EspNeat's code overview

All that happens in **UnityNeat** is coordinated by the `GameObject Evaluator`. This object has a copy of the script `EspNeatOptimizer`, which manages the creation all the elements (visual and virtual) needed to evolve artificial brains for the desired units.

The interface is managed by `GuiManager`. `Visualizer` is also a very important class for some menus. These two classes allow the user to edit the modular structure of the genomes (adding a new module, reseting the module that is being evolved, and so on). When the user is satisfied with some modifications, the information required to actually perform any changes is passed from `GuiManager` to `EspNeatOptimizer` to `EspNeatGenomeFactory` (where it is finally done).

The evolution needs some setup information. For example, it needs to know if neural networks with loops are allowed or not, how long unit trials will be, if we will use Neat or HyperNeat, etc. All these are read from an XML document (`experiment.config`, in the folder "resources") and some are also coded in the script `SimpleExperiment`, which initializes some key classes.

The "brains" that control units are encoded in **genomes**, which are lists of **neurons** (nodes) and **connections**. These nodes and connections are fully described by their corresponding **genes**, organized in `NeuronGeneList` and `ConnectionGeneList`. The first set of Neat genomes in SharpNeat is created by the `NetworkFactory`, which is also responsible for adding new modules to the networks and other tasks with modules.

Connection genes contain an ID, a source neuron, a target neuron, a connection weight (strength) and a module ID. Connection genes can be protected or not protected (non protected connections are those added automatically by the evolution process). Non-protected connections are inside of a module, while protected connections communicate different modules, or modules with input/output, etc.

Neuron genes contain an ID, a neuron type (input, output, hidden), an activation function (how it translates input into output), an auxiliary state label (not used in this version) and a list of target and source neurons. ESPNEAT also requires a module ID, and introduces the neuron types local_input, local_output and regulatory.

Evolution needs a way to evaluate the performance of genomes. When this is required, units are instantiated by `Optimizer`, and each is assigned a brain created from the available genomes. Units are only aware of the input and output values of these brains, but it is up to them what to do with the output they receive as well as how to create the input required by these brains.

A **phenome** is a processed version of a genome, and is the brain itself (it takes input values and gives back output values). The details here depend on the kind of network being used. For example, if no loops are allowed, nodes are classified with respect to their distance from the input nodes. Then signals are propagated in the correct order. But if there are loops this approach is not valid. In any case, the genome contains all the information of the genome, but the functions to *efficiently* turn input into output are in the phenome.

A genome is translated into a phenome via a **decoder**.

Units, which actually work with phenomes (brains), can be evaluated automatically or manually (interactively). Automatic evaluation requires the user to code a fitness function in the `UnitController` script (`UnitController` is actually an abstract class. The user needs to code his or her own complete version to use as unit controller). Automatic evaluation can reward the unit for anything. For example, it could reward units with two inputs and two outputs for producing output_2 equal to input_1 and output_2 equal to input_1 or any other arbitrary function. Most often output values are used for something like locomotion and the unit is evaluated for its success in this task (for example, losing points for collisions).

Beware: `evaluator` and `.Evaluation` may refer to the creation of a unit, allowing it to behave for a time and THEN evaluating it. It is a poor choice of name, specially since the name `.Evaluate` is used in other places meaning only to assign fitness.

Manual evolution requires the user to click on interesting units, which are rewarded with a fitness value. As of this version, units get fitness equal to 1 by default, so if the fitness value for chosen units is 2, they will be two times as good when creating offspring. The default fitness value for chosen units is 10. Interactive evolution also allows to choose non-interesting units. This will get fitness = 0.

Once we have fitness values with each genome in the population, offspring will be created.

First genomes are compared with each other, and the most similar ones are grouped into species.

The fitness values associated with each genome are used to decide the amount of **offspring** each specie should produce.

Reproduction is controlled by an `EvolutionAlgorithm` (see `AbstractGednerationalAlgorithm`). Within each specie, genomes have a chance of producing offspring which is proportional to each genome's fitness. Within each specie there will also occur some sexual reproduction, which uses information from two parent genomes. A small fraction of sexual reproduction happens with parents from different species.

Reproduction will produce offspring which copy the parent's genome with some modifications. Mutations are implemented in `NeatGenome` (or the specific version of `IGenome` used). These mutations can be:

1. Change in the strength (weight) of a connection. Changes in weights can be of different kinds. They can be **resets** or "**jiggles**",

which can be **Gaussian** or **uniform** perturbations. The relative probabilities for these are stored in `NeatGenomeParameters`.

2. Addition of a node. As explained in the code: When a neuron is added to a neural network in NEAT, an existing connection between two neurons is discarded and replaced with the new neuron and two new connections, one connection between the source neuron and the new neuron and another from the new neuron to the target neuron. The new IDs for the elements created are stored, so they can be used if this mutation happens in another genome.
3. Addition of a new connection. Repeated connections are not allowed. Loops may not be allowed either, depending on the phenome type.
4. Deletion of connections (and nodes which as a result end up isolated).
5. The auxiliary state of a neuron may be changed. In this version this is not used at all.

NEAT assigns a unique ID value for each structure element. Each node and connection will share the same ID in different genomes. This allows to compare genomes, which is needed to classify genomes into species. This is also needed to produce coherent sexual reproduction. In sexual reproduction the offspring inherit a copy of every common gene from the parents. It takes a parent at random in each case (both parents share these connections, but not their strength!). The genes that are not common (and we know this thanks to the unique IDs) can then be treated in a special way.