Evolutionary AI Library for PhaserJS

Author: Nuh Furkan ERTURK[1]

Scientific Coordinator: Iuliana MARIN[2]

Abstract: In this project a new javascript library js-gen-lib introduced. The library aims to ease required effort to implement genetic algorithms and evolutionary neural networks in javascript specifically PhaserJs project. Unlike existing javascript libraries for genetic algorithms, js-gen-lib has sophisticated architecture to be compatible with PhaserJs. In order to achieve this goals, a very low level and flexible library developed. Initially the aim was set to implement all the essential features of a genetic algorithm. "Introduction to Evolutionary Computing" of A.E. Eiben and J.E. Smith used to determine main guidelines. Special thanks to the writers.

Keywords: PhaserJS, Genetic Algorithms, Neuro-evolution

1. Introduction

a. What the project does?

Artificial Intelligence (AI) for computer games is a trending topic. There are numerous methods to develop such algorithms. The Genetic Algorithms(GAs), more specifically Neuro-evolution(NE), is one of those methods studied. The library "js-gen-lib"[1] is a sophisticated library to implement NE methods for games developed with the PhaserJs[2] framework.

b. Existing libraries and differences

There are already GAs libraries developed using java-script. However, their applications are relatively abstract and require a vast amount of effort to make implementations in particular fields. In this paper, the library "genetic-js"[3] had taken as a reference to compare with the introduced "js-gen-lib".

c. Why to use Genetic Algorithms?

GAs solve optimisation, simulation and modelling problems[4]. There are numerous players with various skill sets. As a result, a game AI requires constant optimisation. There will be many different and distinct problem cases which could be solved through efficient modelling. GAs are one of the most suitable technologies to solve the problems given and simulate them. Our library yields multiple neural networks for a single case, so we expect to provide a more authentic and personalised experience for the users.

2. Classes

In this section, classes of the "js-gen-lib" were introduced. There are five classes as follows; "Neuron", "Node", "NeuralBucket", "NeuralNetwork", and "Genesis". Apart from all those classes, another file included in the package is "js-gen-util.js". This file contains the necessary files to execute some specific operations of those classes.

---

[1] - Nuh Furkan Erturk, Sophomore, 1221B, Faculty of Engineering in Foreign Languages, nuh_furkan.erturk@stud.fils.upb.ro

[2] - Iuliana Marin, Department of Engineering in Foreign Languages, marin.iulliana25@gmail.com

a. Neuron

The neuron class is a critical class of the library where each object of the Neuron class represents a process inside the neural network. To express different kinds of procedures and diversify one of the same kinds, the constructer of each Neuron object generates a random string of 32-bits.

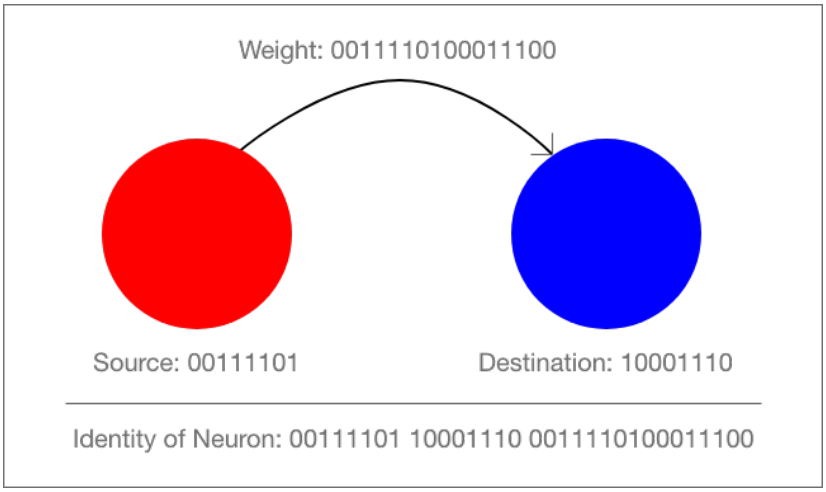

Figure-1: Representation of a Neuron object

A representation of an object of a Neuron is given above on the *Figure-1* with its 32-bit string. The first 8-bit represents the source node, while the second 8-bit represents the destination node. The last 16-bit part is called weight. The output of the source node was multiplied by the value weight to diversify the same kinds of neurones.

b. Node

As was introduced in the previous sub-title, each Neuron object consists of two nodes. All the nodes are objects of the Node class. There are three classes which extend the Node class. Depending on their "Node-ID", every node gets a class.

Table-1

Node-ID Starts With	For Source Nodes	For Destination Nodes
0	InternalNode	InternalNode
1	SensorNode	MotorNode

Guide to determining the node-type

The first bit of the ID determines the type of node. As is given above in *Table-1*, the source node can be a "SensorNode" or "InternalNode", and likewise, a destination node can be an "InternalNode" or “MotorNode".

c. NeuralBucket

A NeuralBucket object collects different implementations node classes. NeuralBucket has three different lists for each distinct node types.

d. NeuralNetwork

NeuralNetwork class creates the networks or the personas of each generation. They represent the neural networks and the AI. They take a number reference and generate random neurones.

A configuration process processes all the neurones generated and creates necessary nodes. When it is initiated, it starts a feed-forward process and give a list of results, called actions. Each action will be associated with the MotorNode label of its origin (the node in which printed). Programmers are expected to analyse actions and choose the most suitable ones to execute in their games.

After executing the neural networks, a fitness value should be set during the run-time of the game. So the rest of the library chooses best-fit networks to breed the next generations.

e. Genesis

This class handles most of the workload to execute and run our implementations. Through this class user creates a generation of networks and tests them. This class also spawns new generations and adds randomness.

f. Utilities

Numerous utility functions were developed to execute sophisticated operations among methods in our classes. This file helped us to avoid the use of any third-party packages. Furthermore, necessary common functions are implemented in the utility file to avoid replicated methods.

3. Implementation on a Game

The implementations of the library highly depend on those nodes. Programmers are expected to develop node classes extending SensorNode, MotorNode or InternalNode. Node class has a method called “setInputVal()”, where the programmer should override this method to change how nodes acquire new inputs. The "SensorNodes" also read their sensory inputs through this method.

There are also “giveOut()” method for InternalNodes and “giveAction()” for MotorNodes. By default, “giveOut()” returns the input value read previously. However, it should be over-written to get a variety of different outputs. “giveAction()” is a special method for MotorNodes working just like the “giveOut()” method.

Then, a "NodeBucket" object should be created, and all the Nodes implemented should be pushed to the NodeBucket object. After then, a Genesis object should be created using the "NodeBucket" object created as a parameter.

After then, the programmer should call the method “populate()” with two parameters where the first one represents the number of distinct networks created in a generation and the second one is the number of neurones in a network.

Finally “readyPopulation()” method should be called to configure networks automatically.

In order to create a new generation, the method “newGeneration()” should be called. In GAs, a mutation might have a significant impact and gives some randomness to GAs[5]. “js-gen-lib” also has a mutation method with two parameters. When the "mutate()” method is called before creating a new generation: a single random bit will be flipped. The probability of a bit-flip happening is the multiplication of the first and second parameters of the “mutate()” function. The first parameter refers chance of choosing a network for mutation and the second parameter refers to the probability for the neurones in the choosen networks.

4. Conclusion

To sum up, “js-gen-lib” has numerous features to implement GAs and gives powerful tools for programmers to develop AIs for web-based games. Insights on the possible applications and further improvement proposals are given below in sub-section “a” and, “b” respectively.

a. Possible applications

The library is meant to be used for game development, specifically for PhaserJs and other javascript applications. Hence, prospected applications are game-development on web platforms.

However, the library has many low-level features and the essential features of GAs such as "mutation", "populate", and "selection" functions. Other GAs applications would be interesting and feasible.

b. Further improvements

Since the whole library has developed in javascript, other options might perform better in the computational processes. Reconsidering those options and implementing especially computational parts of the program on those programming languages/frameworks would improve the capabilities and enwiden the feasible applications.

Another reform area is the methods offered by the library in the implementation of genetic algorithms. Currently, the library allows users to use only one mutation function and limited selection methods. Diversification of those operations would enable users to implement GAs in a larger spectrum.

5. Bibliography

Bibliography content

[1]. https://github.com/nuhfurkan/js-gen-lib

[2]. https://github.com/photonstorm/phaser/tree/v3.55.2

[3]. https://github.com/subprotocol/genetic-js

[4]. *A.E Eiben, J.E. Smith*, Introduction to Evolutionary Computing, Springer, Second Edition, 2015/4, pp. 1-5

[5]. *A.E Eiben, J.E. Smith*, Introduction to Evolutionary Computing, Springer, Second Edition, 2015/4, pp. 49-50