NeuroEvolution of Augmenting Topologies

Nathan Zabriskie

April 17, 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 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 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.

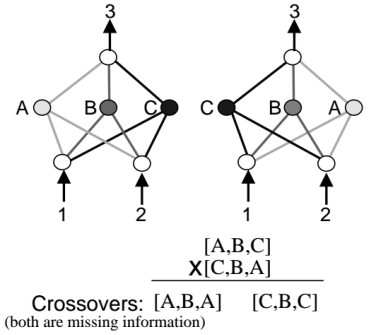


Figure 1: Information loss due to crossover.

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' 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 development.

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. 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 developments 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, often their performance 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

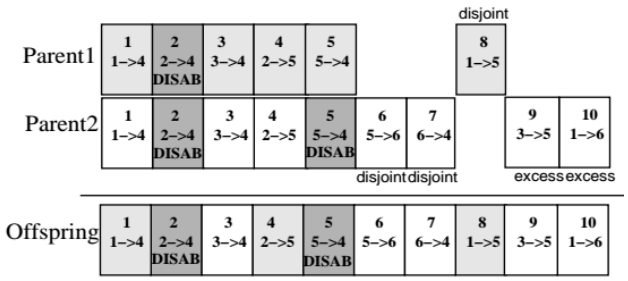
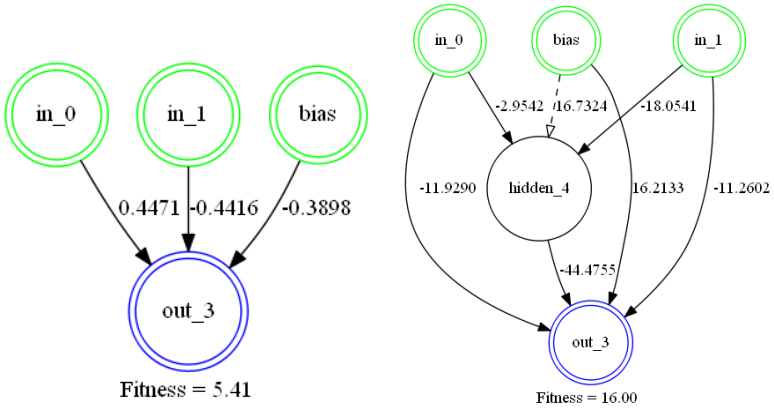


Figure 2: Aligning genomes by innovation number simplifies the counting of differing genes.

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.



(a) Initial network layout (b) Final network layout

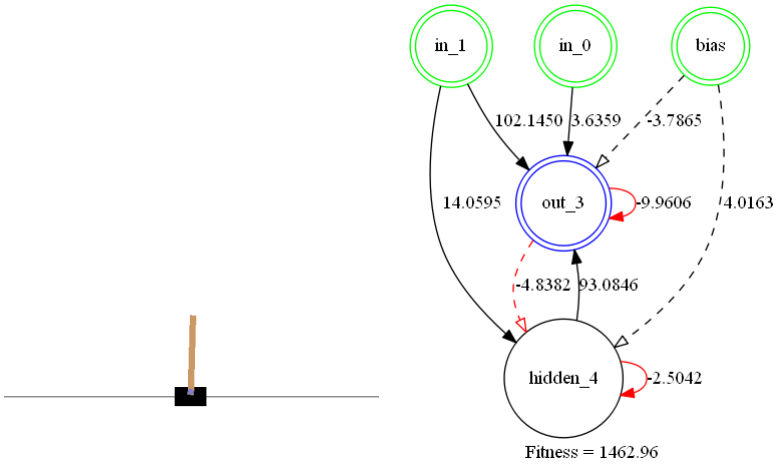
Figure 3: An example XOR solution

3.2 Pole Balancing

After I was satisfied that the system was working correctly I moved onto some more interesting reinforcement learning tasks. 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.

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 deg 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 every run in an average of 109 generations. Figure 4b shows a particularly appealing solution with only one hidden node. Although it's



(a) Example frame of the cart pole simulation.

(b) An example solution.

Figure 4: The pole balancing problem with withheld velocity

3.3 Flappy Bird

4 Modifications

4.1 Speciation

4.2 Innovation Tracking

5 Conclusion

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.