Research Master Internship Report

18/05/2020 - 27/11/2020

NeuroEvolution for Reinforcement Learning

Author:
M. Paul Templier

Supervisor:
Dennis Wilson
Enseignant-Chercheur,
ISAE-SUPAERO

v0.1

Note: This report being due for a defense on 04/09/2020, its content do not account for the entirety of the internship.

Acknowledgements

First and foremost I would like to express my gratitude to my supervisor Dennis Wilson[1] for his guidance, comments and suggestions during this internship, and for giving me the opportunity to finally experience the world of the evolutionary computation that had appealed to me for many years while benefiting from his experience. I would also like to extend my thanks to Emmanuel Rachelson[2], who made our Data & Decision Sciences MS curriculum an unforgettable experience, and I look forward to having the opportunity to work with them in the future.

I also want to thank the whole SuReLI[3] group at ISAE-SUPAERO[4] for their warm welcome, both on site and while working for home, and especially my fellow Research Intern Lucas Hervier[5] whom I had the pleasure to talk and work with.

Finally, a special thanks goes to Carlos Aguilar[6], Blanche Gracia-Faucher and ISAE-SUPAERO for helping me find this internship after borders closed due to Covid-19.

[1] https://people.isae-supaero.fr/dennis-wilson?lang=en
[2] https://people.isae-supaero.fr/emmanuel-rachelson?lang=en
[3] https://sureli.github.io/
[4] https://www.isae-supaero.fr/en/
[5] https://www.linkedin.com/in/lucashervier/
[6] https://scholar.google.fr/citations?user=jfSYshIAAAAJ&hl=fr

Contents

Introduction **5**

1 Prolegomena **8**

1.1 Reinforcement Learning: what is an MDP? 8

1.2 Artificial Neural Networks acting as a policy 9

1.3 Development tools 12

2 Evolutionary Computation **13**

2.1 Evolutionary Algorithms 13

2.2 Evolution of policies 15

2.3 NeuroEvolution 16

2.4 Quality Diversity approaches 19

3 Breezy Project: a Dota 2competition **22**

3.1 Context 22

3.2 Dota 2as a RL environment 24

3.3 Our approach 27

3.4 Results 31

4 Benchmarking Evolutionary Reinforcement Learning **32**

4.1 Motivation and structure 32

4.2 Benchmark tasks 35

4.3 Algorithms in `BERL.jl` 36

5 NeuroEvolution.jl **38**

5.1 Original NEAT algorithm 38
5.2 Implementing NEAT 41
5.3 HyperNEAT 47
5.4 Future developments 48

6 First results and Future work **50**
6.1 NEAT 51
6.2 Algorithm comparison 51
6.3 Future work 52

Conclusion **54**

Acronyms

Concepts

MDP *Markov Decision Process*

ML *Machine Learning*

RL *Reinforcement Learning*

ERL *Evolutionary Reinforcement Learning*

DL *Deep Learning*

EC *Evolutionary Computation*

Algorithms

NEAT *NeuroEvolution of Augmenting Topologies*

CPPN *Compositional Pattern-Producing Network*

HyperNEAT *Hypercube-based NEAT*

CGP *Cartesian Genetic Programming*

CMA-ES *Covariance Matrix Adaptation Evolution Strategy*

BERL *Benchmarking Evolutionary Reinforcement Learning*

Introduction

Technical Context

Evolutionary algorithms are an attractive alternative to gradient-based methods for the optimization of neural network parameters. Evolutionary strategies [Wierstra 2011] estimate the gradient of the objective function but do not require a strict gradient definition, enabling their application to a variety of problems. The evolution of synaptic weights of a neural network has been demonstrated on a variety of tasks [Such 2017] and is competitive to state-of-the-art deep reinforcement learning methods such as Deep Q Networks [Mnih 2015].

However there are many open questions in the application of neural networks. Direct encoding approaches which optimize each synaptic weight often struggle with high-dimensionality, which is required for convolutional neural networks with millions of parameters. Indirect encoding approaches which optimize a surrogate function which is then applied to determine neural network weights might not be able to reach specific maxima due to the limitations of the genetic encoding. Finally, some evolutionary approaches also optimize neural network structure in addition to or instead of synaptic weight optimization.

Neural networks are often applied in supervised learning cases where data is plentiful and a strong error signal can be backpropagated through the network for each set of examples. Stochastic search methods such as evolutionary algorithms, which at best estimate the search space and are intended to optimize independent variables, are not an appropriate choice for supervised learning. However, when the learning signal is weak, such as in reinforcement learning, evolutionary methods can improve on gradient-based methods which must construct a differentiable approximation of the objective. The application domain of this internship is therefore games, which have a weak learning signal and sparse rewards; this is a domain where evolutionary algorithms are a good choice over other gradient-based methods.

Research internship

This internship takes place in the context of both a Master's degree in Engineering (MSc Eng) at ISAE-SUPAERO[7] and a Masters degree in Operations Research[8] led by ENAC (M2 RO) as a research internship in Machine Learning. This internship was completed in the Department of Complex Systems Engineering (DISC) at ISAE-SUPAERO, in the Supaero Reinforcement Learning Initiative (SuReLI)[9] research group.

As the initial goal of my internship was to study neuroevolution, i.e. the optimization of neural networks with evolutionary algorithms, I first joined my supervisor's class on evolutionary computation[10] to consolidate the first knowledge I had into a more solid base. I then took part in a competition to evolve bots to play Dota 2, a complex real-time video game, in a 1v1 setting in order to experiment neuroevolution on a real use case. This first phase allowed me to gain a wider vision of evolutionary computation, along with helping me grasp the concepts and the API of `Cambrian.jl` , a framework for Evolutionary Computation which I later used. For 2 weeks, we also followed the GECCO[11], Alife[12] and ICML[13] conferences held online.

As my internship then focused more on Evolutionary RL in general, I started working on the `BERL.jl` library to develop a benchmak as it was still a missing element in this field, for which I learnt the Julia language.
I then used the knowledge on neuroevolution I had developed during the first phase to develop the `NeuroEvolution.jl` library based on `Cambrian.jl` , to implement standard neuroevolution algorithms for `BERL.jl` and to provide a modular framework which will allow me to try novel neuroevolution algorithms in the coming months.

Layout of this document

This document is a report on my internship, to showcase research work I have already conducted and will continue for the last months. I tried to describe my work and to document the developments made notably on the Dota 2bot, `BERL.jl` , and `NeuroEvolution.jl` .

In chapters 1 and 2, I will lay the bases of Reinforcement Learning, neuroevolution and evolutionary approaches as a first theoretical preamble. Chapter 3 will describe the

[7] https://www.isae-supaero.fr/en/
[8] http://m2rit-ro.recherche.enac.fr/
[9] https://sureli.github.io/
[10] https://d9w.github.io/evolution/
[11] https://gecco.tamps.cinvestav.mx/
[12] https://alife.org/conference/alife-2020/
[13] https://icml.cc/

competition we entered aiming at developing an AI to play 1v1 games of Dota 2. In chapters 4 and 5, I will then introduce `BERL.jl` and `NeuroEvolution.jl` , two Julia libraries I developed to try and benchmark evolutionary RL algorithms. Finally, chapter 6 will summarize results encountered so far and present the leads for future work in this internship.

Chapter 1

Prolegomena

1.1 Reinforcement Learning: what is an MDP?

A **Markov Decision Process** [Bellman 1957] (or *MDP*) is a process that formally describes a fully-observable environment, which means the current state fully characterizes the environment.

It can be defined as:

- a set of states S
- a set of actions A that can be taken from the states in S
- the probability $P_a(s, s')$ of reaching each state s' if action a is chosen from state s
- and the immediate reward $R_a(s, s')$ associated with transitioning from state s to state s' by taking actions a

An environment can be considered to have a **sparse reward**, which means many transitions between states have the same null reward, and only a few have a different information. This raises the issue of reward plateaus, where there is no information on what makes an action better than the others, hence making it harder to evaluate a state.

An example of environments with sparse rewards are games like chess, where only winning or losing give tangible information on the value of a state, but not intermediate game states. Hence, an agent can play many steps without receiving any reward, and will need to explore many trajectories to find information.

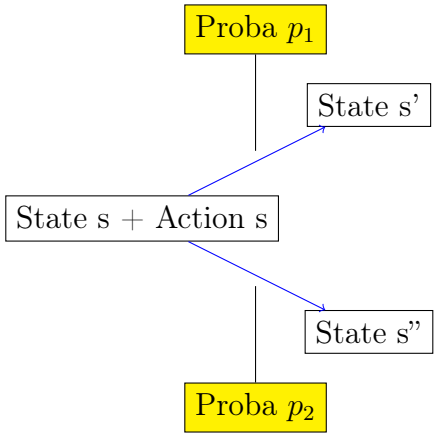

Figure 1.1: Markov Decision Process ($p_1 + p_2 = 1$)

A (deterministic) **policy** π is a mapping from states to actions. $\pi : S \longrightarrow A$.

The goal of **Reinforcement Learning** is to find the optimal policy π^* that maximizes the expected sum of rewards across an experiment.

1.2 Artificial Neural Networks acting as a policy

1.2.1 Artificial Neural Networks

An Artificial Neural Network (or *ANN*, simply called Neural Network in this document) is a bio-inspired computing systems based on some principles present in the biological brain. ANNs can be represented as a directed graph of artificial neurons connected to each other.

An artificial neuron is a simple computing unit which effectuates a weighted sum of multiple inputs x_i, adds a bias b, and then computes a function (called *activation function* f) on this sum. The final value is presented as output Y:

$$Y = f(\sum_i w_i x_i + b)$$

with w_i the weight associated with input i (cf figure 1.2).

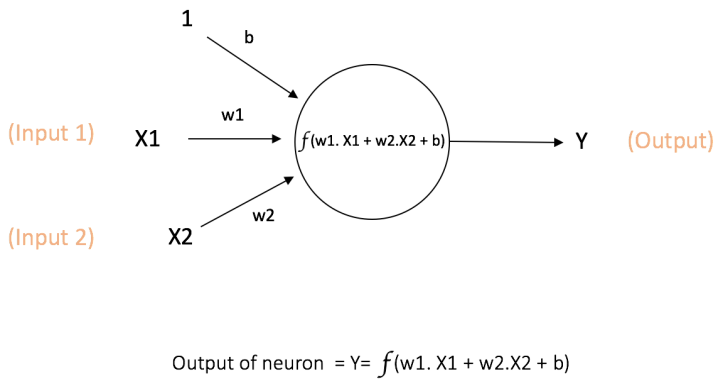

Figure 1.2: Representation of an Artificial Neuron
https://www.kdnuggets.com/2016/11/quick-introduction-neural-networks.html

As illustrated in figure 1.3, ANNs are functions in which input neurons each take the value of an argument, and the output of the function is a vector computed from the concatenation of the values of the output neurons.

Traditionally, neurons are presented as layers in an ANN, units of a layer taking the outputs of neurons of the previous layer as input, and being used by the next layer. Each layer is then computed successively, from the input layer to the output. Recurrent units link the output of neurons to the input of earlier neurons, transmitting information from an evaluation to the next one, which can be useful for series of linked data points.

If this structure is convenient to represent fixed networks, other architectures can exist where each neuron is considered individually and can be linked to any other neuron, without layers. These architectures however require a specific heuristic defining the order in which all neurons are computed.

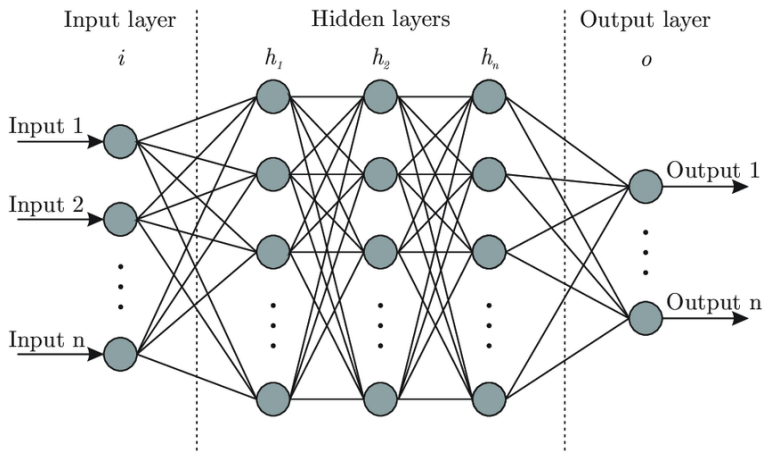

Figure 1.3: Representation of an Artificial Neural Network with layers in [Bre 2017]
https://www.kdnuggets.com/2016/11/quick-introduction-neural-networks.html

All the weights and biases from all the neurons in a network are parameters of the neuron, and changing them induces a change in the output of the function.

1.2.2 Direct Policy search with ANNs

As neural networks [Rosenblatt 1958] can - in theory - approximate any function, they can be used to define a policy mapping any state to an action. In RL, **Direct Policy search** aims at finding the best policy through the optimization of the parameters of this policy network.

In this method, the network is optimized to output the best action to take, from a state given as input. Two optimization methods can be employed: gradient-based, or stochastic.

1.2.3 Gradient-based optimization

Policy gradient methods use gradient descent [Robbins 1951] on the parameters of the neural network to optimize the policy.

As in supervised learning the true value of the target is known, the gradient is readily available, making gradient descent a viable option for neural network optimization.

However, reinforcement learning environments present harder settings for gradient-based optimization:

- The best action is not already known, so the gradient is harder to compute
- As a first-order method, gradient descent can get stuck in a local optimum or a plateau
- RL environments with sparse rewards present many plateaus and make the gradient harder to find

Hence, optimizations methods not based on gradient such as evolutionary methods can be a solution to these sorts of problems.

1.2.4 Evolutionary Reinforcement Learning

Evolutionary Reinforcement Learning (or *ERL*) is the RL approach based on using stochastic optimization, and more precisely evolutionary algorithms, to optimize a policy network without computing the gradient. Some of the evolutionary algorithms ERL can use are described in chapter 2.

As the ERL field is still lacking a tool to compare and benchmark algorithms, I took part in a Dota 2 ERL competition and I am currently working on a benchmarking framework, as described in chapters 3 and 4.

1.3 Development tools

1.3.1 Julia

Julia[14] is a high-level, dynamic programming language designed for high performance with a just-in-time compiler. It is notably interoperable with multiple languages, including C, Fortran, Python, R, MATLAB, Java, or Scala. [Bezanson 2017]

I used Julia in this project to provide a faster environment than Python, while still being compatible with state-of-the-art libraries such as gym in Pyhton or Atari in C.

The principles of **Test-Driven Development** were followed: requirements are first translated into tests, then the code is developed until all tests pass. Tests provide both a way to ensure new code doesn't break previously working code, and a list of use cases of the library, hence improving its documentation.

Additionally an instance of **TravisCI**[15], a Continuous Integration service, was linked to all libraries I developed to build and test all code on remote servers, ensuring they can be run on machines with a clean configuration.

1.3.2 Cambrian

Cambrian.jl[16] is an Evolutionary Computation framework developed in Julia by my supervisor Dennis Wilson. It implements a structure to facilitate the development of evolutionary algorithms, focusing on genetic programming and neuroevolution. It aims at reaching the modularity of the MABE (Modular Agent-Based Evolution) platform[17], while focusing more on the evolutionary computation aspect.

I used Cambrian in my development to provide a consistent framework between projects and ensure compatibility, and for its usefulness in structuring the development.

[14] `https://julialang.org/`
[15] `https://travis-ci.org/`
[16] `https://github.com/d9w/Cambrian.jl`
[17] `https://github.com/Hintzelab/MABE/wiki`

Chapter 2

Evolutionary Computation

2.1 Evolutionary Algorithms

Evolutionary Algorithms, and more globally Evolutionary Computation, are bio-inspired approaches based on the idea of using some mechanisms of biological evolution [Darwin 1859] and genetics [Mendel 1865] in computational methods [Mitchell 1998].

Evolutionary algorithms are stochastic optimization methods that use a population of candidate solutions (or *individuals*) to iteratively reach an optimal or quasi-optimal solution by selecting the fittest individuals at each generation. They can use mutation operators, which modify some characteristics of an individual, and crossover operators that create children sharing traits with 2 (or more) parents from the population, in a process described in figure 2.1.

Evolutionary Algorithms can be characterized at different levels, as seen in figure 2.2:

- **Gene**: Based on biological genetics, genes are the basic blocks of the individual, each defining a parameter such as the value of an argument or a simple rule.
- **Genome**: This group of genes describes an individual
- **Phenotype**: Once expressed, the genome creates the phenotype, for example a graph from a list of genes each defining the weight of a link between 2 nodes.
- **Problem**: Some algorithms can solve static problems like the Rastrigin function while others optimize policies, for example investment strategies.
- **Fitness function**: The fitness function evaluates how well the individual is adapted to the problem, e.g. by computing the mean error or any function designed to be maximized by the algorithm.

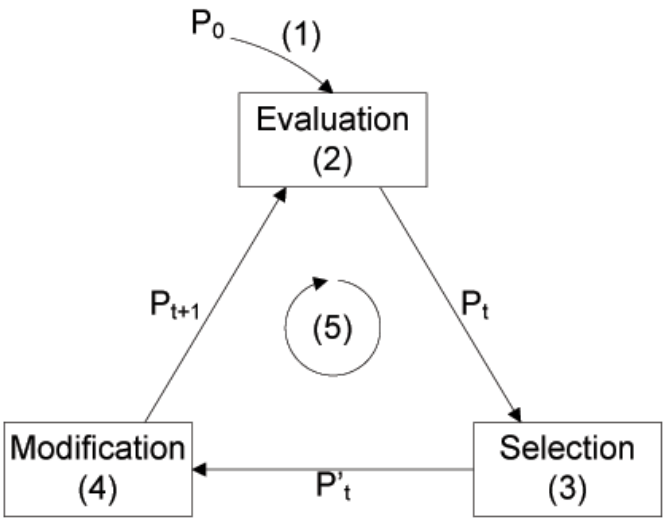

Figure 2.1: Illustration of the steps of an Evolutionary Algorithm.
Dennis Wilson, `https://github.com/d9w/evolution/blob/master/2_ga/1_GAs.ipynb`

Evolutionary algorithms follow the same steps, as illustrated in figure 2.1: (1) A population P_0 of random individuals is created, (2) At each generation t, each individual in population P_t is evaluated and assigned a fitness value based on the evaluation, (3) Some individuals are selected based on their fitness, (4) Mutation and Crossover operators create new individuals and generate the population P_{t+1} for generation $t + 1$, (5) This iteration is repeated until a stopping criterion (e.g. performance, computation cost) is reached.

The defining components of an evolutionary algorithm are the design of a *genetic encoding* which encodes an individual into genes, the *genotype-to-phenotype mapping* which creates an individual from its genes, the *evolutionary operators* of mutation and crossover than create new individuals, and the *population selection* method to conserve the best traits.

Problem	Gene	Genome	Phenotype	Fitness Function
TSP	110	sequence of cities	tour	tour length
Function optimization	3.21	variables x of function	f(x)	lmin-f(x)l
graph k-coloring	permutation element	sequence for greedy coloring	coloring	# of colors
investment strategy	rule	agent rule set	trading strategy	portfolio change
Regress data	Executable sub-expression	Executable expression	model	Model error on training set (L1, L2)

Figure 2.2: Evolutionary Algorithms: examples of problems and individual encodings. Examples in white are static problems which can use a direct problem encoding in the genome. Examples in yellow show the evolution of a policy or function. [O'Reilly 2019]

2.2 Evolution of policies

While Genetic Algorithms are optimization algorithms, Evolutionary Programming evolves functions or policies in an environment they interact with. It relies on the same mechanisms as Evolutionary Algorithms, except the individuals are functions. Mutation and crossover operators are defined on these functions, and selection is based on the behavior of each individual through the total reward they collect during a run. This total reward can then be adjusted to compute the fitness by favoring diverse behaviors or phenotypes. By optimizing for the most efficient behavior, these algorithms look for the function that solved the problem best. These functions can be for example defined as graphs like Cartesian Genetic Programming [Wilson 2018a], or as neural networks like NEAT [Stanley 2002a].

In addition to previously shown features, individuals in Evolutionary Programming algorithms can be characterized by their behavior in an environment. We can hence look at **Genotype**, **Phenotype**, and **Behavior** for a mapping of individuals in a problem. Even though individuals may share similar genotypes or phenotypes, their behaviors may differ due to the impact of the environment.

Optimizing functions to be used as policies through evolutionary algorithms is the field of Evolutionary Reinforcement Learning, as described in section 1.2.4.

2.3 NeuroEvolution

Neuroevolution is the field which tries to evolve neural network, i.e. their weights and/or architecture, through evolutionary algorithms. Two main approaches have arisen to tackle this issue: directly encoding the network, and using a generator to create it indirectly.

2.3.1 Direct encoding of an ANN

The first approach relies on directly encoding the final neural network, used to choose an action during the game, as an individual for the genetic algorithm. Each weight is therefore directly mutated by the algorithm.

Weight Evolution with Evolution Strategies

Similarly to a gradient-based methods, this technique aims to optimize synaptic weights and biases in a neural network by using evolutionary algorithms such as Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [Hansen 1996] [Salimans 2017].

CMA-ES is a population-based stochastic derivative-free method that creates new populations based on a distribution, which evolves from the distribution in the previous generation based on which candidates performed best (Fig. 2.3).

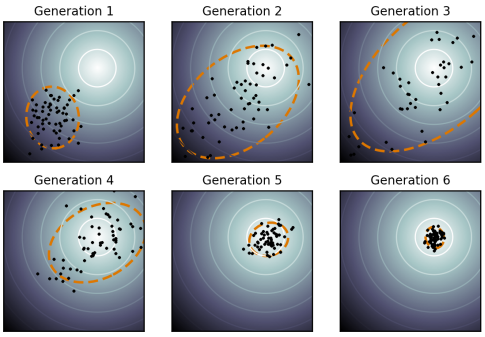

Figure 2.3: Illustration of an actual optimization run with CMA-ES on a simple two-dimensional problem. The spherical optimization landscape is depicted with solid lines of equal-values. The population (dots) is much larger than necessary, but clearly shows how the distribution of the population (dotted line) changes during the optimization. [Wikipedia contributors 2020a]

NeuroEvolution of Augmenting Topologies

Since the architecture of a neural network (e.g. number of layers, size of each layer, recurrent units) can have a strong impact on final performance, this technique evolves simultaneously both network structure and neuron weights of the final network. The NeuroEvolution of Augmenting Topologies (NEAT) algorithm created in 2002 ([Stanley 2002a, Stanley 2002b]) presents solutions to 3 fundamental issues in the evolution of neural networks:

- **Crossover between different structures**: NEAT provides a genotype-to-phenotype mapping based on mutations (Fig. 2.4).

- **Protection of topological innovation**: The speciation mechanism evaluates candidates by comparing them to similar architectures to allow new structures to mature without being instantly eliminated.

- **Minimization of structural complexity**: Nodes are added to a minimal initial structure in order to prevent networks from being too big without benefits.

Since NEAT directly uses the network it evolves as the policy, it is a direct encoding method: the phenotype is built from the genotype in one step.

Multiple variants have been created from NEAT, including rtNEAT [Stanley 2005] for real-time neuroevolution, FS-NEAT [Whiteson 2005] for feature selection, or HyperNeat based on indirect encoding [Stanley 2009].

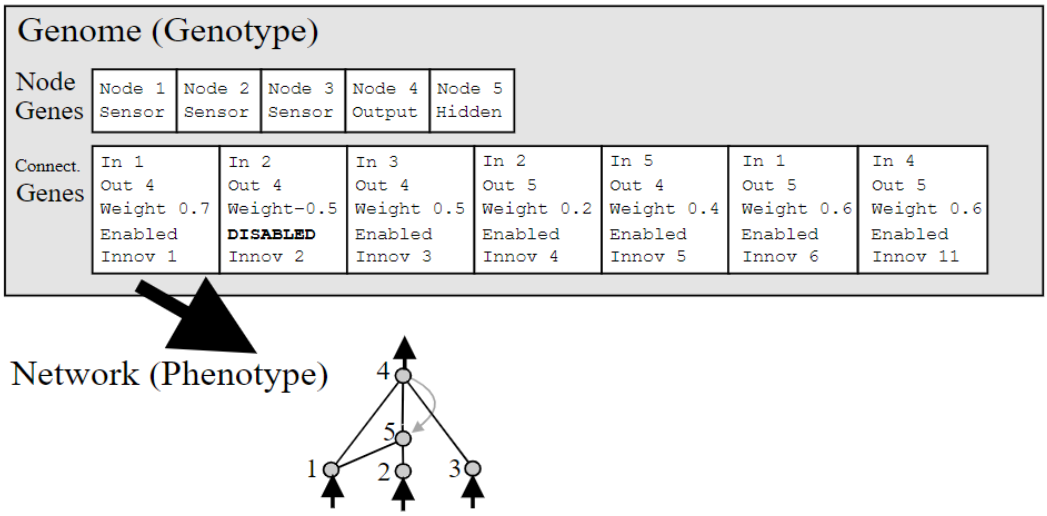

Figure 2.4: NEAT genotype-to-phenotype mapping example. A genotype is depicted that produces the shown phenotype. There are 3 input nodes, one hidden, and one output node, and seven connection definitions, one of which is recurrent. [Stanley 2002b]

Further details on NEAT are given in 5.

2.3.2 Indirect encoding of Neural Networks

The second approach uses an intermediary program, optimized through an evolutionary algorithm, to generate the synaptic weights of the final neural network based on their positions. This method reduces the search space to a more efficient one by reusing the same information multiple times, since the same program is used for each and every connection across the final network.

HyperNEAT [Stanley 2009] uses a Compositional Pattern Producing Network (CPPN) [Stanley] optimized by a NEAT-based process to create the final network. CPPNs are Artificial Neural Networks that implement multiple activation functions including Gaussian, sigmoid, and periodic functions such as sine or cosine, adding symmetry patterns to their outputs relative to their inputs.

Using a CPPN as intermediate network allows the final network to be larger without increasing the size of the space to explore, since the same CPPN can be used on any number of neurons. The symmetry patterns of the CPPN also produce connectivity patterns in the final network based on the positions of the neurons, better exploiting regularities in the task.

2.4 Quality Diversity approaches

While most evolutionary approaches focus on directly finding the individual with the highest fitness, the family of Quality-Diversity algorithms [Cully 2018] favors solutions both diverse and high-performing. Since some RL problems can be very deceptive for neuroevolution (cf figure 2.5), evolving more varied behaviors can lead to better-performing solutions and avoids getting stuck in dead-ends.

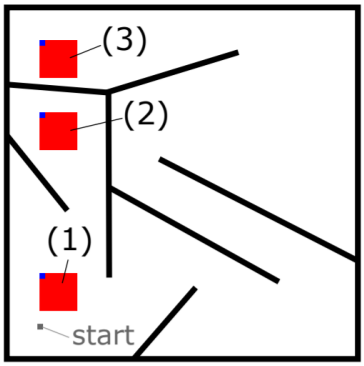

Figure 2.5: Deceptive maze in which (1) is the start region, (2) is the dead-end region, and (3) is the exit region. Evolution based on distance to the exit region leads to individuals being stuck in the dead-end region. [Doncieux 2019]

2.4.1 Novelty search

Novelty Search [Lehman 2011a] replaces the fitness function in a problem with a *novelty score*, which defines how novel the behavior of the individual is compared to previously encountered populations. It hence pushes towards more diversity, to cover the search space: it is a *divergent* evolution technique.

Novelty Search with local competition [Lehman 2011b] applies this principle while still using the fitness score to keep the best-performing individuals in each niche, without inducing competition between individuals of different niches.

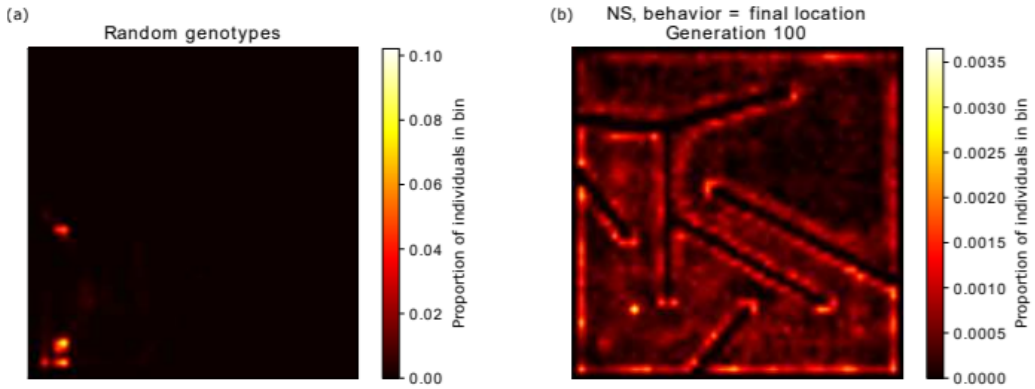

Figure 2.6: Solutions generated by (a) a random search of 10000 individuals in the genotype space and by (b) 100 generations of 100 individuals via novelty search. [Doncieux 2019]

2.4.2 MAP-Elites

In population-based algorithms like Evolutionary programming, elites are the best elements in a population, or in a local sub-population. Keeping the elites from a generation to the next in order to be sure to not lose the best individuals is called *elitism*.

The Multi-dimensional Archive of Phenotypic Elites (*MAP-Elites*) algorithm [Mouret 2015] aims at *illuminating the search space*, i.e. exploring the search space and keeping local elites in a discretized map of the space. By characterizing behavior with features independent from the fitness, MAP-Elites discovers diverse behaviors, and can show the fitness landscape of the problem and how attributes of solutions contribute to performance.

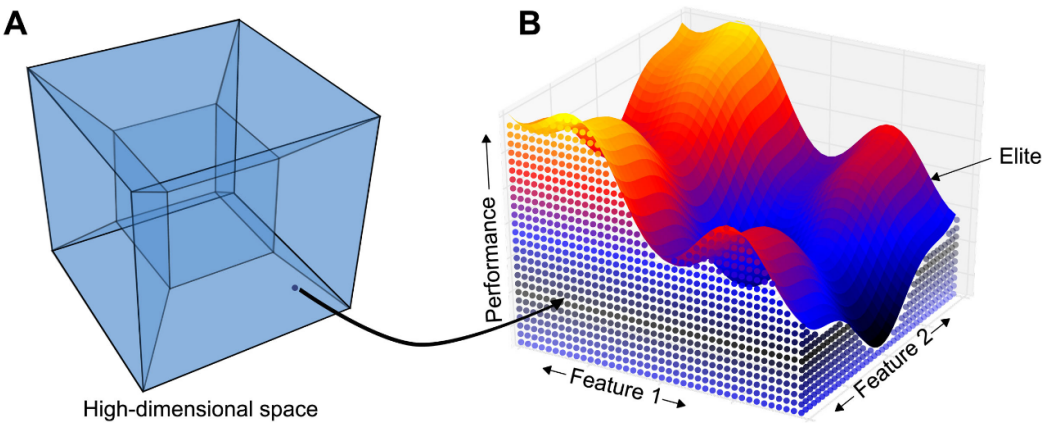

Figure 2.7: The MAP-Elites algorithm searches in a high-dimensional space to find the highest-performing solution at each point in a low-dimensional feature space, where the user gets to choose dimensions of variation of interest that define the low dimensional space. [Mouret 2015]

Chapter 3

Breezy Project: a Dota 2competition

During the first month of my internship I teamed up with my supervisor Dennis Wilson and my fellow research intern Lucas Hervier to participate in an Evolutionary Reinforcement Learning competition held as part of the international Genetic and Evolutionary Computation Conference, known as GECCO 2020, on the Dota 2video game. The agents we submitted are available on GitHub[18], and our code has been made open-source[19].

3.1 Context

3.1.1 Video games and Machine learning

In 2018 the video game industry generated sales of more than US$130 billion, reaching around 2 billion players around the world. Because video games can provide complex tasks necessitating short- and long-term planning in a controlled environment with no risks to humans [Laird 2001], they also provide an excellent test-bed for Artificial Intelligence (AI) systems. Multiple learning agents have been tested on the Arcade Learning Environment based on Atari 2600 games [Bellemare 2012], including Reinforcement Learning (RL) techniques like Deep Q Networks [Mnih 2013] and Agent57 [Badia 2020]. Other games such as StarCraft, Dota 2, Mario, or Doom, have also been extensively used as machine learning benchmarks.

[18] `https://github.com/d9w/DotaBot`
[19] `https://github.com/d9w/Project-Breezy-DOTA-Evolutionnary`

3.1.2 OpenAI Five

From 2016 to 2019, OpenAI[20] published work on the OpenAI Five project[21], an AI trained to play Dota 2first in 1v1, and then extended to 5v5. OpenAI Five achieved a 99.4% win rate[22] against human players of any rank and defeated the world champions[23] twice in a live tournament, while also showing cooperation capabilities with humans.

However, most of the technical details of the AIs trained for this project still remain unknown and unpublished, from the model architecture to the interface used to communicate with the game developed in partnership with Valve. Hence, further work and results reproduction are harder to implement, adding to the computational constraint: OpenAI Five required 764 petaflop/s-day operations in over 10,000 years of games played against itself, totalling around $6.6.10^{22}$ operations. This makes the use of Dota 2as a RL benchmark and its use in the development of open-source RL algorithms still an open problem.

3.1.3 Project Breezy

In this context, Project Breezy[24] is a competition created by Robert Smith and Malcolm Heywood from Dalhousie University to develop a Dota 2-playing program evolved with evolutionary algorithms, in a 1-versus-1 symmetrical match-up.

For this purpose, they implemented and published a server interfacing with the Dota 2game, called *Breezy*. As this work was not officially supported by Valve there may still be a performance difference compared to the system used to train OpenAI Five, but it provides an open-source tool for benchmarking algorithms on Dota 2.

We entered this competition as an opportunity to try and test neuroevolution algorithms on a complex real-time game, and as a step towards the development of a comprehensive benchmark of Evolutionary Reinforcement Learning algorithms as described in Chapter 4.

[20] `https://openai.com/about/`
[21] `https://openai.com/projects/five/`
[22] `https://arena.openai.com/#/results`
[23] `https://openai.com/blog/openai-five-defeats-dota-2-world-champions/`
[24] `https://web.cs.dal.ca/~dota2/?page_id=353`

3.2 Dota 2as a RL environment

3.2.1 The game

Produced by Valve and available on its platform Steam[25], Dota 2[26] is the sequel of Defense of the Ancients (DotA), a community-created game mod for Warcraft III that popularised the Multiplayer Online Battle in Arena (MOBA[27]) video game genre. In Dota 2, 2 teams (called *Radiant* and *Dire*) of 5 players each compete to destroy the main structure of the enemy's base (the *Ancient*) in a real-time strategy game, each player controlling one of the 119 possible characters (the *heroes*).

Figure 3.1: A game of Dota 2in progress, showing the Radiant team inside their base at the beginning of a match. [Wikipedia contributors 2020b]

The game takes place on a square map (cf fig. 3.2) which contains both team-controlled buildings (e.g. towers, barracks) that the other team can attack, and neutral objectives (e.g. Roshan). Team bases are situated in the bottom-left and top-right corners respectively, with 3 lanes joining them called the top (through the top-left corner), the middle (diagonally), and the bottom (through bottom-right corner) lane, each guarded by towers and around which most early gameplay happens. Creeps are basic AI-controlled units that periodically spawn for each team at the barracks and walk down the lane to help destroy enemy buildings. Dealing the last hit ("last hitting") an enemy creep is rewarded with money, which allows to buy items influencing the player's hero strengths. Last hitting a creep of the same team allows to deny this resource from the enemy.

[25] https://store.steampowered.com/app/570/Dota_2/
[26] https://en.wikipedia.org/wiki/Dota_2
[27] https://en.wikipedia.org/wiki/Multiplayer_online_battle_arena

Figure 3.2: Labelled Dota 2map as of version 7.20
https://dota2.gamepedia.com/Map

The 1v1 game mode opposes only 1 player in each team, and can be won by either killing the enemy twice, or destroying the first enemy tower on the middle lane. The game mode used for this competition additionally fixes the hero to be Shadow Fiend[28] for both players, for the purpose of simplicity and balancing.

3.2.2 Project Breezy Software

Project Breezy[29] provides software to enable exchanges with the Dota 2game by opening a server communicating with the game instance. The agent, through a Python server, interacts with the Breezy server, getting the game state and sending action orders. The game state contains 310 features such as health points, heroes positions, creeps positions, or spell cooldowns. 29 actions are available, including movements, spell casting, or attacking.

We noticed the server actually returned the state before effectuating an action, adding a delay in the information processing. Even though evolved models might still work thanks to internal memory, recurrence, or implicit prediction, this made predicting the next state in the simulator and analyzing behaviors significantly harder. This analysis can be seen in figures 3.3 and 4.1, which respectively show the de-synchronisation between actions taken

[28] https://dota2.gamepedia.com/Shadow_Fiend
[29] https://web.cs.dal.ca/~dota2/?page_id=353

and states received, and the irregularity of the timesteps between states.

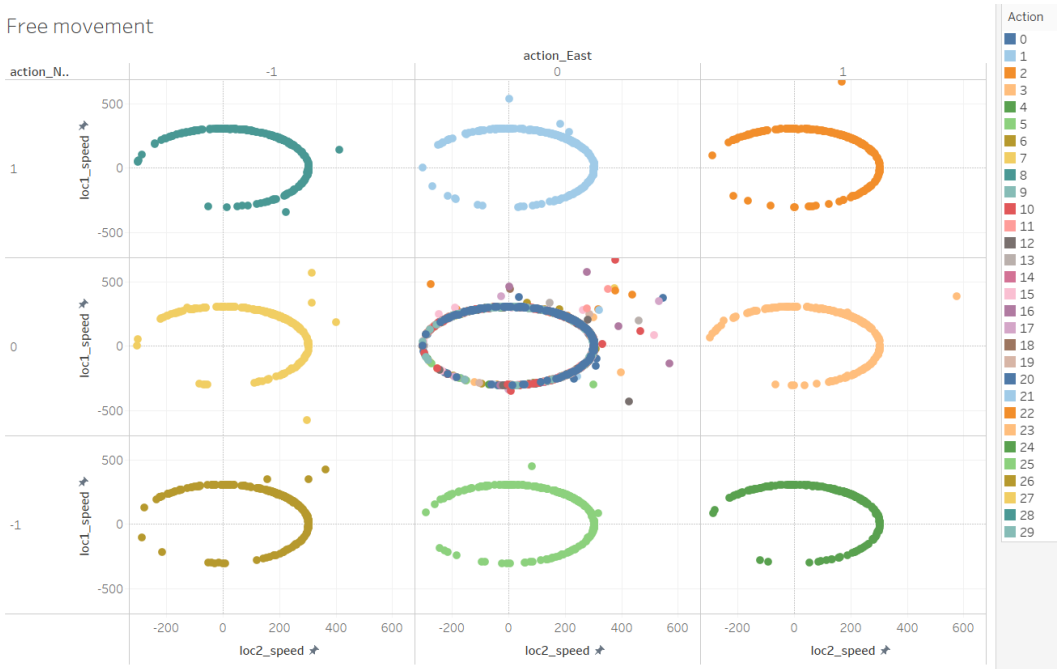

Figure 3.3: Attempted movements as received from the server after sending an action, showing no correlation between the action sent and the movement.

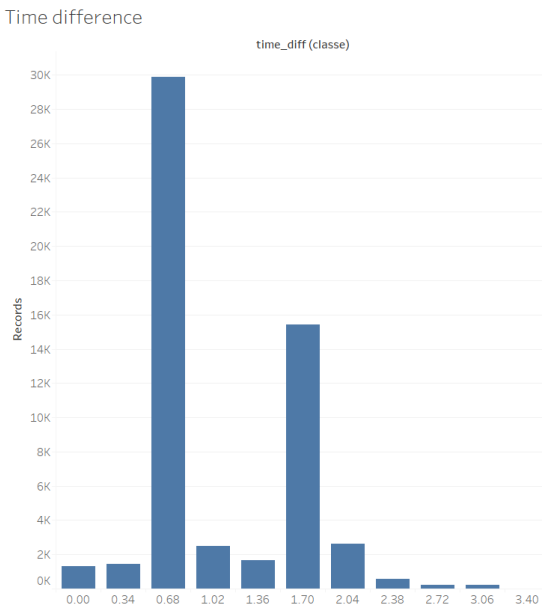

Figure 3.4: Distribution of the in-game time difference between the reception of 2 consecutive states from the game, showing inconsistency in the Breezy interface.

3.2.3 Challenges

Evolving Neural Networks to play Dota 2is subject to multiple constraints:

- **Processing time**: Since Dota 2is a real-time game, the generated neural networks needs to be fast to evaluate at each step.
- **Evaluation time**: Even sped up, evaluating a policy requires running a full game in a simulator, hence the number of evaluations is a strong limiting factor.
- **Network size**: Due to the large size of the game state provided by the Breezy server and the high number of possible actions, generated neural networks might require large architectures, hence increasing computation time and problem dimensionality.
- **Network complexity**: Since Dota 2requires both short- and long-term planning, more complex architectures might be required
- **Fitness sparsity**: As the fitness used as rewards dealing damage and last hitting, many behaviors that do not get close enough to the enemy have the same fitness, hence information about the quality of an individual is rare and requires already advanced policies

3.3 Our approach

3.3.1 Algorithms

We chose to train both CGP Individuals [Wilson 2018a] and NEAT Individuals [Stanley 2002a], as defined in the respective papers, in a MAP-Elites algorithm. They were implemented with **CartesianGeneticProgramming.jl**[30] for the CGP agent, and a custom NEAT agent developed at **NEAT.jl**[31] for this challenge.

As we believed the best way to find interesting individuals, either with NEAT or with CGP, was to explore the behavior space, we based the exploration on the **Illuminating search spaces by mapping elites** [Mouret 2015] work, implementing a MAP-Elites method to find policies of varying styles, focusing more on killing the enemy, on destroying the tower, or balancing both.

As the evaluation of a policy relies on a full game which can be heavily influenced by small perturbations, including time delays due to the Breezy software communication method, the behavior space can be seen as a noisy domain. Hence, we believe using the work of [Flageat 2020] on MAP-Elites in noisy domains using deep grids could improve

[30] `https://github.com/d9w/CartesianGeneticProgramming.jl`
[31] `https://github.com/d9w/NEAT.jl`

future work, since it was published too late for us to implement it.

3.3.2 Problem Modeling

Fitness Function

As in order to win a 1v1 a player needs to either kill the opponent twice or to take his first mid tower, we took into account the number of times we killed the opponent champion, and the opponent tower health in the individual evaluation. Moreover, to succeed in one of those two tasks one would also need good laning skills, so we also added the amount of gold collected (*net worth*), the number of last hits, and the number of denies. We also punished being killed and behaviors causing an early stopping as referenced in section 3.3.3.

The final fitness function is hence as follows:

$$\begin{aligned} fitness = {} & netWorth + 100 * lastHits + 100 * denies \\ & + 2000 * ratioTowerHealth + 1000 * nbKill \\ & - 250 * nbDeath - 500 * earlyStop \end{aligned}$$

Behavior Space

Our goal with a Quality-Diversity approach was to observe individuals with highly diverse behaviors to increase our chances of finding interesting ones. Therefore, we needed a characterization of the behavior space to differenciate playstyles. We described the behavior in a discretized 2-dimensional space featuring the *total damage made to the opponent champion* and the *percentage of the opponent tower health taken* as axes, since the objective was either to kill the opponent champion or to take his tower.

3.3.3 Reducing the evaluation cost

We quickly realized that one of the main difficulties of this challenge was to deal with the evaluation cost of the fitness. Indeed, in order to evaluate an agent we needed him to play a full Dota 2game, which can take up to several minutes in real time even with a tenfold speeding. Since Evolutionnary Algorithms require a large number of evaluations of individuals, we had to find a way to cope with this issue, which we tackled with parallel evaluation, early stopping, and simulator-assisted offspring generation.

Parallelization

In order to accelerate the evaluation of the individuals of a population, we tried to make it run in multiple parallel threads. Since Steam and Dota 2block us from running multiple Dota 2instances on the same machine, we tried to build Docker images based on SteamOS[32] and SteamCMD[33], which are supposed to facilitate the interaction with Steam. However due to multiple issues including missing dependencies in the Linux version of Dota 2, OS incompatibilities between the image and Breezy software, lack of Steam documentation on the topic or difficulties in running Dota 2headless, we did not manage to make the Docker images run Dota 2and Breezy software.

Early stopping

A simple but efficient way of reducing the evaluation time was to use an Early Stopping mechanism. We decided to stop the evaluation of an agent if he did not kill a creep after 5min of DOTA time. This functionality reduced the time loss due to static individual, and the evaluation length of "coward individuals" avoiding fights.

Neural-based simulator

Since an evaluation is time consuming, I developped a **Dota Simulator**[34] to predict game state evolutions from previous state and chosen action, using a neural network trained on more than 100 games played with a random policy.

The main part of the simulator was implemented in object-orient Python to process game data, remove constant features to reduce the size of the network, create and train the neural network, and act as a simulator with a Gym-like API. A Julia wrapper was then added, to allow plugging the simulator into the rest of the project.
Jupyter notebooks were also used to explore data and train the networks on the Google Colaboratory[35] platform.

UNet ([Ronneberger 2015], figure 3.5 is a deep neural network architecture that consists of a contracting path to extract feature information from spatial information, and an expansive path that combines spatial and feature information to avoid the need for the contracting path to pass it along with features.

3 neural networks of growing depths based on UNet were tried, with fully-connected layers in the contracting path, 1-dimensional convolution in the concatenation steps, and

[32] `https://store.steampowered.com/steamos/buildyourown?l=english`
[33] `https://developer.valvesoftware.com/wiki/SteamCMD`
[34] `https://github.com/TemplierPaul/Dota_Simulator`
[35] `https://research.google.com/colaboratory/local-runtimes.html?hl=en`

fully-connected layers when expanding.

A few simple feed-forward fully-connected networks were also tried, with no particular performance difference observed. Finally, a fully-connected network with few layers of many neurons was chosen.

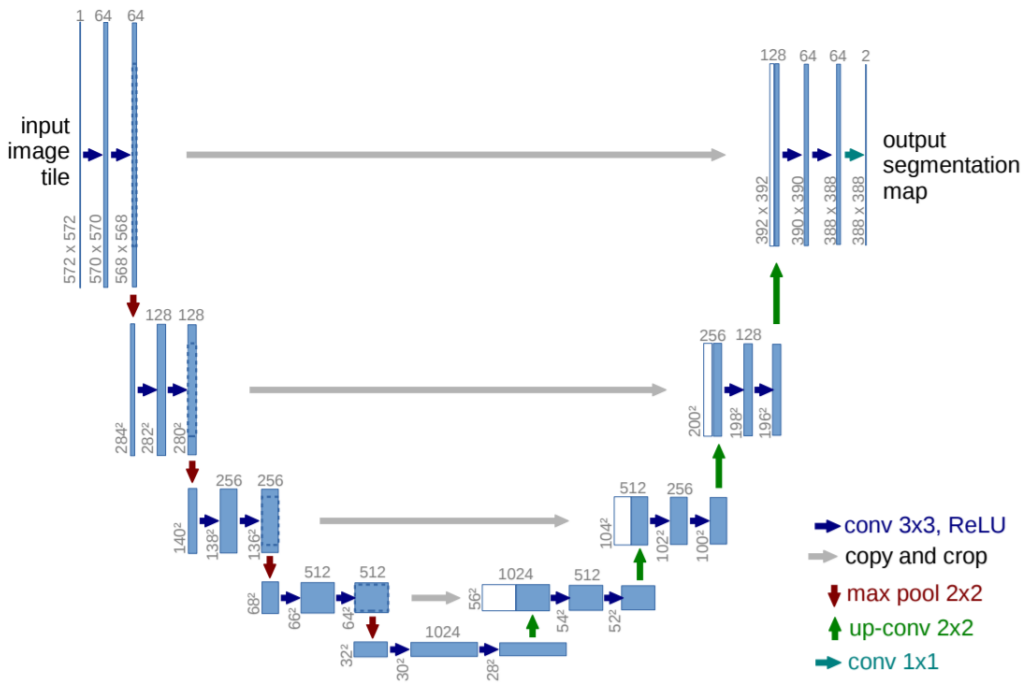

Figure 3.5: Example U-net architecture in [Ronneberger 2015], each blue box corresponding to a multi-channel feature map.

The simulator was used in the offspring generation phase to encourage diversity in the offspring generation, to only evaluate the most promising children and avoid unnecessary computation:

1. Offspring are generated for 10 times the size of the population.

2. 100 game steps are simulated for each offspring with the neural network, and the actions taken at each step are stored.

3. The actions taken by each individual are used to select the *population_size* most unique offspring based on behavior.

4. Those individuals are added to the new population and evaluated in the real game to compute the real fitness.

By doing so we saw many more mutations and/or crossovers, and taking the individuals the more distant from each other increased our chances of covering the behavior space.

3.4 Results

Results were announced at GECCO 2020 on July 12th by the organizers.
9 teams had registered to the competition, but only 2 -including us- provided code to be evaluated, among which we ranked first.

Figure 3.6: Competition results announcement at GECCO 2020

However, the algorithms we developed still showed poor performance against the game AI agent, only evolving basic capabilities like last-hitting or attacking the enemy hero, but dying very often due to over-aggressive behaviors, or still getting stuck far from the mid lane. Hence, we believe there is still a lot of work to be done on this subject, including adding read-write memory and training the agent for a longer period.

Chapter 4

Benchmarking Evolutionary Reinforcement Learning

`BERL.jl` [36], short for *Benchmarking Evolutionary Reinforcement Learning*, is a benchmarking tool created by my supervisor Dennis Wilson to compare the performance of Evolutionary Reinforcement Learning algorithms on a set of standard Reinforcement Learning benchmark tasks, for which I developed the architecture.

4.1 Motivation and structure

Even though benchmarks exist for Evolutionary Algorithms, such as COCO (COmparing Continuous Optimisers)[37], they do not include Evolutionary RL algorithms, and focus on optimization of static problems which would be represented by the white rows in figure 2.2.

While comparisons of multiple RL algorithms are often provided in works such as [Wilson 2018a] defining new methods, they rely on values taken from other publications on the same standard environments, but in potentially different settings. Language speed and actual computational cost may not be the same in every case, introducing bias in the comparison and making it harder to know what algorithm is better.
Non-open-source code, such as OpenAI's algorithms for Dota 2, also raises the question of how to compare algorithms on an equal footing, and implementations adding features that were not included in the initial paper, like NEAT, make it harder to compare the performance of the algorithm behind.

Finally, benchmarking these algorithms is a technical challenge since implementations

[36] `https://github.com/d9w/BERL.jl`
[37] `https://coco.gforge.inria.fr/`

and environments can vary in languages, from Python to C++, and may also be implemented in deprecated versions of these languages.

Hence, as the Evolutionary RL and Genetic Programming fields need better benchmarks [McDermott 2012] to compare algorithms on standardized tests, `BERL.jl` aims to bring a versatile framework that allows this comparison and can be incrementally completed as new methods appear.

The code of `BERL.jl` is voluntarily modular to allow adding more environments and new algorithms to the comparison. Hence, it relies on 3 blocks: algorithm `_setup` files[38], `BERLenv` structures[39], and the `start_berl` method[40] to link them.

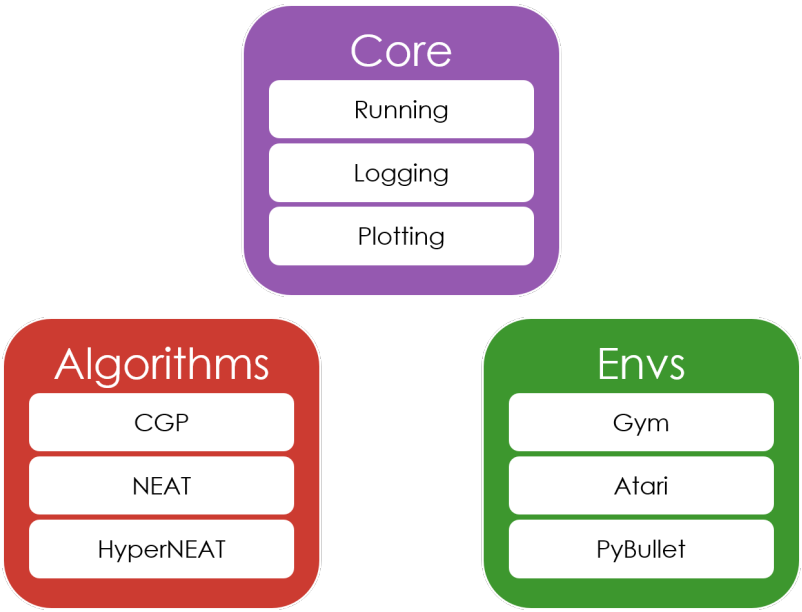

Figure 4.1: Architecture of `BERL.jl` in 3 blocks.

4.1.1 `_setup` files

Each algorithm requires a `_setup` Julia file that includes a function returning a `Cambrian.Evolution` object from a configuration dictionary and a fitness function, and a `process` function to evaluate the algorithm with a specific input. This allows to use all algorithms the same way in the rest of the code through an algorithm-agnostic interface.

Additionally, each algorithm has multiple YAML files[41], each defining the run parameters for an environment. The `_setup` and YAML files are all stored in the same folder

[38] https://github.com/d9w/BERL.jl/blob/master/src/algorithms/CGP/_setup.jl
[39] https://github.com/d9w/BERL.jl/blob/master/src/environments/BERL_env.jl
[40] https://github.com/d9w/BERL.jl/blob/master/src/core.jl
[41] https://github.com/d9w/BERL.jl/blob/master/src/algorithms/CGP/gym.yaml

named after the algorithm.

YAML files are used in this project for their compatibility with all languages, their simplicity of use, and their simplicity to read and write even for researchers not used to Julia.

4.1.2 BERLenv structure

In order to also present a homogeneous interface on the environments side, the BERLenv[42] abstract type is used to wrap each environment and make them all interact with the same functions. For each benchmark, a `mutable struct` is created with all required elements inside, and the function building it is in charge of starting the simulator and setting the parameters.

The `fitness` method takes a `Cambrian.Individual` and evaluates it, returning its fitness in this environment. Finally, the `new_gen` method can be used to reset the environment between generations or to give it a new random seed, making it stochastic between generations but deterministic for all evaluations at the same step.

Listing 4.1: Gym wrapper in `BERL.jl`

```
abstract type BERLenv end

mutable struct GymEnv <: BERLenv
    name::String
    memory::Array # Array of storable data
    gym
    env_name
    gen
    n_steps::Int64
end
```

In the example presented in listing 4.1, the `GymEnv` structure contains all fields required for the processing through `BERL.jl` , such as a name or generation number, along with the gym environment the algorithm will interact with.

4.1.3 start_berl

Finally, the `start_berl` function can be used to run an algorithm in a specific environment, creating all elements and making them interact.

[42] https://github.com/d9w/BERL.jl/blob/master/src/environments/BERL_env.jl

The `berl_run` function can be used to run all wanted algorithms on the same set of benchmarks, as selected in the configuration YAML files[43]. Each combination of an algorithm and an environment can be run multiple times in order to average the metrics, depending on the value of the `runs`[44] parameter.

4.2 Benchmark tasks

A few standard test environments have already been implemented into `BERL.jl` in order both to provide a first benchmark and to give an example of how to integrate RL environments into the architecture of the project.

Standard supervised learning tasks have also been added like the computation of XOR[45], or classification in the Iris dataset[46], to introduce easier tasks for testing purposes and to show more implementation examples.

For Gym, Atari, and PyBullet, the specific environment can be chosen in the YAML configuration file[47] but other run parameters will be the same, as defined in the YAML file for each algorithm[48].

4.2.1 Gym and PyBullet

Gym[49] is a toolkit developed by OpenAI to help in the development and testing of reinforcement learning algorithm. It implements a unified API to interact with various environments, from CartPole's pandulum balancing[50] to Ant's four-legged walk[51] (figure 4.2).

[43] `https://github.com/d9w/BERL.jl/tree/master/run_config`
[44] `https://github.com/d9w/BERL.jl/blob/master/run_config/algorithms.yaml`
[45] `https://en.wikipedia.org/wiki/Exclusive_or`
[46] `https://archive.ics.uci.edu/ml/datasets/iris`
[47] `https://github.com/d9w/BERL.jl/blob/master/run_config/environments.yaml`
[48] `https://github.com/d9w/BERL.jl/blob/master/src/algorithms/CGP/gym.yaml`
[49] `http://gym.openai.com/`
[50] `https://gym.openai.com/envs/CartPole-v0/`
[51] `http://gym.openai.com/envs/Ant-v2/`

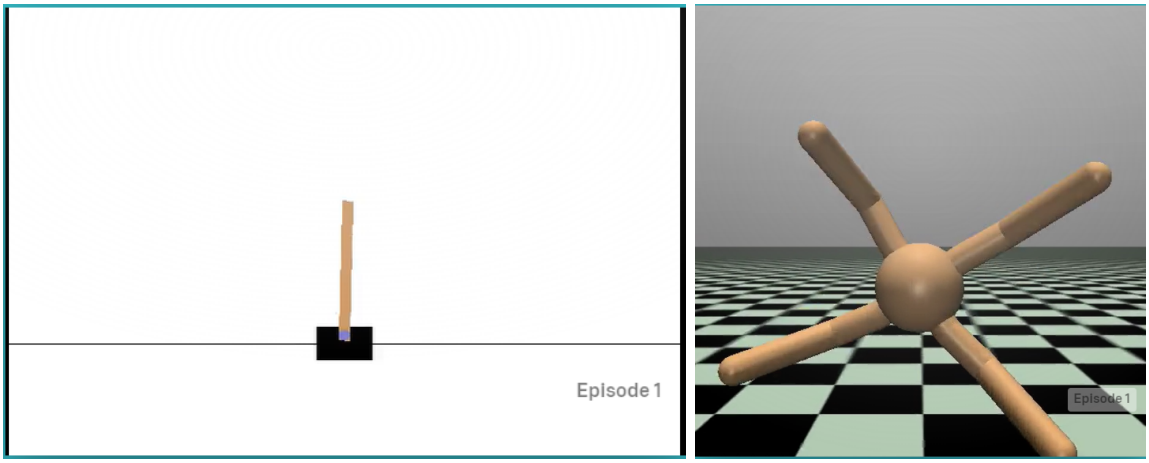

Figure 4.2: Cartpole (left) and Ant (right) environments in Gym
https://gym.openai.com/envs/CartPole-v1/ and http://gym.openai.com/envs/Ant-v2/

4.2.2 Atari RAM

Through Gym and the Arcade Learning Environment (ALE, [Bellemare 2012]), RL algorithms can be trained to play Atari games[52]. While many state-of-the-art reinforcement learning focus on frame-based interaction with the game, requiring GPU computation to process the images through convolutional network, the Atari environments based on RAM (Random-access memory) have been chosen for BERL.jl to reduce training costs and length while still providing complex learning environments. RAM stores all elements and variables of the environment, without introducing the complexity of image-based machine learning. Algorithms benchmarked through BERL.jl hence have to focus more on gameplay than on image processing.

4.3 Algorithms in BERL.jl

BERL.jl aims at regrouping all state-of-the-art evolutionary RL algorithms in order to compare them. As such, the list of modules we wish to benchmark is still to be completed, but CGP, NEAT and HyperNEAT have already been added. Future development will include Artificial Gene Regulatory Networks [Wilson 2018c], Tangled Program Graphs [Kelly 2018] and Grammatical Evolution [Ryan 1998].

[52] https://gym.openai.com/envs/#atari

4.3.1 CGP

The first algorithm added into BERL.jl is Cartesian Genetic Programming, which was proven efficient on Atari game-playing [Wilson 2018a]. The Julia implementation[53] used is based on Cambrian.jl and was developed by Dennis Wilson. The run parameters are based on the CGP parameters used on similar problems.

4.3.2 Neuroevolution algorithms

As there was no neuroevolution library in Julia to fit the purposes of BERL.jl , I developped NeuroEvolution.jl [54] to implement NEAT and HyperNEAT.

This library also lays the foundation for the development of more customized neuroevolution algorithms during the second part of my internship, by providing a modular platform based on Cambrian with multiple independent blocks.

The working principles of NeuroEvolution.jl are detailed in chapter 5.

[53] https://github.com/d9w/CartesianGeneticProgramming.jl
[54] https://github.com/TemplierPaul/NeuroEvolution.jl

Chapter 5

NeuroEvolution.jl

NeuroEvolution.jl[55] is a Julia library I developed to implement multiple neuroevolution algorithms for BERL. It includes **NEAT** (NeuroEvolution of Augmenting Topologies) and **HyperNEAT** (Hypercube-based NEAT), and an implementation of a direct weights optimization in a fix network using **CMA-ES** (Covariance matrix adaptation evolution strategy) is planned to complete it.

The concepts powering the NEAT algorithm may seem relatively simple and clear as described in the literature, making it a quite elegant solution. However, its implementation raises multiple complex issues, turning out to be rather intricate. The obstacles met when implementing part of it during an earlier personal project[56], and in the NEAT agent[57] developed for the Breezy competition (cf chapter 3) helped me design a more reliable structure and reduced the development time.

This chapter aims at presenting the original NEAT algorithm, both in the theory and in its implementation, and then explaining choices made during the design and development of NeuroEvolution.jl, its architecture, and the main principles ruling it.

5.1 Original NEAT algorithm

The NEAT algorithm relies on 3 main solutions to neuroevolution issues:

- **Crossover between different structures**: NEAT provides a genotype-to-phenotype mapping based on mutations (Fig. 2.4 in chapter 2).
- **Protection of topological innovation**: The speciation mechanism evaluates can-

[55] `https://github.com/TemplierPaul/NeuroEvolution.jl`
[56] `https://github.com/TemplierPaul/Genepy`
[57] `https://github.com/d9w/NEAT.jl`

didates by comparing them to similar architectures to allow new structures to mature without being instantly eliminated.

- **Minimization of structural complexity**: Nodes are added to a minimal initial structure in order to prevent networks from growing in size without benefits.

5.1.1 Crossover and Mutation

Mutation

Additionally to weight mutation, NEAT relies on 2 types of structural mutations: adding connections, and adding nodes (figure 5.1).

Adding a connection only requires creating a new gene between two neurons which were not previously linked. This new connection is assigned an incremented innovation number as a unique identifier across the whole population and all generations, which allows to track its existence in the NEAT run.

Adding a node requires splitting a connection into 2 new ones, by creating a new neuron and adding two new connections as two new genes.

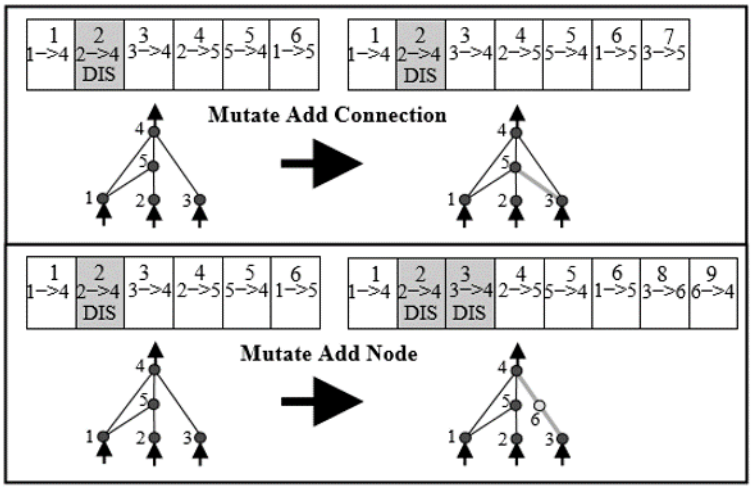

Figure 5.1: The two types of structural mutation in NEAT. Both types, adding a connection and adding a node, are illustrated with the genes above their phenotypes. The top number in each genome is the innovation number of that gene. New genes are assigned new increasingly higher innovation numbers. [Stanley 2002b]

Crossover

NEAT crossover relies on the comparison of the two genomes based on the genes innovation numbers as unique identifiers (figure 5.2). If both parents have a gene with the same innovation number (but possibly different weights), the algorithm selects one at random. If a gene only appears in one parent, it is selected if this parent has the better fitness.

In the case where multiple activation functions are possible (e.g. in a CPPN) the dictionaries linking each neuron to its function in each parent are merged, giving the priority to the fittest parent when a neuron appears in both.

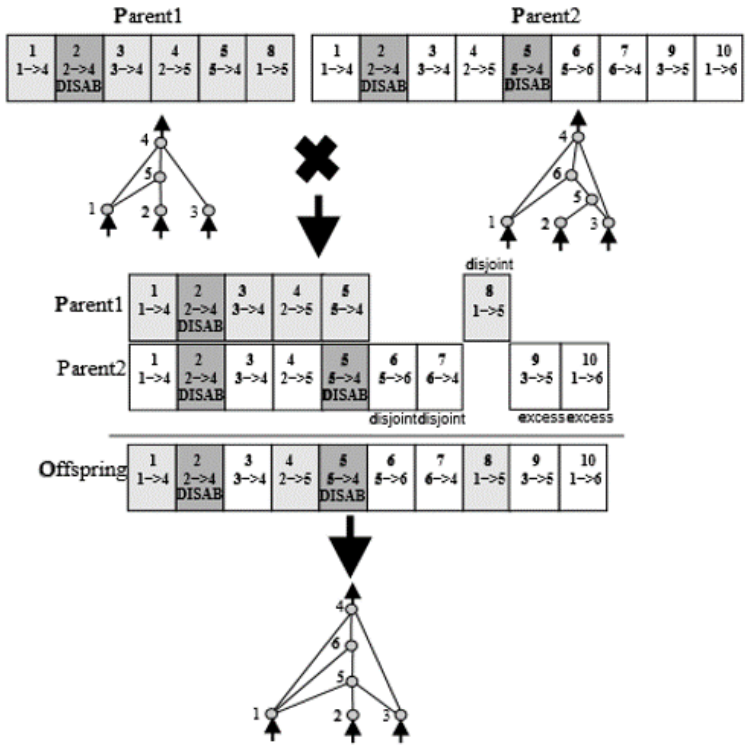

Figure 5.2: NEAT crossover mechanism matching up genomes for different network topologies using innovation numbers. Although Parent 1 and Parent 2 look different, their innovation numbers (shown at the top of each gene) tell us which genes match up with which. For matching genes, the one to keep is selected at random. [Stanley 2002b]

Mutation power

The *mutation_power* parameter implements a stronger mutation amplitude on the weights of the last genes of the genome, as they are more recent. It considers older genes

have had more time to be optimised, and that newer ones may require bigger steps to converge faster. It is not part of the original paper, but has been described on the author's website and in his code.

5.1.2 Speciation

Stolen children

The stolen children feature consists in removing a few offsprings from bad species and adding them to good ones, making the search more greedy. It is not part of the original paper, but has been described on the author's website and in his code.

Elitism

Elitism is the mechanism keeping the best individuals (*elites*) from a generation to the next one, ensuring the best solution is not lost once encountered. There is no elitism in the original implementation, although it has been added in the latest version of my library as an optional feature, for comparison purposes with CGP which uses it.

5.2 Implementing NEAT

I developed the NEAT algorithm based on the original papers [Stanley 2002a] and [Stanley 2002b], taking into account the remarks and indications of the author on his website[58]. The reorganized version[59] of the original C++ implementation of NEAT also provided insight into the algorithm, although some parts were approached differently, taking the papers as reference.

5.2.1 NEAT Individual

Based on `Cambrian.jl`, `NeuroEvolution.jl` uses an abstract type `NEATIndiv` from which the structure `NEATIndividual` inherits. This allows to build new NEAT-like individuals using the same methods with additional features, like HyperNEAT which relies on the NEAT algorithm for evolution, but integrates a different evaluation procedure. `NEATIndiv` inherits from `Cambrian.Individual` to use Cambrian's architecture.

A `NEATIndividual` object contains a dictionary mapping an innovation number to a gene, which represents a connection between two neurons, a list of neurons defined by

[58] `https://www.cs.ucf.edu/~kstanley/neat.html#FAQ2`
[59] `https://github.com/FernandoTorres/NEAT`

their positions, a dictionary mapping each neuron position to the activation function of the corresponding neuron, a `Network` object containing the neural network built from the genome, and a list of the fitness of the individual as required by `Cambrian.jl`.

Listing 5.1: NEAT Individual

```
abstract type NEATIndiv <: Cambrian.Individual end

mutable struct NEATIndividual <: NEATIndiv
    genes::Dict
    fitness::Array{Float64}
    neuron_pos::Array{Float64}
    network::Network
    activ_functions::Dict
end
```

Source code at `https://github.com/TemplierPaul/NeuroEvolution.jl/blob/master/src/individual.jl`

5.2.2 Neuron position and Execution order

On of the main issues encountered when implementing NEAT in previous personal projects is the order of computation of the neurons. Due to the way it grows, the NEAT network can build recurrent connections, including bidirectional relationships with one connection from neuron A to neuron B, and one connection from B to A. Then, propagating depth from the output neurons to make sure all previous neuron values have been computed leads to an infinite loop between A and B, each one needing to be computed before the other.

To prevent that issue and reduce the time needed to build the network, this implementation of NEAT was inspired by the work of [Wilson 2018d] in which the position of a CGP node is described as a floating point value. The execution order is then available by design since they can be executed sequentially based on their positions. Recurrent connections are defined by destination neurons being before origins, which makes possible to allow or block them easily at mutation.

This representation also removes the propagation time concept, making the NEAT network closer to standard neural networks.

The position of a neuron is defined as follows:

- All input neurons are placed with negative integer positions.
- The bias neuron (constant equal to 1) is placed at 0.

- All output neurons are placed at positive integer positions, starting at 1.

- All other neurons are placed with positions in $]0, 1[$.

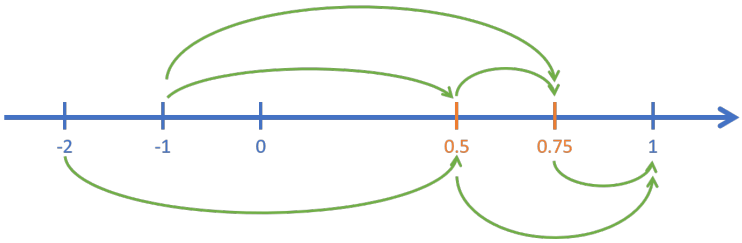

Figure 5.3: Example of NEAT individual as neurons defined by position and feed-forward connections, with 2 input features, 1 output neuron and 2 hidden neurons.

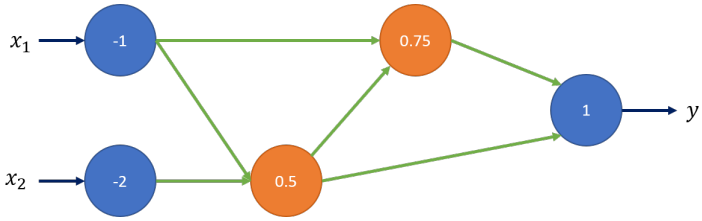

Figure 5.4: Neural Network generated from the above individual.

Neurons added through mutation of a connection are placed at a random position between the origin node of the connection at p_o and the destination node at p_d, so that the order of execution is not modified.
The position of the new neuron is therefore randomly drawn in $]p_o, p_d[\cap]0, 1[$.

Because of this structure, and to avoid edge cases, connections between 2 output neurons are forbidden. Connections towards input and bias neurons are blocked too, since their values are fixed to the input vector and to 1 respectively.

When computing the network's output, its neurons are sorted by ascending position. Since the position determines the neuron type, choosing the update type is simple:

Listing 5.2: NEAT network processing

```
for p in indiv.neuron\_pos
    if p < 0 # Input neurons
        indiv.network.neurons[p].output =
                    last\_features[Int(-p)]
    elseif p == 0 # Bias neuron
        indiv.network.neurons[p].output = 1
    else # Hidden and output neurons
        compute!(indiv.network.neurons[p],
                    indiv.network.neurons)
    end
end
```

Source code at `https://github.com/TemplierPaul/NeuroEvolution.jl/blob/master/src/process.jl`

5.2.3 Run configuration

The parameters to run NEAT are defined in a YAML file[60] that is read at runtime. It includes:

- Cambrian settings: population size, number of generations
- Input and output size of the network
- Speciation parameters
- Mutation probabilities for each mutation type
- Available activation functions including sigmoid, trigonometric functions, absolute value, identity, or ReLU.

One of the available activation functions is randomly selected when a new neuron is created, allowing the implementation of a Compositional Pattern-Producing Network (CPPN) by making multiple ones available. The default activation function for NEAT is the sigmoid.

[60] `https://github.com/TemplierPaul/NeuroEvolution.jl/blob/master/cfg/test.yaml`

5.2.4 Crossover and innovation number

A usually delicate part of the implementation of NEAT lies in the management of the innovation number, which tracks genes across the population and guarantees the same gene is recognized on both sides during crossover.

In this implementation, and according to Cambrian's structure, a configuration dictionary is passed across the algorithm to each method requiring run parameters, built from the YAML configuration file. The innovation number is stored inside and updated after each mutation adding a gene.

Using a dictionary in the individual to store each gene linked to its innovation number allows to compare genomes more efficiently at crossover than an array of genes.

5.2.5 Speciation threshold

NEAT relies on speciation to update its population:

- If individuals are closer than the speciation threshold, they get put in the same species
- The fitness of each individual is computed with explicit fitness sharing, hence divided by the size of the population of their species
- Offspring are distributed among species depending on the mean (or the max) fitness in the current members of the species
- Reproduction is computed in each species by selecting parents randomly in the top members of the species (or based on their individual fitness in a tournament selection) and applying either mutation or crossover.
- All offspring are assigned to the closest species

However as generations occur the distance between genomes grows and the algorithm tends to assign a different species to each individual, hence causing hundreds of species to appear. In order to keep a stable number of species, K.O.Stanley introduces on his website a changing threshold, increasing of decreasing by a fix amount in order to keep the number of active species stable.

I introduced a slightly modified version of this algorithm as `update_threshold!`, which modifies the threshold by an amount scaled on the ratio:

$$\frac{current_species_number}{wanted_species_number}$$

This solution yields a very stable number of species, although usually higher than the wanted number.

5.2.6 Compositional Pattern-Producing Networks

Compositional Pattern-Producing Networks [Stanley], or CPPNs, are neural networks built with multiple possible activation functions, including `sin`, `cos`, or `tanh`, producing patterns with symmetries.

CPPNs can be evolved with CPPN-NEAT, a variant of NEAT where multiple activation functions are used, being determined when the node is created in a mutation. This architecture is used in HyperNEAT so that the final network can benefit from the symmetries.

As a standard NEAT is a CPPN with a single activation function, I implemented the possibility to have CPPNs in the base NEAT code, and the activation functions can be selected in the YAML config file[61].

Listing 5.3: Available activation functions in my implementation

```
# Activation functions
sigmoid: true
sin: false
cos: false
ReLU: false
tanh: false
abs: false
identity: false
gauss: false
```

Source code at `https://github.com/TemplierPaul/NeuroEvolution.jl/blob/master/cfg/test.yaml`

5.2.7 NEAT in a standard Genetic Algorithm

The goal of using Cambrian as a framework in this implementation also was to allow for a high compatibility with other algorithm so that the coding blocks defined for NEAT could be reused in classical evolutionary algorithms, from a Genetic Algorithm (GA) to a Quality-Diversity method as envisioned in the Breezy competition (cf chapter 3).

Hence, I finally implemented a simple wrapper to use the NEAT Individual and its

[61] `https://github.com/TemplierPaul/NeuroEvolution.jl/blob/master/cfg/test.yaml`

mutation and crossover operators in a standard Genetic Algorithm, ignoring the specific species-based populate function for the standard Cambrian `ga_populate` function.

This compatibility with other Cambrian algorithms allows to easily use the NEAT individual in other settings such as Quality-Diversity methods or open-ended evolution, thus helping in new algorithms research.

5.3 HyperNEAT

HyperNEAT [Stanley 2009] introduces indirect encoding by using a CPPN (Compositional Pattern-Producing Network [Stanley]) to determine the weights of a neural network with a fix architecture. Since the CPPN can be evolved with the NEAT algorithm for HyperNEAT, most evolutionary methods in NeuroEvolution.jl can be reused.
However, instead of evaluating the evolved network (the CPPN) to solve the problem, the CPPN is used to determine the weights of a network with fixed architecture, referenced here as the `final network`.

5.3.1 HyperNEAT Individual

HyperNEAT was added to NeuroEvolution.jl by creating a structure `HyperNEATIndiv`, inheriting from `NEATIndividual` like `NEATIndiv`. Thus, functions affecting the evolved network (e.g. mutation, crossover, populate) take as arguments any object inheriting from the abstract type `NEATIndividual`, and are blind to whether NEAT or HyperNEAT is used. Conversely, other functions related to the evaluation of an individual (e.g. evaluate, build, process) are algorithm-specific and are defined for each individual type.

Listing 5.4: HyperNEAT Individual

```
mutable struct HyperNEATIndividual <: NEATIndiv
    genes::Dict
    fitness::Array{Float64}
    neuron\_pos::Array{Float64}
    network::Network
    activ\_functions::Dict
    hn\_net::GridNetwork
end
```

Source code at `https://github.com/TemplierPaul/NeuroEvolution.jl/blob/master/src/HyperNEAT.jl`

`HyperNEATIndividual` uses the same structure as `NEATIndividual`, adding a `hn_network`

field to store the generated final network described in subsection 5.3.2.

5.3.2 Final network definition

The final network is a feed-forward, fully-connected neural network with a fixed architecture. It is build from an array of `Layer` objects, each containing the output values of a neurons layer, and an array of `Connection` objects linking 2 layers and implementing weights.

Listing 5.5: HyperNEAT final network

```
mutable struct GridNetwork
    layers::Array{Layer}
    connections::Array{Connection}
end
```

Source code at `https://github.com/TemplierPaul/NeuroEvolution.jl/blob/master/src/HyperNEAT.jl`

The `build!` method creates the final network and sets its weighs with the CPPN, while the `process!` method computes the output of the final network. This approach allows to only build the network once, but increases complexity by adding a step to place between mutation and evaluation.

5.3.3 Run configuration

Using HyperNEAT requires a specific YAML file[62] which adds information about hidden layers and structure of the generated network to the standard NEAT parameters.

Although the default architecture for the final networks only links a layer of neurons to the previous one, toggling the `hn_link_all_layers` parameter to `true` creates a connection between each layer and all the previous ones, for a more complex architecture. Using a CPPN for weights generation makes it possible to use such architecture without expanding the search space, contrary to direct encoding.

5.4 Future developments

As NeuroEvolution.jl still seems to struggle on harder RL tasks in BERL compared to CGP, some additional features are planned for future development. This includes using the

[62] `https://github.com/TemplierPaul/NeuroEvolution.jl/blob/master/cfg/hyperneat.yaml`

HyperNEAT CPPN to generate biases in the final network as well as connection weights, and adding stolen children to make the search more greedy or implementing mutation power, which have been described in the original code but not in the published work.

Chapter 6

First results and Future work

Although BERL is still in development and NeuroEvolution.jl is being tuned to reach decent performance, a few experiments have been conducted.

6.1 NEAT

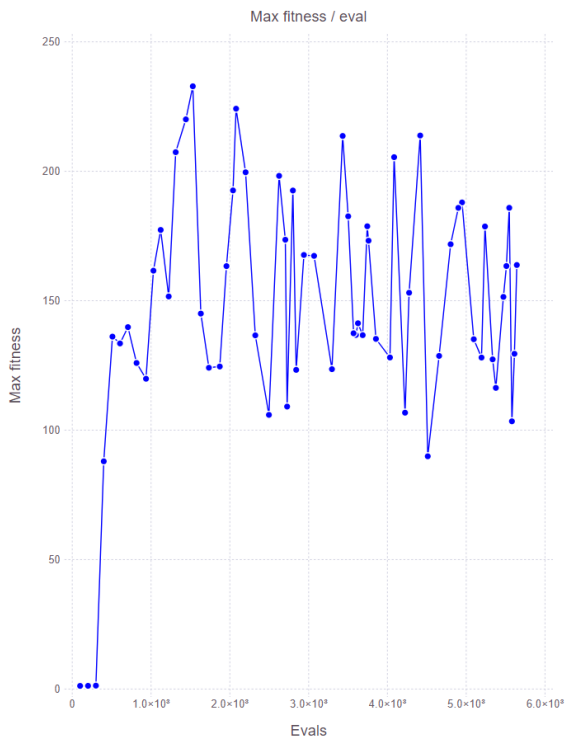

Figure 6.1: NEAT run on an AntBullet environment: evolution of max fitness wrt the number of evaluations

A first experiment of NEAT on an AntBullet environment shows the algorithm learns and its maximum fitness improves, although it quickly stagnates as the best solutions seem to be forgotten between generations. This observation led to the implementation of elitism (cf section 5.1.2) to see if it improves performance in future experiments.

6.2 Algorithm comparison

We compared NEAT and CGP on the same CartPole benchmark. As the random seed of the environment is incremented at every generation, all individuals of a population are evaluated on the same environment but it changes between generations. This leads to a stochastic environment, hence even with elitism maximum fitness can decrease as the best individual in a generation may not perform as well on a slightly different environment.

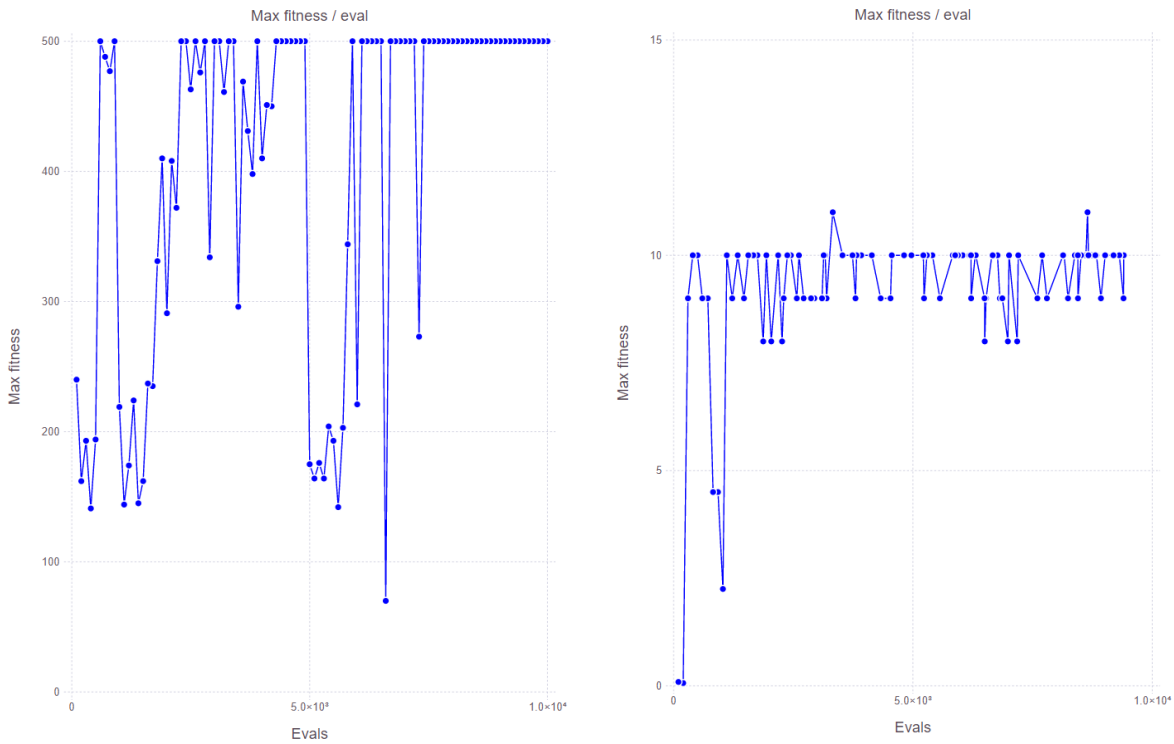

Figure 6.2: CGP (left) and NEAT (right) runs on an Cartpole environment: evolution of max fitness wrt the number of evaluations

Although the environment is stochastic, hence harder to learn, CGP finally reaches a fitness of 100, solving the problem. However, NEAT (without elites) seems to struggle, showing further parameter tuning and features development are still required.

6.3 Future work

In the second part of this internship, I will keep developing `NeuroEvolution.jl` and `BERL.jl` jointly in order both to provide more stable and efficient tools, and to use them as a platform to design, implement and benchmark new algorithms.

In this regard, the following points are currently forecast to be essential components of my research work in the next months:

- Adaptation of NE.jl and BERL.jl to Cambrian v0.2.0, which redefines the structure for a more robust and modular implementation
- Completion of the NEAT implementation with additional features as observed in the

original code

- Development of a CMA-ES-optimized neural network in NE.jl for weight-only evolution
- Study of alternative indirect encoding algorithms, e.g. Artificial Gene Regulatory Networks [Wilson 2018c]
- Introduction of other evolutionary algorithmic mechanisms such as coevolution or Quality-Diversity in the NEAT structure
- Addition of gradient-based methods for weights optimization in a neuroevolution architecture

Conclusion

While comparing evolutionary RL algorithms is a critical part of the understanding of the state of the art and the development of new algorithms in this field, a standardized benchmarking framework has not emerged yet, which is the goal of the development of `BERL.jl` .

More than in the development of the architecture, the challenge also relies in the choices of the tasks to use for the comparison. Standard environments such as Gym's CartPole provide a well-known benchmark, but have been solved by multiple methods. On the other hand, games like Dota 2 are arduous to tackle both for the algorithms, due to the very high complexity of the game and its short- and long-term strategies, and for the researchers willing to use them because of the challenges raising from the interactions with the game. Even though finding a method solving Dota 2 can be interesting for research, its capacity to be applied to other problems, especially real-world tasks, still requires a standardized benchmark.

On the personal side, this internship has given me the opportunity to learn more about the field of Evolutionary Computation, which I had been attracted toward for many years. I got to discover both the quality and the diversity of the work on this topic, and more generally on Artificial Life, through reading bibliography and attending conferences.

Neuroevolution methods have opened to me a different approach to Reinforcement Learning and its questions, building new techniques on algorithmic ideas discovered decades ago and based on natural behaviors evolved for millions of years. The SuReLI team has also been a great inspiration source, joining multiple approaches and competences in the same goal of improving RL research, in a skilled and enjoyable working environment.

I have thoroughly enjoyed working on this project, and I look forward to keeping working with Dennis Wilson and Emmanuel Rachelson in the next years, since I am set to continue this internship with a PhD on bio-inspired methods for Artificial Neural Network learning. With both of them as supervisors, we will tackle the issue of finding policies that help networks actually learn instead of solely optimize for a specific task, based on biological neuronal models and notably building on previous work in Neuromodulation [Wilson 2018b].

Bibliography

[Badia 2020] Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo et Charles Blundell. *Agent57: Outperforming the Atari Human Benchmark*, 2020.

[Bellemare 2012] Marc G. Bellemare, Yavar Naddaf, Joel Veness et Michael Bowling. *The Arcade Learning Environment: An Evaluation Platform for General Agents*. CoRR, vol. abs/1207.4708, 2012.

[Bellman 1957] Richard Bellman. *A Markovian Decision Process*. Indiana Univ. Math. J., vol. 6, pages 679–684, 1957.

[Bezanson 2017] Jeff Bezanson, Alan Edelman, Stefan Karpinski et Viral B. Shah. *Julia: A Fresh Approach to Numerical Computing*. SIAM Review, vol. 59, no. 1, pages 65–98, 2017.

[Bre 2017] Facundo Bre, Juan Gimenez et Víctor Fachinotti. *Prediction of wind pressure coefficients on building surfaces using Artificial Neural Networks*. Energy and Buildings, vol. 158, 11 2017.

[Cully 2018] A. Cully et Y. Demiris. *Quality and Diversity Optimization: A Unifying Modular Framework*. IEEE Transactions on Evolutionary Computation, vol. 22, no. 2, pages 245–259, 2018.

[Darwin 1859] Charles Darwin. The origin of species by means of natural selection: or, the preservation of favored races in the struggle for life. 1859.

[Doncieux 2019] Stephane Doncieux, Alban Laflaquière et Alexandre Coninx. *Novelty search: a theoretical perspective*. In Proceedings of the Genetic and Evolutionary Computation Conference, pages 99–106, 2019.

[Flageat 2020] Manon Flageat et Antoine Cully. *Fast and stable MAP-Elites in noisy domains using deep grids*, 2020.

[Hansen 1996] Nikolaus Hansen et Andreas Ostermeier. *Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation.* In Proceedings of IEEE international conference on evolutionary computation, pages 312–317. IEEE, 1996.

[Kelly 2018] Stephen Kelly et Malcolm Heywood. *Emergent Tangled Program Graphs in Multi-Task Learning.* In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18, pages 5294–5298. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[Laird 2001] John E Laird et Michael Van Lent. *Human-Level AI's Killer Application Interactive Computer Games.* AI Magazine Volume 22 Number 2 (2001) (© AAAI), 2001.

[Lehman 2011a] J. Lehman et K. O. Stanley. *Abandoning Objectives: Evolution Through the Search for Novelty Alone.* Evolutionary Computation, vol. 19, no. 2, pages 189–223, 2011.

[Lehman 2011b] Joel Lehman et Kenneth O Stanley. *Evolving a diversity of virtual creatures through novelty search and local competition.* In Proceedings of the 13th annual conference on Genetic and evolutionary computation, pages 211–218, 2011.

[McDermott 2012] James McDermott, David R White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong*et al.* *Genetic programming needs better benchmarks.* In Proceedings of the 14th annual conference on Genetic and evolutionary computation, pages 791–798, 2012.

[Mendel 1865] Gregor Mendel. *EXPERIMENTS IN PLANT HYBRIDIZATION.* Psychological Review, 1865.

[Mitchell 1998] Melanie Mitchell. An introduction to genetic algorithms. MIT press, 1998.

[Mnih 2013] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra et Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning*, 2013.

[Mnih 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski*et al.* *Human-level control through deep reinforcement learning.* nature, vol. 518, no. 7540, pages 529–533, 2015.

[Mouret 2015] Jean-Baptiste Mouret et Jeff Clune. *Illuminating search spaces by mapping elites.* 2015.

[O'Reilly 2019] Una-May O'Reilly et Erik Hemberg. *Introduction to Genetic Programming*. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '19, page 710–725, New York, NY, USA, 2019. Association for Computing Machinery.

[Robbins 1951] Herbert Robbins et Sutton Monro. *A Stochastic Approximation Method*. Ann. Math. Statist., vol. 22, no. 3, pages 400–407, 09 1951.

[Ronneberger 2015] Olaf Ronneberger, Philipp Fischer et Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*, 2015.

[Rosenblatt 1958] F. Rosenblatt. *The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain*. Psychological Review, pages 65–386, 1958.

[Ryan 1998] Conor Ryan, J. J. Collins et Michael O'Neill. *Grammatical Evolution: Evolving Programs for an Arbitrary Language*. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer et Terence C. Fogarty, editeurs, Proceedings of the First European Workshop on Genetic Programming, volume 1391 of *LNCS*, pages 83–96, Paris, 14-15 Avril 1998. Springer-Verlag.

[Salimans 2017] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor et Ilya Sutskever. *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. 3 2017.

[Stanley] Kenneth O. Stanley. *Compositional pattern producing networks: A novel abstraction of development*.

[Stanley 2002a] Kenneth O Stanley et Risto Miikkulainen. *Efficient evolution of neural network topologies*. In Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600), volume 2, pages 1757–1762. IEEE, 2002.

[Stanley 2002b] Kenneth O Stanley et Risto Miikkulainen. *Evolving neural networks through augmenting topologies*. Evolutionary computation, vol. 10, no. 2, pages 99–127, 2002.

[Stanley 2005] Kenneth O Stanley, Bobby D Bryant et Risto Miikkulainen. *Real-Time Neuroevolution in the NERO Video Game*, 2005.

[Stanley 2009] Kenneth O Stanley, David D ' Ambrosio et Jason Gauci. *A Hypercube-Based Indirect Encoding for Evolving Large-Scale Neural Networks*, 2009.

[Such 2017] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley et Jeff Clune. *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*. CoRR, vol. abs/1712.06567, 2017.

[Whiteson 2005] Shimon Whiteson, Peter Stone, Kenneth O Stanley, Risto Miikkulainen et Nate Kohl. *Automatic Feature Selection in Neuroevolution*, 2005.

[Wierstra 2011] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun et Jürgen Schmidhuber. *Natural Evolution Strategies*, 2011.

[Wikipedia contributors 2020a] Wikipedia contributors. *CMA-ES — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=CMA-ES&oldid=967047119`, 2020. [Online; accessed 20-August-2020].

[Wikipedia contributors 2020b] Wikipedia contributors. *Dota 2 — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Dota_2&oldid=966599163`, 2020. [Online; accessed 20-August-2020].

[Wilson 2018a] Dennis G. Wilson, Sylvain Cussat-Blanc, Hervé Luga et Julian F. Miller. *Evolving simple programs for playing Atari games*. CoRR, vol. abs/1806.05695, 2018.

[Wilson 2018b] Dennis G Wilson, Sylvain Cussat-Blanc, Hervé Luga et Kyle Harrington. *Neuromodulated Learning in Deep Neural Networks*, 2018.

[Wilson 2018c] Dennis G Wilson, Kyle Harrington, Sylvain Cussat-Blanc et Hervé Luga. *Evolving Differentiable Gene Regulatory Networks*, 2018.

[Wilson 2018d] DG Wilson, Julian F. Miller, Sylvain Cussat-Blanc et Hervé Luga. *Positional Cartesian Genetic Programming*, 2018.