Neuroevolution of Augmenting Topologies (NEAT): Flappy Bird

Written by: Emma Kantola and Ryan Yoseph

May 30th, 2020

Abstract

Machine learning is a relatively new and interesting area of research in artificial intelligence, most commonly studied in order to create more fun and challenging video games. In this study, the implementation of Neuroevolution of Augmenting Topologies (NEAT) was applied to a popular side-scrolling mobile game called FlappyBird. With this implementation, the goal is to create a neural network (performing as the agent) that can navigate its way through the pipe gaps more so than a typical human. Through multiple generations, the agent adjusted its genome with respect to the fitness function, and as generations went by, more and more bird agents successfully met this goal of performing the game properly. Thus, the neuroevolution of augmenting topologies proved successful when implemented in Flappy Bird.

I. Presentation and Choice of the Game

FlappyBird (Figure 1) is a side-scrolling mobile game created in 2013 that caught the attention of millions with its simplistic yet addicting concept. In FlappyBird, the user is to direct a bird up or down through pipes as the screen moves to the right. For every pipe that the bird successfully makes it through, the user is rewarded one point, until the user collides with a pipe.

FlappyBird, although simplistic, is an interesting neural network training game in that it is very easy to see how the AI responds to the fitness function and implements that knowledge into future generations, seeing as the neural network learns from the failure of hitting a pipe. Additionally, since FlappyBird only has a few input layers—the bird's y-position, the y-displacement to the top pipe, and the y-displacement to the bottom pipe—it is easy to see how the weights of each input layer affect the output. Therefore, by not picking an overly complicated game, I am able to see and understand more easily how neural network training works.

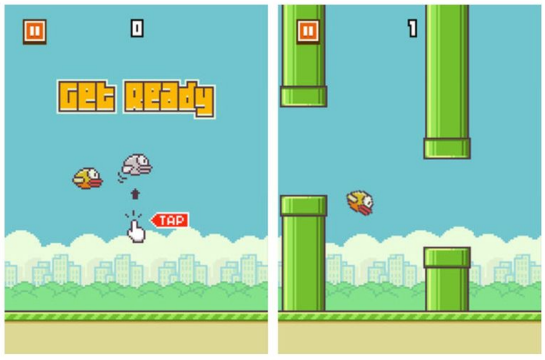

Figure 1: Flappy Bird Game

Possible features that could complicate the learning process for the neuroevolution would be having the pipe gap become smaller, or having the pipes move up and down as the bird tries to maneuver through the obstacles. Both of these new features would require the AI to be more aware of the changing environment, and thus need more accuracy in its generations. Instead of simply adding new features that make it easier for death, adding the possibility to increase its fitness more with a more positive decision would also be an interesting installment. For example, the pipe could not have only one opening, but two, and in one of the openings have a coin that would allow the bird to gain two points (one for going through the pipe and one for acquiring the coin). This would add a higher fitness increase for the birds who choose the path of two points, thus allowing it to evolve to reach an even

higher score with possibly fewer generations. Implementing more hindrances as well as rewards results in more variables for the AI to consider, changing the weights of every decision, and creating a more optimal output for the AI.

In the FlappyBird universe, the state space has static and dynamic information. The static information are the variables kept constant in the environment that Faby has to react to, such as the velocity being 6 and the gap of the pipes being 200. Dynamically, the gap of the pipe remains the same but where the gap occurs in the pipe does not, thus this state space is constantly changing.

In the action space, there are two options in which the user has to control Faby (the bird) - jump up, or jump down. This option correlates to where the pipe opening is on the next pipe that Faby has to fly through. Therefore, given the state of the location of the pipe, Faby makes a decision given the dynamic information received even before crossing the previous pipe. This action is dependent on the placement of the opening of the pipe. If the hyperbolic tangent function outputs 0.5 based on the inputs: the y-position of bird object, distance between top pipe and bird, and distance between bottom pipe and bird, then Faby will not jump. If the hyperbolic tangent function is above 0.5 based on the same inputs, then Faby will jump, as shown with a for loop in Figure 2.

```
for x, bird in enumerate(birds): # If value > 0.5, make bird jump.
  bird.move()
  ge[x].fitness += 0.1 # every second our bird stays alive, we give the bird one fitness point

  #Activate the neural network with our inputs.
  output = nets[x].activate((bird.y, abs(bird.y - pipes[pipe_ind].height), abs(bird.y - pipes[pipe_ind].bottom)))

  if output[0] > 0.5:
    bird.jump()
```

Figure 2: For Loop for Action State

Due to the lack of randomness in FlappyBird from its state and action space, the environment is deterministic. Although the pipe location is random, there is no action that the agent has to take that is random, thus being a deterministic environment.

Instead of implementing the typical reward function associated with reinforcement learning, neuroevolution of augmenting topologies uses a function of fitness to evolve the agent through generations. The fitness function implemented pertains to three possibilities of how long the Faby stays alive, if Faby hits a pipe, and if Faby successfully passes through a pipe. However, to evolve the agent faster, the weight of the fitness is not the same value for all outcomes. For example, for every second Faby is alive, she only gets a fitness point of 0.3 with a clock at 30, for every pipe Faby hits, she loses a point, and for every pipe Faby successfully crosses, she gets +5 points. This, in turn, trains the neural network to change behavior based on highest fitness points, creating more successes with each generation.

II. Implementation of Your Agent

The algorithm implemented to train the agent is the neuroevolution of augmenting topologies, in which multiple neural connections are made and tested, with the best connection being chosen for the agent to use. However, the best connection is not automatically used in just all generations, the failing connections can choose different neural paths, outputting a possibly better agent. With this algorithm, to test different solutions for more optimal outputs we decided to change the fitness points allotted for the three different actions. In the first implementation, the variables were set at:

Population Size = 20
Activation Function = HyperbolicTan

Staying Alive Fitness = +0.3; clock tick at 30 ticks per second
Getting Past Pipe Fitness = +5
Collide with Pipe = -1 fitness
Input Layer = 3
Hidden Layer = 0
Output = 1

In the second implementation, the variable of getting past the pipe was changed to:

Getting Past Pipe Fitness = +1

And finally, in the third implementation, the same variable was changed to:

Getting Past Pipe Fitness = +10

Plotting the generation number vs. the population average fitness (Figure 3), it can be seen that the value of +5 for the "getting past pipe fitness" renders the best output with this changing variable.

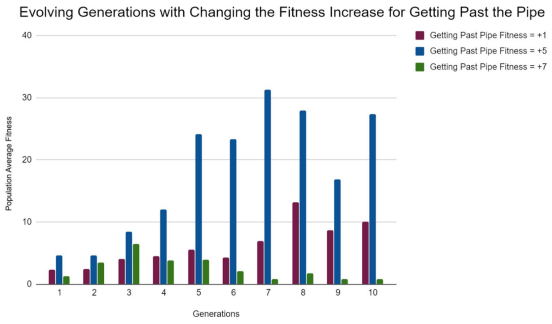

Figure 3: Evolving Generations with Changing the Fitness Increase for Getting Past the Pipe

The implementation decided on had the ending parameters of:

Population Size = 20
Activation Function =
Staying Alive Fitness = +0.3; clock tick at 30 ticks per second
Getting Past Pipe Fitness = +5
Collide with Pipe = -1 fitness
Input Layer = 3
Hidden Layer = 0
Output = 1

An additional parameter that was tested was changing the activation function from HyperbolicTan to Sigmoid with different greater than outputs, which is demonstrated in Figure 4 below.

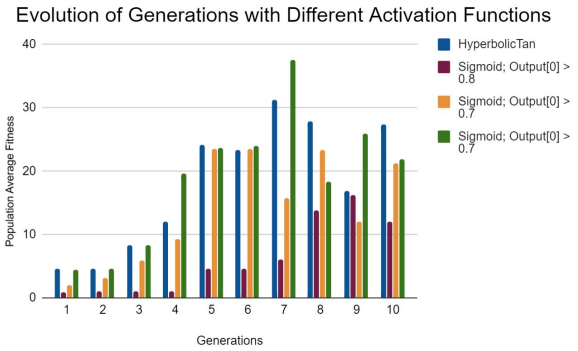

Figure 4: Evolution of Generations with Different Activation Functions

As it can be seen, the activation function with hyperbolic tan proved to not output the largest population average fitness, but did output the most consistent, rendering it with the most efficient parameters of:

Population Size = 20
Activation Function =
Staying Alive Fitness = +0.3; clock tick at 30 ticks per second
Getting Past Pipe Fitness = +5
Collide with Pipe = -1 fitness
Input Layer = 3
Hidden Layer = 0
Output = 1

Thus, our implementation ended with the results of Figure 5.

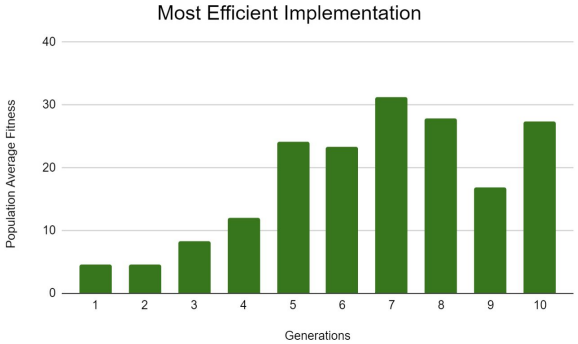

Figure 5: Most Efficient Implementation

With this implementation, the largest problem encountered was the first generation of birds continuously either trying to fly over the pipes or dive-bombing the ground. We were able to fix this with line 287 in our code: if we found birds whose

heights were too close to the top or bottom, we would pop the bird, its genome, and its neural network from our list of birds. The second largest problem we ran into when training was the problem of having two pipes on the screen at the same time. The neural network needs to evaluate the respective displacements from the bird's y-position with the top and bottom of the pipe. Therefore, having two pipes on the screen would prove horrendous, and thus line 237-240 in our code would fix this problem by creating a pipe index.

III. Evaluation of Your Agent

The agent learned this rather simple game fairly quickly, practically mastering the game around the 3rd-4th generation. Its failures were primarily found when the Δy-distance between consecutive pipe gaps was large, and during initialization when all bird genomes were weak. At the second generation, given elitism in [DefaultReproduction] = 2, the bird agents had a stronger genome to pull from than the first generation, and thus profited much better. As generations went by, the fitness of the individual birds grew, therefore the average number of birds that went further into the game grew as well.

When comparing the agents against themselves, there was always an average of 2 birds that would either fly to the top of the screen or would fly to the bottom. During the first generation, a maximum of two birds would make it past the first pipe, and sometimes none of them made it past either. As generations went by, more and more birds would make it past the pipes together. This makes sense because the gene pool increases in fitness, therefore passing down the superior neural network to the next generation.

Due to the simplicity of Flappy Bird, this agent is the same as most artificial intelligence designed for Flappy Bird: with the agent playing the bird will never run into a pipe and thus continue flying forever. Therefore our agent is on par with the majority of machine learning algorithms out there.

As shown in Figure 6, the playing skill of the AI beat Ryan playing Flappy Bird over the 10 generations. Within Generation 3, the agent had successfully mastered the game, acquiring the maximum points that we would let the agent run for, whereas Ryan did not obtain a score over 19. With this information, it is no doubt that the AI would outperform any human "master" playing the game as well.

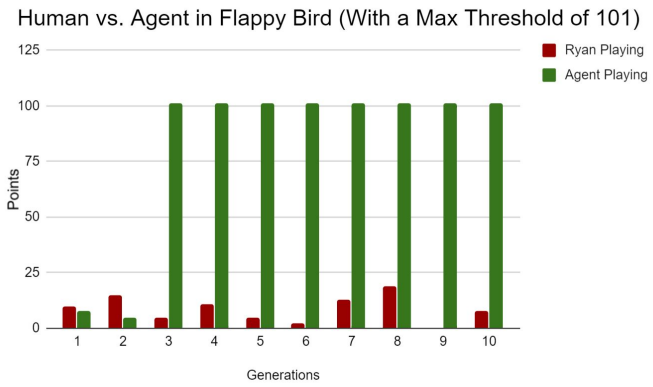

Figure 6: Human vs. Agent in Flappy Bird (With a Max Threshold of 101)

IV. Conclusion

NEAT implements the idea that small neural networks can evolve into complex ones over generations, ending with neural networks that are able to beat games at a rate humans cannot compete with. The application of the NEAT algorithm on the viral game Flappy Bird proved to be very efficient, with the highest average population fitness found when using a sigmoid function with output > 0.75. It only took an average of 4 generations to successfully train and reproduce successful genomes.

REFERENCES:

1. “Configuration File Description¶.” *NEAT*, neat-python.readthedocs.io/en/latest/config_file.html.
2. Stanley, Kenneth O., and Risto Miikkulainen. *Efficient Evolution of Neural Network Topologies*. The University of Texas at Austin.