PROJECT REPORT

NEAT Implementation

Projets 2

Master en Architecture des Systèmes Informatiques

Andrea Dal Molin

2023-2024

Contents

Introduction

This report will go through the making of the NEAT implemenation required for the course "Projects 2". The objective of this project is to build an implementation of NEAT (*NeuroEvolution of Augmenting Topologies*) from scratch. This implementation then had to be tested first with a simple XOR and then with the platformer game that we built during the first project. Sadly, the latter part could not be completed due to time constraints.

We will initially go through a general implementation overview, discuss the different approaches that were attempted. Following this, the report will describe how the final version of the implementation has been achieved. We later discuss the problems that have been encountered, and how they were tackled. Finally, we will mention the possible improvements that could be made to the project.

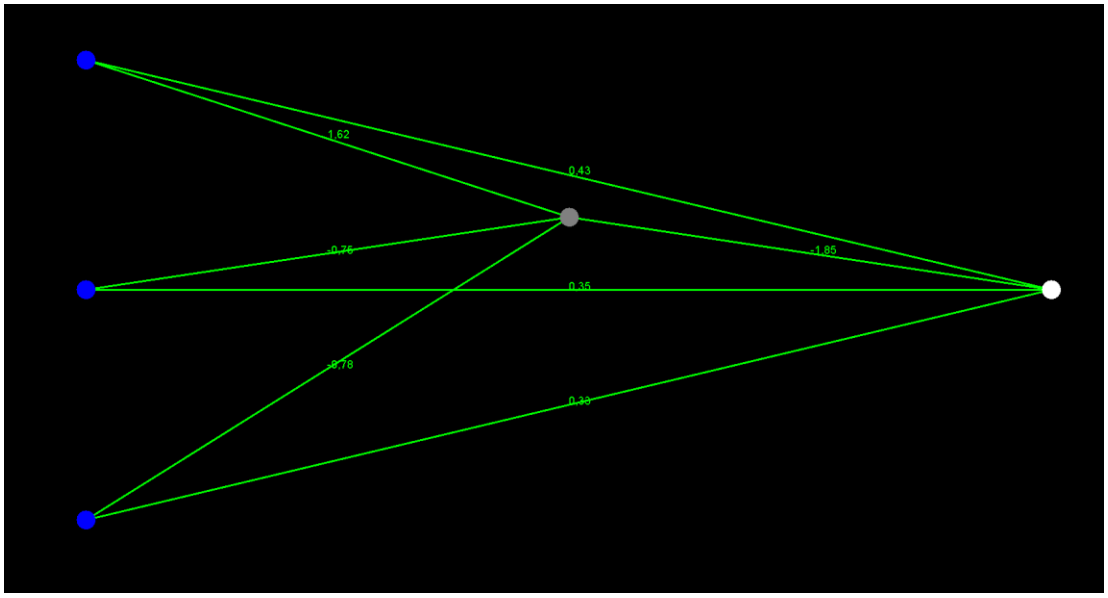


Figure 1 : An example of Genome created by the NEAT algorithm.

Implementation overview

The project is divided into packages that reflect the roles of each part of the application.

The `model` package is the most important one and the densest. It contains the whole logic of the implementation. It contains a sub-package, called `genes`. This one contains the lower-level part of the implementation, including the definition of ConnectionGene, NodeGene, Genome and other corollary classes. Next to the genes package, we can find the following classes: `Individual`, `Species`, `RandomSelector`, and the `Neat` class itself.

The `ui` package simply contains classes that are used to generate a visualization of the final result. It will allow us to see the Genome that has been built.

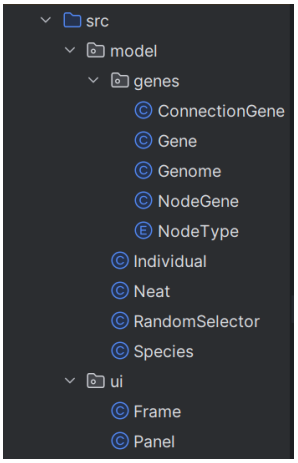


Figure 2 : The code structure of the project

Let's dive into some of the most important classes.

NodeGene

This class is responsible for managing the connections and output calculation of a node in the neural network.

```java
public class NodeGene extends Gene implements Comparable<NodeGene> {
    private double x, y;
    private double output;
    private NodeType type;

    private final List<ConnectionGene> connections = new ArrayList<>();

    public NodeGene(int innovationNumber) {
        super(innovationNumber);
    }
```

It contains an output value, a type (to differentiate between input, output and hidden nodes), and a list of connections related to that node.

Importantly, this class can calculate its own output value, based on it weight and a sigmoid function.

ConnectionGene

```java
public class ConnectionGene extends Gene {

    private final NodeGene from;
    private final NodeGene to;
    private double weight;
    private boolean enabled = true;
```

The `ConnectionGene` class simply represents a connection between two nodes. It therefore defines a start node and an end node. It also contains a weight value and can be toggled on or off.

Genome

This class, much more complex than the first two, is one of the most ones of the project. Here resides the whole logic of inter-node relationships, the mutations, crossover, distance functions and much more. It is structured as follows:

```java
public class Genome {

    private final Neat neat;
    private final Map<Integer, NodeGene> nodes = new TreeMap<>();
    private final Map<Integer, ConnectionGene> connections = new TreeMap<>();
```

A Neat object is present, which we will present later in its own class. Most importantly, it contains a list, or better said a `TreeMap` of nodes and connections. These are fundamental for the operations that the Genome is responsible for.

Individual

Going one step more abstract, we have the individuals. This class is very a "utilitarian" class. We could have easily done without it and directly use the Genomes in the Neat class, but for the sake of code simplicity and SRP good practices, it made sense to implement the `Individual` class. The existence of this class greatly simplifies the Neat class.

Furthermore, the Individual class introduces one of the important aspects of NEAT: speciation.

```
public class Individual {

    private Genome genome;
    private double score;
    private Species species;
```

As we can see, an individual is linked to a specific species.

Neat

Finally, the true heart of the implementation resides here, in the Neat class. It holds arguably too many responsibilities and manages the whole network in a relatively abstract manner (using as much as possible the methods present in the subclasses. It also contains a series of important parameters that are used in the whole project.

```
public class Neat {

    public static final double MUTATE_LINK_RATE = 0.3;
    public static final double MUTATE_NODE_RATE = 0.03;
    public static final double MUTATE_WEIGHT_SHIFT_RATE = 0.02;
    public static final double MUTATE_WEIGHT_RANDOM_RATE = 0.02;
    public static final double MUTATE_TOGGLE_RATE = 0.2;
    public static final double WEIGHT_SHIFT_STRENGTH = 0.3;
    public static final double WEIGHT_RANDOM_STRENGTH = 1;
    public static final double SURVIVAL_PERCENTAGE = 80;
    public static final double C1 = 1;
    public static final double C2 = 1;
    public static final double C3 = 0.4;
    public static final double CP = 4;

    private final HashMap<ConnectionGene, ConnectionGene> allConnections = new HashMap<>();
    private final List<NodeGene> allNodes = new ArrayList<>();
    private int inputSize;
    private int outputSize;
    private final ArrayList<Individual> individuals = new ArrayList<>();
    private final ArrayList<Species> species = new ArrayList<>();

    public Neat(int inputSize, int outputSize, int individuals) {
        this.initialize(inputSize, outputSize, individuals);
    }
```

This class is responsible for the evolution of the population, the breeding, pruning and speciation of the network.

Development

Development iterations

The development of this project has been a rollercoaster of interpretations. The first step was to read but most importantly *understand* the scientific publication published by Kenneth O. Stanley[1] about NEAT. What followed was a few attempts at an implementation. Initially, the main difficulty was to decide on the responsibility of each class, the management of recurrent connections, and the speciation.

Mutations have also been a point of contention. The scientific paper specifies a few types of mutations, but many resources over the Web give other interesting interpretations. Initially, only the mutations included in the paper were included, but I later decided to implement a few more.

Final implementation

The final version of the implementation was the result of merging the information from the scientific paper, with some other resources found around the Web, which served as inspiration for this project.

Results

As said in the introduction, only the XOR test has been implemented.

The main function of the Neat class demonstrates a test of the Neat network done with a XOR fitness function.

[1] https://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

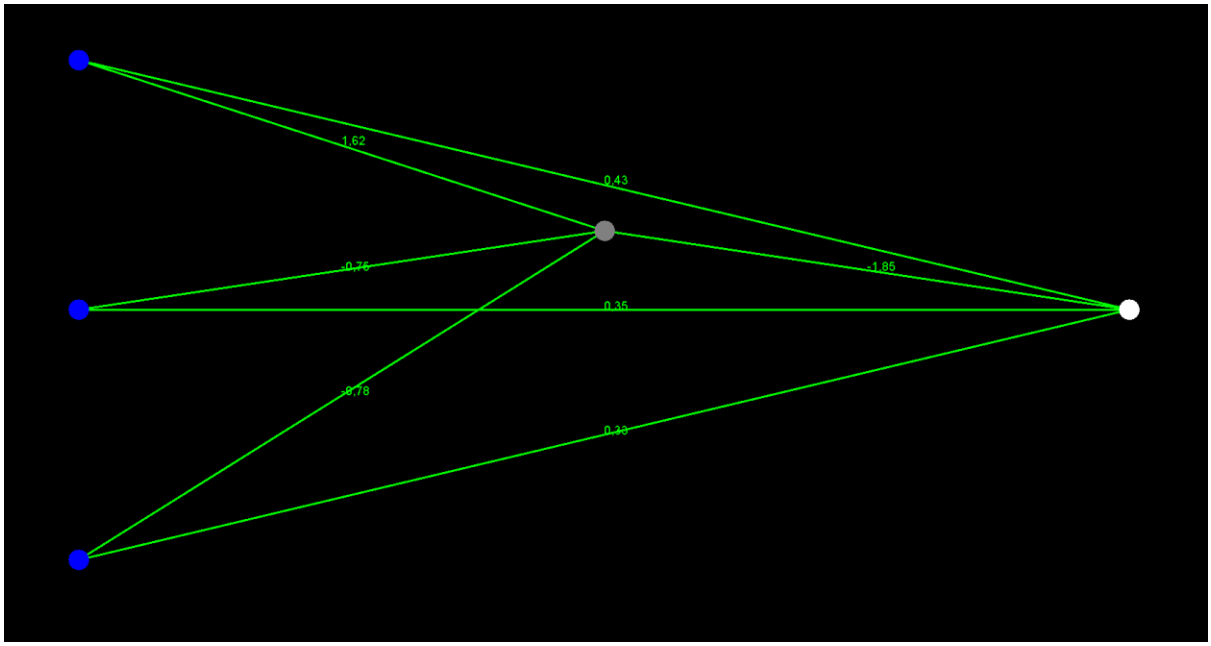
```java
Neat neat = new Neat(3, 1, 250);

double[][] inputs = {{0, 0, 1}, {0, 1, 1}, {1, 0, 1}, {1, 1, 1}};
double[] expectedOutputs = {0, 1, 1, 0};

double fitnessThreshold = 3.9;
double bestFitness = 0;
int generation = 0;

while (bestFitness < fitnessThreshold) {
    bestFitness = 0;
    for (Individual individual : neat.individuals) {
        double fitness = 0;
        for (int i = 0; i < inputs.length; i++) {
            double[] output = individual.calculateOutput(inputs[i]);
            double error = Math.abs(expectedOutputs[i] - output[0]);
            fitness += 1 - error;
        }
        individual.setScore(fitness);
        if (fitness > bestFitness) {
            bestFitness = fitness;
        }
    }

    System.out.println("Generation " + generation + " - Best Fitness: " + bestFitness);
    neat.evolvePopulation();

    if (bestFitness >= fitnessThreshold) {
        System.out.println("Satisfactory fitness level reached at Generation " + generation);
        break;
    }

    generation++;
}
```

A fitness threshold is then given to stop the algorithm when it reaches good performance.

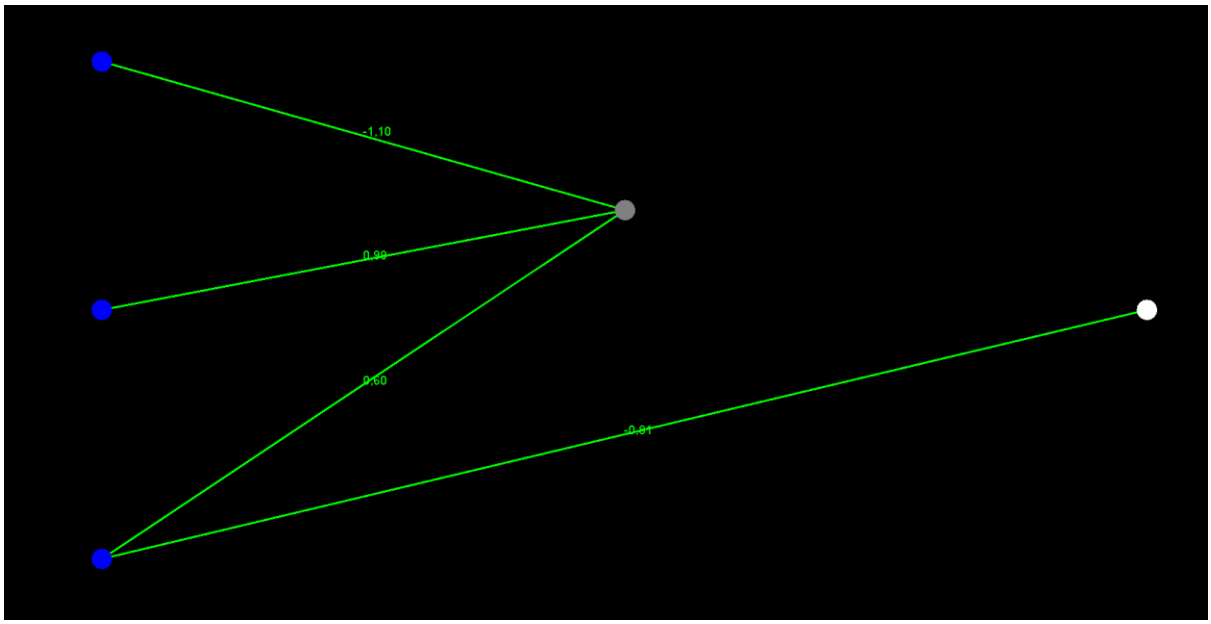
As for the performance results, the network generally manages to converge towards a good solution in around 100-200 generations with 205 individuals. A bias node is used, as is done in the paper.

The implementation can definitely be improved, not only be tweaking the parameters, but also by improving the mutation logic. On some good results, the network achieves results like these.

```
Demonstrating XOR with the best model:
Input1, Input2 -> Predicted Output : Actual Output
0.0, 0.0 -> 0.0214625830717016 97 : 0.0
0.0, 1.0 -> 0.9591992555166767 : 1.0
1.0, 0.0 -> 0.9768862474044889 : 1.0
1.0, 1.0 -> 0.0303440401692351 95 : 0.0
```



These are not too bad, but results are sometimes not as good.



Difficulties and problems

I personally have a great interest in neural networks and had a bit of previous knowledge of NEAT. However, the learning curve for implementing it has been steep.

Of course, some problems occurred along the way. We'll now examine one of them.

Self-decimating population

After the initial implementation, while testing with the XOR function, I noticed that the evolution was showing promising signs, and fitness was gradually increasing. Speciation was also

showing some good signs of progress and it clearly promoted different “approaches’ from different genomes.

However, upon further examination, I noticed that fitness never actually reached its maximum possible value. In fact, the program was stopping by itself before the stopping condition was hit. This is what was the console was showing:

The decimal number was the fitness, while the integer was the number of individuals in the species. As we can see this number was drastically going down.

This turned out to be caused by the fact that an individual that was pruned because of bad fitness, wasn’t reassigned to a new species. The pool of individuals being a fixed number, the more the algorithm continued, the more individual it killed and the less population it had. This resulted in a self-decimating algorithm, that was showing signs of promise, with a growing fitness, but that inevitably struggled to survive.

This problem was in the end quite easily solved by making sure every individual was reassigned to a new species after being killed.

```
################################
model.Species@520a3426 0.5 800
################################
model.Species@520a3426 0.5008115543075617 640
################################
model.Species@520a3426 0.5010749026062074 512
################################
model.Species@520a3426 0.502171305778192 409
################################
model.Species@520a3426 0.5020905314035655 327
################################
model.Species@520a3426 0.5038982801346302 261
################################
model.Species@520a3426 0.5050026323124484 208
################################
model.Species@520a3426 0.5070371184150175 166
################################
model.Species@520a3426 0.5083067377242365 132
################################
model.Species@520a3426 0.5136996223259086 105
################################
model.Species@520a3426 0.5165927939961273 84
################################
model.Species@520a3426 0.5215279779407137 67
################################
model.Species@520a3426 0.5276375603640959 53
################################
model.Species@520a3426 0.534938048007442 42
################################
model.Species@520a3426 0.5440884891522484 33
################################
model.Species@520a3426 0.5561126225574069 26
################################
model.Species@520a3426 0.5617698572143012 20
################################
model.Species@520a3426 0.5875524353392874 16
################################
model.Species@520a3426 0.601884371802099 12
################################
model.Species@520a3426 0.621816095992697 9
################################
model.Species@520a3426 0.6436543423888458 7
################################
model.Species@520a3426 0.6636253573603546 5
################################
model.Species@520a3426 0.6899321027087254 4
################################
model.Species@520a3426 0.7061818411146908 3
################################
model.Species@520a3426 0.7214709541597896 2
```

Conclusion

In this report, we examined how the NEAT project has been implemented, and how it's been tested with the XOR function. We went through the main classes that make up the logic of the algorithm and followed with the development process and the challenges that it presented.

To conclude, personally, this project has been a challenge. NEAT is, in my opinion, an incredibly interesting algorithm and I cannot avoid leaving this project with a slight feeling of bittersweet. I would have enjoyed working on this project more if I had more time. However, I am extremely satisfied with the notions that I have already learned about NEAT and I am sure I will continue to try being more proficient in it.