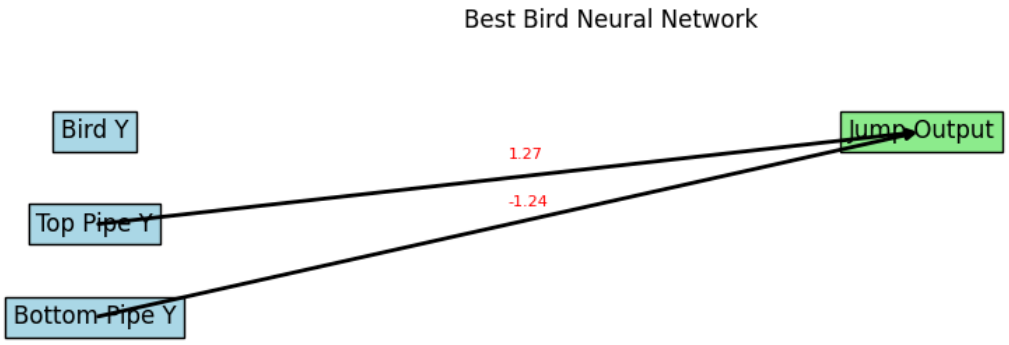


The birds start with the most basic topology possible. They simply have 3 input nodes (Bird Y, Top Pipe Y, and Bottom Pipe Y) and 1 output node (Jump Output). The nodes are not even connected to each other. Thus, when these birds initially played the game, they immediately fell to their deaths. After undergoing the NEAT algorithm and methodology explained earlier, the bird's structures evolved as shown below. I'll call them Bird 1 and Bird 2 respectively.

Bird 1



Bird 2



The most interesting observation is that despite accomplishing the same exact goal, their topology and connection weights are completely unique. Every time that I run the NEAT algorithm, it generates a different output. For example, Bird 1 evolved into 5 nodes and 4 connections. It created a hidden node in a new layer which is called “45” in the diagram. The red line connection means that the bird disabled the connection in favor of the hidden node connections. It utilizes all three inputs to generate a proper output value. Meanwhile, Bird 2 evolved into 4 nodes and only 2 connections. Considering the connection weight values, both birds most likely use the tanh activation function between every layer. Now, despite both birds generating the same fitness value (since they both beat the game), Bird 2 has the more optimal structure, so I’ll go into more depth with this bird. The neural network controlling Bird 2 didn’t even need the bird’s location to perform its tasks. This impressive feat exists because of the other information that I fed the neural network. I gave the network the pipe’s location relative to the bird’s location. More specifically, the network knows the vertical displacements of the pipe pair (the pipes that the bird must go in between) relative to the bird’s vertical position. Thus, applying a linear computation and tanh on these two inputs was enough to generate a valid output for conditional jumping. The pipe heights are random throughout the playthrough, so the neural network must see the randomly generated pipes and then respond accordingly. However, since the gap size is consistent for every pipe pair, and Bird 2 disregards the bird’s vertical position, I’ve realized that the most minimal structure for a successful bird would probably only have 2 nodes: 1 input node and 1 output node. The input node must either be Top Pipe Y or Bottom Pipe Y. Additionally, Bird 2’s weight values are significant for exposing its behavior. The two inputs have roughly the same absolute weight values (|-1.24| and |1.27|). These weights indicate that the inputs contribute roughly equally to the output calculation. Since the Top Pipe Y’s weight is positive while the Bottom Pipe Y’s weight is negative, they must contribute in opposing directions to the output. Bird 2 seeks to “jump up” towards the top pipe’s height while seeking to “fall down” towards the bottom pipe’s height. A balance between these two actions is achieved when the bird goes in between the pipes.

I enhanced my data collection by using neat-python’s Reporter object. This reporter records data beyond that of simple diagrams. Here is what the reporter had to say about my second experiment. (This experiment resulted in Bird 2):

****** **Running generation 0** ******

```
Population's average fitness: 2.79400 stdev: 0.12395
Best fitness: 3.20000 - size: (1, 0) - species 1 - id 32
Average adjusted fitness: 0.094
Mean genetic distance 0.711, standard deviation 0.499
Population of 50 members in 1 species:
   ID   age  size  fitness  adj fit  stag
 ====  ===  ====  =======  =======  ====
    1    0    50      3.2    0.094     0
Total extinctions: 0
Generation time: 1.365 sec

 ****** Running generation 1 ******

Population's average fitness: 2.83000 stdev: 0.19723
Best fitness: 3.50000 - size: (1, 1) - species 1 - id 93
Average adjusted fitness: 0.130
Mean genetic distance 0.946, standard deviation 0.589
Population of 50 members in 1 species:
   ID   age  size  fitness  adj fit  stag
 ====  ===  ====  =======  =======  ====
    1    1    50      3.5    0.130     0
Total extinctions: 0
Generation time: 1.271 sec (1.318 average)

 ****** Running generation 2 ******

Population's average fitness: 3.08400 stdev: 0.77107
Best fitness: 7.50000 - size: (1, 1) - species 1 - id 105
Average adjusted fitness: 0.080
Mean genetic distance 1.227, standard deviation 0.511
Population of 50 members in 1 species:
   ID   age  size  fitness  adj fit  stag
 ====  ===  ====  =======  =======  ====
    1    2    50      7.5    0.080     0
Total extinctions: 0
Generation time: 2.931 sec (1.856 average)

 ****** Running generation 3 ******

Population's average fitness: 3.52000 stdev: 1.59399
Best fitness: 8.40000 - size: (1, 2) - species 1 - id 185
Average adjusted fitness: 0.144
Mean genetic distance 1.408, standard deviation 0.520
Population of 50 members in 1 species:
```

ID age size fitness adj fit stag
==== === ==== ======= ======= ====
1 3 50 8.4 0.144 0
Total extinctions: 0
Generation time: 3.381 sec (2.237 average)

******** Running generation 4 ********

Population's average fitness: 3.85800 stdev: 2.46204
Best fitness: 17.90000 - size: (1, 1) - species 1 - id 230
Average adjusted fitness: 0.076
Mean genetic distance 1.429, standard deviation 0.429
Population of 50 members in 1 species:
ID age size fitness adj fit stag
==== === ==== ======= ======= ====
1 4 50 17.9 0.076 0
Total extinctions: 0
Generation time: 4.573 sec (2.704 average)

******** Running generation 5 ********

Best bird saved successfully!
Population's average fitness: 13.67800 stdev: 62.04256
Best fitness: 443.00000 - size: (1, 2) - species 1 - id 260

Best individual in generation 5 meets fitness threshold - complexity: (1, 2)

The NEAT algorithm underwent 5 generations to finally complete the game. The algorithm, then, generated Bird 2. Bird 2's fitness is 443.00 which equals the total fitness of surviving 35 pipes and thousands of frames. Bird 2 could've kept going, but I stopped it since it met the fitness threshold. Interestingly enough, Bird 2 originates from the very first species ever made (species 1). Since neural network topologies were unbelievably simple in this scenario, structural and value differences were too insignificant to create many species. Bird 2 is the 260th bird produced by NEAT and flew with a population whose average fitness is 13.678. Many of its brethren were vastly weaker than it. Most importantly, throughout every generation, the neural networks only got better. For example, Generation 1's average fitness was 2.794. By Generation 3, the average fitness was 3.52. Stagnation amongst all species stayed 0 throughout the entire experiment. This popular trend for constant improvement proved that I never needed to presuppose the topology or connection space for any neural network in this experiment. The neural networks built themselves, from the ground up, growing connections and (sometimes) neurons along the way. All I had to give them was enough information to play the game.

Conclusion and Learnings

By the end of the experiment, I managed to generate not one but two birds who could theoretically beat Flappy Bird. Both birds started with nothing but floating neurons and 0 connections. Then, they evolved into creatures that can navigate pipes until the end of time. NEAT proved itself once again by creating optimal structures through meticulous and tactful evolution. I only had to make educated guesses on the input information and let NEAT handle everything else. Neural networks may not require their topologies predetermined for them. They can transform themselves much like organisms in the real world.

To implement NEAT, I first had to learn about NEAT. To learn about NEAT, I first had to learn about neuroevolution. To learn about neuroevolution, I first had to learn about the genetic algorithm and neural networks. Fortunately, this class taught me about neural networks. Thus, this project taught me the rest. I learned of newfound methods for neural networks to learn and grow. Then, I learned about how I can implement such methods. I learned reinforcement learning where neural networks were forced to adapt to their Flappy Bird environment. I explored many AI concepts and algorithms along the way and hope to learn many more in the future.