NEAT: An implementation of the NEAT algorithm in Java

Flo Raeymaekers

Q1 2023-2024

1 Introduction

NEAT, which stands for NeuroEvolution of Augmenting Topologies, is a genetic algorithm for the generation of artificial neural networks. It was developed by Kenneth O. Stanley and Risto Miikkulainen in 2002. The algorithm is based on three key techniques:

- Tracking genes with history markers to allow crossover among topologies
- Applying speciation to preserve innovations
- Developing topologies incrementally from simple initial structures

NEAT is also capable of co-evolving the connection weights of multiple neural networks simultaneously. The algorithm has been applied to many different reinforcement learning tasks, including the evolution of neural controllers for physical robots in simulation and reality, the development of neural network agents that play classic arcade games at human or superhuman levels, and the generation of complex artificial organisms called virtual pets that can be trained to perform tasks such as obstacle avoidance and pathfinding.

In this paper, though, I will only cover the implementation of a XOR gate using NEAT. This is a simple task, but it will allow me to explain the algorithm in a clear and concise way.

2 Implementation

2.1 The augmented topology

The first step in the NEAT algorithm is to have a way to represent the topology of a neural network. The typical neural network has an input layer, some hidden layers and an output layer, which every one of these are densely connected to one another.

This very simplistic topology is simple to implement as it is just a matter of creating a matrix of weights and biases. However, this topology is not very flexible: any addition or removal of neurons or connections would require a complete rework of the matrix.

As NEAT requires the ability to add and remove neurons and connections, a more flexible topology is required. This is where the augmented topology comes in. In my implementation, the augmented topology is represented by a Directed Acyclic Graph (DAG). This DAG is composed of nodes and edges. The nodes represent neurons, and the edges represent connections between neurons (which will be called Axons). The DAG is acyclic, meaning that there are no cycles in the graph. This is important as it prevents the network from looping back on itself, which would cause the network to never stop processing.

In Java, the JGraphT library can be used to represent a DAG. Its implementation handles cyclic prevention, so I don't have to worry about that. But most importantly, it allows me to sort the nodes in topological order. This is important as it allows me to process the network in the correct order.

The neurons and axons

```java
package be.floshie.neat.ai.graph;

import ...;

public class Neuron {
    private final int id;
    private final Function1<Double, Double> activationFunction;
}
```

Listing 1: The Neuron class

In the neuron class, there are two fields: the unique identifier of the neuron and the activation function.

```java
package be.floshie.neat.ai.graph;

import lombok.*;

public class Axon {
    private final int id;
    private final int innovation;

    private final boolean enabled;

    private double weight;
}
```

Listing 2: The Axon class

In the axon class, there are four fields: the unique identifier of the axon, the innovation number of the axon, the enabled state of the axon and the weight of the axon. The innovation number is used to track the history of the axon. This is important as it allows the algorithm to perform crossover between two different topologies.

The feed through process

JGraphT provides great tools to handle graphs and DAGs, but it is not a neural network library. So I had to implement the feed through process myself.

I created a class called `AdvancedNeuralNetwork` which decorates the DAG with the ability to perform the feed through process. This class is composed of two fields: the DAG and a map of neurons to their value. The map is used to store the value of the neurons after the feed through process. When the process is done, the map is cleared.

2.2 Genetics

In this section, I will cover the genetics of the NEAT algorithm. This includes the mutation, crossover, selection, fitness but also the speciation of the topologies.

The mutation

In this speceific implementation, there are four types of mutations:

The mutation of the weights goes through every axon and mutates the weight of the axon by adding a random value between a predefined range to the weight.

The mutation of the connectivity of the axons goes through every axon and mutates the enabled state of a random axon.

The addition of a neuron goes through every axon and mutates the axon by splitting it into two axons. The first axon will have the same weight as the original axon, but will have a weight of 1. The second axon will have a weight of 1 and will be connected to the first axon. The second axon will also have the same enabled state as the original axon.

The addition of an axon goes through every possible connection between neurons and mutates the connection by adding a new axon. The new axon will have a weight of 1 and will be enabled.

Each of these mutations have a chance of happening. This chance is defined in the configuration of the algorithm.

The common interface of these mutations is the `MutationStrategy` interface. This interface has a single method: `Individual mutate(Individual)`. This method takes a topology and mutates it. A new instance of individual is returned, meaning that the original topology is not mutated.

The crossover

This algorithm is based on the crossover described in the original paper. Each axon can be of three states: disjoint, excess or matching. In the crossover algorithm, we only care about whether the axon's innovation number matches the other parent or not.

If it does match, then the axon is inherited from a random parent. If it doesn't match, then the axon is inherited from the fittest parent.

Inside the code, this is implemented in the `CrossoverStrategy` interface. This interface has a single method: `Individual crossover(List<Individual>)`. The need of the whole population is due to the fact that the fitness depends on the whole population. We'll see that later.

The selection

The algorithm uses a tournament selection. This means that a random number of individuals are selected from the population. The fittest individual of this selection is then returned.

The fitness

The fitness of an individual is calculated by the XOR gate using the Mean Square Error (MSE) as the loss function. The fitness is then calculated by taking the inverse of the MSE. This means that the fittest individual has the highest fitness.

The fitness is then adapted with the speciation algorithm.

The speciation

The main purpose of the speciation algorithm is to prevent the population from converging to a single topology. This is done by calculating the distance between two topologies. If the distance is below a certain threshold, then the two topologies are considered to be in the same species.

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

The distance is calculated using the formula above. The c_1, c_2 and c_3 are constants that are defined in the configuration of the algorithm. The E is the number of excess genes, the D is the number of disjoint genes between the two topologies and the $\bar{W}$ is the average weight difference of matching genes.

After the distance between an individual and every other individual from the population is calculated, its fitness is adapted by the following formula:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} \delta_{(i,j)}} \tag{2}$$

3 Results

Sadly, I was not able to get the algorithm to work. I tried to debug the algorithm, but I was not able to find the issue.

Where the algorithm should have converged to a topology that is able to solve the XOR gate, it instead converged to a topology that is outputing the same value for every input, that is 0.5.

```
    [main] INFO be.floshie.neat.NeatMain - Generation: 234
    [main] INFO be.floshie.neat.NeatMain - Best fitness: 0.022857142857142857
    [main] INFO be.floshie.neat.NeatMain - Best individual: Individual@4f704591
    [main] INFO be.floshie.neat.NeatMain - Best individual performance:
        0, 0:    0.5
        0, 1:    0.5
        1, 0:    0.5
        1, 1:    0.5
```

Listing 3: The output of the logs

4 Conclusion

In conclusion, even though the project was itself a failure, I can say that I learned a lot from it. I've gone deeper into the NEAT algorithm and I've learned a lot about neural networks. I've also learned a lot about the Java language and the JGraphT library.

I am saddened by the fact that I was not able to get the algorithm to work, but I am still proud of what I've accomplished. I hope that someday I will be able to get the algorithm to work. This is only the beginning of my journey into the world of artificial intelligence.