NeuroEvolution of Augmenting Topologies

Nathan Zabriskie

April 18, 2017

1 Introduction

Genetic algorithms learn by heuristically sampling possible solutions to some problem, assigning a fitness score to each solution based on its performance, and then combining and tweaking the most "fit" solutions to create more samples for the next step of the simulation. Traditionally, genetic algorithms would operate on a fixed network topology specified by the user, with each genome representing some possible weights for connections within the network. Although researchers made some early attempts to develop genetic algorithms that evolved topologies along with weights (called Topology and Weight Evolving Artificial Neural Networks or TWEANNs), these algorithms often suffered from a number of problems including information loss due to competing conventions and overly penalizing new, larger topologies.

In *Evolving Neural Networks through Augmenting Topologies* [1] Stanley and Miikkulainen present their algorithm NeuroEvolution of Augmenting Topologies (NEAT) which addresses some of these problems. This paper examines the contributions of NEAT to genetic learning and evaluates its performance on a number of tasks.

2 NEAT Overview

As mentioned above, early TWEANN models often suffered from information loss during the crossover phase at the end of each generation due to what Stanley and Miikulainen term the competing conventions problem. When two genomes contain hidden nodes that encode the same information, if their hidden nodes are stored in different orders their offspring will invariably lose the capabilities of at least one of the hidden nodes as shown in figure 1.

NEAT addresses this problem by assigning each connection within the network an innovation number or ID based on when the connection first appears. Each time a connection is added during the offspring creation phase at the end of a generation it is compared with other connections that have appeared in the same generation. If a matching connection is found then the connection is assigned the same innovation number as the earlier connection, otherwise it is given a new, unique number. In future generations when performing crossover, connections are aligned according to their innovation numbers which helps prevent loss due to competing conventions.

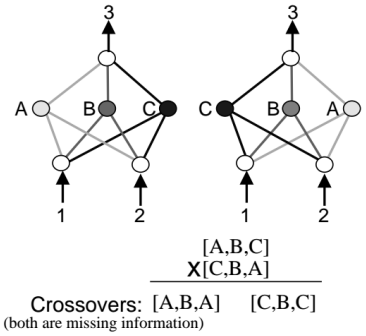


Figure 1: Information loss due to crossover.

Tracking connections in this way also helps solve another problem in TWEANN's, the penalizing of new developments. In order to encourage the development of simple networks many TWEANN algorithms add a penalty term to fitness calculations based on the number of nodes in a network. Although this will in theory keep networks small, it can have some unintended side effects. Often when a new node is added to a network it takes some number of generations before the node develops any useful function and contributes to the genome's fitness. If a genome with a new node is overly penalized just for having an extra node it might be eliminated from the population before it has a chance to develop thus preventing what could potentially be a useful new topology. NEAT solves this problem by organizing genomes into species based on how similar each genome is to each other. Although NEAT is not the first algorithm to speciate genomes, the tracking of innovation numbers makes calculating the distance between genomes trivial. Again connection genes within each genome are simply aligned according to their innovation numbers and the number of differing genes are counted as

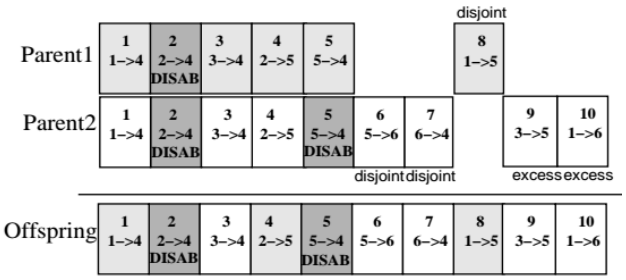


Figure 2: Aligning genomes by innovation number simplifies the counting of differing genes.

shown in figure 2. Then the distance δ between the two genomes is calculated by

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

where E is the number of excess genes (genes in one genome that have higher IDs than the maximum ID in the other genome), D is the number of disjoint genes, $\overline{W}$ is the average weight difference between matching genes, and c_1, c_2, and c_3 are user-defined constants. Genomes with δ smaller than some value are put into the same species and compete only within that species. If a species does not improve its fitness for a number of generations then it is eliminated. By protecting new innovations in this way, new topologies are given time to develop but are eventually trimmed out if they fail to produce positive results.

3 Experiments

Although genetic algorithms can be used to do supervised learning, their performance often lags behind more mathematically rigorous methods such as simple back-propagation. After ensuring that my implementation of NEAT worked correctly, I tested my model on several classic datasets such as the iris and diabetes datasets but I was very unimpressed by the results. Although the final accuracies on these problems was comparable to standard neural networks, NEAT was much slower to train and very sensitive to overfit. I eventually moved on to reinforcement learning tasks because as detailed in [2] genetic algorithms excel in situations where gradients are difficult to calculate and standard gradient-descent algorithms break down. As the reinforcement learning tasks better showcase the strengths of this model this section will primarily deal with those experiments. NOTE: In all network diagrams green nodes are inputs, blue nodes are outputs, black edges represent forward connections, red edges represent recurrent connections, and dashed lines represent disabled connections that have no effect.

3.1 XOR Verification

I used the XOR task to ensure that my NEAT implementation worked properly as Stanley and Miikkulainen explicitly mention it in their paper as a good test problem. For this experiment I initialized a population of 150 genomes to each have 2 input nodes and 1 output node. Each genome received a fitness calculated by the formula

$$fitness = (4 - \sum_n |t_n - z_n|)^2 \tag{2}$$

where t_n is the desired output for input n and z_n is the actual output for each of the 4 possible inputs. After I discovered and fixed several bugs the system performed quite well on this task. In no run did it fail to find a solution and over 10 runs the final network had an average of 2.6 hidden nodes where 1 is optimal. Although it did not find the optimal solution in every run it did develop a network with 1 hidden node in $\approx 20\%$ of runs.

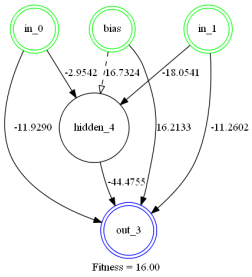
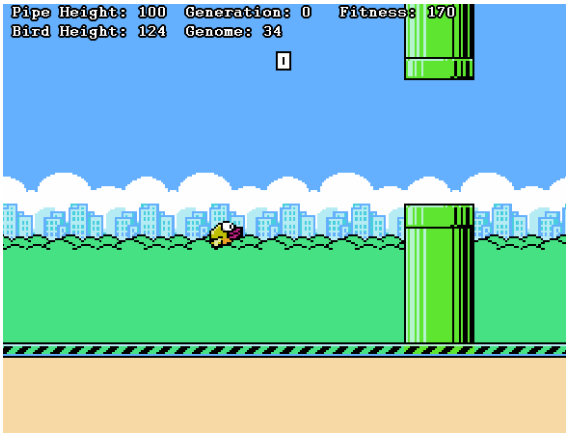


Figure 3: An example XOR solution

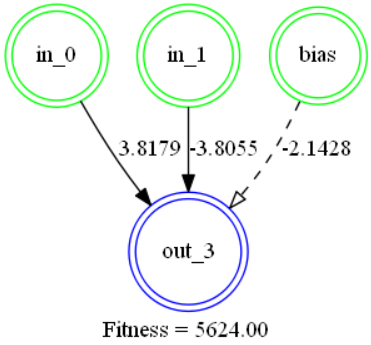
3.2 Flappy Bird

After I was satisfied that my system was working correctly, I created a reinforcement learning environment by hooking my NEAT implementation up to the Bizhawk Nintendo Entertainment System emulator and had it learn to play the game Flappy Bird. In Flappy Bird the player is tasked with guiding a bird through a series of vertical gaps by having the bird flap its wings to gain height. This sounds simple enough but it turns out to be a very difficult game for a human player due to the unforgiving collision detection and narrow gaps.

Unfortunately, Bizhawk can only interact with Lua scripts whereas I implemented NEAT in Python. To communicate between these platforms I created a server that runs NEAT in a Python environment and



(a) A frame from the Flappy Bird simulation



(b) An example solution network

Figure 4: Flappy Bird reinforcement learning task

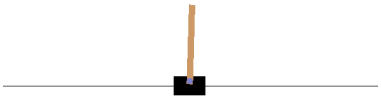
receives commands over a socket connection. The Lua client simply sends commands to the server to perform tasks like ending a generation, getting a genome's output, or assigning a fitness score to a genome. Although this adds some undesired overhead, this setup was more than fast enough to handle all the commands necessary to run the emulator at full speed and quickly iterate through generations.

In order to get any meaningful information out of the emulator it is necessary to read values out of the game's RAM which I had not had previous experience with. Luckily I was able to determine how to get the current height of the bird and the height of the next upcoming gap so these two values served as the inputs to the system. Each genome had to take these two inputs and then return whether the bird should flap its wings or stay idle. Genomes received a fitness based on how many frames they survived plus a bonus for each gap they successfully navigated.

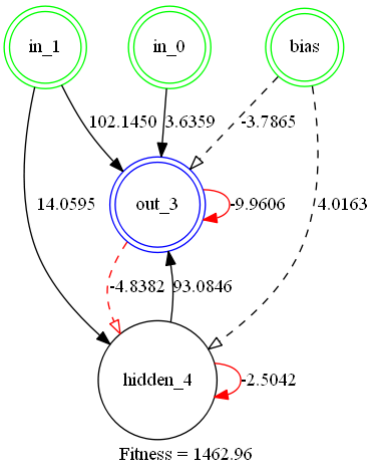
Using only these two inputs NEAT created an agent that could play the game indefinitely in an average of 11.3 generations. As shown in figure 4b the networks that evolved were not complicated but they performed very well. While watching the agents play the game I noticed that a large number of the agents in early generations would press the flap button at every possible opportunity, quickly flying off the top of the screen. This prevents them from crashing into the ground immediately but also prevents them from ever clearing one of the vertical gaps. To discourage this behavior I eventually added a penalty term to the fitness calculation for every flap taken. This penalty helped focus the efforts of the genomes and decreased the average number of generations required to 4.6.

3.3 Cart Pole (Pole Balacning)

Since Flappy Bird ended up being a little on the easy side, I moved on to what I hoped would be a more difficult reinforcement learning task. As described in [3], the pole balancing problem is a classic control task in which systems must balance an inverted pendulum on a cart by pushing the cart left or right. The original formulation of the problem provides the agent with 4 observations at each time step: the position of the cart, the velocity of the cart, the angle of the pole, and the velocity of the tip of the pole.



(a) Example frame of the cart pole simulation.



(b) An example solution.

Figure 5: The pole balancing problem with withheld velocity

At each step of the simulation the agent is provided with the 4 observations and must return its chosen action, to push the cart left or to push the cart right. In the Python OpenAI Gym [4] the simulation ends when the pole angle is $> 20.2°$ from vertical or the cart is > 2.4 units from the center of the track. For each frame the agent survives, its fitness increases by one point. The problem is considered solved when an agent receives > 475 fitness averaged over 100 runs.

Using all 4 observations, NEAT solved the problem using the initial population in $\approx 80\%$ of runs, meaning that the system never had to change a single genome in order to achieve the required fitness. This is not particularly interesting so I decided to increase the challenge of the problem by only providing each network with the positions of the cart and pole, withholding the velocity. Under these conditions the system must evolve recurrent connections in addition to the normal forward connections in order to recover the missing information. Even with the missing information NEAT handled the problem very nicely. Over 10 runs it solved the problem in an average of 109 generations. Figure 5b shows a particularly appealing solution with only one hidden node. Although it's difficult to determine exactly what each connection represents, the two active recurrent connections indicate that the system retains some history of previous positions in order to reconstruct the velocity.

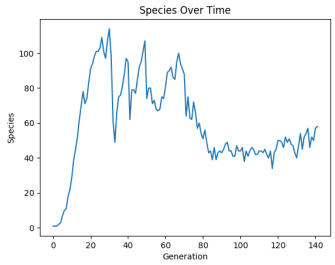
4 Modifications

In order to better understand the reasoning behind some design decisions in the original NEAT algorithm I made several modifications to the base system.

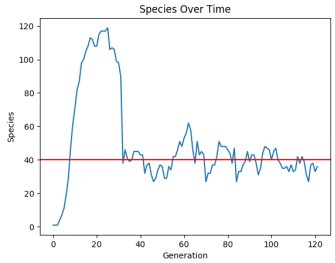
4.1 Dynamic Speciation Threshold

While experimenting with the NEAT algorithm I was often frustrated by how sensitive it was to the changes in its hyper parameters. Often it would take many runs to find parameters that allowed the system to find a solution and any small perturbations to these parameters would drastically influence run times. One particularly sensitive parameter was the distance threshold for δ that determines whether two genomes are in the same species or not. If this threshold was too low then the number of species would explode. If it was too high then there would only be one species and the system would not have enough diversity.

While searching for how other people dealt with this problem I came across [5], a website created by one of the original paper authors, that has some helpful information about how to improve stability



(a) Species over time with no dynamic threshold



(b) Species over time with dynamic threshold. The target number of species is marked with a red line

Figure 6: Effects of dynamic δ threshold

in NEAT. One item suggested making the δ threshold dynamic in order to maintain a target number of species. Making this addition proved very useful. With this change in place I could simply set the threshold to a reasonable value and let the system run without needing to repeatedly tweak the threshold to find one that led to a solution. It is worth mentioning that although figure 6a and 6b look similar, this is mostly due to the fact that figure 6a was generated only after testing many threshold values to find one that allows the system to find a solution whereas 6b was generated in one shot.

4.2 Innovation Tracking

During my initial reading of [1] I found it odd that innovation numbers were only shared among connections created during the current generation instead of among all connections in every network. It seemed to me that by always giving connections that connected the same nodes the same ID you could decrease the number of connections present in the final solution. To test this hypothesis I ran the pole balancing problem 5 times with the default ID sharing strategy and 5 times with the IDs shared across all connections regardless of generation. The results are summarized in table 1.

		Per Generation	Per Run
Generations	Mean	94.8	124.0
	SD	40.16	37.89
Average No. of Hidden Nodes	Mean	4.6	5.2
	SD	1.14	3.19
No. of Connections	Mean	15.8	13.8
	SD	5.11	6.41

Table 1: Effects of sharing IDs within each generation compared to each run.

Overall the change in ID tracking seemed to make the algorithm perform worse overall. On average it took significantly more generations to find a solution. Although on average it did decrease the number of connections present in the final solution, it also greatly increased the variability in how many hidden nodes were present in the solution, meaning that solutions were less consistent. I assume these decreases in performance are the result of unrelated connections sharing the same ID. Even though two connections might connect the same two nodes, the time at which they are introduced into the system is an important part of their identity. A connection between nodes A and B in generation 1 probably has a different meaning than a connection between the same nodes added in generation 100 as the rest of the network has changed. Forcing these connections to share the same ID essentially reintroduces the competing conventions problem which NEAT tries to avoid in the first place.

5 Conclusion

NEAT provides several useful extensions to both the basic genetic algorithm and TWEANN algorithms. By evolving networks starting from a minimal solution, NEAT frees the user from having to make any decisions about the topology of the network before execution begins. In addition, because it starts from minimal networks, NEAT tends to create minimal solutions which can be desirable in many contexts. Although, like many genetic algorithms, NEAT is not as ideal as standard gradient-descent algorithms for normal supervised learning tasks, it excels in reinforcement learning environments.

This model does suffer from a high sensitivity to choice of hyper parameters but this sensitivity can be somewhat lessened by making some parameters dynamic. These changes make the algorithm even easier to use as the user can more fully trust the algorithm not to explode.

If I were to continue working with this model I would like to explore more ways to decrease the importance of hyper parameters and increase stability of the algorithm. I would also like to further optimize NEAT for use in supervised learning by experimenting with ways to reduce its tendency to overfit, possibly using dropout or other regularization techniques. Using the Bizhawk framework I built up I would also like to have NEAT learn to play more complicated games to push it to its limit. It would also be interesting to run NEAT using the output of a CNN as input so that it could potentially utilize information from raw pixel values rather than pulling data straight out of RAM.

References

[1] Kenneth Stanley and Ristro Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[2] Darrell Whitley, Stephen Dominic, Rajarshi Das, and Charles W Anderson. Genetic Reinforcement Learning for Neurocontrol Problems. In *Genetic Algorithms for Machine Learning*, pages 103–128. Springer, 1993.

[3] Charles W Anderson. Learning to Control an Inverted Pendulum Using Neural Networks. *IEEE Control Systems Magazine*, 9(3):31–37, 1989.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[5] Kenneth Stanley. The NeuroEvolution of Augmenting Topologies (NEAT) Users Page. `https://www.cs.ucf.edu/~kstanley/neat.html`. Accessed: 2017-04-01.