Diploma project

EVOLVING SWITCH NEURON NETWORKS

Christodoulos Hajdichristodoulou

University of Cyprus

Department of Computer Science

May 2021

University of Cyprus

Computer Science Department

EVOLVING SWICH NEURON NETWORKS

Christodoulos Hadjichristodoulou

Supervisors:

Dr. Chris Christodoulou

Dr. Vassilis Vassiliades

This diploma project is submitted to the department for partial fulfillment of the requirements for the degree of Bachelor of Science at the University of Cyprus

May 2021

Acknowledgments

This whole undertaking would be impossible to get to this stage without the supervision of Dr. Chris Christodoulou. The continuous support and knowledge he has provided throughout this past year have proven to be extremely useful in my endeavors.

I am also deeply thankful to Dr. Vassilis Vassiliades, who has guided me through some of the hardest parts of this project and was always happy to share his insights and expertise on the subject.

ABSTRACT

Neural networks have seen widespread use during the past few decades for solving a diverse array of problems, both in the field of computer science as well as several other fields, such as biology and finance. Many researchers have devoted a lot of effort into studying these networks and brewing novel ideas in order to improve their performance. One of those novelties is the switch neuron, a type of neuron with a unique behavior pattern that will be the main feature of this diploma thesis.

In this project an effort is made to automate the process of finding the optimal architecture and weights for a network containing switch neurons, through the utilization of already existing modern neuroevolutionary algorithms, specifically the NeuroEvolution of Augmenting Topologies algorithm and novelty search. The success of this process is then gauged by comparing the performance of the resulting networks at a reinforcement learning environment with that of networks which were manually designed to be optimal. The reinforcement learning tasks used are the T-Maze and the binary association task.

The results of this experiment are mixed. For the simplest one of the tasks – the T-Maze - the evolutionary algorithms were able to find an optimal network within a few generations and with a less complex architecture than the ones designed by hand. However, for the rest of the problems both algorithms struggle to evolve an optimal network even after a substantial amount of evaluations and reaching a high level of complexity in the population's architectures.

Contents

1 Introduction

In my thesis, I expand on the work done by Vassiliades and Christodoulou (2016) on a novel type of neuron proposed by them, the switch neuron. The switch neuron aims to provide new utility to artificial neural networks by propagating signals in a unique way which is explained in detail in section 2.1. In their work, Vassiliades and Christodoulou (2016) successfully used this type of neuron to design artificial neural networks that solve the T-Maze problem and the binary association problem in a standard reinforcement learning environment. The task of this project is to expand genetic algorithms for neural networks – specifically NeuroEvolution of Augmenting Technologies (NEAT) (Stanley & Miikkulainen, 2002) – along with certain network encoding schemes and alternative search methods with the ability to use switch neurons in their networks' architectures. The two encoding schemes used are the direct encoding (the one NEAT uses by default) and the map-based encoding (Doncieux, Mouret, Pinville, Tonelli, & Girard, 2011), which introduces the idea of creating neural networks with maps of neurons as the building block instead of isolated neurons. The novelty search algorithm (Lehman & Stanley, 2008) is also incorporated in our experiments, which is a search method that disregards the objective function and focuses solely on discovering novel patterns . Our goal with this approach is to remove the necessity for a designer to manually construct the network's architecture, which could possibly be very time-consuming. We judge the viability of the evolved networks by testing them on the tasks mentioned above and comparing their results with those of neural networks designed by hand.

The first pillar upon which my work stands on is reinforcement learning. Reinforcement learning is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the cumulative reward. A basic reinforcement learning agent interacts with its environment in discrete time steps. At each time step, the agent receives the current state and reward. It then chooses an action from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state and the reward associated with the transition is determined. The goal of a reinforcement learning agent is to learn a policy which maximizes the expected cumulative reward. Figure 1 illustrates a bird's eye view of the described framework. An interested reader is referred to *Reinforcement Learning: A Survey* (Kaelbling, Littman, & Moore, 1996) for a more in-depth introduction.

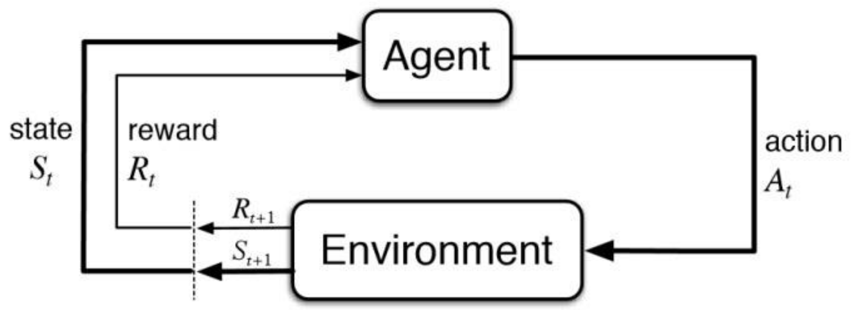


Figure 1: A basic representation of the reinforcement learning framework

Neural networks are commonly used as agents in a reinforcement learning environment, aside from all their numerous other applications. Neural networks are a set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns. A network's primary components are neurons, which act as computation units and work together to produce the final output. Neurons are connected with each other and a weight value is assigned to each connection. This weight determines how much influence the output of one neuron will have on the output of the other. By adjusting the weights of these connections in respect to some desired output, the network is essentially learning – approximating the function between the inputs and the outputs. A simple neural network is depicted in figure 2.

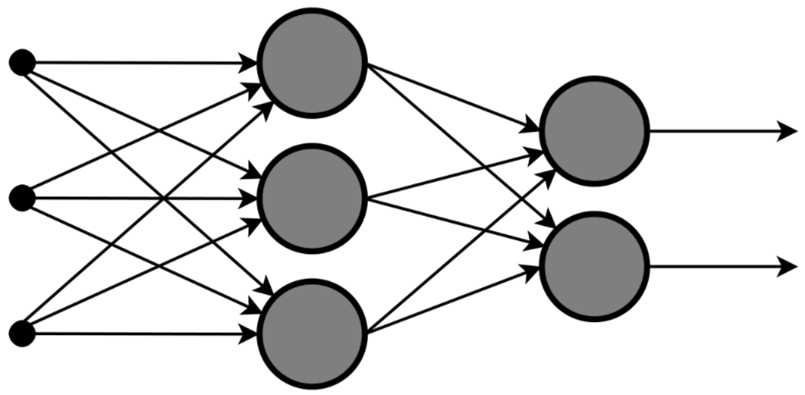


Figure 2: An illustration of an arbitrary neural network.

The third major theme in this project are genetic algorithms and more specifically neuro-evolution. A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring for the next generation. For this project, we focus on algorithms designed to evolve neural networks. Conventional learning is taking place with a fixed topology and the network tries to adapt its weights to represent the data provided to it as best as possible. A genetic algorithm, however, is also tasked with figuring out the optimal topology for the given problem, allowing for automatic discovery of the topology rather relying on the experience and insights of the designer. A more detailed introduction to genetic algorithms would be *Genetic Algorithms: Concepts and Applications* (Man, Tang, & Kwong, 1996)

In the following pages, the central components the experiment (switch neurons, NEAT, map-based encoding and novelty search) are discussed in detail through an overview of the relevant literature. In section 3 the specifics of the implementation of these concepts are presented as well as a few points where some extra attention is needed so some traps can be avoided. The procedure needed to recreate the experiments is also explained. Further down, in section 4, the results of the various experiments we have run are and our interpretation of them are presented.

2 Literature Review

2.1 Switch neurons

As mentioned above, the switch neuron plays a central role in this project. In this section, the definition of the switch neuron is introduced, as described by Vassiliades and Christodoulou (2016).

2.1.1 Context

The usual formulation of an artificial neuron involves the integration of incoming signals and parameters through an accumulation or integration function resulting in the neuron's activity. This activity is then fed through an activation function resulting in the neuron's output. In order to define the switch neuron model, the following extensions to the traditional model are adopted (Soltoggio, Bullinaria, Mattiussi, Durr, & Floreano, 2008) (Vassiliades & Christodoulou, 2016):

- A new type of connection is introduced, the modulatory. This type of connection allows for neurons to emit modulatory signals which affect the synaptic plasticity of a target neuron.
- Each neuron has an internal value for a modulatory activation in addition to its standard activation.

In this context, a neuron consists of two parts that are responsible for the calculation and storage of its standard activation and modulatory activation:

$$n_i := \langle s_i^{(std)}, s_i^{(mod)} \rangle$$

where $s_i^{(std)}$ and $s_i^{(mod)}$ are tuples that hold the parameters for computing and storing the standard output and the modulatory output of the neuron respectively.

2.1.2 The switch neuron

The core concept of switch neurons is that, in contrast with traditional neurons, the incoming signals are not integrated, but instead only one is allowed to be propagated forward, while all the others are blocked (Vassiliades & Christodoulou, 2016). Modulatory signals change the level of modulatory activity of the switch neuron and the level of modulatory activity determines from which of its incoming connections the signal is allowed to be propagated forward.

The integration function of the switch neuron calculates its standard activation as

$$a_i^{(std)}(t) = w_{ji} \cdot y_i^{(std)}(t - d_{ji})$$

where $a_i^{(std)}(t)$ is the standard activation of the switch neuron at time t, $y_i^{(std)}(t)$ is the standard output of the j^{th} presynaptic node at time t, w_{ji} is the weight of the (standard) connection and d_{ij} is the delay of the connection. The jth index is selected according to:

$$j = \lfloor n \cdot y_i^{(mod)}(t) \rfloor$$

where $n \in N+$ is the number of incoming standard connections and $y_i^{(mod)}(t) \in [0, 1)$ is the modulatory output of switch neuron i at time t. This equation effectively partitions the range of modulatory output [0, 1) into n homogeneous intervals, with the size of each interval being equal to 1/n. It also implies an 'order relation' between the connection indices, i.e., connection k comes after connection $k - 1$, meaning that if currently connection $k - 1$ is selected and the next modulatory output is at most 1/n greater than the current, then the next selected connection will be connection k. This means that the modulatory signals *indirectly* decide which signal is propagated by the switch neuron by changing its modulatory activity. The switch neuron's modulatory activity is kept in the range [0,1) by implementing a cyclic pattern in the space of indices. Figure 3 shows how the level of modulatory activation of the switch neuron affects its decision.

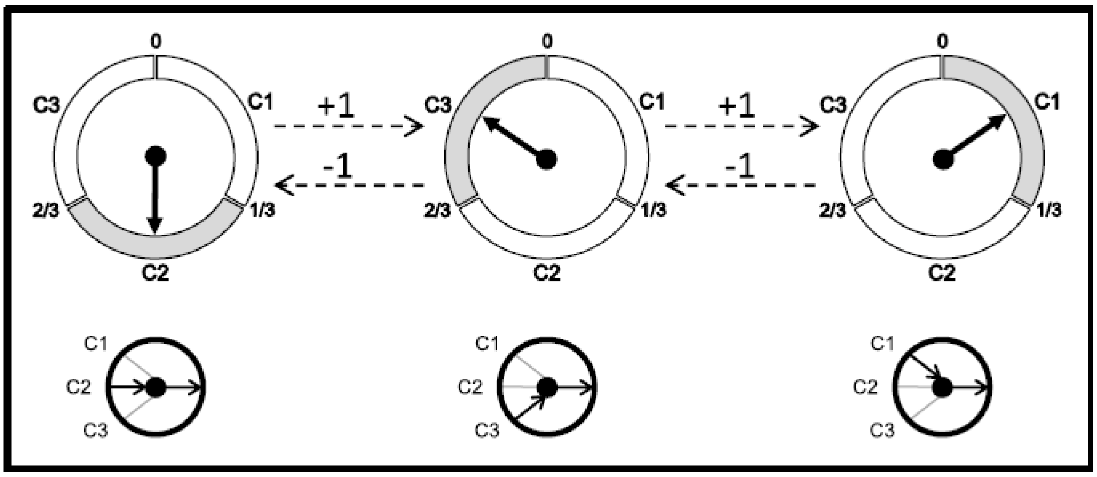


Figure 3. Modulatory "wheel". Redrawn from Vassiliades & Christodoulou (2016)

2.1.3 The switch module

The switch module is a structure of three neurons which allows for switch neurons to modulate other switch neurons and its design concept is presented in figure 4.

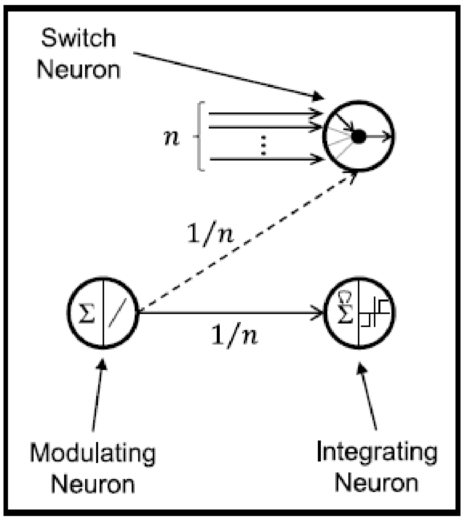


Figure 4. The switch module. Redrawn from Vassiliades & Christodoulou (2016).

The switch module consists of a switch neuron (as described above), a modulating neuron and an integrating module. The ‘‘modulating neuron’’ is responsible for altering the level of modulatory activation of the switch neuron and for this reason it connects to the switch neuron with a modulatory connection (shown by a dashed line). The ‘‘integrating neuron’’ integrates the modulatory signals emitted from the modulating neuron, using a standard connection that has the same weight as the modulatory one, which is equal to 1/n, and fires when its activation exceeds a threshold value; this neuron is responsible for connecting different switch modules/neurons (Vassiliades & Christodoulou, 2016). Effectively, this setup makes the switch module able to fire *both* an activation signal (from the switch neuron itself) and a modulatory signal (from the integrating neuron) at the same time.

The modulating neuron uses the weighted-sum integration function and the linear activation function for calculating its standard activation and standard output respectively. Therefore,
its (standard) output is computed as:

$$y^{(std)}(t) = a_i^{(std)}(t) = \sum_{w_{ij} \in std} w_{ij} * y_j^{(std)} * (t - d_{ji})$$

The integrating neuron uses a perfect integrator as the integration function and the linear activation function. When the output is not zero, i.e., 1 or −1, the activation of the neuron is reset to a baseline value, b, which is set to 0 in all experiments of this study.

$$a_i^{(std)}(t) = a_i^{(std)} * (t-1) + \sum_{w_{ij} \in std} w_{ij} * y_j^{(std)} * (t - d_{ji})$$

$$y_i^{(std)}(t) = \begin{cases} 1, & a_i^{(std)}(t) \geq \theta \\ -1, & a_i^{(std)}(t) < \theta \\ 0, & otherwise \end{cases}$$

where θ > 0 is a threshold value that is set to 1 in all experiments.

2.2 NEAT

2.2.1 Genetic Algorithms

As described above, genetic algorithms are search based optimization techniques that reflect the process of natural selection where the fittest individuals are selected for reproduction. In this section the basic framework in which this family of algorithms operates is laid.

The process begins with a population of (random) candidate solutions to the problem we are trying to optimize – these are referred to as chromosomes. Each individual consists of a number of genes, with each gene controlling one parameter of the problem. The values of the parameters are encoded in the context of a specific alphabet to enable meaningful crossovers between genes, with the most usual encoding being a binary string.

The fitness function evaluates the performance of each gene with respect to the task at hand. Thus, a higher fitness score assigned to a chromosome means the solution encoded within it is better. After every chromosome is evaluated, a subset of the population is selected in pairs for reproduction, with the probability of selection being proportional to the fitness score.

Each pair of selected chromosomes produces one or more offspring (depending on the algorithm), primarily through the means of crossover. The process of crossover involves exchanging certain genes between the parents, resulting in new chromosomes which are added to the population. New offspring are also subjected to mutation with a low probability. When a chromosome mutates, some genes are altered randomly. Mutation in genetic algorithms aims to maintain diversity and introduce novel genes to the gene pool. The evaluation – reproduction cycle is repeated until one of the candidate solutions satisfies our fitness criteria or the population converges, i.e. no further optimization is possible.

2.2.2 Neuroevolution of Augmenting Topologies

First introduced in (Stanley & Miikkulainen, 2002) ,Neuroevolution of Augmenting Topologies (NEAT for short) is such a genetic algorithm that is specifically tuned for evolving neural networks, both their weights and structure. From this point on, we refer to the individuals in the population as genomes or genotypes and the networks that correspond to them phenotypes.

NEAT uses a direct scheme for encoding neural networks into genotypes. Each genome includes a list of node genes and a list of connection genes, each of which refers to two node genes being connected. A node gene describes a neuron in the network: its bias, integration function and activation function. Each connection gene specifies the in-node, the out-node, the weight of the connection, whether the connection gene is expressed (an enable bit), and an innovation number, which allows finding corresponding genes during crossover. Figure 5 shows the mapping from genotype to phenotype.

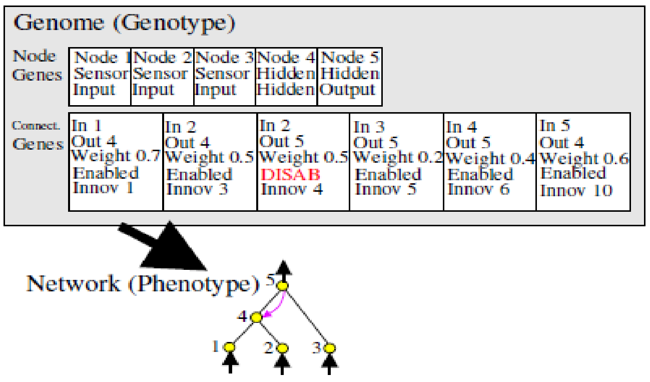

Genome (Genotype)
Node Genes
Node 1 Sensor Input
Node 2 Sensor Input
Node 3 Sensor Input
Node 4 Hidden Hidden
Node 5 Hidden Output
Connect. Genes
In 1 Out 4 Weight 0.7 Enabled Innov 1
In 2 Out 4 Weight 0.5 Enabled Innov 3
In 2 Out 5 Weight 0.5 DISAB Innov 4
In 3 Out 5 Weight 0.2 Enabled Innov 5
In 4 Out 5 Weight 0.4 Enabled Innov 6
In 5 Out 4 Weight 0.6 Enabled Innov 10
Network (Phenotype)
5
4
1
2
3

Figure 5. Genotype to phenotype mapping. Redrawn from Stanley & Miikkulainen (2002).

The initial population consists of neural networks with minimal topologies, that is without any hidden nodes, and grows more and more complex with each generation. The aim behind this decision is to minimize the initial dimensionality of the search space and thus speed up the evolutionary process.

Mutation in NEAT can change both connection weights and network structures. Connection weights mutate with a small probability over a gaussian function, with a constant mean and standard deviation, multiplied by a mutation factor. Structural mutations, which expand the genome, occur in two ways: *add connection* and *add node.* Figure 6 shows how each operation affects the genotype and the phenotype.

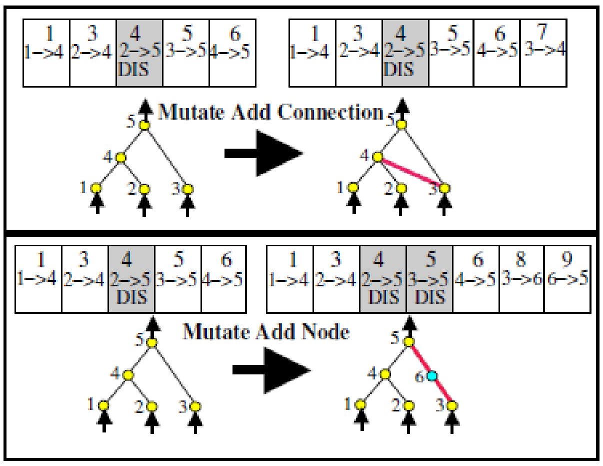


Figure 6. Add connection and add node. Redrawn from Stanley & Miikkulainen (2002).

The NEAT algorithm has been used with success widely, especially in combination with reinforcement learning. The applications range from researchers tackling classic problems in the field such as pole balancing to hobbyists using it to evolve agents capable of playing video games. It is also easily accessible since there exist libraries that implement it for many popular programming languages. These qualities portray NEAT as a promising tool and a useful ally in our endeavors.

2.3 Map based encoding

The map-based encoding is an indirect encoding scheme used for evolving neural networks. In Doncieux, Mouret, Pinville, Tonelli, & Girard (2011) a map is described as 1D or a 2D grid of identical neurons and the main idea behind this approach is connecting these maps with regular connection schemes. This allows such neural networks to scale up to larger maps while maintaining the same overall structure.

The aim of the designers of this scheme was to improve the quality of the neural networks resulting from neuroevolution through imitating better biological systems (brains), which rely on repetition and combination of hierarchically organized modules.

Connection schemes between maps are restricted to three cases, as depicted in figure 7: one to one connection with constant weights, one to all with constant weights and one to all with weights following a Gaussian distribution (although for this project the Gaussian is substituted with a custom function which better fits our needs).

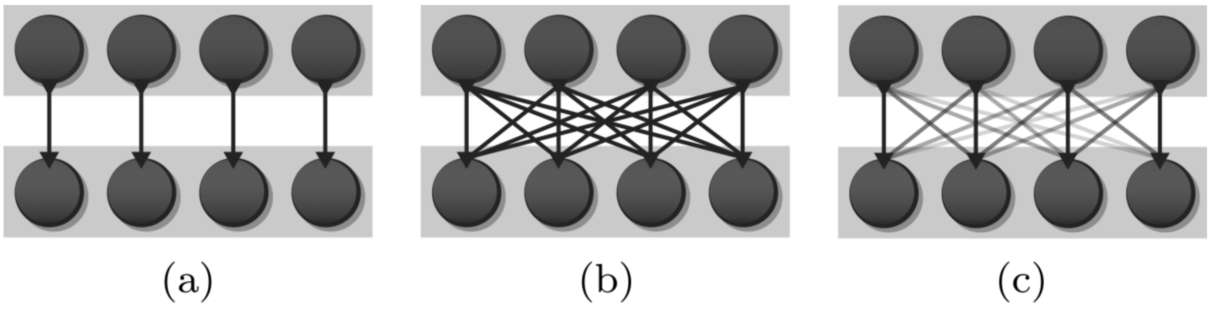


Figure 7: The three connection schemes of map-based encoding, in order: one-to-one, one-to-all with constant weights and one-to-all with the gaussian distribution.

This encoding can be easily adapted to be used in conjunction with the NEAT algorithm by extending the parameters that node and connection genes control. For node genes, we add a parameter which indicates whether a node should be scaled to a map or not and for connection genes we allow evolution to determine the connection scheme between two maps.

2.4 Novelty search

Novelty search is a new method of achieving optimal solutions with neuro-evolution introduced by Lehman & Stanley (2008) and is based on the radical idea of ignoring the objective. The idea is to identify novelty as a proxy for stepping stones. That is, instead of searching for a final objective, the learning method is rewarded for finding any behavior whose functionality is significantly different from what has been discovered before. Thus, instead of an objective function, this algorithm employs a novelty metric.

That way, no attempt is made to measure overall progress. In effect, such a process gradually accumulates novel behaviors.

2.4.1 The novelty metric

There are many potential ways to measure novelty by analyzing and quantifying behaviors to characterize their differences. Importantly, like the fitness function, this measure must be fitted to the domain. The way this is achieved is by defining a behavioral descriptor for the problem we are trying to optimize, a compact representation, that is, of the way an agent behaves in the environment. Then, using these behavioral descriptors we can measure how far apart two agents are in the behavioral space.

The novelty of a newly generated individual is computed with respect to the observed behaviors of an archive of past individuals whose behaviors were highly novel when they originated. The aim is to characterize how far away the new individual is from the rest of the population and its predecessors in novelty space, i.e. the space of unique behaviors. A good metric should thus compute the sparseness at any point in the novelty space. Areas with denser clusters of visited points are less novel and therefore rewarded less. A simple measure of sparseness at a point is the average distance to the k-nearest neighbors of that point, where k is a fixed parameter that is determined experimentally. Intuitively, if the average distance to a given point's nearest neighbors is large then it is in a sparse area; it is in a dense region if the average distance is small. The sparseness p at point x is given by:

$$p(x) = \frac{1}{k}\sum_{i=1}^{k} dist(x, m_i)$$

where m_i is the ith-nearest neighbor of x with respect to the distance metric.

If novelty is sufficiently high at the location of a new individual, i.e. above some minimal threshold, then the individual is entered into the permanent archive that characterizes the distribution of prior solutions in novelty space.

2.4.2 Novelty in the T-Maze domain

The T-Maze is a domain of problems designed to evaluate reinforcement learning agents and it is described by Soltoggio, Bullinaria, Mattiussi, Durr, and Floreano (2008). The single T-Maze consists of two arms that either contain a high or low reward (illustrated in figure 8). The agent begins at the bottom of the maze and its goal is to navigate to the reward position and return home. This procedure is repeated many times during the agent's lifetime. The goal of the agent is to maximize the amount of reward collected over deployments, which requires it to memorize the position of the high reward in each deployment. When the position of the reward sometimes changes, the agent should alter its strategy accordingly to explore the other arm of the maze in the next trial.

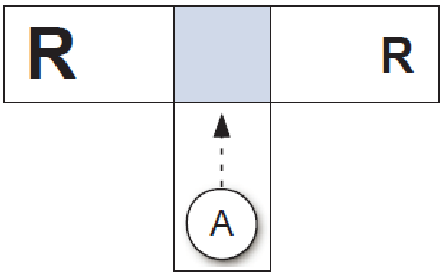


Figure 8: The single T-Maze

The behavior of an agent in the T-Maze domain is characterized by a series of trial outcomes. It is necessary to include multiple trials because an agent that learns can only be distinguished from one that does not by observing its behavior before and after the reward switch. Each trial outcome is characterized by two values: (1) the amount of reward collected (high, low, none) and (2) whether the agent crashed. These outcomes are assigned different distances to each other depending on how similar they are. In particular, an agent that collects the high reward and returns home successfully without crashing (HN) should be more similar to an agent that collects the low reward and also returns home (LN) than to one that crashes without reaching any reward location (NY). The novelty distance metric is ultimately computed by summing the distances between each trial outcome of two individuals over all deployments.

In their experiments, Risi, Vanderbleek, Hughes, & Stanley (2009) found out that novelty search outperformed the classic fitness-based search, in the sense that it was able to evolve optimal agents in a shorter time span.

3 Design and Implementation

3.1 Bird's eye view

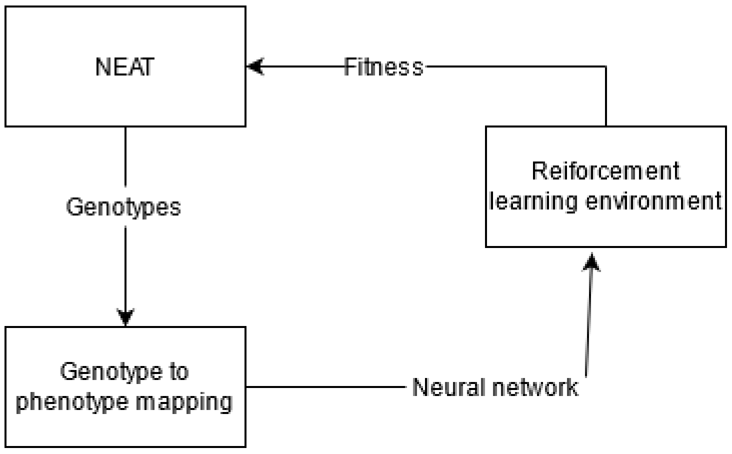

NEAT
Fitness
Reinforcement learning environment
Genotypes
Genotype to phenotype mapping
Neural network

Figure 9: Overview of the flow of an evolutionary experiment

On the highest abstraction level, the flow of an evolutionary experiment is simple and can be described in a couple of steps. To start off, the evolutionary algorithm (NEAT or novelty search) initializes the population of genotypes. Then we map these genotypes to the phenotypes of the type of neural networks using the encoding schemes we designated. In this project we conduct experiments with two schemes: the direct encoding and the map-based encoding, both enhanced with switch neurons. After building the neural networks, we expose each one of them to our selected reinforcement learning environment in order to evaluate their fitness, or in the case of novelty search to extract their behavioral descriptors. Finally, the evolutionary algorithm selects the genomes that will reproduce and pass their genes to the next generation based on the metric score of each one. The process is repeated until either a genome reaches the maximum fitness for a task or the maximum number of generations has been reached.

3.2 Modelling the networks

The aim of building a network with switch neurons, of course, is not for the whole network to consist of switch neurons, but rather have them cooperate with "traditional" neurons in order to enable more complex behavior. Unfortunately, finding a package that suits our needs proved to be challenging, either because some of them did not allow for arbitrary connection schemes, which is essential, or were not easily extendable. As such, it was deemed necessary to create a neural network definition by code tailored to our needs. The main concepts of the implementation are discussed below, while the more specific parts can be studied from the code listed under Appendix A, and specifically the switch_neuron.py file (page A-47).

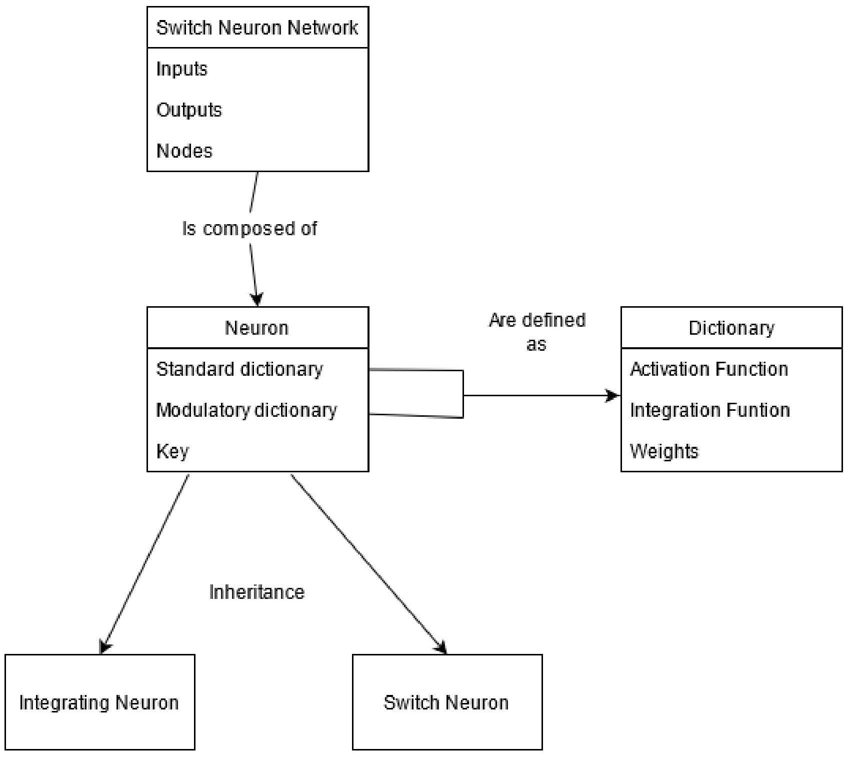


Figure 10: How the network implementation is organized

Figure 10 showcases the structure of the network implementation. A switch neuron network is composed of neurons, which in turn are defined by their standard and modulatory parts. A neuron can be a traditional neuron, in which case it only has a standard part, an integrating neuron or a switch neuron. Integrating and switch neurons are just substantiations of their superclass with some of the parameters pre-defined and with some exclusive methods to accommodate for their expected functionality.

Arbitrary connection structure unfortunately means that computations for the activation of the neural network are hard to vectorize and be implemented with matrix multiplication as would be the case with feedforward networks. Thus, each neuron contained in the network is activated in a serial manner in the order they are presented in the container of the nodes (a simple list), first activating the modular part of the neuron and then the standard part. At this point we are not concerned with checking if the order of activation is correct since this is taken care of during the creation of the network.

3.3 Learning environments

We employ two major reinforcement learning environments for evaluating the neural networks: the T-Maze and the binary association task. Both are developed with the help of OpenAI Gym (https://gym.openai.com/), a toolkit for standardization of the definition of an environment. It comes with several example environments ready to go but also offers the ability to developers to define their own environments through a specific interface.

For every environment defined with OpenAI Gym, an episode has a very simple and standard structure: the environment provides an "observation" to the agent (i.e. the state) and the agent returns one of the possible actions. Then the environment makes any necessary alterations to the state and presents it back to the agent along with the corresponding reward, repeating the cycle until the episode reaches a termination condition. An important implementation detail is that, contrary to evaluating traditional networks, when dealing with switch neuron networks it is advised to activate the network twice with the same input, first without the reward and then with it. This allows for the network to calibrate its state in case of a negative reward and prepare for the next one.

3.3.1 The T-Maze

For the implementation of the T-Maze environment, the gym-minigrid package (https://github.com/maximecb/gym-minigrid) is also used, which is an extension of OpenAI Gym that specializes on grid-like environments where the agent needs to reach a goal.

The state for this domain is expressed as a tuple of three boolean values in this order:

1. True if the agent is at the starting position, else false.
2. True if the agent is at a junction, else false.
3. True if the agent is at a maze end, else false.

However, because the inputs to the networks are floats, we map the true value to 1 and the false value to 0. The possible rewards at each step are the following:

- -0.4 if the agent crashes on a wall
- -0.3 if the agent fails to return to the starting position
- 0.2 if it finds the low reward
- 1 if it finds the high reward

The reward for failing to return the starting position does not apply for the versions of the environment where the episode ends when the agent reaches a maze end. The possible actions at each step are: turn left, turn right and move forward. Because the output of the network is also a float, we map the ranges of possible outputs to these discrete actions. In particular, an output of less than -0.33 maps to turning left, more than 0.33 to turning right and an output in between to moving forward.

With this setup, we define three versions of the T-Maze environment:

- The homing T-Maze (meaning the agent has to return to the starting position after it reaches a maze end).
- The non-homing T-Maze
- The double T-Maze, a variation where after the first turning point there is a second one before the agent reaches a maze end. So, essentially, there are four possible maze ends and the agent needs to decide twice in which direction to turn.

In order to be able to judge the agent's adaptability, the evaluation should consist of multiple episodes – this parameter is configurable at execution time. After a random but restricted number of episodes, the high reward changes its location. An evaluation like this is repeated a few times to punish agents that do not fit our criteria but "got lucky" during the first evaluation. For the experiments, four such evaluations are being carried out for each network.

3.3.2 Binary association task

The general setting in the binary association task is to make an agent learn the random associations between binary input patterns and binary output patterns based on feedback that comes in the form of a reward signal. These associations can be re-randomized at certain points in time, requiring from the agent to unlearn the previous associations and re-learn the new ones. The number of inputs is n and the number of outputs is m. There are four types of association problems: (i) one-to-one, where each of the n inputs needs to be associated with one of the m outputs, (ii) one-to-many, where each of the n inputs needs to be associated with one of the 2^m possible output patterns, (iii) many-to-one, where each of the 2^n input patterns needs to be associated with one of the m outputs, and (iv) many-to-many, where each of the 2^n input patterns needs to be associated with one of the 2^m possible output patterns. Therefore, in each case, all possible input vectors need to be associated with some output vectors. Although all of the variations are implemented, evolutionary experiments were run only for the one-to-one association problem because the hand-crafted solutions for the rest use some features that are not supported in the current neural network implementation and relying on the algorithm to discover new alternatives to them was deemed excessive.

The state of this environment at a time step is simply a tuple containing the n inputs – a tuple of 1's and 0's. The possible actions an agent can take are the set of the possible outputs. However, we do not evolve agents with m outputs but rather one, in order to reduce the complexity. Thus, the ranges of the possible outputs are mapped to the possible actions, similarly to above. For example, for the n=3, m=3 one-to-one variation, a value of more than 3.3 maps to (1,0,0), less that -3,3 to (0,0,1) and anything in between to (0,1,0). The environment awards a reward of 1 to the agent if the guessed association is correct and 0 otherwise.

3.4 Evolutionary algorithms

3.4.1 NEAT

As mentioned above, there exist libraries of code with NEAT implementations for many popular programming languages. My language of choice is python, because it offers many useful tools in its standard library and it speeds up development. The package used for this project is neat-python , which is open source and available on github (https://github.com/CodeReclaimers/neat-python). It was developed by CodeReclaimers and is under the BSD-3-Clause License.

Thankfully, the package is easily extendable. In order allow for NEAT to evolve networks with switch neurons, all that was needed was to define a custom type of genes for both nodes and connections that inherit a basic gene defined in the library and programmatically add to it all the evolved parameters. The same course of action was taken regarding the map-based encoding integration with NEAT.

3.4.2 Novelty search

The implementation of novelty search was also done as an extension to the NEAT package. A behavioral descriptor and a distance function using the descriptor and the hamming distance was defined for each problem in order to give the algorithm the ability to compare the behaviors of two agents. This distance function is utilized by an evaluation function that computes the novelty score for each agent using the nearest neighbors algorithm as described above and decides if it will be added to the archive. The evaluation function replaces the objective function in the NEAT algorithm and essentially transforms it into a novelty-based algorithm. When the evolutionary experiment is finished, the winner is derived from the archive rather than the resulting population. *(*Implementation detail: the archive is re-evaluated at the end of each generation to ensure that we maintain a diverse collection of adaptive agents).*

For the single T-Maze domain, the behavioral descriptor presented by Risi, Vanderbleek, Hughes, & Stanley (2009) is employed (described in section 2.4.2). The descriptor for the homing T-Maze is a variation of this one as well; in addition to the markings regarding whether the high or low reward was collected and whether the agent crashed, we add a third marking for tracking if the agent managed to return to the starting point after collecting the reward. For example, a descriptor for a single trial may be "LNY": the agent collected the low reward, it did not crash, and it returned to the starting point. For the double T-Maze, we take a slightly different approach to the matter. For each trial we record if the agent reached a maze end and which one it reached (each maze end is labeled with a number). So, if the agent reached the third maze end its descriptor would be 3, and if it crashed instead, it would be 0. The quantified distance between two episode descriptors would be 0 if in the two episodes the agents ended up on the same maze end, and 0 if they ended up on different ends.

Regarding the binary association task descriptor, we needed to design a different type of descriptor, since the number of episodes for each evaluation is much larger and keeping track of every single one would become too resource-heavy and time consuming. So, the solution proposed is to take snapshots of the agent's responses to each input at regular intervals and record changes to them. This way, we end up with an n-dimensional array of 1's and 0's, where a 1 indicates that between the snapshot indicated by the index and the previous one the agent changed its responses and a 0 means it did not.
Consider the following example:

Suppose that at episode 100 the agent has the following response table for the 2 by 2 one-to-one association problem:

(1,0) -> (0,1)

(0,1) -> (1,0)

Let the snapshot interval be 100 episodes. If, now, at episode 200 this response table stays the same, then we append a '0' to the behavioral descriptor. If, however, the table changes to:

(1,0) -> (1,0)

(0,1) -> (0,1)

then we append a '1'.

The result for the distance metric for two agents would be the hamming distance between the two descriptors. To further demonstrate, let agent A's behavior be described by (1,0,1,0) and agent B's behavior by (1,1,0,0). The hamming distance between the two is calculated as the number of indexes where the descriptors differ, so in this case where the descriptors differ in the indexes 1 and 2, the distance would be 2.

The dimensionality of the descriptor is configurable at execution time through specifying the number of episodes and the snapshot interval. It should be noted, though, that setting a very small snapshot interval results in a non-representative descriptor since it would not account for the episodes the agent needs to re-learn the associations after a shuffle.

3.5 Mapping genotypes to phenotypes

3.5.1 Direct encoding

Mapping a genotype to a phenotype is mostly a straightforward process. One only has to translate all the information contained in the genes to the components of the network. Special attention must be paid to the order of activations of the nodes inside the network. Because arbitrary connection schemes are possible, grouping neurons into layers may be impossible. What is proposed instead is to use the topological sorting algorithm to get all the hidden nodes in one of the possible correct orders, by making sure that the nodes from all the incoming connections of a specific node are added to the ordered list before that specific one.

Of course, sometimes, this will not be possible as loops can emerge, in which case a random neuron from those in the loop is added first.

Another point of interest is the case when two switch neurons are connected with each other via a modulatory connection. When this is detected by the part of the code responsible for constructing the network, the first switch neuron is replaced by a switch module (as described in section 2.1.3). The evolutionary algorithm is not given the ability to use switch modules in its architectures, although the emergence of the pattern with individual neurons is still possible.

3.5.2 Map-based encoding

Although the same principles apply for mapping genotypes to phenotypes using the map-based encoding scheme, some additional steps must be taken before we end up with the complete network. When evaluating a node gene and translating it to a neuron, the program also checks the parameter of the gene that controls if that node is isolated. If it is not isolated, then n more neurons are added to the network with identical parameters with the original one, with n being a customizable parameter at execution time (map size). Also, when evaluating connections, we must check if the connection is between two maps as well as the connection scheme used between them. The different connection schemes are described in section 2.3.

Initially, the gaussian function was used in one of the connection schemes, as proposed in the original paper presenting this encoding (Doncieux, Mouret, Pinville, Tonelli, & Girard, 2011). However, the randomness that it introduced lead to inconsistent mappings between genotypes and networks, i.e., one genotype did not always map to the same phenotype. This noise is undesirable in our networks, as switch neurons are sensitive to it. For this reason, we chose to replace the Gaussian with another deterministic function. This function translates the weight evolved from NEAT, w, to n different weight values (where n = map size) by splitting the range [-w, w] into n-1 equal ranges. For example, let w = 1 and n =5. Then this function f would derive these 5 values: (-1, -0.5, 0, 0.5, 1).

3.5.3 The agent class

After creating the network, we encapsulate it into an instance of the Agent class. This class serves two purposes: a) to pre-process the inputs to the network and b) to post-process the network's outputs. This processing is sometimes necessary to be able to translate an environment status to a valid input, or a numerical output to a valid action. An example of this is the mapping from the network's numerical output to a valid action for the binary association task.

3.6 Wrapper and experiments

This section is dedicated to explaining how to configure and run an evolutionary experiment using the code implementing the above, as well as laying out the various setups for the experiments that were conducted.

Assuming python3 is installed (code tested for python 3.7) and the whole directory was cloned from github, the required external packages can be installed with the help of pip and the file requirements.txt. First off, we must configure the settings for NEAT, which is done by creating a configuration file. Configuration files for all scenarios are included, thus only changing the various parameters' values as one sees fit is necessary. This configuration file controls parameters like the maximum fitness threshold, the population size and the number of inputs and outputs of the networks. More details can be read at the wiki page of the neat-python package (https://neat-python.readthedocs.io/en/latest/config_file.html).

The rest of the customizable parameters that are relevant to the other parts of the project are defined at execution time. The whole process is controlled by the file evolve.py (appendix A - page 18), which based on the arguments given will run the relevant experiment. The arguments taken by evolve.py are:

- -s SCHEME: choose the encoding scheme used for mapping phenotypes to genotypes. Possible choices: switch, switch_maps
- -c CONFIG: path to the NEAT configuration file.
- -g GENERATIONS: the maximum number of generations that you want NEAT to evaluate networks for.
- -p PROBLEM: which problem we are trying to optimize. Possible choices: binary_association, tmaze, double_tmaze and homing_tmaze. The binary association option referes to the 3x3 one-to-one variation.
- --num_episodes EPISODES: the number of episodes for each evaluation. Keep in mind that for the T-Maze problems each agent is evaluated four times.
- --switch_interval INTERVAL: only applicable for the binary association task. Specifies the interval between shuffling the input – output pairs.

- --map_size MAP_SIZE: only required if the map-based encoding was chosen. Specifies the size of the maps used in the networks.
- --novelty: if used, NEAT will use the novelty metric instead of the objective function for the chosen problem.
- --threshold THRESHOLD: the minimum novelty score for a genotype to enter the archive.
- --snap_iter SNAP_ITER: only applicable for the binary association task along with novelty search. Specifies the interval between snapshots.
- --dump: if used, the program will save the winner network to a binary file (may be useful for debugging).
- --draw FILE: if used, the program creates a diagram depicting the winner network at path FILE.
- --log FILE: optional. Directs the program to log the maximum fitness per generation to a text file in append mode.

4 Results and Discussion

A total of twelve sets of experiments have been executed, three sets for each of the four problems: binary association task (the 3x3 one-to-one variation), the single non-homing T-Maze, the double non-homing T-Maze and the single homing T-Maze. The first set uses the NEAT algorithm with the direct encoding scheme, the second one uses NEAT with the map-based encoding scheme and the third one uses the novelty search algorithm. In this section the results are presented and some statistics that may help us interpret them.

The sample size for the experiments is rather small, due the computational intensity of the tasks. Execution times on the machine we used vary from a couple of hours for the simplest ones and a few days (and maybe even a week, depending on the configuration) for the more complex tasks, the ones with the novelty search algorithm for example.

4.1 Success in simple tasks

The first problem we examine is the single non-homing T-Maze. For these experiments, the number of episodes per evaluation is 8 and in every generation we evaluate all networks 4 times. For each evaluation, the position of the high reward is changed once at a random episode between episode 3 and 7. To calculate the satisfactory fitness threshold, we consider the behavior of an optimal agent. We would expect an optimal agent to miss the high reward a total of 5 times: 4 times because we change the location of the high reward, and possibly once in the start of the evaluation. So, since the high reward is 1, the low reward is 0.2 and we have a total of 32 episodes, the fitness threshold is computed as 27 * 1 + 0.2 * 5 = 28. For all three sets of experiments, the algorithm was able to evolve a competent agent with performance comparable to the ones designed by hand in most of the runs. Figure 11 is an illustration of an optimal agent from the NEAT-direct set. The grey boxes denote the input nodes, the white circles the hidden neurons and the blue circle an output node. A double circle (a circle within a circle) means that the neuron is a switch neuron. The activation function of a hidden neuron is written inside the circle (the identity function is the linear activation function). A solid line between two nodes represents a standard connection between them while a dashed line represents a modulatory connection. However, a modulatory connection to a non-switch neuron is parsed as a standard connection by the program. Each hidden and output neuron has an evolvable parameter that controls its bias, which is not shown in the diagram.

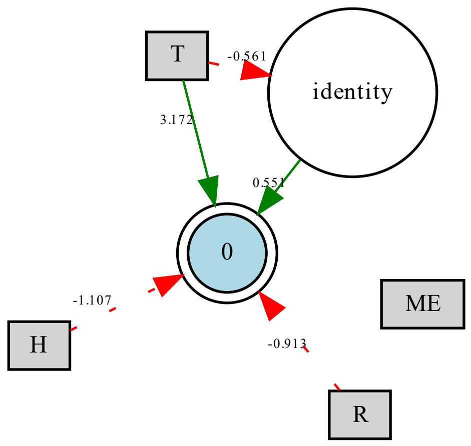


Figure 11: Illustration of an optimal agent from the NEAT-direct experiment set for the single non-homing T-Maze.

As described in section 3.3.1, the inputs are:

- H: 1 if the agent is at the starting position, else 0.
- T: 1 if the agent is at a turning point, else 0.
- ME: 1 if the agent is at a maze end, else 0.
- R: the reward from the environment for the current action (we activate the networks twice with the same inputs so they can calibrate in case of undesired behavior).

Inspecting the network, we observe some interesting properties that are worth noting. First of all, the complexity of it is very low, with only one hidden neuron and a total of five connections. This, of course, does not come in contrast with our expectations, since NEAT searches for the simplest solution by design. However, this is not the case for every run; some networks that were evolved are much more complex, with random redundant hidden neurons and connections that do not meaningfully contribute to the output of the network. Results like these are also somewhat justifiable, with the idea being that those redundant components gave the network an evolutionary advantage in early generations, and thus were kept, but were later rendered useless by the introduction of new patterns.

Secondly, we notice that the (ME) input is not used, which is only logical, since this is the non-homing task and the agent does not need to know when it has to return to the starting point. Another thing that we expected is that if a solution were to be found, it was necessary for NEAT to discover the pattern of the reward input modulating the switch neuron, which, as we can see, has happened (the R input is connected with a modulatory connection to the switch neuron 0). In addition, a peculiar pattern that has emerged and is present in many of the resulting networks, especially for this task, is successfully using the switch neuron as the output node. This challenges the design patterns of the hand-crafted networks, where the switch neurons are used towards the "end" of the network, a few steps before the output nodes, but never as the output. Finally, in this network the identity function is used for the activation of the hidden neuron, but a variety of other functions is used in other solutions, such as the sigmoid and the heaviside function.

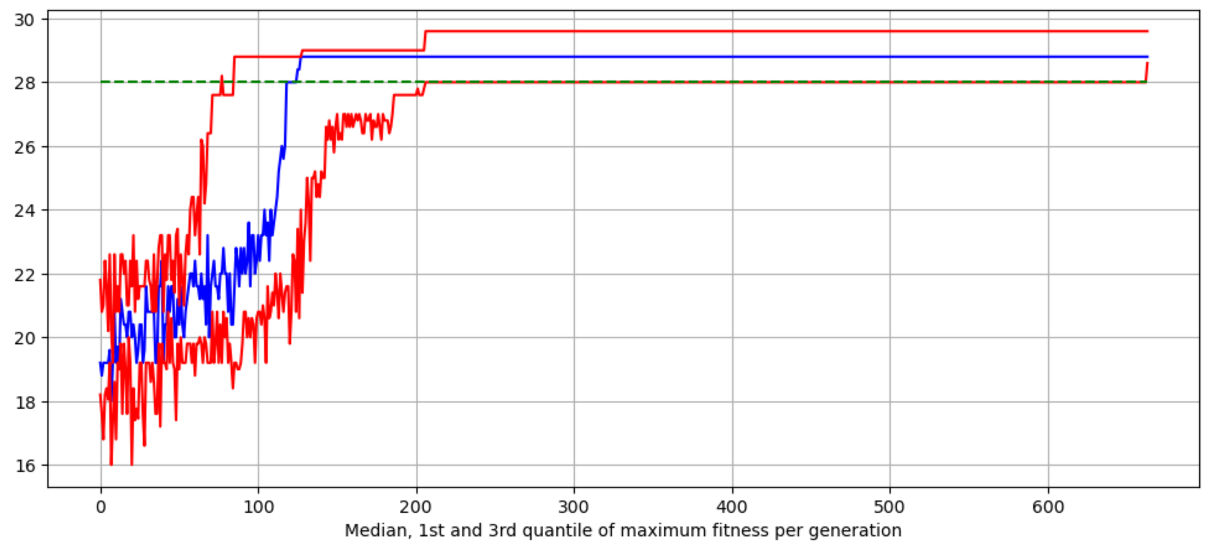


Figure 12: Chart of the 1st quartile (red line), median (blue line) and 3rd quartile (red line) -in this order - of the maximum fitness for each generation (Sample size = 16). The green dashed line is the fitness threshold.

Figure 12 depicts the progress of NEAT trying to find a solution for 16 different runs of the experiment. In most cases, the algorithm was able to evolve an optimal agent from relatively early – in less than 200 generations, while for some cases it needed 600 generations, or it did not manage to find a solution in the 1000 generations limit. We are speculating that by taking time to fine tune some parameters in the configuration file that control the mutation rate and power of the architecture and weights could result in faster convergence, but it was not deemed necessary to do so since this project is mainly about proving that the concept can work.

The above data concern the experiment set with NEAT and the direct encoding. The results for the set with the indirect encoding are also similar, in the sense that the algorithm can consistently evolve an optimal agent. In this set, 8 runs were executed with every run having a different map size – from 4 to 11. The resulting networks vary from an architecture standpoint. Some of them do not use maps at all and are similar to the network in figure 11, while some others do, and with similar success rate. An example of a satisfactory network that incorporates maps in its architecture is shown in figure 12. Although it is a little cluttered, the regularity of the connection weights between the output neuron and a group of hidden neurons is apparent – specifically it uses the one-to-one scheme with constant weights.

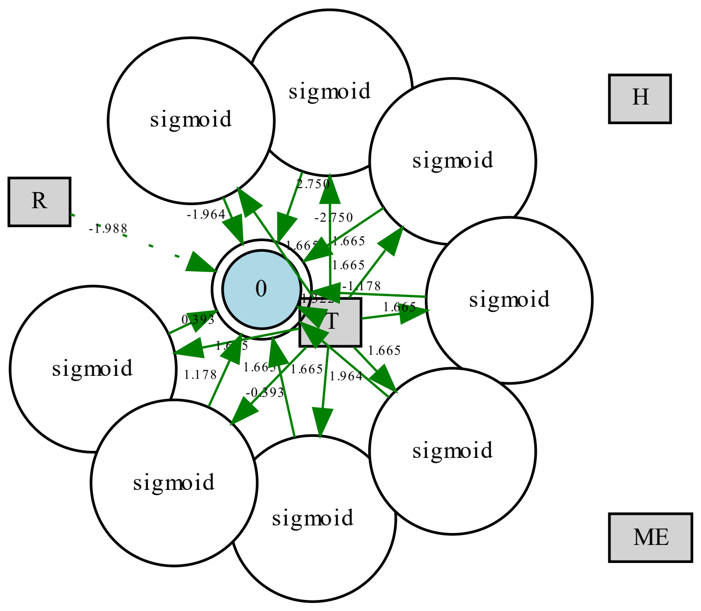


Figure 13: A network that solves the non-homing single T-Maze task that uses neuron maps. Small implementation detail: a number is written inside the white circles instead of the name of a function because a lambda function is used.

The respective fitness graph for this set of experiments is displayed in figure 14. The trend is like the one for the direct encoding scheme, but we notice that finding a solution now takes more generations on average. This suggests that the indirect encoding hampers the evolution process rather than helping it, at least for simpler tasks.

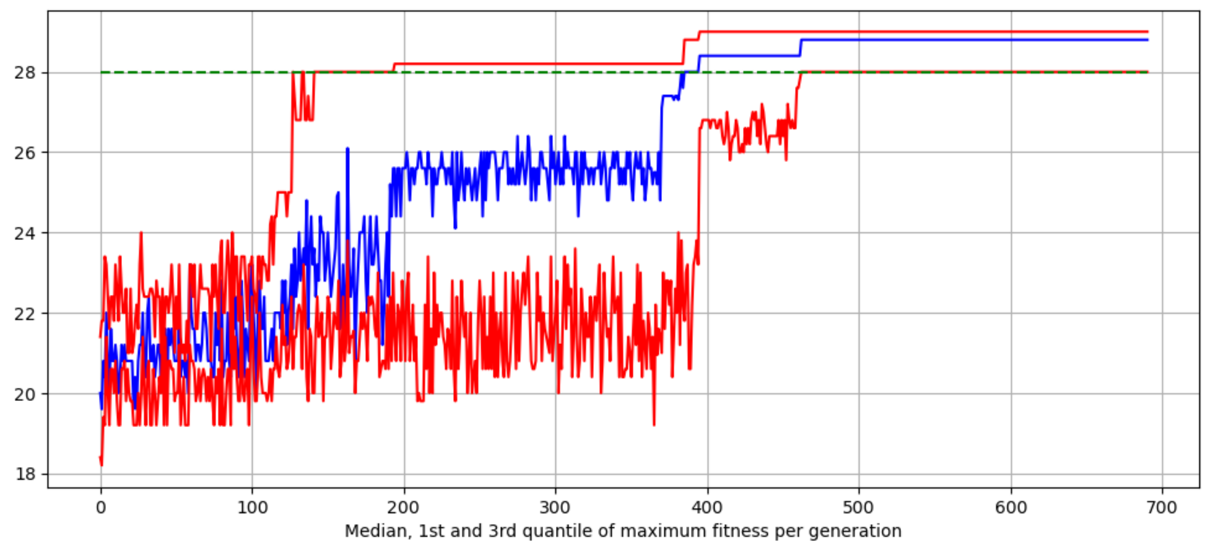


Figure 14: Chart of the 1st quartile (red line), median (blue line) and 3rd quartile (red line) -in this order - of the maximum fitness for each generation for the NEAT- map-based algorithm (Sample size = 8). The green dashed line is the fitness threshold.

Moving onto the set using the novelty search algorithm, we find consistent success here as well. A total of 8 experiments were run using these settings, each one with different novelty thresholds for a genome to enter the archive. Upon viewing the resulting networks, though, an astounding difference in complexity from the other two sets emerges. An example is illustrated in figure 15 for reference. This sudden jump in complexity can probably be attributed to the fact that in this set we do not use a fitness threshold, rather we let the algorithm evaluate solutions for 1000 generations for every experiment. As such, simple solutions may appear early on but larger networks that emerge later have a slightly higher fitness and thus rise as the “winners”. This suggests that a revision in the termination condition of the algorithm may be beneficial.

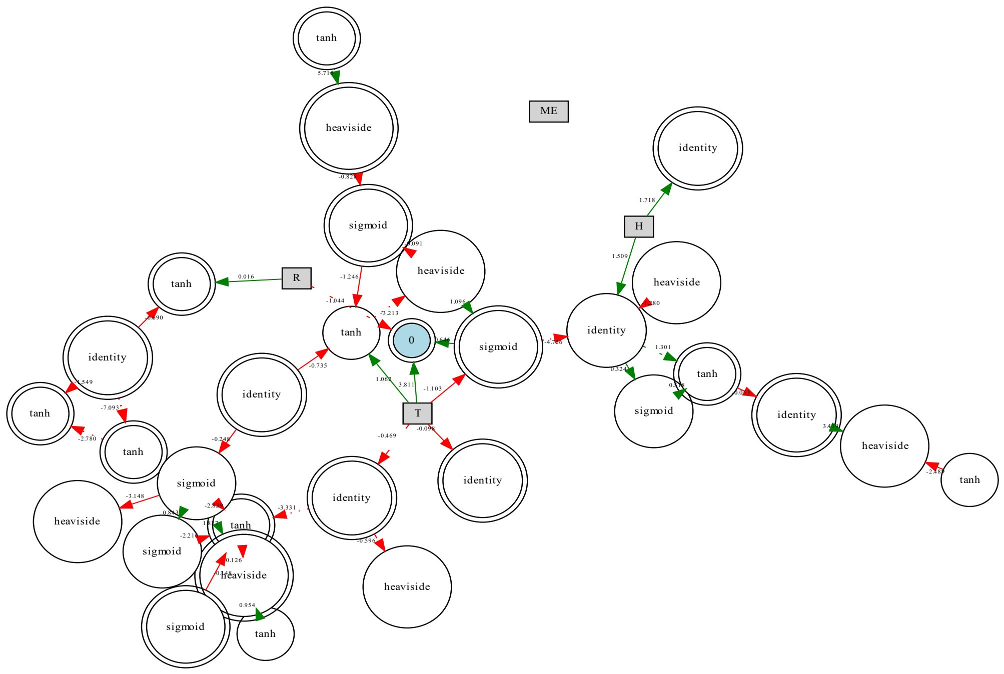


Figure 15: A network-solution evolved with the novelty search algorithm.

4.2 Shortcomings

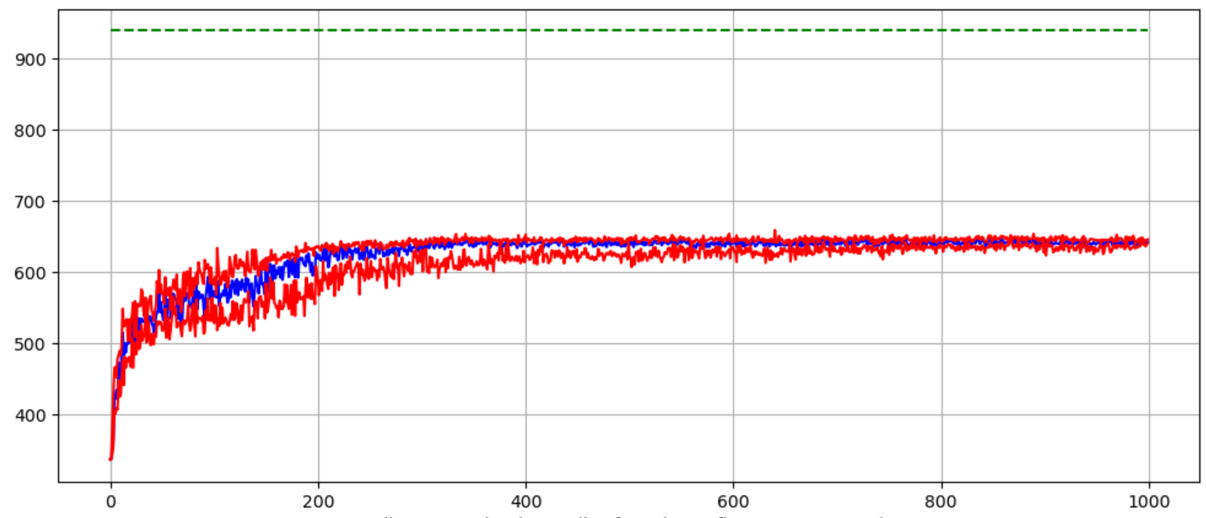
Before the results for the rest of the problems are presented, the basic settings for each experiment set are stated so that a better picture of the context is portrayed.

- Non-homing double T-Maze: Similarly to the single T-Maze, we perform four evaluations of each network for each generation, with each evaluation consisting of twelve trials. Similarly to the single T-Maze task, the high reward location is changed once in every evaluation. As such, an optimal agent would need to re-learn the path to it 5 times: 4 times because we change its location and possible once in the beginning. Each time the agent re-learns the location of the high reward, it would make a maximum of three mistakes before finding it. Considering we have a total of 48 episodes, the fitness threshold is computed as: 5*3*0.2 + 33*1 = 36.
- Homing single T-Maze: Once again, we perform four evaluations, each with 8 episodes. The fitness threshold is set to 28, with the same logic used for the single non-homing T-Maze.
- Binary association task: For the 3x3 one-to-one association task, we perform 1000 trials for each network evaluation, with a switch interval of 100 trials. This means that an agent would have to re-learn the associations 10 times during one evaluation. Each time the associations are randomized, an optimal agent would make a maximum of n*(m-1) = 6 mistakes to re-learn

them. Considering that the high reward is 1 and the low 0, the fitness threshold is calculated as 6*10*0 + 940*1 = 940.

For all the above, a maximum of 1000 generations is allowed. In the case of using the map-based encoding, each experiment is run with a different map size in the range [2, 9] and in the case of the novelty search algorithm each run has a different archive entrance threshold which is capped to 25% of the maximum distance between two descriptors.

For all the cases described, none of the three algorithms was able to find a solution. In figures 16, 17, 18 the progression of the maximum fitness per generation is shown for all three problems for the NEAT-direct set of experiments (sample size = 8). For the binary association task (first graph) and the homing T-Maze (second graph) it is apparent that the algorithm plateaus at a certain fitness point below our set threshold after 100-200 generations. On the other hand, the respective graph for the double T-Maze problem is closer to the desired fitness but much noisier, which suggests that the results are sensitive to the parameters of the algorithm and perhaps with a much larger generation cap we could see some interesting results emerge. However, upon further inspection of the results, it seems that in this domain there is high variance between two evaluations of the same network, which could mean a failure in evolving adaptive agents – and subsequently in the punishment of non-adaptive agents by the environment.



Median, 1st and 3rd quantile of maximum fitness per generation
Binary assocation task.

Figure 16: Chart of the 1st quartile (red line), median (blue line) and 3rd quartile (red line) of the maximum fitness for each generation for the NEAT- direct set for the binary association task (sample size = 8).

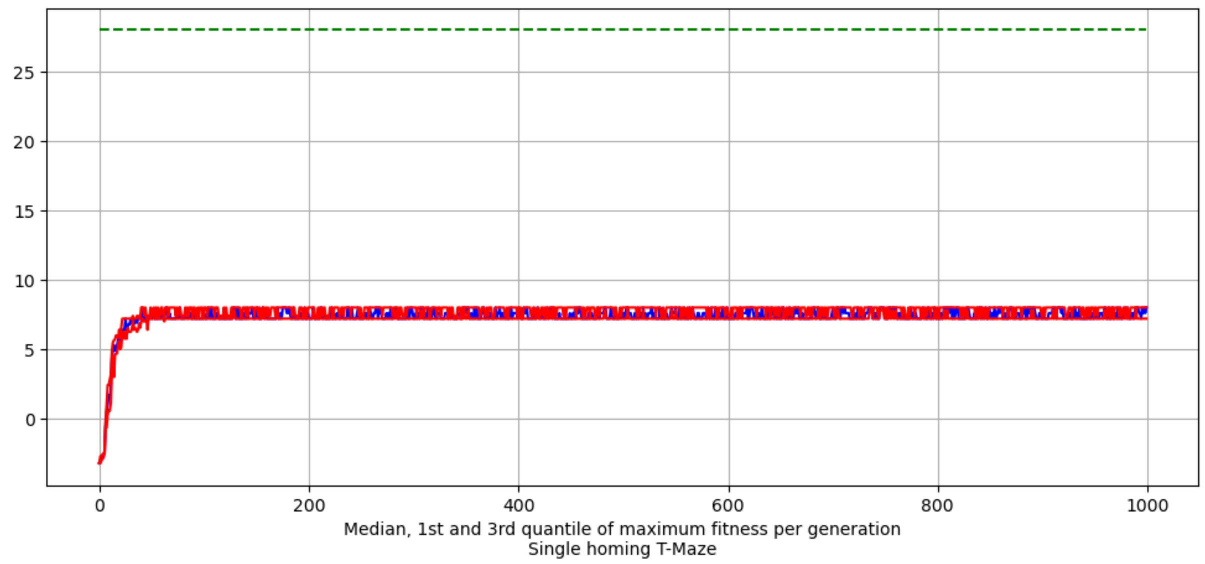


Figure 17: Chart of the 1st quartile (red line), median (blue line) and 3rd quartile (red line) of the maximum fitness for each generation for the NEAT- direct set for the homing T-Maze (sample size = 8).

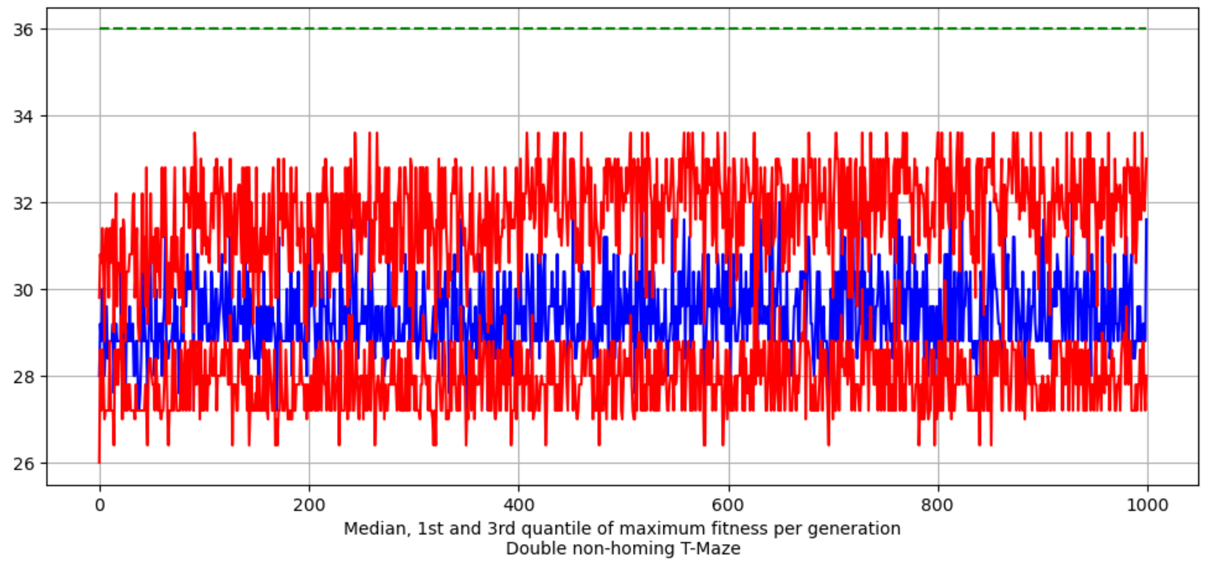


Figure 18: Chart of the 1st quartile (red line), median (blue line) and 3rd quartile (red line) of the maximum fitness for each generation for the NEAT- direct set for the double T-Maze (sample size = 8).

The same patterns are detected in the respective graphs for the map-based encoding set of experiments, with the only difference being the variance mentioned for the double T-Maze domain resulting in some fault positives, i.e. a genome was flagged as satisfactory during the evolution but was found lacking during a second evaluation.

4.3 Discussion

The success that we have had with the task of the single non-homing T-Maze constitutes proof that switch neurons can be successfully used alongside neuroevolution algorithms to automatically discover optimal network architectures. It could become an additional tool in such frameworks and assist the evolutionary process, even if only for simple tasks. It would seem, however, that the success rate of the three algorithms is at least somewhat correlated, when we in fact expected the novelty search algorithm to decouple at a degree from the performance of NEAT.

The failure at the rest of the tasks can point us to mistakes in our approach to the issue and the implementation, both technical and not. Firstly, there is the possibility that the connectivity patterns we gave NEAT to work with were not enough for it to develop optimal agents for the more complex tasks. The hand-crafted networks utilize some structures that are impossible to occur through this implementation of NEAT, such as two connections with different weights between the same neurons, delayed connections, and vector inputs to the nodes. One other weakness in our approach could be the sensitiveness of switch neurons to noise, which means that a small difference in the incoming weights can dramatically alter the output of the whole network. The networks that were designed manually have had their weights carefully picked and architectures that may be too specific for a genetic algorithm to approximate. A final misstep that was also mentioned above is specific to the double T-Maze domain and concerns the failure of the environment to punish “lucky” non-adaptive agents.

5 Conclusion and future work

In this diploma thesis our attempt at evolving switch neuron networks is outlined. We begin by introducing some basic principles in the areas that are relevant to our study: reinforcement learning, neural networks and genetic algorithms. Then, we proceed to analyze the specific theories and algorithms we used, with the central piece being the switch neuron, a unique type of artificial neuron that was introduced recently by Vassiliades & Christodoulou (2016), the supervisors of this project. The NeuroEvolution of Augmenting Topologies algorithm, the map-based encoding scheme and novelty search are also reviewed with the relevant references to the original studies. A discussion of the structure of the experiments, as well as the design and implementation decisions behind them follows, dissecting each part of the project separately and how they work together towards the goal of evolving adaptive agents using switch neurons in their architectures. The results of this attempt are mixed. We found success in evolving optimal agents for the simplest one of the tasks but failed for the rest. Still, this indicates that the unique properties of the switch neuron can be a beneficial addition to evolutionary algorithms and help with the evolutionary process, especially when the objective is adaptive behavior.

There are various methods one could try in future work in order to achieve better performance. One of these is the enhancement of the evolutionary algorithm with the structures mentioned above, giving them some additional tools that can help the process. Another venue one could pursue is the adaptation of this experiment to a different evolutionary algorithm. For instance, quality-diversity optimization algorithms (Chatzilygeroudis, Cullyz, Vassiliades, & Mouret, 2021) were proposed by Dr. Vassiliades during our sessions, but were ultimately dismissed due to time constraints and the difficulty of fitting them to our needs. This family of algorithms builds on the premise of novelty search and try to explore the search space to find the best solution and could prove to be a powerful tool in such an effort.

References

Chatzilygeroudis, K., Cullyz, A., Vassiliades, V., & Mouret, J.-B. (2021). Quality-Diversity Optimization:a novel branch of stochastic optimization. In P. M. Pardalos, V. Rasskazova, & M. N. Vrahatis, *Black Box Optimization, Machine Learning, and No-Free Lunch Theorems.* Springer International Publishing.

Doncieux, S., Mouret, J.-B., Pinville, T., Tonelli, P., & Girard, B. (2011). The EvoNeuro Approach to Neuro-Evolution. *Proceedings of the Workshop on Development & Learning in Artificial Neural Networks (DevLeANN), Oct. 2011.* Paris, France.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning: A Survey. *Journal of Artifical Intelligence Research*, 237-285.

Lehman, J., & Stanley, K. O. (2008). Exploiting open-endedness to solve problems throughthe search for novelty. *Proceedings of the Eleventh International Conference on Artificial Life(ALIFE XI),* (pp. 329-336).

Man, K. F., Tang, K. S., & Kwong, S. (1996). Genetic Algorithms: Concepts and Applications. *IEEE Transactions on industrial electronics*, 519-534.

Risi, S., Vanderbleek, S. D., Hughes, C. E., & Stanley, K. O. (2009). How novelty search escapes the deceptive trap of learning to learn. *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (pp. 153-160). Montreal Québec Canada: Association for Computing Machinery, New York, NY, United States.

Soltoggio, A., Bullinaria, J. A., Mattiussi, C., Durr, P., & Floreano, D. (2008). Evolutionary advantages of neuromodulated plasticity in dynamic, reward-based scenarios. In S. Bullock, J. Noble, R. Watson & M. A. Bedau (eds.). *Artificial Life XI* (pp. 569-579). Cambridge: MIT Press.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary computation, 10(2),* 99-127.

Vassiliades, V., & Christodoulou, C. (2016). Behavioral plasticity through the modulation of switch neurons. *Neural Networks, 74,* 35-51.

t_maze.py:

```
from gym_minigrid.minigrid import *
from gym_minigrid.register import register

LOW = 0.2
HIGH = 1
CRASH_REWARD = -0.4
FAIL_HOME = -0.3
STEP_PENALTY = -0.005
#Helper class for keeping track of reward positions
class MazeEnd(Ball):
    reward = LOW
    def __init__(self, reward='low'):
        color = 'yellow'
        if reward == 'high':
            color = 'green'
            self.reward = HIGH
        super().__init__(color)

    def can_overlap(self):
        return True

    def set_high(self):
        self.reward = HIGH

class TMazeEnv(MiniGridEnv):

    #Actions that the agent is permitted to make
    class Actions(IntEnum):
        # Turn left, turn right, move forward
        left = 0
        right = 1
        forward = 2

    def __init__(self, high_reward_end=0, size = 7, isHoming = False):
        self.agent_start_pos = (size // 2, 1)
        self.agent_start_dir = 1
        #Which maze end has the high reward. The rest have the low reward
        self.high_reward_end = high_reward_end
        self.reward_range = (-1, 1)
        self.actions = TMazeEnv.Actions
        self.isHoming = isHoming
        max_steps = 8
        if self.isHoming:
            max_steps = 15
        super().__init__(
            grid_size= size,
            max_steps = max_steps,
            agent_view_size = 3
        )


    def _gen_grid(self, width, height):
        #Setting the coordinates for each maze end
        self.LEFT_END = (1,self.height - 2)
        self.RIGHT_END = (self.height - 2, self.height - 2)
        self.MAZE_ENDS = [self.LEFT_END, self.RIGHT_END]
        self.middle = self.width // 2
        #Setting the coordinates for the turning points
        self.TURNING_POINTS = [(self.middle, self.width - 2)]
        self.reward = 0
        self.grid = Grid(width, height)

        #Here we are creating the landscape which looks like this
        #       X
        #       X
        #       X
        # X X X X X X X
        #
        for i in range(width):
            for j in range(height):
                self.put_obj(Lava(),i,j)

        for i in range (width):
            self.grid.set(i,height-2,None)
            self.grid.set(self.middle,i,None)
        self.mission = "Find the highest reward"

        x=0
        y=0
        w=width
        h=height
        self.grid.horz_wall(x, y, w,Lava)
        self.grid.horz_wall(x, y + h - 1, w,Lava)
        self.grid.vert_wall(x, y, h,Lava)
        self.grid.vert_wall(x + w - 1, y, h,Lava)

        #Place the agent
        if self.agent_start_pos is not None:
            self.agent_pos = self.agent_start_pos
            self.agent_dir = self.agent_start_dir
        else:
            self.place_agent()

        #Set the maze ends with the correct rewards
        self.set_reward_pos(self.high_reward_end)
        #Set the home square
        self.put_obj(Goal(),self.agent_start_pos[0],
self.agent_start_pos[1])

    def set_reward_pos(self, high_reward_end):
        self.high_reward_end = high_reward_end
        for i in range(len(self.MAZE_ENDS)):
            if i == high_reward_end:
                self.put_obj(MazeEnd('high'),self.MAZE_ENDS[i][0],
self.MAZE_ENDS[i][1])
            else:
                self.put_obj(MazeEnd('low'), self.MAZE_ENDS[i][0],
self.MAZE_ENDS[i][1])


    def step(self, action):
        #Some of the following code is copied from MiniGrid's step function
        self.step_count += 1
        reward = 0
        done = False

        # Rotate left
        if action == self.actions.left:
            self.agent_dir -= 1
            if self.agent_dir < 0:
                self.agent_dir += 4
            action = self.actions.forward

        # Rotate right
        elif action == self.actions.right:
            self.agent_dir = (self.agent_dir + 1) % 4
            action = self.actions.forward

        # Get the position in front of the agent
        fwd_pos = self.front_pos

        # Get the contents of the cell in front of the agent
        fwd_cell = self.grid.get(*fwd_pos)

        # Move forward
        if action == self.actions.forward:
            if fwd_cell == None or fwd_cell.can_overlap():
                self.agent_pos = fwd_pos
            if fwd_cell != None and fwd_cell.type == 'goal':
                done = True
                reward = self._reward()
            #If the agent crashes the episode ends and a negative reward is
given
            if fwd_cell != None and fwd_cell.type == 'lava':
                done = True
                reward = CRASH_REWARD
            # When we are simulating the homing T-maze task, the reward is
delayed
            if fwd_cell != None and fwd_cell.type == 'ball':
                self.reward = fwd_cell.reward
                self.grid.set(self.agent_pos[0],self.agent_pos[1],None)
                if not self.isHoming:
                    done = True
                    reward = self.reward
                self.agent_dir = (self.agent_dir + 2) % 4

        # Done action (not used by default)
        elif action == self.actions.done:
            pass

        else:
            assert False, "unknown action"

        #Checking for number of step taken to punish the agent when it
fails to return home
        if self.step_count >= self.max_steps:
            done = True
            reward = FAIL_HOME

        atHome = list(self.agent_pos) == list(self.agent_start_pos)
        atTurning = False
        for turningPoint in self.TURNING_POINTS:
            if list(turningPoint) == list(self.agent_pos):
                atTurning = True
        atMazeEnd = False
        for maze_end in self.MAZE_ENDS:
            if list(maze_end) == list(self.agent_pos):
                atMazeEnd = True
        #observation format = [is agent at home, is agent at turning point,
is agent at maze end]
        obs = (float(atHome),float(atTurning),float(atMazeEnd))
        #Comment out for debugging purposes
        #print(obs)
        return obs, reward, done, {}

    def reset(self, reward_pos = -1):
        super().reset()
        atHome = list(self.agent_pos) == list(self.agent_start_pos)
        atTurning = False
        for turningPoint in self.TURNING_POINTS:
            if list(turningPoint) == list(self.agent_pos):
                atTurning = True
        atMazeEnd = False
        for maze_end in self.MAZE_ENDS:
            if list(maze_end) == list(self.agent_pos):
                atMazeEnd = True
        # observation format = [is agent at home, is agent at turning
point, is agent at maze end]
        obs = (float(atHome), float(atTurning), float(atMazeEnd))
        #print(obs)
        if reward_pos in range(len(self.MAZE_ENDS)):
            self.set_reward_pos(reward_pos)
        return obs

    def _reward(self):
        return self.reward


class DoubleTMazeEnv(TMazeEnv):

    def __init__(self, high_reward_end = 0, isHoming = False):
        super().__init__(high_reward_end=high_reward_end,size=9, isHoming=
isHoming)
        if self.isHoming:
            self.max_steps = 23
        else:
             self.max_steps = 12


    def _gen_grid(self, width, height):
        if self.agent_start_pos is not None:
            self.agent_pos = self.agent_start_pos
            self.agent_dir = self.agent_start_dir
        else:
            self.place_agent()

        self.mission = "Find the highest reward"
        self.END1 = (1, self.height - 2)
        self.END2 = (1, 1)
        self.END3 = (self.width - 2, 1)
        self.END4 = (self.width - 2, self.height - 2)
        self.MAZE_ENDS = [self.END1, self.END2, self.END3, self.END4]
        self.middle = self.width // 2
        self.TURN1 = (self.middle, self.middle)
        self.TURN2 = (1, self.middle)
        self.TURN3 = (self.width-2, self.middle)
        self.TURNING_POINTS = [self.TURN1, self.TURN2, self.TURN3]
        self.reward = 0
        self.grid = Grid(width, height)
        for i in range(width):
            for j in range(height):
                self.put_obj(Lava(), i, j)

        x, y = self.agent_start_pos[0], self.agent_start_pos[1]
        self.put_obj(Goal(), x, y)
        while (y < self.TURN1[1]):
            y += 1
            self.grid.set(x, y, None)

        i = 0
        while x + i < self.TURN3[0]:
            i += 1
            self.grid.set(x+i, y, None)
            self.grid.set(x-i,y,None)

        j = 0
        while y+j < self.END4[1]:
            j+=1
            self.grid.set(x + i, y+j, None)
            self.grid.set(x - i, y+j, None)
            self.grid.set(x + i, y-j, None)
            self.grid.set(x - i, y-j, None)

        self.set_reward_pos(self.high_reward_end)
        self.put_obj(Goal(),self.agent_start_pos[0],
self.agent_start_pos[1])

if __name__ == "__main__":
    #Matplotlib crashes on render for now but it is not going to be a
problem
    tmaze = TMazeEnv()
    tmaze.set_reward_pos(1)
    tmaze.render()
    x = input("Press enter when finished")

#The following code is executed when this file is imported and is necessary
for this environment to work
#with manual_control.py
class TMazeHoming(TMazeEnv):

    def __init__(self):
        super().__init__(isHoming = True)
```

association_task.py

```
import random

import gym
import numpy as np
from gym import Space
from gym.utils import seeding


class BinaryList(Space):

    def __init__(self, n, one_hot=False):
        super().__init__()
        self.n = n
        self.actions = BinaryList._gen_bin_space(n, one_hot)
        self.seed()

    def sample(self):
        return self.actions[np.random.choice(range(self.n))]

    def contains(self, x):
        for a in self.actions:
            if tuple(a) == tuple(x):
                return True
        return False

    def _to_list(self):
        return self.actions[:]

    @staticmethod
    def _bit_tuple(x, padding=0):
        return tuple([1 if digit == '1' else 0 for digit in
bin(x)[2:].rjust(padding, '0')])

    @staticmethod
    def _gen_bin_space(n, one_hot=False):
        bin_space = []
        if one_hot:
            for i in range(n):
                bin_space.append(BinaryList._bit_tuple(2 ** i, n))
        else:
            for i in range(2 ** n):
                bin_space.append(BinaryList._bit_tuple(i, n))
        return bin_space

#If the agent guessed correctly a reward of 0 is given, else -1

class AssociationTaskEnv(gym.Env):

    metadata = {'render.modes': ['human']}
    step_count = 0

    def __init__(self, input_num, output_num, mode='one-to-one', rand_inter
= 0):

        self.n = input_num
        self.m = output_num
        self.reward_range = (-1,0)
        self.rand_inter = rand_inter
        self.obs_gen = self.next_obs()
        if mode == 'one-to-one':
            self.action_space = BinaryList(output_num, one_hot= True)
            self.observation_space = BinaryList(input_num, one_hot= True)
        elif mode == 'one-to-many':
            self.observation_space = BinaryList(input_num,one_hot=True)
            self.action_space = BinaryList(output_num,one_hot=False)
        elif mode == 'many-to-one':
            self.observation_space = BinaryList(input_num,one_hot=False)
            self.action_space = BinaryList(input_num,one_hot=True)
        elif mode == 'many-to-many':
            self.observation_space = BinaryList(input_num, False)
            self.action_space = BinaryList(input_num,False)
        else:
            raise ValueError('Unknown mode used: {}'.format(mode))
        self.associations = {}
        self.reset()

    def step(self, action):

        #print("Observation: {}, Expected: {}, Agent action:
{}".format(self.observation, self.associations[self.observation],action))
        action = tuple(action)
        isValid = False
        for output in self.action_space._to_list():
            if output == action:
                isValid = True
        if not isValid:
            raise ValueError('Invalid action taken {}'.format(str(action)))
        self.reward = -1

        if self.associations[self.observation] == action:
            self.reward = 0

        self.step_count += 1
        if self.rand_inter > 0 and self.step_count % self.rand_inter == 0:
            self.randomize_associations()

        #self.observation = list(self.associations.keys())[self.step_count
% len(self.associations)]
        self.observation = next(self.obs_gen)
        done = True

        return self.observation, self.reward, done, {}

    def reset(self, rand_iter = -1):
        self.step_count = 0
        self.randomize_associations()
        #self.observation = list(self.associations.keys())[0]
        self.observation = next(self.obs_gen)
        if rand_iter > 0:
            self.rand_inter  = rand_iter
        return self.observation

    def render(self, mode='human'):
        assert mode == 'human', '{} mode not supported'.format(mode)
        if self.step_count > 0:
            print("Reward: {}".format(self.reward))
        print("Observation: {}".format(self.observation))
        print("Actions:")
        print(self.action_space._to_list())

    def randomize_associations(self):
        self.associations = {}
        input = [i for i in self.observation_space._to_list()]
        output = [o for o in self.action_space._to_list()]
        np.random.shuffle(input)
        for pattern in input[:]:
            idx = np.random.choice(range(len(output)))
            self.associations[pattern] = output[idx]
            input.remove(pattern)
            output.pop(idx)
            if len(output) == 0:
                output = [o for o in self.action_space._to_list()]

        return self.associations

    def next_obs(self):
        while(True):
            obs = self.observation_space._to_list()
            random.shuffle(obs)
            while(len(obs) > 0):
                yield obs.pop()

class OneToOne2x2(AssociationTaskEnv):
    def __init__(self):
        super().__init__(2,2)

class OneToOne3x3(AssociationTaskEnv):
    def __init__(self):
        super().__init__(3,3)

class OneToMany3x2(AssociationTaskEnv):
    def __init__(self):
        super().__init__(3,2,'one-to-many')

class ManyToMany2x2Rand(AssociationTaskEnv):
    def __init__(self):
        super().__init__(2,2,'many-to-many',500)

if __name__ == '__main__':
    env = OneToMany3x2()
    while True:
        env.render(mode='human')
        x = input("Select action with binary string e.g. 011 or quit: ")
        if x == 'quit' or x == 'Quit' or x =='q':
            exit(0)
        action = tuple([int(c) for c in x])
        env.step(action)
```

eval.py

```python
import copy
import math
import random

import gym
import gym_association_task
import t_maze
from functools import partial
###
#All the following evaluation functions take as argument an agent variable.
It is assumed that the agent has
#an activate function which takes as input a vector (list) and returns an
output which corresponds to the action
#of the agent. The actions are described in each environment
##############


from utilities import shuffle_lists


def eq_tuples(tup1, tup2):
    for x,y in zip(tup1, tup2):
        if x != y:
            return False
        return True

def eq_snapshots(s1,s2):
    for key in s1:
        if not eq_tuples(s1[key], s2[key]):
            return False
    return True

#For a network to be considered to be able to solve the one-to-one 3x3
association task in this case it needs to
#to achieve a score of at least 1976 (2000 - (5*(3*2))  = steps -
(association_changes+1)*(n*(m-1)).
#Note that scores above this threshold do not mean better performance since
the score of 1976 is already considered optimal.
#The network here needs to accept 4 inputs (3 for observation and 1 for
reward) and return a vector with 3 binary values.
#For num_episodes = 100 | 1000
#    rand_iter = 25     | 100
#    max fitness = 70   | 934
def eval_one_to_one_3x3(agent, num_episodes = 1000, rand_iter=
100,snapshot_inter=50, descriptor_out=False):
    env = gym.make('OneToOne3x3-v0')
    s = num_episodes
    observation = env.reset(rand_iter=rand_iter)
    input = tuple(list(observation) + [0])
    responses = {}
    prevsnapshot = {}
    bd = []
    for i_episode in range(num_episodes):
        action = agent.activate(input)
        if descriptor_out:
            t_in = input[:-1]
            if t_in not in prevsnapshot:
                prevsnapshot[t_in] = action
            responses[t_in] = action
            if i_episode != snapshot_inter and i_episode%snapshot_inter ==
0 and i_episode>0:
                if eq_snapshots(responses, prevsnapshot):
                    bd.append(0)
                else:
                    bd.append(1)
                prevsnapshot = responses
        observation, reward, done, info = env.step(action)
        input = list(input)
        input[-1] = reward
        agent.activate(input)
        input = tuple(list(observation) + [0])
        s += reward
    env.close()
    if descriptor_out:
        return s, bd
        #print(bd)
    else:
        return s

#For a network to be considered to be able to solve the one-to-many 3x2
association task in this case it needs to
#to achieve a score of at least 1964 (2000 - 4*(3*(4-1)) = steps -
association_changes*(n*(2^m - 1)).
#Note that scores above this threshold do not mean better performance since
the score of 1964 is already considered optimal.
#The network accepts 4 inputs (3 for observation and 1 for reward)  and
return a vector with two binary values.
def eval_one_to_many_3x2(agent):
    env = gym.make('OneToMany3x2-v0')
    num_episodes = 2000
    sum = num_episodes
    observation = env.reset(rand_iter=500)
    input = list(observation)
    input.append(0)
    for i_episode in range(num_episodes):
        action = agent.activate(input)
        observation, reward, done, info = env.step(action)
        input = list(input)
        input[-1] = reward
        agent.activate(input)
        input = list(observation)
        input.append(0)
        sum += reward
    env.close()
    return sum

def int_to_action(x):
    if x == 0:
        return "Left"
    if x ==1:
        return "Right"
    return "Forward"

#Assuming 8 episodes and one switch, the optimal fitness for an agent would
be 6 * 1 + 2 * 0.2 = 6.4
#The one switch will occur at 2 + y
class TmazeEvaluator():
    DOMAIN_CONSTANT = 2
    def __init__(self, num_episodes = 8,samples = 4, debug =  False,
descriptor_out = False):
        self.maxparam = num_episodes - 2*self.DOMAIN_CONSTANT
        self.param_list = [i for i in range(0,self.maxparam+1)]
        self.samples = samples
        self.params = random.sample(self.param_list, self.samples)
        self.num_episodes = num_episodes
        self.debug = debug
        self.descriptor_out = descriptor_out
        self.eval_func = self.eval_tmaze

    def eval_tmaze(self, agent):

        env = gym.make('MiniGrid-TMaze-v0')  #init environment
        s = 0       #s = total reward
        pos = 0     #pos = the initial position of the high reward
        bd = []     # behavioural descriptor

        for param in self.params:
            #pos = 0
            for i_episode in range(self.num_episodes):
                reward = 0
                #swap the position of the high reward every s_inter steps
                if i_episode == self.DOMAIN_CONSTANT + param:
                    pos = (pos + 1) % 2
                observation = env.reset(reward_pos= pos)
                #print(f"y = {param}, pos = {pos}")
                #append 0 for the reward
                input = list(observation)
                input.append(0)
                done = False
                #DEBUG INFO
                if self.debug:
                    print("Episode: {}".format(i_episode))
                    print("High pos: {}".format(pos))
                while not done:
                    action = agent.activate(input)
                    observation, reward, done, info = env.step(action)
                    input = list(observation)
                    input.append(reward)
                    #DEBUG INFO
                    if self.debug:
                        print("     {}".format(int_to_action(action)))
                if self.debug:
                    print(input)
                s += reward
                #Add this episode to the behavioural descriptor
                if self.descriptor_out:
                    if math.isclose(reward, t_maze.LOW):
                        des = 'l'
                    elif math.isclose(reward, t_maze.HIGH):
                        des = 'h'
                    else:
                        des = 'n'
                    if math.isclose(reward, t_maze.CRASH_REWARD):
                        des += 'y'
                    else:
                        des += 'n'
                    bd.append(des)
                agent.activate(input)
                #DEBUG INFO
                if self.debug:
                    print("Reward: {}".format(reward))
                    print("---------------")
        env.close()
        if self.debug:
            print(f"Total reward: {s}")
        if self.descriptor_out:
            return s, bd
        return s



class DoubleTmazeEvaluator():

    DOMAIN_CONSTANT = 4
    def __init__(self, num_episodes=12, samples=4, debug=False,
descriptor_out=False):

        self.maxparam = num_episodes - 2 * self.DOMAIN_CONSTANT
        self.param_list = [i for i in range(0,self.maxparam+1)]
        self.samples = samples
        self.params = random.sample(self.param_list, self.samples)
        self.num_episodes = num_episodes
        self.debug = debug
        self.descriptor_out = descriptor_out
        self.eval_func = self.eval_double_tmaze

    #num episodes = 12
    #samples 4
    #max fitness = 36
    def eval_double_tmaze(self, agent):

        env = gym.make('MiniGrid-DoubleTMaze-v0')  #init environment
        env = env.unwrapped
        s = 0         #s = total reward
        pos = 0       #pos = the initial position of the high reward
        bd = []       # behavioural descriptor
        maze_ends = [0,1,2,3]

        for param in self.params:
            for i_episode in range(self.num_episodes):
                reward = 0
                #swap the position of the high reward every s_inter steps
                if i_episode == self.DOMAIN_CONSTANT + param:
                    choices = copy.deepcopy(maze_ends)
                    choices.remove(pos)
                    pos = random.choice(choices)

                observation = env.reset(reward_pos= pos)
                #append 0 for the reward
                input = list(observation)
                input.append(0)
                done = False
                #DEBUG INFO
                if self.debug:
                    print("Episode: {}".format(i_episode))
                    print("High pos: {}".format(pos))
                while not done:
                    action = agent.activate(input)
                    observation, reward, done, info = env.step(action)
                    input = list(observation)
                    input.append(reward)
                    #DEBUG INFO
                    if self.debug:
                        print("     {}".format(int_to_action(action)))
                if self.debug:
                    print(input)
                s += reward
                #Add this episode to the behavioural descriptor
                if self.descriptor_out:
                    if eq_tuples(env.agent_pos, env.END1):
                        des = 1
                    elif eq_tuples(env.agent_pos, env.END2):
                        des = 2
                    elif eq_tuples(env.agent_pos, env.END3):
                        des = 3
                    elif eq_tuples(env.agent_pos, env.END4):
                        des = 4
                    else:
                        des = 0
                    bd.append(des)
                agent.activate(input)
                #DEBUG INFO
                if self.debug:
                    print("Reward: {}".format(reward))
                    print("--------------")
        env.close()
        if self.debug:
            print(f"Total reward: {s}")
        if self.descriptor_out:
            return s, bd
        return s

class HomingTmazeEvaluator():

    DOMAIN_CONSTANT = 2
    def __init__(self, num_episodes=8, samples=4, debug=False,
descriptor_out=False):

        self.maxparam = num_episodes - 2 * self.DOMAIN_CONSTANT
        self.param_list = [i for i in range(0,self.maxparam+1)]
        self.samples = samples
        self.params = random.sample(self.param_list, self.samples)
        self.num_episodes = num_episodes
        self.debug = debug
        self.descriptor_out = descriptor_out
        self.eval_func = self.eval_tmaze_homing

    def eval_tmaze_homing(self, agent):

        env = gym.make('MiniGrid-TMazeHoming-v0')  #init environment
        s = 0       #s = total reward
        pos = 0     #pos = the initial position of the high reward
        bd = []     # behavioural descriptor

        for param in self.params:
            for i_episode in range(self.num_episodes):
                reward = 0
                #swap the position of the high reward every s_inter steps
                if i_episode == self.DOMAIN_CONSTANT + param:
                    pos = (pos + 1) % 2
                observation = env.reset(reward_pos= pos)
                #append 0 for the reward
                input = list(observation)
                input.append(0)
                done = False
                #DEBUG INFO
                if self.debug:
                    print("Episode: {}".format(i_episode))
                    print("High pos: {}".format(pos))
                while not done:
                    action = agent.activate(input)
                    observation, reward, done, info = env.step(action)
                    input = list(observation)
                    input.append(reward)
                    #DEBUG INFO
                    if self.debug:
                        print("     {}".format(int_to_action(action)))
                if self.debug:
                    print(input)
                s += reward
                #Add this episode to the behavioural descriptor
                if self.descriptor_out:
                    if math.isclose(reward, t_maze.LOW):
                        des = 'l'
                    elif math.isclose(reward, t_maze.HIGH):
                        des = 'h'
                    else:
                        des = 'n'
                    if math.isclose(reward, t_maze.CRASH_REWARD):
                        des += 'y'
                    else:
                        des += 'n'
                    if math.isclose(reward, t_maze.FAIL_HOME):
                        des += 'n'
                    else:
                        des += 'y'
                    bd.append(des)

                agent.activate(input)
                #DEBUG INFO
                if self.debug:
                    print("Reward: {}".format(reward))
                    print("--------------")
            env.close()
            if self.debug:
                print(f"Total reward: {s}")
            if self.descriptor_out:
                return s, bd
            return s

#The simplest case for test the network's implementation. Use when in doubt
about the rest.
#Optimal network should have a score of 4 or very close to 4.
#The network takes 2 input and returns one output

xor_inputs = [(0.0, 0.0), (0.0, 1.0), (1.0, 0.0), (1.0, 1.0)]
xor_outputs = [   (0.0,),     (1.0,),     (1.0,),     (0.0,)]
```

```
TRIALS = 10

def eval_net_xor(net):
    sum = 0
    for i in range(TRIALS):
        fitness = 4
        trial_in, trial_out = shuffle_lists(xor_inputs, xor_outputs)
        for xi, xo in zip(trial_in, trial_out):
            output = net.activate(xi)
            fitness -= abs(output[0] - xo[0])
        sum += fitness
    return sum/TRIALS

class NoveltyEvaluator():

    #distance_func: a function that computes the distance of two
behavioural descriptors
    #k: the number used for the nearest neighbour calculation
    #threshold: the sparseness threshold for a gene to enter the archive
    #The archive has the key of the genome as the key and another
dictionary as values with bd, novelty and fitness, genome
    def __init__(self, eval_func, threshold = 1, k = 15):
        self.eval_func = eval_func
        self.threshold = threshold
        self.k = k
        self.archive = {}
        self.visited_novelty = {}

    def distance_func(self, bd1, bd2):
        assert False, "NoveltyEvaluator.distance_func needs to be
overloaded!"

    def get_best_id(self):

        maxid = list(self.archive.keys())[0]
        maxfitness = self.archive[maxid]['fitness']
        for key in self.archive:
            if self.archive[key]['fitness'] > maxfitness:
                maxfitness = self.archive[key]['fitness']
                maxid = key

        return maxid

    #Find the novelty score for an agent.
    def eval(self,key,genome ,agent):
        #if key in self.visited_novelty:
        #    return self.visited_novelty[key]
        fitness, bd = self.eval_func(agent)

        cache = []
        if not self.archive:
            self.archive[key] = {'bd': bd, 'novelty': len(bd),
'fitness':fitness, 'agent' : agent, 'genome': genome}
            return len(bd)
        for k in self.archive:
            dist = self.distance_func(bd, self.archive[k]['bd'])
            if len(cache) > self.k and dist < min(cache):
                cache.remove(min(cache))
            if len(cache) < self.k:
                cache.append(dist)

        novelty = sum(cache) / len(cache)
        self.visited_novelty[key] = novelty
        if novelty > self.threshold:
            self.archive[key] = {'bd': bd, 'novelty': novelty,
'fitness':fitness, 'agent': agent}
        return novelty

    def reevaluate_archive(self):
        if not self.archive:
            return
        for key in self.archive:
            agent = self.archive[key]['agent']
            fitness, bd = self.eval_func(agent)
            self.archive[key]['bd'] = bd
            self.archive[key]['fitness'] = fitness
            #print(bd)

class TmazeNovelty(NoveltyEvaluator):

    def __init__(self, n_episodes, samples, threshold = 8):
        self.evaluator =
TmazeEvaluator(num_episodes=n_episodes,samples=samples, debug =  False,
descriptor_out = True)
        eval_func = self.evaluator.eval_tmaze
        super().__init__(eval_func, threshold=threshold)

    def distance_func(self, bd1, bd2):
        total = 0
        for t1, t2 in zip(bd1, bd2):
            for i in range(len(t1)):
                if t1[i] != t2[i]:
                    total += 1

        return total

class DoubleTmazeNovelty(NoveltyEvaluator):

    def __init__(self, n_episodes, samples, threshold=10):
        self.evaluator =
DoubleTmazeEvaluator(num_episodes=n_episodes,samples=samples,debug=False,de
scriptor_out=True)
        eval_func = self.evaluator.eval_double_tmaze
        super().__init__(eval_func, threshold=threshold)

    def distance_func(self, bd1, bd2):
        score = 0
        for i, j in zip(bd1, bd2):
            if i != j:
                score += 1
        return score

class HomingTmazeNovelty(NoveltyEvaluator):

    def __init__(self, n_episodes, samples, threshold=1):
        self.evaluator = HomingTmazeEvaluator(num_episodes=n_episodes,
samples=samples,debug=False, descriptor_out=True)
        eval_func = self.evaluator.eval_tmaze_homing
        super().__init__(eval_func, threshold=threshold)
    def distance_func(self, bd1, bd2):
        total = 0
        for t1, t2 in zip(bd1, bd2):
            for i in range(len(t1)):
                if t1[i] != t2[i]:
                    total += 1

        return total

class AssociationNovelty(NoveltyEvaluator):

    def __init__(self, num_episodes, rand_iter,snapshot_inter,threshold=2):
        eval_f = partial(eval_one_to_one_3x3,num_episodes = num_episodes,
rand_iter= rand_iter,
                                                     snapshot_inter=snapshot_inter,
descriptor_out=True)
        super().__init__(eval_f, threshold=threshold)

    def distance_func(self, bd1, bd2):
        sum = 0
        for x,y in zip(bd1,bd2):
            if x!=y:
                sum += 1
        return sum
```

evolve.py

```
import pickle
import argparse
from functools import partial, partialmethod
import gym_association_task
from eval import eval_one_to_one_3x3, eval_net_xor, TmazeNovelty, \
    DoubleTmazeNovelty, HomingTmazeNovelty, TmazeEvaluator,
DoubleTmazeEvaluator, HomingTmazeEvaluator, \
    AssociationNovelty
import switch_neat
from maps import MapNetwork, MapGenome
import switch_maps
from recurrent_neat import RecurrentNetwork
from solve import convert_to_action, convert_to_direction
import neat
import Reporters
from utilities import heaviside
from switch_neuron import Agent
import render_network
from neat.statistics import StatisticsReporter

def identity(x):
    return x

def main():
    schemes = {'switch':switch_neat.create , 'maps' : MapNetwork.create,
'recurrent': RecurrentNetwork.create,
               'switch_maps' : switch_maps.create}
    problems = {'xor' : eval_net_xor,
'binary_association':eval_one_to_one_3x3, 'tmaze':
TmazeEvaluator().eval_tmaze,
                'double_tmaze':
                DoubleTmazeEvaluator.eval_double_tmaze, 'homing_tmaze':
HomingTmazeEvaluator().eval_tmaze_homing}

    domain_constant = {'tmaze': 2, 'double_tmaze': 4, 'homing_tmaze':2}

    parser = argparse.ArgumentParser(description="Evolve neural networks
with neat")
    parser.add_argument('-s', '--scheme', help=f"Choose between the
available schemes: {','.join(schemes.keys())}",
                         type=str, required=True, choices=schemes.keys())
    parser.add_argument('-c', '--config', help="The config file",
required=True, type=str)
    parser.add_argument('-g', '--generations', help="The number of
generations", type=int)
    parser.add_argument('-p', '--problem', help=f"Available problems:
{','.join(problems.keys())}", required=True, type=str,
                        choices=problems.keys())
    parser.add_argument('--map_size', help="Set the map size for the
relevant schemes", type=int)
    parser.add_argument('--dump', help="Dump the network in a binary file",
type=str)
    parser.add_argument('--num_episodes', help="Number of episodes for
tmaze/binary_association", type=int)
    parser.add_argument('--switch_interval', help="Interval of episodes for
"
                                                  "shuffling the
associations", type=int )
```

```
    parser.add_argument('--novelty', help='Use the novelty metric instead
of the fitness function', action="store_true")
    parser.add_argument('--threshold', help='Threshold for a new genome to
enter the archive', type=float, default=1)
    parser.add_argument("--snap_inter", help="Snapshot interval for
association problem novelty search", type = int)
    parser.add_argument("--draw", help='Render the network to a picture.
Provide the name of the picture.', type=str, default=None)
    parser.add_argument("--log", help="Log the max fitness per generation
to text file. (Append)", type=str, default=None)
    args=parser.parse_args()

    eval_f = problems[args.problem]
    in_f = identity
    out_f = identity
    genome = neat.DefaultGenome
    evaluator = None
    #Configure genome based on the encoding scheme and neurons used
    if args.scheme == 'switch':
        genome = switch_neat.SwitchGenome
    elif args.scheme == 'maps':
        genome = MapGenome
    elif args.scheme == 'switch_maps':
        genome = switch_maps.SwitchMapGenome

    #Configure the pre-processing and post-processing functions based on
    #the environment
    if args.problem == 'binary_association':
        out_f = convert_to_action
    elif args.problem in ['tmaze', 'double_tmaze', 'homing_tmaze'] :
        out_f = convert_to_direction

    #If we use the map-based encoding scheme add the map size parameter to
the function
    #responsible for creating the network from the genotype.
    create_f = None
    if args.map_size is not None and (args.scheme in ['maps',
'switch_maps']):
        create_f = partial(schemes[args.scheme], map_size=args.map_size)
    else:
        create_f = schemes[args.scheme]

    num_episodes = 100
    s_inter = 20
    if args.num_episodes is not None:
        num_episodes = args.num_episodes
    if args.switch_interval is not None:
        s_inter = args.switch_interval
    #If the problem is the t-maze task, use the extra parameters episodes
and switch interval
    if args.problem == 'tmaze':
        if args.novelty:
            evaluator = TmazeNovelty(num_episodes,samples=4,
threshold=args.threshold)
            eval_f = evaluator.eval
        else:
            evaluator = TmazeEvaluator(num_episodes, samples=4)
            eval_f = evaluator.eval_tmaze
    elif args.problem == 'double_tmaze':
        if args.novelty:
            evaluator = DoubleTmazeNovelty(num_episodes,samples=4,
threshold=args.threshold)
            eval_f = evaluator.eval
        else:
            evaluator = DoubleTmazeEvaluator(num_episodes, samples=4)
            eval_f = evaluator.eval_double_tmaze
    elif args.problem == 'homing_tmaze':
        if args.novelty:
            evaluator = HomingTmazeNovelty(num_episodes,samples=4,
threshold=args.threshold)
            eval_f = evaluator.eval
        else:
            evaluator = HomingTmazeEvaluator(num_episodes, samples=4)
            eval_f = evaluator.eval_tmaze_homing
    elif args.problem == 'binary_association':
        if args.novelty:
            evaluator =
AssociationNovelty(num_episodes,rand_iter=args.switch_interval,snapshot_int
er=args.snap_inter,
                                            threshold=args.threshold)
            eval_f = evaluator.eval
        else:
            eval_f = partial
(eval_one_to_one_3x3,num_episodes=num_episodes, rand_iter=s_inter)

    def make_eval_fun(evaluation_func, in_proc, out_proc, evaluator=None):

        def eval_genomes (genomes, config):
            for genome_id, genome in genomes:
                net = create_f(genome,config)
                #Wrap the network around an agent
                agent = Agent(net, in_proc, out_proc)
                #Evaluate its fitness based on the function given above.
                genome.fitness = evaluation_func(agent)

        def eval_genomes_novelty(genomes, config):
            evaluator.reevaluate_archive()
            for genome_id, genome in genomes:
                net = create_f(genome,config)
                #Wrap the network around an agent
                agent = Agent(net, in_proc, out_proc)
                #Evaluate its fitness based on the function given above.
                genome.fitness = evaluation_func(genome_id, genome, agent)

        if args.novelty:
            return eval_genomes_novelty
        else:
            return eval_genomes

    config = neat.Config(genome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
                         args.config)
    config.genome_config.add_activation('heaviside', heaviside)


    # Create the population, which is the top-level object for a NEAT run.
    p = neat.Population(config)
    #Add a stdout reporter to show progress in the terminal.
    p.add_reporter(neat.StdOutReporter(True))
    stats = StatisticsReporter()
    p.add_reporter(stats)
    if args.problem in  ['double_tmaze', 'tmaze', 'homing_tmaze']:
        if args.novelty:
            mutator = Reporters.EvaluatorMutator(evaluator.evaluator)
        else:
            mutator = Reporters.EvaluatorMutator(evaluator)
        p.add_reporter(mutator)

    # Run for up to ... generations.
    if args.novelty:
        f = make_eval_fun(eval_f, in_f, out_f, evaluator)
    else:
        f = make_eval_fun(eval_f, in_f, out_f)
    winner = p.run(f, args.generations)

    #If we are using the novelty metric get the winner from the archive
    if args.novelty:
        winnerid = evaluator.get_best_id()
        winner = evaluator.archive[winnerid]['genome']
        winner_agent = evaluator.archive[winnerid]['agent']
    else:
        print('\nBest genome:\n{!s}'.format(winner))
        winner_net = create_f(winner, config)
        winner_agent = Agent(winner_net,in_f, out_f)


    if args.novelty:
        if args.problem == 'binary_association':
            score = evaluator.eval_func(winner_agent)[0]
        else:
            score = evaluator.evaluator.eval_func(winner_agent)[0]
    else:
        score = eval_f(winner_agent)
    print("Score in task: {}".format(score))
    if args.draw is not None:
        if args.scheme in ['maps', 'switch_maps']:
            map_size = args.map_size
        else:
            map_size = -1
        render_network.draw_net(config, winner,filename = args.draw,
map_size=map_size)

    if args.log is not None:
        fp = open(args.log, 'a')
        best_fitness = [str(c.fitness) for c in stats.most_fit_genomes]
        mfs = ' '.join(best_fitness)
        fp.write(mfs)
        fp.write("\n")
        fp.close()

    if args.dump is not None:
        fp = open(args.dump,'wb')
        pickle.dump(winner,fp)
        fp.close()
        print(f'Agent pre-processing function: {in_f.__name__}')
        print(f'Agent post-processing function: {out_f.__name__}')

if __name__ == "__main__":
    main()
```

maps.py

```
from neat.attributes import FloatAttribute, BoolAttribute, StringAttribute
from neat.genes import BaseGene, DefaultNodeGene
from neat.genome import DefaultGenomeConfig, DefaultGenome
from neat.graphs import required_for_output
from neat.six_util import itervalues
import numpy as np
from switch_neuron import Neuron
from utilities import order_of_activation

class MapNetwork:

    #A lot of this code is taken directly from neat.nn.RecurrentNetwork and 
modified because it is very close to
    #the desired outcome
    def __init__(self,inputs, outputs, nodes):
        self.input_nodes = inputs
        self.output_nodes = outputs
        self.nodes = nodes
        self.nodes_dict = {}
        for node in nodes:
            self.nodes_dict[node.key] = node


    @staticmethod
    def create(genome, config, map_size):
        """ Receives a genome and returns its phenotype (a MapNetwork). """
        genome_config = config.genome_config
        required = required_for_output(genome_config.input_keys, 
genome_config.output_keys, genome.connections)

        # Gather inputs and expressed connections.
        node_inputs = {}
        children = {}
        node_keys = list(genome.nodes.keys())[:] #+ 
list(genome_config.input_keys[:])
        # for key in genome_config.input_keys + genome_config.output_keys:
        #     children[key] = []
        #     for i in range(1,map_size):
        #         if key < 0:
        #             new_idx = min(node_keys) - 1
        #         else:
        #             new_idx = max(node_keys) + 1
        #         children[key].append(new_idx)
        #         node_keys.append(new_idx)

        for cg in itervalues(genome.connections):
            if not cg.enabled:
                continue

            i, o = cg.key
            if o not in required and i not in required:
                continue

            for n in [i,o]:
                if n in children.keys():
                    continue
                children[n] = []
                if n in genome_config.input_keys or n in
genome_config.output_keys:
                    continue
                if not genome.nodes[n].is_isolated:
                    for _ in range(1,map_size):
                        new_idx = max(node_keys) + 1
                        children[n].append(new_idx)
                        node_keys.append(new_idx)

            in_map = [i] + children[i]
            out_map = [o] + children[o]
            for n in out_map:
                if n not in node_inputs.keys():
                    node_inputs[n] = []

            if len(in_map) == map_size and len(out_map) == map_size:
                #Map to map connectivity
                if cg.one_to_one:
                    #1-to-1 mapping
                    weight = 5*cg.weight
                    for i_n in range(map_size):
                        node_inputs[out_map[i_n]].append((in_map[i_n],
weight))

                else:
                    #1-to-all
                    if cg.is_gaussian:
                        #Gaussian
                        for o_n in out_map:
                            for i_n in in_map:
                                node_inputs[o_n].append((i_n,
np.random.normal(cg.weight,cg.sigma)))
                    else:
                        #Uniform
                        for o_n in out_map:
                            for i_n in in_map:
                                node_inputs[o_n].append((i_n, 5*cg.weight))

            else:
                #Map-to-isolated or isolated-to-isolated
                if cg.is_gaussian:
                    # Gaussian
                    for o_n in out_map:
                        for i_n in in_map:
                            node_inputs[o_n].append((i_n,
np.random.normal(cg.weight, cg.sigma)))
                else:
                    # Uniform
                    for o_n in out_map:
                        for i_n in in_map:\
                            node_inputs[o_n].append((i_n, 5 * cg.weight))

        input_keys = genome_config.input_keys
        output_keys = genome_config.output_keys
        conns = {}
        for k in genome.nodes.keys():
            if k not in node_inputs:
                node_inputs[k] = []
                if k in children:
                    for c in children[k]:
                        node_inputs[c] = []
            conns[k] = [i for i, _ in node_inputs[k]]
        sorted_keys = order_of_activation(conns, input_keys, output_keys)
        nodes = []
        for node_key in sorted_keys:
            if node_key not in genome.nodes.keys():
                continue
            node = genome.nodes[node_key]

            activation_function =
genome_config.activation_defs.get(node.activation)
            aggregation_function =
genome_config.aggregation_function_defs.get(node.aggregation)
            nodes.append(Neuron(node_key, {
                'activation_function': activation_function,
                'integration_function': aggregation_function,
                'bias': node.bias,
                'activity': 0,
                'output': 0,
                'weights': node_inputs[node_key]
            }))

            if node_key not in children:
                continue

            for n in children[node_key]:
                nodes.append(Neuron(n, {
                    'activation_function': activation_function,
                    'integration_function': aggregation_function,
                    'bias': node.bias,
                    'activity': 0,
                    'output': 0,
                    'weights': node_inputs[n]
                }))



        # for key in genome_config.input_keys:
        #     input_keys.append(key)
        #     if key in children.keys():
        #         for child in children[key]:
        #             input_keys.append(child)
        #
        # for key in genome_config.output_keys:
        #     output_keys.append(key)
        #     if key in children.keys():
        #         for child in children[key]:
        #             output_keys.append(child)

        return MapNetwork(input_keys, output_keys, nodes)

    #Perform a forward pass in the network with the given inputs. Since we
are working with recurrent networks
    #and arbitrary connections, no separation of the neurons is performed
between the neurons and the activation
    #sequence is determined from their order in the node_evals array.
    def activate(self, inputs):
        if len(self.input_nodes) != len(inputs):
            raise RuntimeError("Expected {0:n} inputs, got
{1:n}".format(len(self.input_nodes), len(inputs)))
```

```
        ivalues = {}
        for i, v in zip(self.input_nodes, inputs):
            ivalues[i] = v

        for node in self.nodes:
            standard_inputs = []
            # Collect the weighted inputs in an array
            for key, weight in node.standard['weights']:
                if key in ivalues.keys():
                    val = ivalues[key]
                else:
                    val = self.nodes_dict[key].standard['output']
                standard_inputs.append(val * weight)
            # add the bias
            standard_inputs.append(node.standard['bias'])
            # Calculate the neuron's activity and output based on it's
standard functions.
            node.standard['activity'] =
node.standard['integration_function'](standard_inputs)
            node.standard['output'] =
node.standard['activation_function'](node.standard['activity'])

        output = [self.nodes_dict[key].standard['output'] for key in
self.output_nodes]
        return output

class MapConnectionGene(BaseGene):

    #Various parameters for defining a connection.
    _gene_attributes = [BoolAttribute('one_to_one'), #1-to-1 or 1-to-all
scheme
                        BoolAttribute('is_gaussian'),  #Gaussian or uniform
distribution
                        FloatAttribute('weight'),#Weigth is used as the
mean of the normal distribution for 1-to-all
                        FloatAttribute('sigma'), #The standard deviation
for the gaussian
                        BoolAttribute('enabled')] #<- maybe remove this
trait

    def __init__(self, key):
        assert isinstance(key, tuple), "DefaultConnectionGene key must be a
tuple, not {!r}".format(key)
        BaseGene.__init__(self, key)

    #Define the distance between two genes
    def distance(self, other, config):
        d = abs(self.sigma - other.sigma) + abs(self.weight - other.weight)
+ int(self.one_to_one != other.one_to_one)
        + int(self.is_gaussian != other.is_gaussian)
        return d * config.compatibility_weight_coefficient

class MapNodeGene(DefaultNodeGene):

    _gene_attributes = [FloatAttribute('bias'), #The bias of the neuron
                        StringAttribute('activation', options='sigmoid'), #
The activation function, tunable from the config
                        StringAttribute('aggregation', options='sum'), #The
aggregation function
```

```
                        BoolAttribute('is_isolated')] #Map vs isolated
neuron

    def distance(self, other, config):
        d = 0
        if self.activation != other.activation:
            d += 1.0
        if self.aggregation != other.aggregation:
            d += 1.0
        if self.is_isolated != other.is_isolated:
            d += 1
        return d * config.compatibility_weight_coefficient

class MapNode:

    def __init__(self,key, activation_function, aggregation_function,
bias,is_isolated, links):
        self.key = key
        self.activation_function = activation_function
        self.aggregation_function = aggregation_function
        self.bias = bias
        self.is_isolated = is_isolated
        self.links = links
        self.activity = 0

class MapGenome(DefaultGenome):
    @classmethod
    def parse_config(cls, param_dict):
        param_dict['node_gene_type'] = MapNodeGene
        param_dict['connection_gene_type'] = MapConnectionGene
        return DefaultGenomeConfig(param_dict)
```

render_network.py

```
import copy
import warnings
import graphviz



def draw_net(config, genome, view=False, filename=None, node_names=None,
show_disabled=True, prune_unused=False,
             node_colors=None, fmt='svg', map_size = -1):
    """ Receives a genome and draws a neural network with arbitrary
topology. """
    # Attributes for network nodes.
    if graphviz is None:
        warnings.warn("This display is not available due to a missing
optional dependency (graphviz)")
        return

    if node_names is None:
        node_names = {}

    assert type(node_names) is dict

    if node_colors is None:
        node_colors = {}

    assert type(node_colors) is dict

    node_attrs = {
        'shape': 'circle',
        'fontsize': '9',
        'height': '0.1',
        'width': '0.1'}


    dot = graphviz.Digraph(format=fmt, node_attr=node_attrs,
engine='neato')
    maps = {}
    inputs = set()
    for k in config.genome_config.input_keys:
        inputs.add(k)
        name = node_names.get(k, str(k))
        input_attrs = {'style': 'filled',
                       'shape': 'box'}
        input_attrs['fillcolor'] = node_colors.get(k, 'lightgray')
        dot.node(name, _attributes=input_attrs)
        maps[str(k)] = [str(k)]

    outputs = set()
    for k in config.genome_config.output_keys:
        outputs.add(k)
        name = node_names.get(k, str(k))
        node_attrs = {'style': 'filled'}
        node_attrs['fillcolor'] = node_colors.get(k, 'lightblue')
        node_attrs['shape'] = 'doublecircle' if genome.nodes[k].is_switch
else 'circle'

        dot.node(name, _attributes=node_attrs)
        maps[str(k)] = [str(k)]

    if prune_unused:
        connections = set()
        for cg in genome.connections.values():
            if cg.enabled or show_disabled:
                connections.add((cg.in_node_id, cg.out_node_id))

        used_nodes = copy.copy(outputs)
        pending = copy.copy(outputs)
        while pending:
            new_pending = set()
            for a, b in connections:
                if b in pending and a not in used_nodes:
                    new_pending.add(a)
                    used_nodes.add(a)
            pending = new_pending
    else:
        used_nodes = set(genome.nodes.keys())




    for n in used_nodes:
        if n in inputs or n in outputs:
            continue

        attrs = {'style': 'filled',
                 'fillcolor': node_colors.get(n, 'white')}
        attrs['shape'] = 'doublecircle' if genome.nodes[n].is_switch else
'circle'
        dot.node(str(n),
label=str(genome.nodes[n].activation),_attributes=attrs)
        if map_size > 0:
            if not genome.nodes[n].is_isolated:
                maps[str(n)] = [str(n) + ' ' + str(i) for i in
range(map_size)]
                maps[str(n)].append(str(n))
            else:
                maps[str(n)] = [str(n)]

    for cg in genome.connections.values():
        if cg.enabled or show_disabled:
            #if cg.input not in used_nodes or cg.output not in used_nodes:
            #    continue
            input, output = cg.key
            a = node_names.get(input, str(input))
            b = node_names.get(output, str(output))
            style = 'solid' if not cg.is_mod else 'dotted'
            color = 'green' if cg.weight > 0 else 'red'
            width = str(0.1 + abs(cg.weight / 5.0))
            fontsize = '5'
            eattrs = {'style': style, 'color': color, 'penwidth': width,
'fontsize': fontsize}

            if map_size > 0:
                in_map = maps[a]
                out_map = maps[b]
                if len(in_map) == map_size and len(out_map) == map_size:
                    # Map to map connectivity
                    if cg.one2one:
                        # 1-to-1 mapping
                        for i in range(map_size):
                            dot.edge(in_map[i], out_map[i], label=
f"{cg.weight:.3f}",
                                         _attributes=eattrs)

                    else:
                        # 1-to-all
                        if not cg.uniform:
                            # Step
                            start = -cg.weight
                            end = cg.weight
                            step = (end - start) / map_size
                            for o_n in out_map:
                                s = start
                                for i_n in in_map:
                                    dot.edge(i_n, o_n, label= f"{s:.3f}",
                                                 _attributes=eattrs)
                                    s += step
                        else:
                            # Uniform
                            for o_n in out_map:
                                for i_n in in_map:
                                    dot.edge(i_n, o_n, label=
f"{cg.weight:.3f}",
                                                 _attributes=eattrs)

                else:
                    # Map-to-isolated or isolated-to-isolated
                    if not cg.uniform:
                        # Step
                        start = -cg.weight
                        end = cg.weight
                        step = (end - start) / map_size
                        for o_n in out_map:
                            s = start
                            for i_n in in_map:
                                dot.edge(i_n, o_n, label= f"{s:.3f}",
                                             _attributes=eattrs)
                                s += step
                    else:
                        # Uniform
                        for o_n in out_map:
                            for i_n in in_map:
                                dot.edge(i_n, o_n, label=
f"{cg.weight:.3f}",
                                             _attributes=eattrs)

            else:
                dot.edge(a, b, label= f"{cg.weight:.3f}",
_attributes=eattrs)

    dot.render(filename, view=view)

    return dot
```

Reporters.py

```
from neat.statistics import StatisticsReporter
from neat.reporting import BaseReporter
from neat.six_util import iteritems
from switch_neat import *
import random

#Return the low quartile of the values in the *values array
def low_quartile(values):
    values = list(values)
    n = len(values)
    if n == 1:
        return values[0]
    values.sort()
    if n < 4:
        return (values[1] + values[0]) / 2.0
    if (n % 4) == 1:
        return values[n//4]
    i = n//4
    return (values[i - 1] + values[i])/ 2

#Return the high quartile of the values in the *values array
def high_quartile(values):
    values = list(values)
    n = len(values)
    if n == 1:
        return values[0]
    values.sort()
    if n < 4:
        return (values[-1] + values[-2]) / 2.0
    if (n % 4) == 1:
        return values[n // 4 * 3]
    i = n // 4 * 3
    return (values[i - 1] + values[i]) / 2

#Inherit the StatisticsReported from the neat package to add some desired
functionality
class StatReporterv2(StatisticsReporter):

    def get_fitness_low_quartile(self):
        return self.get_fitness_stat(low_quartile)

    def get_fitness_high_quartile(self):
        return self.get_fitness_stat(high_quartile)

    def get_fitness_best(self):
        return self.get_fitness_stat(max)

#This is a reporter that saves the best network genome to a binary file
every generation.
class NetRetriever(BaseReporter):

    def __init__(self):
        self.g = 0

    def end_generation(self, config, population, species_set):
        max_fit = 0
        best_genome = None
        #Find the best genome based on fitness
        for id, genome in iteritems(population):
            if not genome.fitness:
                continue
            if genome.fitness > max_fit:
                max_fit = genome.fitness
                best_genome = genome
        try:
            #Save it to the binary file
            fp = open(f"net_gen_{self.g}.bin", 'wb')
            pickle.dump(create(best_genome, config), fp)
            fp.close()
        except:
            pass
        if self.g > 0:
            #Remove the previous genome to avoid cluttering
            os.remove(f"net_gen_{self.g-1}.bin")
        self.g += 1

class EvaluatorMutator(BaseReporter):

    def __init__(self, evaluator):
        self.evaluator = evaluator

    def end_generation(self, config, population, species_set):
        self.evaluator.params = random.sample(self.evaluator.param_list,
self.evaluator.samples)
```

solve.py (contains functions that build the hand-designed networks):

```
import copy
from switch_neuron import Neuron, SwitchNeuron, SwitchNeuronNetwork, Agent
from math import tanh
from t_maze.envs import TMazeEnv
from utilities import mult, clamp, heaviside


def convert_to_action(scalar):
    if scalar[0] > 3.3:
        return (1,0,0)
    if scalar[0] < -3.3:
        return (0,0,1)
    return (0,1,0)
#Returns an agent which solves the 3x3 one-to-one association task
#We say that a network solves this problem when it manages to learn a new
association within n*(m-1) steps,
#in this case 6 steps.

def solve_one_to_one_3x3():
    input_keys = [-1, -2, -3, -4]
    output_keys = [0]
    switch_keys = [1, 2, 3]
    node_keys = [4, 5, 6]

    nodes = []

    modulating_nodes_dict = {
        'activation_function': lambda x: clamp(x,-10,10),
        'integration_function': mult,
        'activity': 0,
        'output': 0,
        'bias':1
    }

    node_weights = {4: [(-1, 1), (-4, 1)], 5: [(-2, 1), (-4, 1)], 6: [(-3,
1), (-4, 1)]}
    for i in node_keys:
        node_dict = copy.deepcopy(modulating_nodes_dict)
        node_dict['weights'] = node_weights[i]
        nodes.append(Neuron(i, node_dict))

    switch_std_weights = {
        1: [(-1, 10), (-1, 0), (-1, -10)],
        2: [(-2, 10), (-2, 0), (-2, -10)],
        3: [(-3, 10), (-3, 0), (-3, -10)]
    }
    switch_mod_weights = {
        1: [(4, 1 / 3)],
        2: [(5, 1 / 3)],
        3: [(6, 1 / 3)]
    }
    for key in switch_keys:
        nodes.append(SwitchNeuron(key, switch_std_weights[key],
switch_mod_weights[key]))

    node_0_std = {
        'activation_function': lambda x: clamp(x,-10,10),
        'integration_function': sum,
        'activity': 0,
        'output': 0,
        'weights': [(1, 1), (2, 1), (3, 1)],
        'bias' : 0
    }
    nodes.append(Neuron(0, node_0_std))

    net = SwitchNeuronNetwork(input_keys, output_keys, nodes)
    agent = Agent(net,lambda x: x,lambda x: convert_to_action(x))
    return agent

#Returns an agent which solves the 3x3 one-to-one association task
#We say that a network solves this problem when it manages to learn a new
association within n*(2^m - 1) steps,
#in this case 9 steps.
def solve_one_to_many():

    input_keys = [-1, -2, -3, -4]
    output_keys = [0,1]
    node_keys = [3,4,5]
    switch_keys = [7,8,9,10,11,12]

    nodes = []

    node_weights = {3: [(-1, 1), (-4, 1)], 4: [(-2, 1), (-4, 1)], 5: [(-3,
1), (-4, 1)]}
    modulating_nodes_dict = {
        'activation_function': lambda x: clamp(x,-1,1),
        'integration_function': mult,
        'activity': 0,
        'output': 0,
        'bias' : 1
    }
    for i in node_keys:
        node_dict = copy.deepcopy(modulating_nodes_dict)
        node_dict['weights'] = node_weights[i]
        nodes.append(Neuron(i, node_dict))

    slow, fast = 0,0
    switch_std_w = {}
    while fast < len(switch_keys):
        switch_std_w[switch_keys[fast]] = [(input_keys[slow], 1),
(input_keys[slow], -1)]
        fast += 1
        switch_std_w[switch_keys[fast]] = [(input_keys[slow], 1),
(input_keys[slow], -1)]
        fast+=1
        slow+=1

    w1, w2 = 0.5, 1
    switch_mod_w = {7: [(3,w2)], 8: [(7,w1)], 9: [(4,w2)], 10:[(9,w1)], 11:
[(5,w2)], 12: [(11,w1)]}

    for key in switch_keys:
        nodes.append(SwitchNeuron(key,switch_std_w[key],switch_mod_w[key]))

    out_w = {0 : [(8,1), (10,1), (12,1)], 1: [(7,1), (9,1), (11,1)]}
    out_dict = {
        'activation_function': heaviside,
        'integration_function': sum,
        'activity': 0,
        'output': 0,
        'bias' : 0
    }
    for key in output_keys:
        params = copy.deepcopy(out_dict)
        params['weights'] = out_w[key]
        nodes.append(Neuron(key,params))

    net = SwitchNeuronNetwork(input_keys,output_keys,nodes)
    return net

def convert_to_direction(x):
    if x[0] < -0.33:
        return  TMazeEnv.Actions.left
    if x[0] > 0.33:
        return TMazeEnv.Actions.right
    return TMazeEnv.Actions.forward

#Returns an agent which solves the single t-maze non-homing task.
#We say a network solves the problem when it it needs at most one step
figure out that the high reward has switched
#positions.
def solve_tmaze():

    input_keys = [-1,-2,-3,-4,-5]
    output_keys = [0]
    node_keys = [1,2,3]

    nodes = []

    #Aggregating neuron
    params = {
        'activation_function' : lambda x : x,
        'integration_function' : sum,
        'activity': 0,
        'output' : 0,
        'weights' : [(-1,-1), (-5,1)],
        'bias':0
    }
    nodes.append(Neuron(1,params))

    m_params = {
        'activation_function': lambda x: clamp(x, -0.8,0),
        'integration_function': mult,
        'activity': 0,
        'output': 0,
        'weights': [(1, 1), (-4, 1)],
        'bias' : 1
    }
    nodes.append(Neuron(2,m_params))

    std_weights = [(-3,5), (-3,-5)]
    mod_weights = [(2,-1.25*0.5)]
    nodes.append(SwitchNeuron(3,std_weights,mod_weights))

    o_params = {
        'activation_function': tanh,
        'integration_function': sum,
        'activity': 0,
        'output': 0,
        'weights': [(3,1)],
        'bias' : 0
    }
    nodes.append(Neuron(0,o_params))

    net = SwitchNeuronNetwork(input_keys,output_keys,nodes)
    agent = Agent(net, lambda x: x.insert(0,1), lambda x:
convert_to_direction(x))
    return agent

def solve_xor_rec():

    in_keys = [-1,-2]
    out_keys = [0]
    hidden_keys = [1]

    nodes = []
    nodes.append(Neuron(1, {
        'activation_function': heaviside,
        'integration_function': sum,
        'bias' : -1.5,
        'activity': 0,
        'output': 0,
        'weights': [(-1, 1),(-2,1)]
    }))

    nodes.append(Neuron(0, {
        'activation_function': heaviside,
        'integration_function': sum,
        'bias': -0.5,
        'activity': 0,
        'output': 0,
        'weights': [(-1,1),(-2,1),(1,-10)]
    }))

    net = SwitchNeuronNetwork(in_keys,out_keys,nodes)
    return net
```

switch_maps.py

```
import os
import pickle

import Reporters
import neat
import numpy as np
from neat.attributes import FloatAttribute, BoolAttribute, StringAttribute
from neat.genes import BaseGene, DefaultNodeGene
from neat.genome import DefaultGenomeConfig, DefaultGenome
from neat.graphs import required_for_output
from switch_neuron import Neuron, SwitchNeuronNetwork, SwitchNeuron, Agent
from neat.six_util import itervalues
from itertools import chain

from utilities import order_of_activation


class SwitchMapConnectionGene(BaseGene):

    #Various parameters for defining a connection.
    _gene_attributes = [BoolAttribute('one2one'), #if true then the
connection scheme is one to one, else one to all
                        BoolAttribute('uniform'),  #step or uniform
                        FloatAttribute('weight'),#Weigth is used as the
mean of the normal distribution for 1-to-all
                        BoolAttribute('enabled'),
                        BoolAttribute('is_mod')]

    def __init__(self, key):
        assert isinstance(key, tuple), "DefaultConnectionGene key must be a
tuple, not {!r}".format(key)
        BaseGene.__init__(self, key)

    #Define the distance between two genes
    def distance(self, other, config):
        d = abs(self.weight - other.weight) + int(self.one2one ==
other.one2one) \
            + int(self.uniform == other.uniform) + int(self.enabled ==
other.enabled) + int(self.is_mod == other.is_mod)
        return d * config.compatibility_weight_coefficient

class SwitchMapNodeGene(DefaultNodeGene):

    _gene_attributes = [FloatAttribute('bias'), #The bias of the neuron
                        StringAttribute('activation', options='sigmoid'), #
The activation function, tunable from the config
                        StringAttribute('aggregation', options='sum'), #The
aggregation function
                        BoolAttribute('is_isolated'),
                        BoolAttribute('is_switch')] #Map vs isolated neuron

    def distance(self, other, config):
        d = abs(self.bias - other.bias) + int(self.activation ==
other.activation) + int(self.aggregation == other.aggregation)\
            + int(self.is_isolated == other.is_isolated) +
int(self.is_switch - other.is_switch)
        return d * config.compatibility_weight_coefficient
```

```python
class SwitchMapGenome(DefaultGenome):
    @classmethod
    def parse_config(cls, param_dict):
        param_dict['node_gene_type'] = SwitchMapNodeGene
        param_dict['connection_gene_type'] = SwitchMapConnectionGene
        return DefaultGenomeConfig(param_dict)


def create(genome, config, map_size):
    """ Receives a genome and returns its phenotype (a
SwitchNeuronNetwork). """
    genome_config = config.genome_config
    required = required_for_output(genome_config.input_keys,
genome_config.output_keys, genome.connections)

    input_keys = genome_config.input_keys
    output_keys = genome_config.output_keys
    # Gather inputs and expressed connections.
    std_inputs = {}
    mod_inputs = {}
    children = {}
    node_keys = set(genome.nodes.keys())  # +
list(genome_config.input_keys[:])

    # Here we populate the children dictionay for each unique not isolated
node.
    for n in genome.nodes.keys():
        children[n] = []
        if not genome.nodes[n].is_isolated:
            for _ in range(1, map_size):
                new_idx = max(node_keys) + 1
                children[n].append(new_idx)
                node_keys.add(new_idx)
    # We don't scale the output with the map size to keep passing the
parameters of the network easy.
    # This part can be revised in the future
    for n in chain(input_keys, output_keys):
        children[n] = []
    #Iterate over every connection
    for cg in itervalues(genome.connections):
        #If it's not enabled don't include it
        if not cg.enabled:
            continue

        i, o = cg.key
        #If neither node is required for output then skip the connection
        if o not in required and i not in required:
            continue

        #Find the map corresponding to each node of the connection
        in_map = [i] + children[i]
        out_map = [o] + children[o]
        #If the connection is modulatory and the output neuron a switch
neuron then the new weights are stored
        #in the mod dictionary. We assume that only switch neurons have a
modulatory part.
        if cg.is_mod and genome.nodes[o].is_switch:
            node_inputs = mod_inputs
        else:
            node_inputs = std_inputs
        for n in out_map:
            if n not in node_inputs.keys():
                node_inputs[n] = []

        if len(in_map) == map_size and len(out_map) == map_size:
            # Map to map connectivity
            if cg.one2one:
                # 1-to-1 mapping
                weight = cg.weight
                for i in range(map_size):
                    node_inputs[out_map[i]].append((in_map[i], weight))

            else:
                # 1-to-all
                if not cg.uniform:
                    # Step
                    start = -cg.weight
                    end = cg.weight
                    step = (end - start) / (map_size - 1)
                    for o_n in out_map:
                        s = start
                        for i_n in in_map:
                            node_inputs[o_n].append((i_n, s))
                            s += step
                else:
                    # Uniform
                    for o_n in out_map:
                        for i_n in in_map:
                            node_inputs[o_n].append((i_n, cg.weight))

        else:
            # Map-to-isolated or isolated-to-isolated
            if not cg.uniform:
                # Step
                start = -cg.weight
                end = cg.weight
                step = (end - start) / (map_size - 1)
                for o_n in out_map:
                    s = start
                    for i_n in in_map:
                        node_inputs[o_n].append((i_n, s))
                        s += step
            else:
                # Uniform
                for o_n in out_map:
                    for i_n in in_map:
                        node_inputs[o_n].append((i_n, cg.weight))

    nodes = []

    #Sometimes the output neurons end up not having any connections during
the evolutionary process. While we do not
    #desire such networks, we should still allow them to make predictions
to avoid fatal errors.
    for okey in output_keys:
        if okey not in node_keys:
            node_keys.add(okey)
            std_inputs[okey] = []
    # While we cannot deduce the order of activations of the neurons due to
the fact that we allow for arbitrary connection
    # schemes, we certainly want the output neurons to activate last.
    input_keys = genome_config.input_keys
    output_keys = genome_config.output_keys
    conns = {}
    for k in genome.nodes.keys():
        if k not in std_inputs:
            std_inputs[k] = []
            if k in children:
                for c in children[k]:
                    std_inputs[c] = []
        conns[k] = [i for i, _ in std_inputs[k]]
    sorted_keys = order_of_activation(conns, input_keys, output_keys)

    for node_key in sorted_keys:
        #if the node we are examining is not in our keys set then skip it.
It means that it is not required for output.
        if node_key not in node_keys:
            continue

        node = genome.nodes[node_key]
        node_map = [node_key] + children[node_key]
        if node.is_switch:
            # If the switch neuron has both modulatory and standard weights
then we can add it normally to the nodes
            # of the network.
            if node_key in std_inputs.keys() and node_key in
mod_inputs.keys():
                for n in node_map:

nodes.append(SwitchNeuron(n,std_inputs[n],mod_inputs[n]))
                continue
            # if the switch neuron only has modulatory weights then we copy
those weights for the standard part as well.
            # this is not the desired behaviour but it is done to avoid
errors during forward pass.
            elif node_key not in std_inputs.keys() and node_key in
mod_inputs.keys():
                for n in node_map:
                    std_inputs[n] = mod_inputs[n]
            else:
                for n in node_map:
                    std_inputs[n] = []
        else:
            for n in node_map:
                if n not in std_inputs:
                    std_inputs[n] = []

        # Create the standard part dictionary for the neuron
        for n in node_map:
            params = {
                'activation_function':
genome_config.activation_defs.get(node.activation),
                'integration_function':
genome_config.aggregation_function_defs.get(node.aggregation),
                'bias': node.bias,
                'activity': 0,
                'output': 0,
                'weights': std_inputs[n]
            }
            nodes.append(Neuron(n, params))

    return SwitchNeuronNetwork(input_keys, output_keys, nodes)

MAP_SIZE = 5
def make_eval_fun(evaluation_func, in_proc, out_proc):

    def eval_genomes (genomes, config):
        for genome_id, genome in genomes:
            net = create(genome,config,MAP_SIZE)
            #Wrap the network around an agent
            agent = Agent(net, in_proc, out_proc)
            #Evaluate its fitness based on the function given above.
            genome.fitness = evaluation_func(agent)

    return eval_genomes
#A dry test run for the xor problem to test if the above implementation
works
def run(config_file):

    #Configuring the agent and the evaluation function
    from eval import eval_net_xor
    eval_func = eval_net_xor
    #Preprocessing for inputs: none
    in_func = out_func = lambda x: x
    #Preprocessing for output - convert float to boolean

    # Load configuration.
    config = neat.Config(SwitchMapGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
                         config_file)
    # Create the population, which is the top-level object for a NEAT run.
    p = neat.Population(config)

    # Add a stdout reporter to show progress in the terminal.
    p.add_reporter(neat.StdOutReporter(True))
    stats = Reporters.StatReporterv2()
    p.add_reporter(stats)
    #p.add_reporter(Reporters.NetRetriever())
    #p.add_reporter(neat.Checkpointer(5))

    # Run for up to 300 generations.
    winner = p.run(make_eval_fun(eval_func, in_func, out_func), 1000)

    # Display the winning genome.
    print('\nBest genome:\n{!s}'.format(winner))

    # Show output of the most fit genome against training data.
    print('\nOutput:')
    winner_net = create(winner, config, MAP_SIZE)
    winner_agent = Agent(winner_net,in_func, out_func)
    print("Score in task: {}".format(eval_func(winner_agent)))
    for i, o in (((0, 0), 0), ((0, 1), 1), ((1, 0), 1), ((1, 1), 0)):
        print(f"Input: {i}, Expected: {o}, got {winner_agent.activate(i)}")
    #Uncomment the following if you want to save the network in a binary
file
    fp = open('winner_net.bin','wb')
    pickle.dump(winner_net,fp)
    fp.close()
    #visualize.draw_net(config, winner, True)
    #visualize.plot_stats(stats, ylog=False, view=True)
    #visualize.plot_species(stats, view=True)

def main():
    # Determine path to configuration file. This path manipulation is
    # here so that the script will run successfully regardless of the
    # current working directory.
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'config/config-switch_maps')
    run(config_path)

if __name__ == '__main__':
    main()
```

switch_neat.py

```python
from __future__ import print_function
import pickle
import visualize
from neat.attributes import FloatAttribute, BoolAttribute, StringAttribute
from neat.genes import DefaultNodeGene, DefaultConnectionGene
from neat.genome import DefaultGenomeConfig, DefaultGenome
from neat.graphs import required_for_output
from neat.six_util import itervalues
from switch_neuron import SwitchNeuron, SwitchNeuronNetwork, Neuron, Agent
import os
import neat
import Reporters
from utilities import order_of_activation

#The class for the gene representing the neurons in our network.
class SwitchNodeGene(DefaultNodeGene):

    _gene_attributes = [FloatAttribute('bias'),
                        StringAttribute('activation', options='sigmoid'),
                        StringAttribute('aggregation', options='sum'),
                        BoolAttribute('is_switch')]

    def distance(self, other, config):
        d = abs(self.bias + other.bias)
        if self.activation != other.activation:
            d += 1.0
        if self.aggregation != other.aggregation:
            d += 1.0
        if self.is_switch != other.is_switch:
            d =3
        return d * config.compatibility_weight_coefficient

#The gene for the connections in our network.
class SwitchConnectionGene(DefaultConnectionGene):
    _gene_attributes = [FloatAttribute('weight'),
                        BoolAttribute('is_mod'),
                        BoolAttribute('enabled')] #The neat package
complains if this attribute is not present.

    def distance(self, other, config):
        d = abs(self.weight - other.weight)
        if self.enabled != other.enabled:
            d += 1.0
        if self.is_mod != other.is_mod:
            d += 1
        return d * config.compatibility_weight_coefficient

#Create a switch genome class to replace the default genome class in our
experiments.
class SwitchGenome(DefaultGenome):
    @classmethod
    def parse_config(cls, param_dict):
        param_dict['node_gene_type'] = SwitchNodeGene
        param_dict['connection_gene_type'] = SwitchConnectionGene
        return DefaultGenomeConfig(param_dict)

#Takes a genome and the configuration object and returns the network
encoded in the genome.
```

```
def create(genome, config):
    genome_config = config.genome_config
    required = required_for_output(genome_config.input_keys,
genome_config.output_keys, genome.connections)
    input_keys = genome_config.input_keys
    output_keys = genome_config.output_keys

    #A dictionary where we keep the modulatory weights for every node
    mod_weights = {}
    #A dictionary where we keep the standard weights for every node
    std_weights = {}
    #Create a set with the keys of the nodes in the network
    keys = set()
    #Iterate over the connections
    for cg in itervalues(genome.connections):
        #if not cg.enabled:
        #    continue

        i, o = cg.key
        #If neither of the nodes in the connection are required for output
then skip this connection
        if o not in required and i not in required:
            continue

        if i not in input_keys:
            keys.add(i)
        keys.add(o)
        #In this implementation, only switch neurons have a modulatory part
        if genome.nodes[o].is_switch:
            #Add the weight to the modulatory part of the o node and
continue with the next connection
            if cg.is_mod:
                if o not in mod_weights.keys():
                    mod_weights[o] = [(i,cg.weight)]
                else:
                    mod_weights[o].append((i,cg.weight))
                continue
        #If the connection is not modulatory
        #Add the weight to the standard weight of the o node.
        if o not in std_weights.keys():
            std_weights[o] = [(i,cg.weight)]
        else:
            std_weights[o].append((i,cg.weight))
    #Create the array with the network's nodes
    nodes = []

    #Sometimes the output neurons end up not having any connections during
the evolutionary process. While we do not
    #desire such networks, we should still allow them to make predictions
to avoid fatal errors.
    for okey in output_keys:
        keys.add(okey)

    for k in keys:
        if k not in std_weights:
            std_weights[k] = []

    #While we cannot deduce the order of activations of the neurons due to
the fact that we allow for arbitrary connection
    #schemes, we certainly want the output neurons to activate last.
    conns = {}
    for k in keys:
        conns[k] = [i for i, w in std_weights[k]]
    sorted_keys = order_of_activation(conns, input_keys, output_keys)

    #Create the nodes of the network based on the weights dictionaries
created above and the genome.
    for node_key in sorted_keys:
        if node_key not in keys:
            continue
        node = genome.nodes[node_key]
        if node.is_switch:
            #If the switch neuron has both modulatory and standard weights
then we can add it normally to the nodes
            #of the network.
            if node_key in std_weights.keys() and node_key in
mod_weights.keys():
                nodes.append(SwitchNeuron(node_key, std_weights[node_key],
mod_weights[node_key]))
                continue
            #if the switch neuron only has modulatory weights then we copy
those weights for the standard part as well.
            #this is not the desired behaviour but it is done to avoid
errors during forward pass.
            elif node_key not in std_weights.keys() and node_key in
mod_weights:
                std_weights[node_key] = mod_weights[node_key]
            else:
                std_weights[node_key] = []
        else:
            if node_key not in std_weights:
                std_weights[node_key] = []

        #Create the standard part dictionary for the neuron
        params = {
            'activation_function' :
genome_config.activation_defs.get(node.activation),
            'integration_function' :
genome_config.aggregation_function_defs.get(node.aggregation),
            'bias' : node.bias,
            'activity' : 0,
            'output' : 0,
            'weights' : std_weights[node_key]
        }
        nodes.append(Neuron(node_key,params))

    return SwitchNeuronNetwork(input_keys,output_keys,nodes)

#This function wraps a given evaluation function to make it suitable for
evaluating an array of genomes
#produced by neat and returns it.
def make_eval_fun(evaluation_func, in_proc, out_proc):

    def eval_genomes (genomes, config):
        for genome_id, genome in genomes:
            net = create(genome,config)
            #Wrap the network around an agent
            agent = Agent(net, in_proc, out_proc)
            #Evaluate its fitness based on the function given above.
            genome.fitness = evaluation_func(agent)
    return eval_genomes

#A dry test run for the one to one association task. I haven't ran it to
end yet because it needs too much time as of now and at some point
#it even reaches a plateau.
def run(config_file):

    #Configuring the agent and the evaluation function
    from eval import eval_net_xor
    eval_func = eval_net_xor
    #Preprocessing for inputs: none
    in_func = out_func = lambda x: x


    #Preprocessing for outputs: one-hot max encoding.


    # Load configuration.
    config = neat.Config(SwitchGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
                         config_file)

    from utilities import heaviside
    config.genome_config.add_activation('heaviside', heaviside)
    # Create the population, which is the top-level object for a NEAT run.
    p = neat.Population(config)

    # Add a stdout reporter to show progress in the terminal.
    p.add_reporter(neat.StdOutReporter(True))
    stats = Reporters.StatReporterv2()
    p.add_reporter(stats)
    #p.add_reporter(Reporters.NetRetriever())
    #p.add_reporter(neat.Checkpointer(5))

    # Run for up to 300 generations.
    winner = p.run(make_eval_fun(eval_func, in_func, out_func), 2000)

    # Display the winning genome.
    print('\nBest genome:\n{!s}'.format(winner))

    # Show output of the most fit genome against training data.
    print('\nOutput:')
    winner_net = create(winner, config)
    winner_agent = Agent(winner_net,in_func, out_func)
    print("Score in task: {}".format(eval_func(winner_agent)))

    #for i, o in (((0,0),0), ((0,1),1), ((1,0),1), ((1,1),0)):
    #    print(f"Input: {i}, Expected: {o}, got
{winner_agent.activate(i)}")
    #Uncomment the following if you want to save the network in a binary
file
    fp = open('winner_net.bin','wb')
    pickle.dump(winner_net,fp)
    fp.close()
    #visualize.draw_net(config, winner, True)
    visualize.plot_stats(stats, ylog=False, view=True)
    #visualize.plot_species(stats, view=True)

def main():
    # Determine path to configuration file. This path manipulation is
    # here so that the script will run successfully regardless of the
    # current working directory.
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'config/config-switch')
    run(config_path)

if __name__ == '__main__':
    main()
```

switch_neuron.py

```
import math
import sys
from math import floor
from utilities import identity


#This is a wrapper class for the neural networks to allow pre-processing of
the input and the output which may help the
#network
class Agent():

    def __init__(self, network, setup_inputs, prepare_outputs):
        self.network = network
        self.setup_inputs = setup_inputs
        self.prepare_outputs = prepare_outputs

    def activate(self,inputs):
        proc_inputs = self.setup_inputs(inputs)
        output = self.network.activate(proc_inputs)
        return self.prepare_outputs(output)

#This is the basis class for the neurons that we will be using for our
experiments.
class Neuron():

    #Consider abolishing the dictionaries and go for a more object-oriented
approach
    def __init__(self,key, standard_dict, modulatory_dict = None):
        #Each dictionary contains the following: activation_function,
integration_function, activity, output, weights
        #Standard dictionary also contains bias
        self.key = key
        self.standard= standard_dict
        self.modulatory = modulatory_dict
        self.has_modulatory = (self.modulatory is not None)

class SwitchNeuron(Neuron):

    #Each of std_weights and mod_weights is an array of tuples of the form
(<node>,<weight>)
    def __init__(self,key, std_weights, mod_weights):

        #The standard part of the neuron is initialized according to the
definition of the switch neuron in Christodoulou's
        #and Vassiliades paper
        std_dict = {'activation_function': identity,
                    'integration_function': self.std_integration_function,
                    'bias' : 0,
                    'activity': 0,
                    'output': 0,
                    'weights': std_weights}

        #Initialize the standard activity of the neuron
        if not std_weights:
            mod_activity = 0
        else:
            mod_activity = 1 / (2 * len(std_weights))
        mod_dict = {
            'activation_function': identity,
            'integration_function': self.mod_integration_function,
            'activity': mod_activity,
            'output': mod_activity,
            'weights': mod_weights
        }

        super().__init__(key,std_dict, mod_dict)

    #The standard integration function is basically a gate that propagates 
one signal from its inputs based on
    #the neurons modulatory activity.
    def std_integration_function(self, w_inputs):
        idx = floor(len(self.standard['weights']) * 
self.modulatory['output'])
        return w_inputs[idx]

    #The modulatory function acts as perfect integrator which we make 
behave in a cyclical fashion in order to keep
    #its value in bounds. This cyclical behaviour is key to the switch 
neuron's role as it enables the  alternating (when
    #needed) propagation of the signals from the input.
    def mod_integration_function(self, w_inputs):
        self.modulatory['activity'] += sum(w_inputs)
        ###################
        #The following bounding of the activity very rarely happens and is 
there to combat overflows.
        #Although, these overflows shouldn't even be happening in the first 
place.
        if math.isnan(self.modulatory['activity']):
            self.modulatory['activity'] = 0
        self.modulatory['activity'] = max(min(self.modulatory['activity'], 
sys.maxsize), -sys.maxsize)
        ###################
        self.modulatory['activity'] -= floor(self.modulatory['activity'])
        return self.modulatory['activity']

#This class models the integrating neuron used in the switch module. It 
continually integrates the signals from
#its inputs and when the sum is over a certain threshold it fires a signal 
itself. It is comparable to the
#integrate-and-fire neurons .
class IntegratingNeuron(Neuron):

    THETA = 1
    BASELINE = 0

    def __init__(self, key, weights):
        params = {
            'activation_function' : self.tri_step,
            'integration_function' : self.perfect_integration,
            'bias' : 0,
            'activity' : IntegratingNeuron.BASELINE,
            'output' : 0,
            'weights': weights
        }

        super().__init__(key,params)
    def perfect_integration(self, w_inputs):
        self.standard['activity'] += sum(w_inputs)
        return self.standard['activity']

    #When the neuron's activity goes over the threshold we fire a 1, and
when the neuron's activity goes below the
    # -threshold we fire a -1. The we reset the activity to 0.
    def tri_step(self, activity):
        if activity >= IntegratingNeuron.THETA:
            self.standard['activity'] = 0
            return 1
        elif activity < -IntegratingNeuron.THETA:
            self.standard['activity'] = 0
            return -1
        return 0

class SwitchNeuronNetwork():

    #inputs: an array containing the keys of the input pins
    #outputs: an array containing the keys of the output neurons
    #nodes: an array containing the neurons of the network, in the order
they are supposed to fire.
    def __init__(self,inputs, outputs, nodes):
        self.inputs = inputs
        self.outputs = outputs
        self.nodes = nodes
        #We create in memory a dictionary of the nodes for faster on-demand
indexing
        self.nodes_dict = {}
        for node in nodes:
            self.nodes_dict[node.key] = node

        #We convert any switch neurons that need to be converted to switch
modules, i.e. the switch neurons that modulate
        #other neurons.
        temp_nodes = nodes[:]
        for node in temp_nodes:
            if isinstance(node, SwitchNeuron) and
len(node.standard["weights"]) > 0:
                self.make_switch_module(node.key)

    #Perform a forward pass through the network. Since arbitrary connection
schemes are allowed, the neurons are not
    #divided into layers and its order of activation is assumed based on
their order of the nodes array.
    def activate(self, inputs):
        assert len(self.inputs) == len(inputs), "Expected {:d} inputs, got
{:d}".format(len(self.inputs), len(inputs))

        #Store the values from the input pins in a dictionary
        ivalues = {}
        for i, v in zip(self.inputs, inputs):
            ivalues[i] = v

        for node in self.nodes:
            #First calculate the modulatory output of the neuron
            if node.has_modulatory:
                mod_inputs = []
                #Collect all the weighted inputs in array
                for key, weight in node.modulatory['weights']:
                    if key in ivalues.keys():
                        val = ivalues[key]
                    else:
                        val = self.nodes_dict[key].standard['output']
                    mod_inputs.append(val*weight)
                #Calculate the neurons modulatory activity and output based
on it's functions.
                node.modulatory['activity'] =
node.modulatory['integration_function'](mod_inputs)
                node.modulatory['output'] =
node.modulatory['activation_function'](node.modulatory['activity'])

            standard_inputs = []
            #Collect the weighted inputs in an array
            for key, weight in node.standard['weights']:
                if key in ivalues.keys():
                    val = ivalues[key]
                else:
                    val = self.nodes_dict[key].standard['output']
                standard_inputs.append(val * weight)
            #add the bias
            standard_inputs.append(node.standard['bias'])
            #Calculate the neuron's activity and output based on it's
standard functions.
            node.standard['activity'] =
node.standard['integration_function'](standard_inputs)
            node.standard['output'] =
node.standard['activation_function'](node.standard['activity'])

        output =  [self.nodes_dict[key].standard['output'] for key in
self.outputs]
        return output

    #Check if a switch neuron needs to be converted to a module AND
converts it if so. A neuron needs to be
    #converted if it modulates other neurons.
    def make_switch_module(self, key):
        #Check if the key of the switch neuron is valid.
        idx = None
        for i, node in enumerate(self.nodes):
            if node.key == key:
                idx = i
                break

        assert idx != None, "Switch Neuron passed not found in the network
nodes"
        s_node = self.nodes_dict[key]
        assert isinstance(s_node, SwitchNeuron), \
            "Argument passed in is {} instead of SwitchNeuron
type".format(s_node.__class__.__name__)
        out_mod = []
        #Find if this switch neurons modulates other neurons
        for node in self.nodes:
            if node.has_modulatory:
                found = False
                for i, w in node.modulatory['weights']:
                    if i == key:
                        out_mod.append(node.key)
                        found = True
                        break
                if found:
                    break
        #If not, then stop the process
        if not out_mod:
            return

        #Create a unique key for the modulating neurons
        modulating_key = max(list(self.nodes_dict.keys())) + 1
        #The modulating neuron inherits the initial switch neuron's
modulatory weights as its standard weights.
        modulating_weights = s_node.modulatory['weights']
        modulating_dict = {
            'activation_function': identity,
            'integration_function': sum,
            'bias' : 0,
            'activity' : 0,
            'output' : 0,
            'weights': modulating_weights
        }
        #Create the modulating neuron and add it to the network's neurons
        modulating_neuron = Neuron(modulating_key,modulating_dict)
        self.nodes_dict[modulating_key] = modulating_neuron
        #The switch neuron's is only modulated by the modulating neuron
with a weight inverse to it's initial
        #input neurons.
        s_node.modulatory['weights'] = [(modulating_key,
1/len(s_node.standard['weights']))]

        #Create a unique key for the integrating neuron
        integrating_key = max(list(self.nodes_dict.keys())) + 1
        #The integrating neuron shares the connection from the modulating
neuron with the switch neuron.
        integrating_weights = [(modulating_key,
1/len(s_node.standard['weights']))]
        #Create the neuron and add it to the network's neurons.
        integrating_neuron =
IntegratingNeuron(integrating_key,integrating_weights)
        self.nodes_dict[integrating_key] = integrating_neuron

        self.nodes.insert(idx,integrating_neuron)
        self.nodes.insert(idx,modulating_neuron)

        #Modify the input weights of each neuron that was modulated by the
switch neuron to be instead modulated
        #by the integrating neuron.
        for n in out_mod:
            node = self.nodes_dict[n]
            for ind, (i, w) in enumerate(node.modulatory['weights']):
                if i == key:
                    node.modulatory['weights'][ind] = (integrating_key, w)
```

test_networks.py (functions that test if a solution is correct):

```python
import unittest
import solve
import eval

class TestNetworks(unittest.TestCase):

    def test_one_to_one(self):
        agent = solve.solve_one_to_one_3x3()
        optimal_score = 1975
        scores = [eval.eval_one_to_one_3x3(agent) for _ in range(100)]
        for score in scores:
            self.assertGreaterEqual(score, optimal_score)

    def test_one_to_many(self):
        agent = solve.solve_one_to_many()
        optimal_score = 1964
        scores = [eval.eval_one_to_many_3x2(agent) for _ in range(100)]
        for score in scores:
            self.assertGreaterEqual(score,optimal_score)

    def test_t_maze(self):
        agent = solve.solve_tmaze()
        optimal_score = 95
        scores = [eval.eval_tmaze(agent) for _ in range(100)]
        for score in scores:
            self.assertGreaterEqual(score, optimal_score)

if __name__ == '__main__':
    unittest.main()
```

utilities.py

```python
import random

def identity(activity):
    return activity

def clamp(x,low,high):
    if x < low:
        return low
    if x > high:
        return high
    return x

def heaviside(x):
    if x < 0:
        return 0
    return 1

def mult(w_inputs):
    product = 1
    for w_i in w_inputs:
        product *= w_i
    return product

def shuffle_lists(list1, list2):
    temp = list(zip(list1, list2))
    random.shuffle(temp)
    list1, list2 = [], []
    for a, b in temp:
        list1.append(a)
        list2.append(b)
    return list1, list2

#Return the order of neuron activation in a cyclical directed graph.
#conns is a dictionary with:
#   key: the name of the node
#   value: the nodes inputs in this form: [node1, node2, node3]
def order_of_activation(conns, inputs, outputs):
    ordered_list = []
    visited = set()
    #The number of incoming connections
    #If a neuron has a recurrent connection to itself we don't count it
    magnitude = {k: len([i for i in conns[k] if i != k]) for k in
conns.keys()}
    #Inputs always activate first
    for i in inputs:
        magnitude[i] = 0

    frontier = [i for i in magnitude.keys() if magnitude[i] ==
min(magnitude.values())]
    def activate_n(node):
        new_nodes = []
        for k in magnitude.keys():
            if k in inputs:
                continue
            if node in conns[k]:
                magnitude[k] -= 1
                new_nodes.append(k)
        return new_nodes
    while len(visited) <  len(conns.keys()) + len(inputs):
        if frontier == []:
            minmagn = min([magnitude[v] for v in magnitude.keys() if v not
in visited])
            frontier = [i for i in magnitude.keys() if magnitude[i] ==
minmagn]
        minn = frontier[0]
        for node in frontier:
            if magnitude[node] < magnitude[minn]:
                minn = node

        if minn not in inputs:
            ordered_list.append(minn)
        frontier.remove(minn)
        visited.add(minn)
        new_nodes = activate_n(minn)
        frontier.extend([n for n in new_nodes if n not in visited and n not
in frontier])

    return ordered_list
```