Using NeuroEvolution of Augmenting Topologies (NEAT) Genetic Algorithm to beat the Snake Game.

Marc Fernández Martínez

July 2024

CHAPTER 1

Genetic Algorithms

SECTION 1 Introduction

In the decades of the 50s and 60s, several computer scientists studied independently from each other the concept of using evolutionary systems as an optimization tool for engineering problems. The basic idea behind these systems was to *evolve* a population of candidate solutions to a given problem, using operators inspired by Nature's **genetic variation** and **natural selection**.

These evolution strategies, used together with programming, gave the name to the field of ***evolutionary programming***.

Genetic Algorithms (GAs) [1] are a class of optimization algorithms inspired by the process of natural selection. They belong to this larger class of evolutionary algorithms (EAs) and were originally designed to formally study the phenomenon of adaptation and develop ways to translate the mechanisms of natural selection into computer systems.

However, over the course of the years, the boundaries between GAs, EAs and evolutionary programming have broken down. Today, researchers and computer scientists often use the term *genetic algorithm* to refer to the algorithms that are able to find approximate solutions to complex problems through iterative improvements based on the principles of genetics and natural selection.

SECTION 2 Biological terminology

Since genetic algorithms are based on how real genetics works in nature, it also adopts some of the terminology to describe the solutions, parameters and operators. Here are some of these terms:

- A **population** refers to a set of candidate solutions.
- The **chromosomes**, or **genome**, typically refers to a candidate solution to a problem, often represented as strings of binary, real numbers, or other data types.
- The **fitness function** evaluates and assigns a *fitness score* to each chromosome, indicating how good the solution is. Just as in nature, a fitter individual has better chances at survival.

- Speaking of survival, the process of **selection** involves choosing the fittest individuals from the population to pass their genes to the next generation.
- The two main genetic operators used in this algorithms are:
    1. **Crossover**, in which two parents are chosen at random and their chromosomes (genomes) are combined by parts to produce an offspring.
    2. **Mutation** on the other hand, introduces random changes to an offspring's chromosome to maintain genetic diversity within the population.

Over successive generations, the population evolves towards better solutions.

SECTION 3 Implementation

The general structure of a GA involves the following steps:

1. **Initialization:** The first step is to generate an initial population of chromosomes randomly. In order to do this, we consider an initial structure of an artificial neural network.

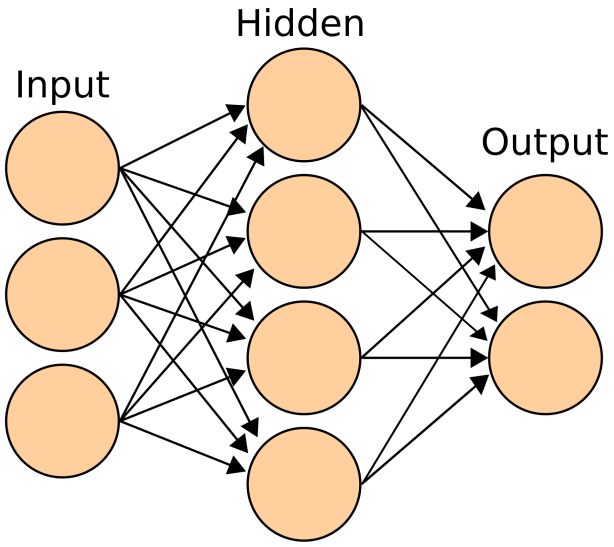


Figure 1.1: An artificial neural network [3].

   The values of the weights in between layers can be thought of as our genes, as well as any additional bias that we introduce. Together, these parameters form the chromosome, or genome, of a certain candidate.

2. **Evaluation:** Next, we compute the fitness of each individual in the population with some particular function that usually reflects how good was the output of a candidate given the input.

3. **Selection:** Select individuals for reproduction based on their fitness. For instance, a roulette wheel selection, where fitter candidates have better odds at survival, or a tournament selection with some survival threshold.

4. **Crossover:** Create offspring by combining pairs of selected individuals.
5. **Mutation:** Apply random changes to the offspring with some probability to introduce diversification.
6. **Replacement:** Form a new population, either replacing the least fit individuals with new offspring, or directly removing the whole previous generation.
7. **Termination:** Repeat steps 2. to 6. until a stopping criterion is met. Namely, a solution with an acceptable fitness is found, or a maximum number of generations is reached.

Below you can find a pseudo-code implementation of this algorithm.

Algorithm 1 Genetic Algorithm

Input: population, threshold, max_generations
Outputs: survivors

```
population ← random initialization
fitness ← EVALUATE(population)
survivors ← population
generations ← 0
while fitness < threshold or generations < max_generations do
    parents ← RANDOMSELECTION(survivors)
    offspring ← CROSSOVER(parents)
    offspring ← MUTATION(offspring)
    fitness ← EVALUATE(offspring)
    survivors ← RANDOMSELECTION(parents + offspring)    ▷ Or other selection criteria
    generation ← generation + 1
```

CHAPTER 2

NEAT

SECTION 1 Introduction

NeuroEvolution of Augmenting Topologies (NEAT) is an advanced form of genetic algorithm specifically designed for evolving artificial neural networks (ANNs). Developed by Kenneth O. Stanley [2], NEAT is able to simultaneously evolve the topology and the weights of the networks, therefore allowing the structure of the networks to becomes increasingly complex over generations.

A major question that arises with this algorithm is how does this poses an advantage over standard genetic algorithms. On one hand, it could overcomplicate the search, but it could also provide the network with the right amount of hidden neurons and connections for a particular problem automatically.

Other major challenges for this kind of evolving-topology algorithms are:

- How to represent different topologies in terms of genes such that we can perform the crossovers easily?
- How can a topological innovation survive the few generations it needs to optimize its parameters?
- Is there a way to minimize the complexity of the topology without adding a corresponding term explicitly in the fitness function?

We will see in the following sections how the NEAT algorithm tackles these problems.

SECTION 2 Genetic Encoding

In NEAT, each genome (also known as chromosomes) includes a list of *connection genes*, which refer to the *node genes* being connected. Namely, the connection gene specifies the in-node, out-node, weight of the connection, its activation state (ON/OFF) and the *innovation number*, a sort of identification which allows us to find corresponding genes during crossover.

For example, below is an example implementation in `C++` of the node and connecting gene structs.

```
struct NeuronGene {
    int neuron_id;
    double bias;
    Activation activation;
}

struct LinkId {
    int input_id;
    int output_id;
}
```

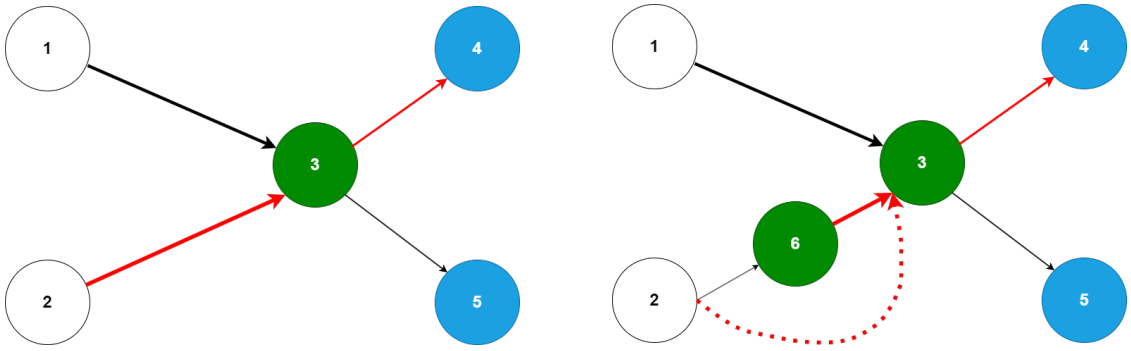
```
struct ConnectionGene {
    LinkId link_id;
    double weight;
    bool is_enabled;
}

struct Genome {
    int genome_id;
    int num_inputs;
    int num_outputs;
    std::vector<NeuronGene> neurons;
    std::vector<ConnectionGene> links;
}
```

By keeping track of a global innovation number, whenever a new gene appears, we assign that number as the new gene's id. This is called a **historical marking**, and it allows us to know exactly which genes match up with which.

SECTION 3 Crossover and mutation

Mutation in NEAT can change both the connection weights and the structure of the network. The latter can occur in two ways:

1. It can **add a connection** by creating a connection gene between to previously unconnected nodes.
2. It may also **add a node** to an existing connection. In this case, the connection is split and the new node placed where the old connection used to be, disabling it in the process. The new connection weight is set to 1, so that the overall behaviour of this connection should be the same, despite the different topology.



(a) Topology of the genome. Input nodes are white, hidden are green and output are blue.

(b) A new node is added on the link. New link has weight 1.

Figure 2.1: Split link mutation diagram. Weights represented by the thickness, color indicates sign, and dotted lines are disabled links.

The method of crossover in NEAT is very simple. Given two structures, they can always be combined in an ordered manner without any topological analysis, that is, the problem is recast as a problem of historical matching of the id's.

When crossing over, the genes in both genomes with the same innovation number are lined up and chosen at random. Genes that do not match are inherited from the more fit parent.

SECTION 4 **Protecting innovation**

Regarding the second challenge, adding new structure nodes to a network usually reduces fitness (initially). However, in NEAT, we speciate the population, meaning that individuals compete within their species (individuals with similar topologies) instead of the whole population. This way, topological innovations are protected at first, and have time to optimize their structure before having to compete for survival.

We can measure the compatibility distance δ of different structures as a linear combination of the number of excess E, disjoint D genes as well as the average weight differences $\bar{W}$:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \bar{W}. \tag{2.1}$$

This equation allows us to speciate using a compatibility threshold δ_t. Each genome is tested one at a time, and if its distance with a particular species is less than this threshold, it is placed into that species (recall the genome_id property of the Genome struct).

Species grow or shrink depending on whether the adjusted fitness of the species is above or below the population average, as:

$$N'_j = \frac{\sum_{i=1}^{N_j} f_{ij}}{\bar{f}}, \tag{2.2}$$

where N_j and N'_j are the old and new number of individuals in species j. The best performing individuals of each species are randomly mated to generate N'_j offspring, replacing their species population entirely.

SECTION 5 **Minimizing dimensionality**

In contrast to other genetic algorithms, NEAT begins with a uniform population of networks with **no hidden nodes**. Because NEAT protects innovation through speciation, it can start this way and grow new structures only when necessary. The structural mutations give rise to new structures, that only survive if they are found to be useful through fitness evaluations.

In conclusion, NEAT searches through a minimal number of weight dimensions, which requires a significantly reduced number of generations to find a solution.

Bibliography

[1] Melanie Mitchell. *Introduction to genetic algorithms*. MIT Press, 1998.

[2] Kenneth O. Stanley and Risto Miikkulainen. Efficient evolution of neural network topologies. *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, May 2002.

[3] User:CBurnett. An artificial neural network, 2006. File: `Artificial neural network.svg`.