FlapPyBI/O

Michael Iuzzolino Brennan McConnell Allie Morgan Nicole Woytarowicz

1 Introduction

Our goal was to implement the NEAT algorithm to solve the Pygame Flappy Bird [6]. As we learned, developing a tool in a short amount of time that learns how to play Flappy Bird is a complex and ambitious task. Existing work such as MarI/O was a great representation of the potential ability of NEAT. Though, the MarI/O implementation, while very impressive, simply handles a single level in the game where all enemy movement and maps are predetermined and unchanging. We wanted to consider if NEAT could handle non-deterministic problems [3]. Flappy Bird has an element of entropy - pipes are randomly placed. With this in mind, we set out on the adventure of a lifetime: solving Flappy Bird with NEAT.

2 Background

2.1 Neural Networks

An artificial neural network (ANN) is a machine learning model based loosely on the way brains solve complex problems biologically. Neural networks are composed of three types of layers of nodes (neurons): input, output and hidden. Each neuron receives input(s) with some weight associated with each and computes a single output based on an activation function. The input layer typically normalizes the inputs received. Hidden layers create additional non-linearity to our decision boundary. Finally, the output layer produces a result(s) which we may threshold before taking action on. In a feed-forward neural network, the outputs of each layer are fed to the next, with no loops tracing back to earlier layers [4].

2.2 Genetic Algorithms

Genetic algorithms (GAs) are iterative methods used to solve optimization or search problems, drawing their inspiration from natural selection. A genetic algorithm begins with a population of solutions (or species) and then randomly combines pairs of fit solutions over time (an analogue to mating, where different parts of each parent solution are inherited) to produce solutions that are potentially more fit. Fitness is evaluated using some fitness function, which defines what parameters should be maximized by any given solution. Random mutations introduced in the mating process are a key feature of genetic algorithms. Although in many problems GAs rarely converge to a global optimum; yet, they tend to perform well when finding local optima [2].

3 NeuroEvolution of Augmenting Topologies

The NeuroEvolution of Augmenting Topologies (NEAT) algorithm was developed in 2002 by researchers Ken Stanley and Risto Miikkulainen in the computer science department at the University of Texas, Austin. The algorithm is based on three key features that allow it to outperform fixed-topology neuroevolution algorithms: employing crossover of different network topologies, utilizing speciation of networks to preserve structural innovations, and incrementally growing networks from simple structures.

The principle of NEAT is to start with as simple a network as possible – only an input and output layer. The complexities of the hidden layers will emerge throughout evolution, as needed. That is, the network topology will augment via selective evolution; hence, neuroevolution of augmenting topologies [5].

This genetic algorithm parallels a real-life biological system. There exist 3 main components to GAs: generations, species, and organisms. A single organism, commonly referred to as a genome, is represented as a single neural network. A species is a collection of these organisms. Organisms similar in structure and topology would be part of the same species. If an organism becomes too different from other members of its species, it will become its own species. Finally, GAs work by iteratively evolving these species over time. Each iteration in NEAT is called a generation. During each successive generation, the organisms of each species will execute.

NEAT encodes neural networks with nodes (neurons) and connection edges (genes). Genes describe the connection between two neurons by specifying the input neuron, output neuron, the weight of said connection, whether the connection is enabled/disabled, and an innovation number that identifies the origin of the connecting genes [5]. In this section we will provide an overview of the algorithm.

3.1 Fitness

The way NEAT and other GAs discover solutions is through random mutations. Initially, a single species is created with n organisms. Each of these genomes consists of only an input layer and output layer. All genes of each genome are assigned randomly. A fitness function is used to score each organism and indicate how well that organism performs some objective task.

Since we do not have labeled data to train on, we can instead determine the success and ability of an organism based on its corresponding fitness score. Thus, the goal becomes to maximize the fitness score an organism can achieve instead of trying to maximize its accuracy relative to labeled data.

3.2 Selection and Replication

At the end of each generation, organisms are ranked according to their fitness performance. The more fit organisms survive in order to create offspring (progeny) and the less fit organisms that did not have useful weights/structure are killed. Following this, the species is re-populated for the upcoming generation. Three distinct methods of creating progeny arise.

First, some progeny are created by randomly cloning some of the surviving genomes and then undergoing mutation. In the second technique, two parents are chosen randomly from the survivors and mate to create a progeny that also undergoes mutation. Finally, the champion of the previous generation is always replicated exactly and passed on as a progeny for the upcoming generation. In this way, we never allow our most fit organism to disappear in the case that it was not selected during cloning or crossover. This method of selection and replication occurs for each species during every generation.

3.3 Mutation

NEAT's main performance boost comes from allowing almost all parts of a neural network to change. This gives NEAT the advantage of being able to find not only correct weights for neural networks, but also the optimal topologies. Mutations occurs in two ways: weight and structural. Weight mutations consist of either modifying the value of a connection, or enabling/disabling a connection. During structural mutations either a single new connection gene with a random weight is added between previously unconnected nodes or a new neuron is added. When a new neuron is added, an existing connection is split in two. The new connection leading into the neuron has a weight of 1, while the out-connection retains the weight of the original connection [5]. In this way, we can minimize the impact that the addition of this neuron has on the current network. These mutations allow new network to topologies arise and new structures to evolve into ever-increasing complexity as is necessary to find the optima.

3.4 Crossover and Innovation Numbers

Global innovation numbers are used to identify historic origins of a gene. Whenever a new gene is added by mutation, the gene's innovation number is incremented [5]. This gives NEAT the advantage of knowing which genes originated when in the genome's evolution and is a method of tracking the ancestry of any given genome.

Innovation numbers are critical for crossover — a process in which the genomes are lined up to create progeny (see Fig. 1). During crossover, the parents' genes are lined up with genes that have matching innovation numbers. These are

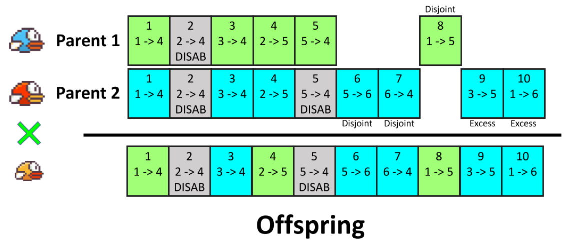

Figure 1: Example of crossover

known as matching genes and are inherited randomly. Furthermore, during crossover, the matching genes' weights are averaged to create a connection gene for the child.

The remaining unmatched genes are either disjoint (those that do not match in the middle of the genome) or excess genes (those that do not match in the end of the genome). The progeny inherit the disjoint and excess genes from the fitter of the parents [5]. Innovation numbers and crossover give NEAT its ability to augment topologies.

3.5 Speciation

Smaller networks tend to have higher fitness and optimize faster than larger networks. There is a chance that after adding structural mutations to a genome, it will not survive the selection process despite having possibly critical innovations. NEAT solves this problem through a method called speciation.

After species go through the selection, replication, and mutation processes, all genomes are analyzed to determine their species membership. Each species has a representative genome that signifies the topology of organisms within that species. In every generation, all genomes in a species are compared to the species representative genome. If they are compatible, the genome stays in the current species. If the genome has evolved topologically to the point where it is no longer compatible with its current species, it is then analyzed to determine if it is compatible with any other currently existing species, and is reallocated respectively. If no such species exists, a new species is created and the initial genome becomes the representative genome for the new species. This process of speciation allows for new structure to be given time to optimize and necessary complexity to be derived.

Compatibility is determined via distance, δ, representing the similarity of genomes. Compatibility distance is a linear combination of the total number of genes in the genome N, the total number of excess E and disjoint D genes, and the average difference in weight of matching genes $\overline{W}$:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} \qquad (1)$$

The coefficients c_1, c_2, and c_3 can be adjusted to weigh the importance of each of the terms [5]. The normalization factor N is set to the number of genes of the smaller genome. Intuitively, the impact of adding a single neuron in a small genome may be very large on δ, suggesting that the two genomes are no longer compatible. In the other case, adding a neuron in an already very large network does not make as much of an impact and the two genomes may still be compatible. Speciation is the crux of NEAT and shows how structure incrementally evolves to the necessary complexity to solve the objective.

3.6 Population Control

During the execution of NEAT, the ecosystem maintains a constant population n. As mentioned previously, the first species is given n organisms. As new species crop up, the population is divided among all existing species. Species that are improving and show promise are given greater portions of the population. Consequently, species that begin to optimize and show promise will be given more genomes.

If a species' fitness is increasing, it means the topological changes that created that species represented a necessary derived feature with respect to the objective task. If a species does not perform well, it can be culled in two ways. First, if the population of a species decays, it indicates that unnecessary complexity arose that did not represent a useful feature. If the population drops low enough, this species is culled. Second, it may be true that a species is scoring well and is maintaining a healthy population, but that this species has found a local optima and not a solution. If the average species fitness score does not improve after k generations, there are two choices: you can re-populate the species with new random weight connections to bump it out of the non-solution local optima, or you can cull the species. Typically a certain number of re-populations are allowed when stagnation occurs before the species is culled entirely.

4 Implementation

The first step was to obtain and customize a Pygame version of Flappy Bird to interface with neural networks [6]. Next, we developed a fixed topology GA (algorithm 1 in Appendix A). We found moderate success at this point and continued to implement NEAT to see how augmenting topologies performed relative to fixed topologies. All of our code was written in Python.

4.1 Interfacing with Pygame

First, it was necessary to interface Pygame with our genetic algorithm. We modified Flappy Bird by adding a flag that could allow us to set the game to either random pipes (as in the original game), or as uniform pipes (constant unchanging pipes). After our GA solved uniform pipes, we added a third option which enabled pre-configured patterns of pipes. Under this more complicated deterministic scenario, convergence by a GA would be harder than uniform pipes. Furthermore, we added the ability to see species number, generation number, organism population, topology for the fixed topology GA, and various metrics related to the fitness score. Lastly, we modified the game to handle multiple birds playing simultaneously.

4.2 Neural Network

The neural network was originally implemented via TensorFlow. However, given the difficulty in deep copying the networks for the replication phase of the GA and the lack of intuitive,

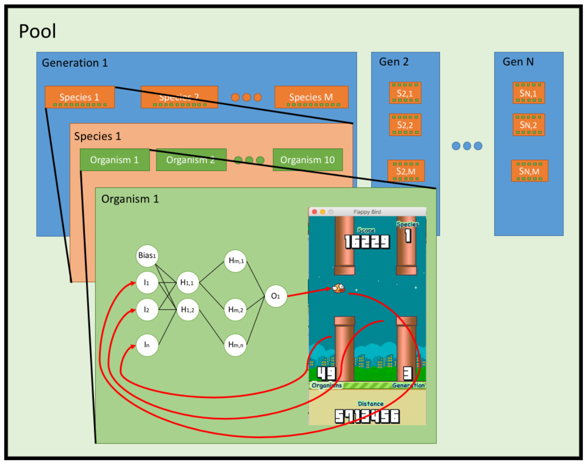

Figure 2: Data Structure Hierarchy

readily available tools for neuroevolution, we implemented our own neural network.

The network topology initiates with 10 input neurons and 1 output neuron. The input neurons correspond to Flappy Bird's game state: the x,y coordinates of a bird, and the x,y coordinates of the upper and lower pipe's center for the next two pairs of pipes. The output neuron yields a 1 or 0, resulting in a flap or no flap, respectively, which is fed as the input back to the Pygame.

The neural network inputs are regularized via Gaussian normalization to prevent hidden layer saturation. The activation function of each neuron is a sigmoid.

Due to NEAT's genetic encoding scheme, matrices were not used. Instead we implemented the neural network in a much more object-oriented approach. The neural net was implemented as a graph. Feed-forward worked as a graph traversal that involved recursively firing a neuron's output when it received all inputs.

4.3 Fixed Topology GA

Our first implementation involved a fairly straightforward genetic algorithm that consists of a selection process where the most fit organisms survive every generation and half are chosen to spawn 2 progeny each. No speciation or population control was implemented and the topology of the neural network was fixed at runtime. We specified the input layer size, output layer size, hidden layer size, and number of hidden layers as command line arguments. Experimentation with this algorithm yielded expected results; if too large of a fixed topology were specified, the search space was too large for convergence to occur. And if the network topology were too simple, it could not converge for more a complicated objective (random pipes).

4.4 NEAT

We then proceeded to implement NEAT according to the description in section 3 (drawing inspiration from [1]). The primary alterations were to the genetic encoding, replication, and mutation schemes. First, we converted the genetic encoding from a set of matrices into a linear genome consisting of genes, each uniquely representing a connection between nodes of the graph and marked by a innovation, as detailed above in section 3.4.

Second, we altered the crossover aspect of NEAT's replication phase. The standard implementation samples the prospective parent organisms from a uniform distribution. This allows for the possibility that the fittest organisms will never mate; simultaneously, the most unfit parents could dominate and produce the entirety of the new generation. We altered this sample space and introduced a skewed distribution where the parent organisms with a higher fitness are more likely to be selected for replication.

Third, we modified the mutation scheme such that new nodes and new links between existing nodes could be added. This is the driving force behind augmenting topologies. Along with these mutations come a large set of parameters: mutation thresholds, rates, etc. Our implementation explored a range of these parameters to discover an optimal combination that would accelerate the learning of Flappy Bird. This was arguably the most difficult task given the immense combinatorial space. Further, there was no intuitive explanation for parameter choice, leading to further difficulty in choosing the correct combinations.

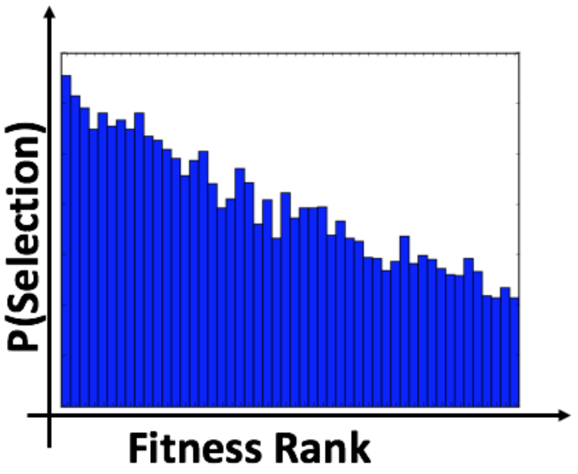

Figure 3: Replication

5 Results

5.1 Fixed Topology GA

Using our fixed topology genetic algorithm, we were able to effectively solve Flappy Bird for uniformly distributed pipes and score relatively well on randomly distributed pipes (around 20 pipes). We ran multiple different trials with this algorithm using different topologies, varying from a simple network with no hidden layers to networks with one hidden layer and up to 8 hidden neurons.

We found that the networks with the most hidden neurons outperformed the simpler structures in terms of how many generations it took before solving the uniform pipes problem, but even simpler structures were eventually able to solve this problem.

5.2 NEAT

We were able to solve the uniform pipes problem as well using NEAT. The main advantage we found with NEAT was its boost in terms of performance; NEAT could solve uniform pipes in as quickly as 3 generations, while it took our the fixed topology genetic algorithm much longer. We also noticed that, despite NEAT not solving random pipes, speciation occurred much more quickly than in the case of uniform pipes. This makes sense because NEAT allows for more complicated structures and random pipes introduce added complexity to the problem. We also found instances of NEAT converging to multiple local optima, which makes sense given that it is a genetic algorithm, as shown below in Fig. 4.

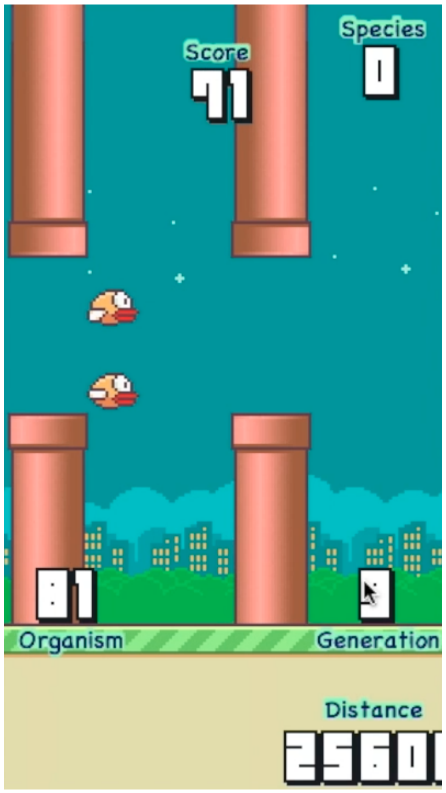

Figure 4: NEAT converging to 2 different solutions

6 Conclusions

Despite our inability to fully solve the random pipes problem within Flappy Birds using a simple fixed topology genetic algorithm or NEAT, we were able to solve the uniform pipes problem with both and show some learning with random pipes. Given the slightly arbitrary nature of GAs, we ended up struggling with deciding the most optimal parameters. It is possible that if we had used a different selection of parameters we would have had better results, but again, as GAs are mostly random, we chose our parameters based on what seemed most logical.

In the future, we could reattempt this project using a different approach instead of GAs. Simulated annealing (SA), for example, is another method used to solve optimization and search problems which tends to perform much better than genetic algorithms. This is because simulated annealing uses transition probabilities to move between solution states. Even if we find a solution at a local optimum, simulated annealing can still stochastically reach optima that are potentially better [7].

References

[1] Adamski, A. (2015) *NEATFlappyBird*, GitHub repository. `https://github.com/NeatMonster/NEATFlappyBird`

[2] Beasley, D., Bull, D., Martin, R. (1995) An Overview of Genetic Algorithms. University of Cardiff, UK. `http://mat.uab.cat/~alseda/MasterOpt/Beasley93GA1.pdf`

[3] Bling, S. (2015) MarI/O - Machine Learning for Video Games, YouTube. `https://www.youtube.com/watch?v=qv6UVOQ0F44`

[4] Neilsen, M. A. (2015) *Neural Networks and Deep Learning*, Determination Press. `http://neuralnetworksanddeeplearning.com/chap1.html`

[5] Stanley, K. O., & Miikkulainen, R. (2002) Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2), 99–127.

[6] Sourabh, V. (2014) *FlapPyBird*, GitHub repository. `https://github.com/sourabhv/FlapPyBird`

[7] Skiena, S. (1997) The Algorithm Design Manual. *Simulated Annealing.*

A Our Algorithm

Algorithm 1 Optimize FlapPy BI/O

```
while FlapPy BI/O not optimized do
  for species in current generation do
    for organism in current species do
      generate fitness of organism by running game
    end for
    cull species
    remove stale species and weak organisms from pool
    globally rank organisms and select top $X$ organisms from generation
    randomly replicate organism to produce progeny (mutation and crossover)
  end for
  for new organism in progeny do
    for representative genome in current species do
      if new organism's genome matches representative genome then
        add new organism to current species
      else
        speciate organism into new species
      end if
    end for
  end for
  if FlapPy BI/O optimized then
    break
  else
    increment to next generation
  end if
end while
```

B Team

- **Michael Iuzzolino**: Team lead, implemented NEAT, designed presentation slides
- **Brennan McConnell**: Implemented fixed topology GA, implemented NEAT
- **Allie Morgan**: Modified Pygame, interfaced genetic algorithms with Pygame
- **Nicole Woytarowicz**: Decided inputs, decided fitness functions, implemented sampling during selection, researcher