Evolutionary Dynamics of Neuroevolution of Augmenting Topologies (NEAT)

APPPHYS 237 Final Project

Ellen Xu

June 10, 2024

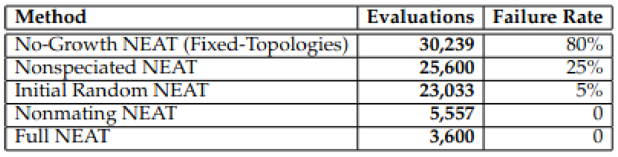
1 Introduction

Evolution is nature's best optimization scheme. Can evolution also be used to optimize artificial systems? Neuroevolution seeks to use evolutionary algorithms to generate neural networks to solve complex tasks. Similar to biological evolution, the classical evolutionary algorithm setup involves the following steps: Population pool Compete/evaluate fitness Selection Mutation Produce offspring

Traditional NE is fixed-topology, i.e. a topology is chosen beforehand by a human experimenter and only network weights are evolved. In their paper, Stanley and Miikulainen (2002)[1] proposed a method of evolving neural network topologies along with weights, resulting in faster convergence and greater success rate on the double pole balance task (a complex RL task). They introduced a few key insights which made evolving topologies possible: 1) homologous recombination for crossover of different topologies, 2) protecting structural innovation using speciation, and 3) "complexifying", or evolving from a simple, homogeneous initial population.

In each problem, we focus on a specific innovation from NEAT. Can we apply insights from evolutionary dynamics to understand neural network evolution?

[1]"Evolving NN's through Augmenting Topologies": https://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf



Method	Evaluations	Failure Rate
No-Growth NEAT (Fixed-Topologies)	**30,239**	80%
Nonspeciated NEAT	**25,600**	25%
Initial Random NEAT	**23,033**	5%
Nonmating NEAT	**5,557**	0
Full NEAT	**3,600**	0

Figure 1: NEAT ablations summary for the double pole balancing with velocities task. Evaluations refers to the average number of evaluations in order to reach a solution (proxy for time to convergence).

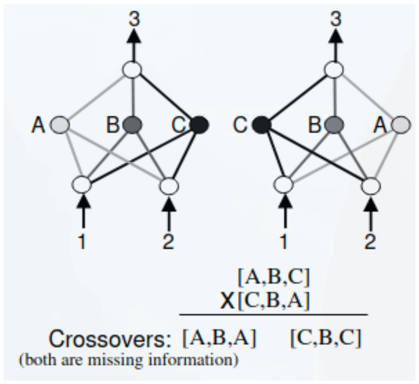


Figure 2: The competing conventions problem. After swapping, the resulting offspring would both miss 1/3 genes.

2 Some Very NEAT Problems

2.1 Homologous recombination

We first examine one of the key difficulties in evolving neural network topologies. When we have two different encodings (chromosomes) which are functionally equivalent, a recombination event can lead to loss of functionality (LOF). This is called the *competing conventions* or *permutations* problem. The example in Figure 2 shows a crossover event results in a loss of a gene.

Consider the simple example of a fixed-topology network. We use a graph encoding G=(V,E), where V is a list of node genes and E is a list of connection tuples containing attributes such as (inode, onode, weight). In the following examples, we simplify our calculations by only considering node encodings/permutations.

Problem 1. *For a NN with a single hidden layer of n units, how many functionally equivalent encodings are there? What about k hidden layers with units n1,...,nk? If we had taken into account weight permutations, would our answer change?*

Proof. With n hidden units we have n! permutations. For k hidden layers, each can permute independently, so we have $\Pi_i n_i!$ permutations. Our answer would not stay the same since we have more permutations if we consider weights. Even with k=1 single layer, the same functionally equivalent networks have different weights matrices $W_{in,hidden}$ and W_{hidden_out}, since the system of transformations is underparameterized. For example, with input=2, n=2, output=1:

$$\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$$

which has 6 unknowns and only 2 linear equations. So we can rewrite our original weights matrices $A \times B$ with different weights $C \times D$ without changing the underlying system of equations.

□

Note that in this simplified example we aren't even considering all the possible weight symmetries or topologies of different sizes which compute the same function[2]. The challenge of finding symmetries with connectivity and weights has been called "Holy Grail of this area" (Radcliffe 1993). We will later discover another method of aligning genomes without needing to sift through all the possible equivalences among permutations.

To get a feel for how much of an issue loss of information during crossover is, we'll consider the simplest type of crossover: the single-point crossover[3]. A crossover point is a point between two genes (not counting the ends), e.g. for above example A—BC or AB—C. Randomly choose a crossover point and exchange all segments after this point. Call an offspring *strange* if it is missing one or more genes.

Problem 2. *Assume we have two parent chromosomes, each with n genes, and all permutations of length n are equally likely. What is the probability that two randomly chosen networks will produce a "strange" offspring?*

Proof. There are n! total permutations and n-1 crossover points. Each time we recombine we get two possible offsprings, but since they are complementary, either both or neither are strange, so we can ignore the factor of two. Conditioning on the crossover split k,

$$P(\text{strange}) = 1 - P(\text{normal}) = \sum_{k=1}^{n-1} P(\text{split at k})P(\text{normal}|\text{split at k}) = \frac{1}{n-1}\sum_{k=1}^{n-1}\frac{k!(n-k)!}{n!} = \frac{1}{n-1}\sum_{k=1}^{n-1}\frac{1}{\binom{n}{k}}$$

where the offspring is normal if the part of the genome before and after the split are the same for both parents, so we have $k!(n-k)!$ choices for how to rearrange the k elements before and n-k elements after.

We can check our answer by sampling and simulating the solution in code (Appendix). □

The essence of the problem is that with many random permutations, "strange" offspring are highly probable with single-point crossover. How can we recombine without losing functionality? Nature's solution uses *homology*, where two genes are homologous if they are alleles of the same trait. In other words, we keeps track of a mapping of corresponding traits which can be safely swapped ($A \leftrightarrow A$ and $C \leftrightarrow C$ in example above). Homologous recombination is the process behind chromosomal crossover: the exchange of homologous genes during sexual reproduction.

Problem 3. *What is the probability that two networks will share a homologous component?*

Proof. This is the same as T_{MRCA}, probability of sharing a common ancestor. coalsecence rate = 1/N, where N is the size of the population (number of networks) □

We already discovered by the competing conventions problem that it is difficult to tell network homology by their structures alone (since there are many possible permutations). Instead, Stanley's solution is to keep track of historical origin of each gene. Every time a new

[2]"Symmetries of Neural Networks": http://bactra.org/notebooks/symmetries-of-neural-networks.html

[3]For more kinds of crossover, e.g. k-point and uniform, see Wikipedia page: https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)

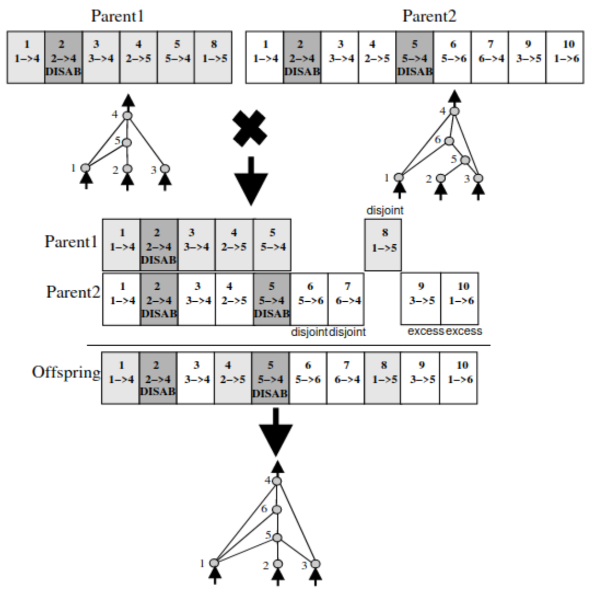


Figure 3: Homologous recombination by matching genes with innovation number

structural mutation occurs, we assign an *innovation number* sequentially, and during mating these respective innovation numbers are inherited and preserved. Then during crossover, we match up the innovation numbers for "matching" genes to ensure none are replaced, and exchange "disjoint" genes separately.

Homologous recombination solves the competing conventions problem and allows us to introduce mating to NEAT. How does this perform in practice? We saw in class that sexual reproduction promotes genetic variation and adaptation. In their ablation study, Stanley also showed that allowing mating/sexual reproduction converges faster compared to asexual, leading to $\approx 35\%$ reduction in number of evaluations needed (Figure 1).

2.2 Speciation

Another challenge is protecting topological innovations. Introducing a new structure often causes a loss in fitness, e.g. adding a new activation changes the function, so it needs time for weights to adapt and converge to a new optimal. New innovations, even when potentially beneficial, are initially selected against and are likely to die out.

In nature, *speciation* allows different structures present in different species to compete in their own niche, instead of the entire population. The idea is to group individuals into separate species based on genetic similarity. Define genetic similarity in terms of two individuals' *compatibility distance* where E and D are excess and disjoint genes (non-matching), c are weighting constants, N is genome length, and $\bar{W}$ is average weight difference (matching),

$$\delta = \frac{c_1 E + c_2 D}{N} + c_3 \bar{W}$$

Let's incorporate the idea of speciation and explicit fitness sharing by modifying our evolutionary model of barcode lineages. Each barcode will represent a different species with barcode fitness S_i, individual fitnesses s, total population N, species size $n_l = \sum_{j=1}^{N} 1(\delta(i,j) < \delta_t)$, with adjusted fitness

$$f_i' = \frac{f_i}{n_l}$$

which is normalized by the number of organisms in the same species n_l. Every species is assigned a number of offspring proportional to sum of adjusted fitnesses, $X_l = \sum_i f_i' = \frac{1}{n_l}\sum_i f_i$, and the entire population is replaced by the offspring each generation.

Problem 4. *Simulate with and without fitness sharing. In NEAT, they used parameters* $c_1 = c_2 = 1.0, c_3 = 0.4, \delta_t = 3.0$.

2.3 Complexifying

Our final goal is to reduce dimensionality of the search space to find a solution as efficiently as possible. In NEAT, we start with a minimal search space by initializing the population as the simplest possible model with 0 hidden nodes, instead of initializing to random individuals. This parallels incremental adaptation in evolution, where we start off from a minimal cell and evolve.

We'll now examine why starting off minimally is beneficial, e.g. evolving from no hidden layers, formulated as a branching process. A branching process describes how our population explores the solution space over many generations, where each branching is a potential network topology.

Problem 5. *If we didn't start off from the simplest genome, but say, initialized at random individuals from generation t, how many potential mutations do we lose?*

Proof. Let $\mu = E[N]$ be our linear branching factor. Then the expected number of offspring at generation t is $E[N_t] = \mu^t$. We can calculate the total number of potential mutants over k generations if we had initialized at generation t:

$$L(t,k) = \sum_{i=t+1}^{t+k} \mu^i = \mu^{t+1}\frac{\mu^k - 1}{\mu - 1}$$

□

By committing to a particular configuration too early, we lose potential for further branching and exploration. In addition, we saw that jackpot mutations are rare but beneficial mutations that occur early on, which can dramatically improve the fitness of a population.

Problem 6. *Consider the number of jackpot mutations as a function of the number of branches we explored. How would losing jackpots effect the average fitness of the population?*

Proof. □

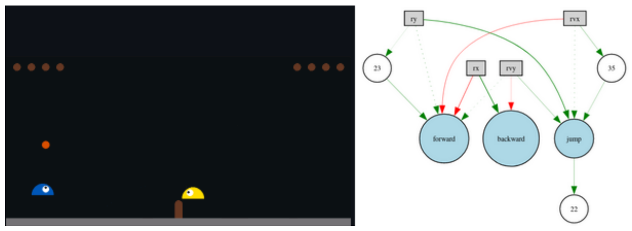


Figure 4: Evolved agent/example topology using NEAT to play neural slime volley

3 Bonus: Running NEAT

Now that we've understood the key innovations and evolutionary theories behind NEAT, let's explore NEAT to evolve and solve a classic RL problem.

Problem 7. *Choose an RL problem on Gym*[4]*. Using NEAT-Python*[5] *or PrettyNEAT*[6]*, modify config.yml to experiment with parameters. Once you have evolved an initial agent, play around with trying to converge to a solution faster or adding your own innovations!*

Proof. NEAT for evolving an agent to play neural slimevolley[7]. I used a self-play ("learning from experience"), multi-agent tournament setup, feature engineering, and reward annealing to speed up convergence. Parameters I used:

`n_inputs: 6, max steps: 1000, n_rounds: 50, max n_gens: 500, pop size: 15`

Within 150 generations we have a solution which beats the in-built agent!

□

[4]https://www.gymlibrary.dev/index.html
[5]https://github.com/CodeReclaimers/neat-python
[6]https://github.com/google/brain-tokyo-workshop/tree/master/WANNRelease/prettyNEAT
[7]https://github.com/ellenjxu/neuralslimevolley

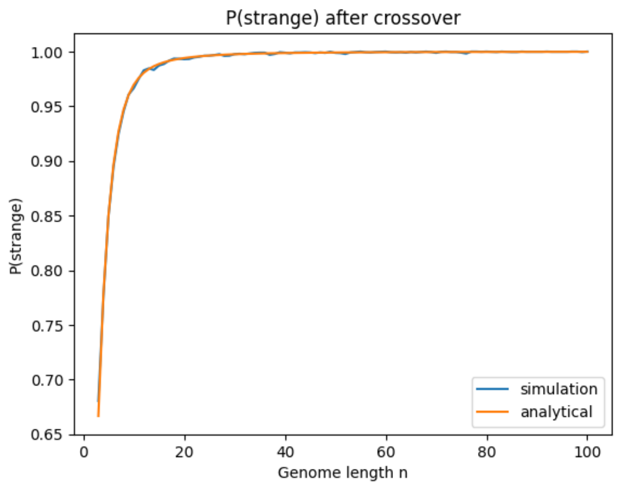
4 Appendix

4.1 Problem 3 simulation code

```python
# simulation
def simulate(n, num_rounds=5000):
    """
    Simulate with genome length n, pick random split, then swap all segments after the split
    Estimate probability that the offspring is missing a segment.
    """
    num_strange = 0
    for _ in range(num_rounds):
        genome1, genome2 = np.random.permutation(n), np.random.permutation(n)
        split = np.random.randint(1, n)  # pick random split from 1 to n-1
        offspring1 = np.concatenate([genome1[:split], genome2[split:]]) # swap
        n_genes = len(set(offspring1))
        if n_genes != n:
            num_strange += 1
    return num_strange / num_rounds

# analytical
def p_strange(n):
    """1 - sum_k 1/(n-1) (comb(n, k)) """
    return 1 - sum(1 / ((n - 1) * math.comb(n, k)) for k in range(1, n))

sim_probs = [simulate(n) for n in range(3, 101)]
act_probs = [p_strange(n) for n in n_values]
```


P(strange) after crossover
P(strange)
Genome length n
simulation
analytical