Neuroevolution of Artificial General Intelligence

Implementation and study of a framework towards the evolution of artificial general intelligence

Kristoffer Olsen

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

June 2020

Neuroevolution of Artificial General Intelligence

Implementation and study of a framework towards the evolution of artificial general intelligence

Kristoffer Olsen

Neuroevolution of Artificial General Intelligence

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

This thesis explores the properties of a novel framework aspiring to achieve General Intelligence via Neuroevolution, called 'Neuroevolution of Artificial General Intelligence' (NAGI). A weight agnostic neuroevolution technique based on 'Neuroevolution of Augmenting Topologies' (NEAT) was used to evolve Spiking Neural Networks (SNNs), which function as controllers for agents interacting with mutable environments in silico, gaining rewards and learning unsupervised through embodiment throughout their lifetime. Synaptic weights are excluded from the genome, ensuring that intrinsic knowledge about the environment is not passed on from parent to offspring, while network topology, the type of neurons and the type of learning are all subject to the evolutionary process.

The results showed that the agents emerging from the framework were able to achieve a high accuracy of correct actions when interacting with mutable environments, even new environments that were never encountered during training. It also showed that great care must be taken in when designing a neuroevolution technique in order to properly guide the evolution towards agents with competing desirable properties.

Preface

This report, combined with the authors implementation of the framework it is based on, concludes the research and findings from the work done on the authors master thesis at University of Oslo (UiO). The code for the implementation is available at the GitHub repository *neat-nagi-python* (`https://github.com/krolse/neat-nagi-python`).

I want to thank main supervisor Sidney Pontes-Filho, as well as co-supervisors Stefano Nichele, Pål Halvorsen, Michael Riegler and Anis Yazidi for the opportunity to be a part of this exciting research, and for their invaluable feedback while working on the thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives a brief overview of the master thesis by presenting central topics and the motivation behind the work. It also defines the goals and research questions for the thesis.

1.1 Background and Motivation

Research on Artificial Intelligence (AI) has led us to outstanding advancement in different scientific fields. Today, AI models are ubiquitous and can be found in almost any large scale system. However, the majority of the AI models in use today are trained and specialized for specific tasks, for which they are becoming increasingly good at performing. Machines have caught up to humans (and are poised to surpass us) in image recognition tasks, something our bodies have naturally evolved to be specialists at. However, many AI models arguably lack the ability of generalization and self-adaptation. These are properties being explored in the research field of Artificial General Intelligence (AGI) or strong AI, with the ultimate goal being a reproduction of life-like, or more specifically human-like intelligence in machines.

Living beings in the real world learn primarily by interaction with their environment (including other living beings), which provides endless unlabeled and mutable data. Our means of interaction are our senses, such as sight, hearing, taste, smell, touch and so on. Our brain, the organ that interprets the encoded signals from our sensory organs, has through continuous evolution gained the ability of being able to distinguish between positive and negative sensory experiences depending on what is good or harmful to us, such as pain and pleasure, which serves as reward and penalty mechanisms that affect our learnt behavior.

The natural brain is a product of the evolutionary process, and is therefore the reason living beings have gained the ability to learn through interaction. Evo-

lution is an extremely complex process on a micro scale level, but it is driven by two simple concepts: survival and reproduction. If you are well suited for survival in your environment, you're likely to live longer and have more opportunities to reproduce, both by simply being alive for longer and having desirable traits that are sexually attractive to potential mates. These mechanisms ensure that desirable traits to a larger degree than undesirable traits are passed on through generations in a population.

In the search for AGI in its simplest form, we will in this thesis explore a framework designed with three key points of biological inspiration in mind: i) the structure of interconnected processing units that compose the brain, ii) the evolutionary process that evolved it, and iii) the sensory interaction with the world that makes learning in the brain possible. Translated into AI models, they can be summarized as follows:

1. Biologically plausible neuron models (Spiking Neurons);
2. Evolution of neural network structures (Neuroevolution);
3. Simulated agent-environment interaction promoting self-learning through embodiment.

The hypothesis is that an approach to natural intelligence with a smaller level of abstraction may lead to an AI with the same general and self-adapting properties found in intelligent beings in the natural world.

1.2 Goals and Research Questions

The goal of this thesis is to explore how EAs can be applied to SNNs to evolve agents that are able to self-learn throughout their lifetime by interacting with environments that are constantly changing. To reach this goal, we define the following research questions that we aim to answer:

- **Research Question One:** Is it feasible to evolve controllers that are able to learn varied decision boundaries[1] without any existing knowledge about the environment by using a weight-agnostic neural network?
- **Research Question Two:** Do the controllers emerging from the evolution display general, problem-independent learning capabilities by being able to perform in never before seen environments?
- **Research Question Three:** With only the sensory feedback from the interactions with the environment, are the agents able to learn by themselves?

[1] In classification problems, the vector space is divided into decision regions where all vectors in that region are assigned to a single category. Decision regions are divided by a *decision boundary*, where there is a tie between two or more categories [1].

1.3 Research Method

In order to achieve the goal of this thesis and answer the research questions, a 'proof of concept' system was designed by the supervisors and the author, and implemented from scratch by the author. Experiments with agent-environment interaction were designed and conducted through simulation with these approaches and metrics in mind in order to measure and analyze how the solutions performed.

1.4 Report Structure

All the theories central to understanding the approaches explored in the thesis are explained in Chapter 2. Related work and approaches are presented and discussed in Chapter 3. The approaches that are used in the framework explored in the thesis are explained in full in Chapter 4. In Chapter 5 we briefly discuss the implementation of the code used in the thesis. The design and setup of the experiments are explained in Chapter 6, and the results from these are presented in Chapter 7, which in turn are evaluated in Chapter 8. Chapter 9 presents suggestions for further research. Chapter 10 contains the conclusion of the thesis.

Chapter 2

Background Theory

In this chapter we explain the relevant background theory necessary to understand the approaches used in the thesis.

2.1 Artificial General Intelligence

AGI [2] is quite young as a research area and does not have a precise definition. Sometimes it is compared to 'natural intelligence' or 'strong AI', which in broad strokes describe intelligence comparable to human or natural intelligence. It is therefore worth elaborating on what is meant by AGI in the context of NAGI. We focus on the 'general' learning and adaptation capabilities of an AGI model, as opposed to the bulk of mainstream AI models that are specialized at certain tasks. In this work, we are not looking for models with comparable or superior intelligence to humans, but for a rather simple model able to continuously adapt in a changing and increasingly complex environment through self-learning.

2.2 Neural Networks

Neural Networks are computational models of neural circuits found in the brain of animals. A neural circuit consists of neurons interconnected by synapses. Neurons are electrically excitable nerve cells which pass electronic signals along the synapses [3, p. 39].

Artificial neural networks attempt to mimic the behavior and adaptive features of biological neural circuits, and are integral to modern Machine Learning (ML), Deep Learning (DL) and Artificial Intelligence (AI) systems. They are useful for solving complex problems where it is difficult to manually code an analytical solution. In this section we will detail two types of neural networks, as well as

the different learning paradigms that they utilize.

2.2.1 Classical Artificial Neural Networks

Artificial Neural Networks (ANNs) [3, pp. 39–49] consist of computational units called neurons that are interconnected by weighted connections, often simply referred to as 'weights'. ANNs are organized into layers, of which there are three types: input, hidden and output. The input layer does not actually contain neurons, but nodes that receive information from some source, which is passed forward throughout the neurons in the hidden layers until it reaches the neurons in the output layer. A generic ANN architecture can be seen in Figure 2.1.

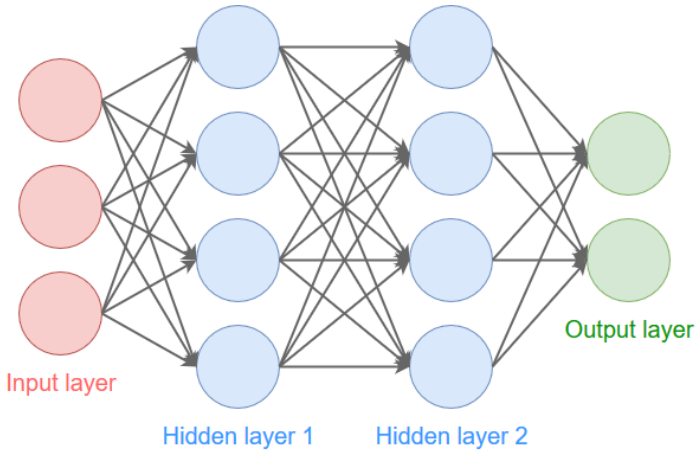


Figure 2.1: Graph representation of a generic ANN with three input nodes, two hidden layers with four neurons each and two output neurons.

So how does a neuron compute its output? A commonly used neuron is the McCulloch and Pitts neuron [3, pp. 40–42] (also called a perceptron), where each neuron computes the sum of products y between its inputs $\{x_1, x_2, ..., x_n\}$ and the weights of the connections the inputs are passed through $\{w_1, w_2, ..., w_n\}$, given by

$$y = \sum_{i=1}^{n} w_i x_i. \tag{2.1}$$

Given some activation function[1] φ, the output of the neuron is given by $\varphi(y)$. Figure 2.2 illustrates this. An ANN consisting of perceptrons is called a multi-layer perceptron.

[1] In the original McCulloch and Pitts neuron, the activation function was a simple threshold function. In modern ANNs, we use other activation functions such as the sigmoid function or the rectifier function (ReLU).

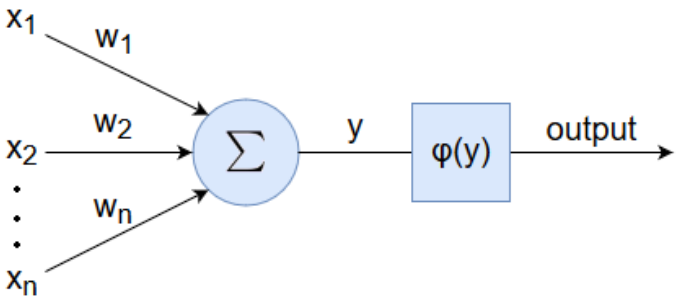


Figure 2.2: McCulloch and Pitts neuron.

2.2.2 Spiking Neural Networks

Spiking Neural Networks (SNNs) [4] [5] are a type of ANNs with a lower level of abstraction (more biologically plausible) than regular ANNs. Unlike ANNs, they incorporate a temporal dimension, or a concept of time. In regular ANNs, the information passed to the inputs are values. In SNNs, however, input comes in the form of sequences of spikes, like a binary pulse signal. Just like biological neurons, the neurons in SNNs have a membrane potential that changes each time the neuron receives a spike as input. Once the membrane potential passes a certain threshold, it gives an output in the form of a spike. This means that neurons in an SNN do not give an output at every propagation cycle, unlike regular ANNs.

Neuroplasticity

Neuroplasticity [6] is the continuous process of change of the synapses in the brain in response to sensory stimuli such as pain, pleasure, smell, sight, taste, hearing and any other sense an animal can have. This is the idea behind how weights are adjusted in SNNs in order to learn. More specifically, they incorporate Hebbian Learning in the form of Spike Timing Dependent Plasticity (STDP) [7], where the neuron's weights of incoming connections (dendrites) are adjusted depending on the relative timing of input and output spikes. Neurons in SNNs have an associated learning rule that determines how to adjust the weights of its inputs once it spikes. Four different Hebbian learning rules are illustrated in Figure 2.3.

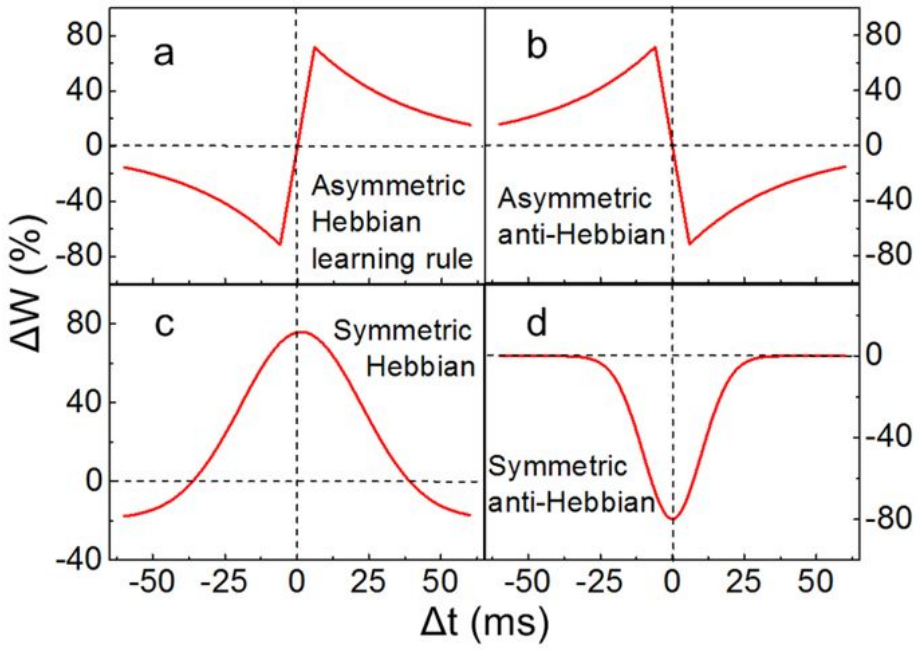


Figure 2.3: Hebbian learning rules. The graphs illustrate the adjustment of a weight (in percent) Δw dependent on the relative timing between input and output spikes Δt. Image taken from [7].

2.2.3 Learning Paradigms

There are several different approaches to ML/DL. Which paradigm an approach belongs to depends on the learning algorithm and to which degree supervision is required in order to learn. The three common paradigms are called supervised learning, unsupervised learning and reinforcement learning. Our work in this thesis fits into a different paradigm, which is self-supervised learning.

Supervised learning

Supervised learning [3, p. 6] is the ML/DL paradigm of learning an approximation of some function that best maps a set of inputs to their correct outputs, given a training set that contains example pairs of inputs and correct outputs (also called labels or ground truths). With this training set, we are able to let our model 'practice' predictions and gradually learn to approximate the function we are interested in.

To train the model, an error function to be optimized is defined, based on the difference between the predicted outputs and the ground truths. During training, the network receives input examples from the training set which are fed forward throughout the neural network, resulting in the output. We then calculate the

error of each such output with regards to the ground truths. The actual learning happens when the weights are adjusted. A common method for adjusting the weights of the model with supervised learning is called backpropagation, where the error at the output layer is propagated backwards throughout the neural network, allowing the adjustment of the weights in the hidden layers so that it can make better approximations in the future. There are also other training methods available for supervised learning, such as evolutionary algorithms.

Supervised learning is well suited for classification and prediction problems. For example, it can be used to predict the score of a movie review based on the text in the reviewer's comment, or the classic ML example of predicting if an image contains a cat or a dog. However, it is not very general, as a model resulting from supervised learning is specialized at solving a specific problem. Training sets are also required, and ideally these are very large. Large data sets can be expensive to produce, and in the cases that they are not available, supervised learning may not be applied.

Unsupervised learning

In unsupervised learning [3, p. 181], no ground truth is provided from the training data during training[2]. Since unsupervised learning methods have nothing to practice against, they instead look at the input data and try to categorize them together based on intrinsic similarities. Unsupervised learning models can also mimic input data, that is to generate new data samples with the same statistics as the input data set. In other words, the data itself is used to guide the learning.

In order to understand how such methods work in practice, we will take a look at a simple clustering algorithm called K-means Clustering [3, pp. 282–285]. This method does not involve neural networks, but is easy to understand. In K-means Clustering, the data is divided into K classes (or clusters) given that you are able to determine how many clusters your data set should be divided into. The algorithm is initialized by placing K center points for clusters in your data set at random. Each data point is assigned to the class representing the closest center point. Once all the data points have been assigned to a cluster, the center point is moved to the mean of the points assigned to it. This process is iterated until convergence. An illustration of how the algorithm clusters data points is shown in Figure 2.4.

Another unsupervised approach that involves neural networks is called Generative Adversarial Nets (GAN) [8]. In GAN, two neural networks are trained simultaneously: one generative model G that generates new 'fake' data similar to the training data, and one discriminative model D that estimates the probability that an input sample came from G rather than the training data. G is trained in a way so that it maximizes the error of D's predictions.

[2]Ground truths may be used in unsupervised learning for performance validation.

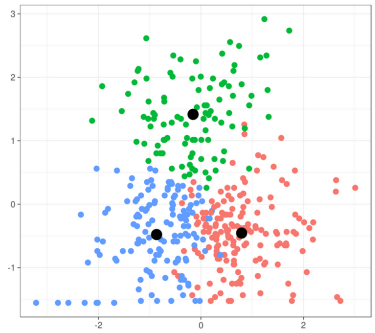


Figure 2.4: Visualization of the K-means Clustering algorithm. Image taken from `https://rpubs.com/cyobero/k-means`.

Reinforcement learning

We have seen how supervised learning trains a model by providing it the correct answers from the training data, and how unsupervised learning trains by exploiting similarities in the data. Reinforcement learning [3, pp. 231–246] can be described as a paradigm somewhere in between the two. In reinforcement learning, the model receives feedback that quantifies how good an answer is, but not how to improve it. In other words, the reinforcement learner needs to search for different strategies in an attempt to figure out which one gives the best solution.

It can be explained in terms of an agent interacting with an environment. In the context of reinforcement learning, the agent is the learner, and the environment is where it is learning and what it is learning about. The environment provides the agent with input in the form of states, and gives feedback about how good a strategy is through some reward function. The ultimate goal of the agent is to find a strategy that maximizes the total reward.

Reinforcement learning is based on the concepts of states and actions. The states are given as input to the agent from the environment, which maps them to actions that will maximize the total reward. This mapping from states to actions is called the policy, π. Given a state S_t at time step t, the agent performs an action A_t and receives a reward R_{t+1} from the environment which ends up in state S_{t+1}. This cycle is illustrated in Figure 2.5.

Since future rewards are uncertain, and we often care more about immediate rewards, future rewards are usually discounted[3]. The total future reward R from time step t is therefore given by

$$R = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \gamma \in [0, 1], \tag{2.2}$$

where γ is the discount factor for future rewards.

[3]Temporal discount is also a phenomenon found in human decision making [9].

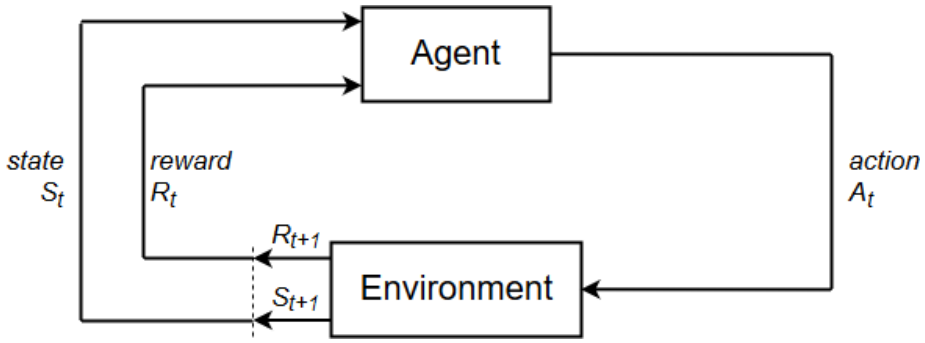


Figure 2.5: The reinforcement learning cycle.

Generally, there are two classes of reinforcement learning problems: episodic and continuous. In episodic problems, learning is split into episodes where each episode has a terminal state, which means that the majority of the rewards can be given at the end. An example is the video game Super Mario Bros., where the goal is to navigate Mario to a flag pole (the terminal state) at the end of each episode. Continuous problems, on the other hand, continue forever. In other words, they have no terminal state. An example is the video game Flappy Bird, where the goal is to navigate a bird between as many pipes as possible without crashing.

Self-supervised learning

In self-supervised learning [10], there is no external supervision involved in the learning of the model. Instead, the input itself is used to adjust the controller of the model through some sensory response. A type of self-supervised learning is called embodied learning, where an agent interacts with an environment through its senses.

To get an idea of what self-supervision through embodiment means, we take a look at a simple scenario. Imagine someone tasting some food they have never seen or tasted before. It might taste disgusting or delicious, and their sense of taste will provide them with a response corresponding to a punishment or reward respectively. If it was disgusting, they are less inclined to eat it again. On the other hand, if it was delicious they have now learned a new type of food that they like. The taste for delicious and disgusting food was acquired through several years of supervised evolution [11]. So now, even without supervision from the environment, one can interact and learn something from one's own sensory experience.

2.3 Evolutionary Algorithms

An evolutionary algorithm (EA) [12, pp. 25–34] [13] is a meta-heuristic[4] optimization algorithm that draws inspiration from biological evolution. EAs are population-based, which means that the algorithm processes a set of multiple candidate solutions simultaneously. The underlying idea stems from Charles Darwin's theory of evolution [15], where the individuals in the population compete within some environment under environmental pressure, which in turn causes natural selection. For a given optimization problem with a fitness function to be maximized, we can then apply this function to the candidate solution as a measure of fitness. Since the superior solutions and their offspring are more likely to survive, the overall performance of the solutions in the population increases.

There are several subclasses of EAs that are specialized for different applications and problems. Some of the most popular variants are called Genetic Algorithms (GA), Genetic Programming (GP), Evolutionary Programming (EP) and Evolutionary Strategies (GS). While these differ in certain areas, such as how solutions are represented (genotype), they all share the same general scheme of an EA.

2.3.1 General Scheme

At the very beginning of a run of an EA, an initial population is generated. The individuals in the initial population are typically randomly generated, but it is also possible to apply problem-specific knowledge during initialization.

The next step, called parent selection, is to sample candidate solutions to be recombined with each other into new solutions, similar to mating between individuals in a biological population.

Once pairs of parents are selected, a binary variation operator, called recombination or crossover, is applied to the two individuals. Recombination merges information from both parents into one or two offspring.

In addition to recombination, most EAs also use a mechanism called mutation to explore diverse solutions. A mutation is a binary operation on a single individual. Typically, an individual has a low chance of mutating between generations, where the mutation is a small change in the solution. A mutation can be applied to an offspring, or an offspring can be created by mutation alone.

After the algorithm has generated an additional set of new candidates (the offspring), the fitness function is then applied to all the solutions to measure their performance. Based on their fitness score, both offspring and parents are selected for a place in the new population of the next generation, where individuals with higher fitness have a higher chance of being selected.

[4] A meta-heuristic is a higher level, problem independent, iterative generation strategy that helps to guide an underlying search algorithm [14].

This whole iteration may be repeated until some termination condition has been achieved. This could be that a satisfactory solution has been found, or a computational limit has been reached, at which point the algorithm will terminate and the individual with the highest fitness is selected as the solution.

This general scheme is visualized in Figure 2.6. It is also worth noting that not all EAs include every step in the scheme. EP, for example, does not use recombination.

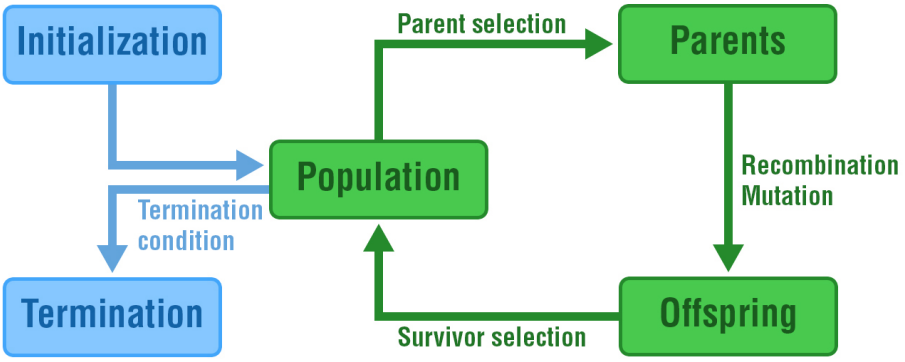


Figure 2.6: Flowchart representation of an EA.

2.3.2 Genotypes

The first step in defining an EA is how to link the original problem in the 'real world' to the problem solving space where the EA operates. Optimization problems can be complex, like optimizing the height and width of an airplane wing or the path of the Travelling Salesman Problem. To be able to apply EAs, it is often necessary to simplify or abstract some aspects of the real world. In other words, we need to define a way to represent the objects that we wish to optimize in a way that can be stored, evaluated and manipulated by a computer.

The solutions in the context of the 'real world' are called phenotypes, while the encoding of the phenotype, or the individuals within the context of the EA, are called genotypes (also called genomes). In the example of the airplane wing, the phenotype is the actual physical wing itself, and the genotype could for example be a collection of floating point numbers containing the dimensions of the wing.

There are numerous possible data structures that can be used as genotypes. Genotypes can be simple or they can be more complex objects that include multiple features, which are often referred to as 'genes'. Some common genotype representations are listed below:

- Binary representation (bit strings);
- Integer representation;

- Real-valued or floating-point representation;
- Permutation representation;
- Tree representation [12, pp. 75–76].

Choosing the genotype has multiple implications for the further definition of the EA. How the fitness function is defined, as well as recombination and mutation must be done with the genotypic representation in mind. If the genotype is a permutation, it will need a different kind of strategy for recombination than a genotype that is a floating point number. As for mutation, if the genotype is a permutation, a mutation could be to swap the order of two subsequent members of the permutation, while a mutation of a floating point number could be a small increase or decrease to the value of a gene.

There are two main approaches to how the genotype can be mapped to its phenotype. They are called *direct* and *indirect encoding*. In direct encoding, every feature of the phenotype are explicitly defined directly in the genotype. The genotype in an approach using indirect encoding is more like a compressed representation that does not directly represent what the phenotype will look like, but functions more like a set of instructions on how the phenotype should develop, much like DNA in living beings.

2.4 Neuroevolution

Neuroevolution [16] is the application of EAs on neural networks. In neuroevolution, EAs may be utilized to evolve features of the neural networks, such as weights, topology, learning rules and activation functions, with the ultimate goal of optimizing the performance of the neural network.

The different features of a neural network are topics for evolution. We can roughly divide the levels that neuroevolution operates on into three categories: connection weights, network topology and learning rules. Historically, the most common way of evolving neural networks is by evolving the weights. Crossover and mutation is utilized to evolve the weight values of a network instead of using gradient based methods such as backpropagation. Evolving the topology enables neuroevolution to explore neural network architectures and adapt to problems without the need of human design and predefined knowledge. Evolving the learning rules is especially interesting with regards to Artificial General Intelligence (AGI), because it can be regarded as a process of 'learning how to learn'. Like with evolving topologies, this also enables automatic design and discovery of novel learning rule combinations within the neural network.

From the perspective of the EA, the neural networks serve as the phenotype. Just as with the general case of EAs, a central problem when defining any neuroevolution algorithm is how to encode the neural network into a genetic representation. Genotype encoding schemes can be divided into two main categories: direct and indirect encoding. In direct encoding schemes (used by

most neuroevolution algorithms) the genome contains information about every node and the connections between the nodes, while indirect encoding usually only specifies the rules for how a neural network may be constructed from the genome.

2.4.1 Neuroevolution of Augmenting Topologies

NeuroEvolution of Augmenting Topologies (NEAT) [17] is a Topology and Weight Evolving Artificial Neural Network (TWEANN) neuroevolution algorithm. NEAT presents a solution to the competing conventions problem, in addition to removing the necessity of designing topologies for the networks to be evolved.

Two networks can represent the exact same function even though their encodings are different, as illustrated by Figure 2.7. This is called the competing conventions problem, and offspring generated by competing conventions typically have poor fitness since it will lose desirable properties from its parents. NEAT's solution is to historically mark each gene with an innovation number that identifies the historical ancestor of each gene. Whenever a new connection gene emerges, it is assigned a new innovation number and stored in a global innovation list. This way, the algorithm is able to look up and identify equal genes to avoid competing conventions.

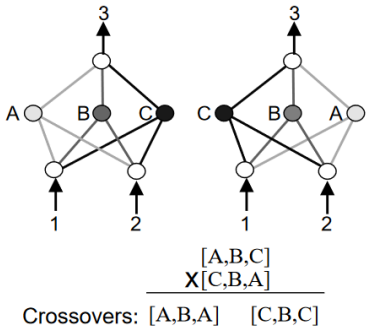


Figure 2.7: The competing conventions problem. The figure illustrates how recombination between two functionally equivalent networks produce damaged offspring. Image taken from [17].

NEAT's genotype represents the linear connectivity of a neural network and contains genes for both nodes and connections. The node genes contain a unique node identifier and information about what type of node it is (input, hidden or output). Each connection gene contains references to the in-node and the out-node that it connects, the weight value of the connection, an enable-bit that determines whether the connection gene is expressed or not, and an innovation number as previously described. If a gene is not expressed in the genotype, it will not be included in the phenotype. The relationship between genotype and phenotype is illustrated in Figure 2.8. NEAT features three different mutation operators: mutating weight values, adding connections and adding nodes. The node and connection mutations are illustrated in Figure 2.9. The weights are

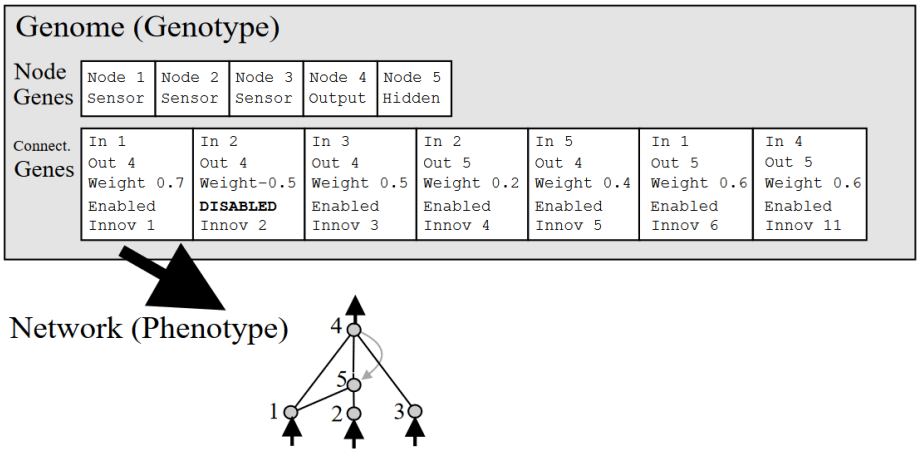


Figure 2.8: An example of genotype to phenotype mapping in NEAT. Image taken from [17].

mutated by simply modifying the value of the weight. When adding a connection, a new connection gene is created and added to the genotype. When adding a node, it's in practice inserted in between a connection. The old connection is disabled, and the new node is added to the genotype together with two new connection genes: one from the input to the disabled gene to the new gene, and one from the new gene to the output of the old gene. When crossover is applied,

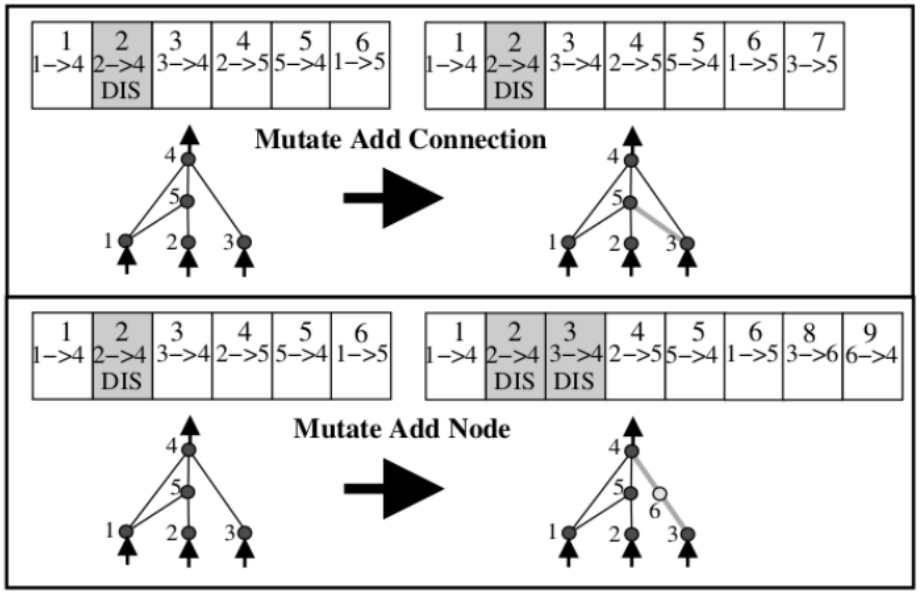


Figure 2.9: Illustration of how nodes and connections are added to neural networks through mutation. Image taken from [17].

connection genes with the same innovation numbers in both parents are lined up. The genes with matching innovation numbers are randomly chosen from

either parent, and the rest of the genes are inherited from the parent with the best fitness value.

In order to protect the emergence of innovative new topologies and allow them time to optimize their structure, NEAT features speciation. The idea of speciation is that individuals primarily compete within their own niche rather than with the entire population. NEAT implements speciation by calculating a measure of distance δ between individuals based on the number of excess genes E and disjoint genes D, as well as the average weight difference of matching genes $\overline{W}$, given by

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W}, \tag{2.3}$$

where N is the total number of nodes in a gene and c1, c2 and c3 are coefficients that can be modified to adjust the importance of the different factors.

If the distance δ between two individuals is above some threshold δ_{th}, the two individuals are defined as belonging to different species and are unable to be crossed over with one another.

To ensure diversity in the population and to prevent a single species from being too dominant in the population, NEAT also uses a population management approach called fitness sharing. This means that all individuals of a species share the fitness of their niche, which encourages exploring a variety of diverse solutions even though they might have a lower fitness than a more populous niche.

2.4.2 Neuroevolution of Artificial General Intelligence

The neuroevolution component of NAGI [18] is a modification of NEAT. The actual algorithm is largely the same, but the features in the genotype of the individuals are different, partly because the phenotypes intended for NAGI come in the form of SNNs. The genotype of individuals in NEAT includes information about the weights of a network. NAGI's focus is on the self-learning capabilities of artificial neural networks. We do not want new individuals to inherit knowledge about the environment, and information about the weights of the SNNs are therefore not included in the genotype, but instead randomly initialized for each individual in every generation. In the absence of weights, the genotype includes information about the type of neuroplasticity in a neuron, and if a neuron is either inhibitory or excitatory.

Fitness of individuals is measured by their lifetime after a simulation in a reactive and mutable environment. When an agent interacts with a reactive environment, it must create some sort of reaction in the agent which is either positive, negative or neutral. This is necessary in order for learning to happen. A mutable environment means an environment that can change its structure and rules during simulation.

Chapter 3

Related Work

In this chapter we refer to previous research that has used similar approaches to those used in the NAGI framework.

3.1 Adaptive NEAT

Stanley et al. [19] present an approach combining NEAT with adaptive synapses, utilizing local Hebbian learning rules to adjust the weights of a network. The intent of the approach is to train controllers for agents interacting in an environment where they forage for food. The authors verified the performance of this approach both with and without adaptive synapses, and their results show that networks in both cases reach maximum fitness. Because of this, they argued that they were displaying adaptive properties.

This approach is quite similar to the approach in the NAGI framework, with some key differences. Importantly, agents were interacting with static environments, meaning an environment never changed its state during the agents lifetime. Secondly, learning rules were associated to connections, as illustrated in Figure 3.1. In the NAGI framework a learning rule is instead associated to a neuron and affects all the incoming connections to that neuron.

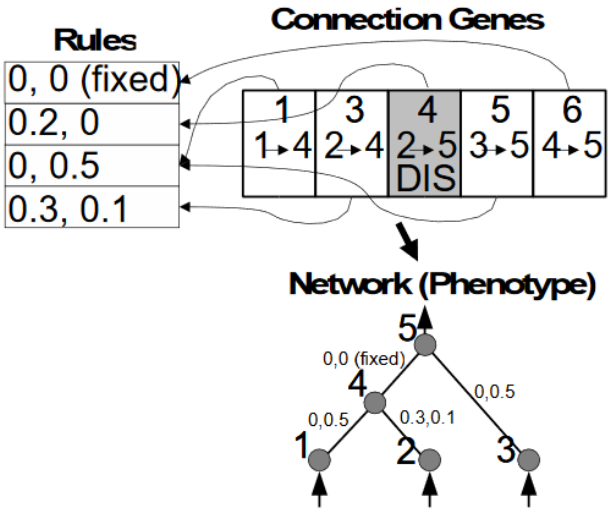


Figure 3.1: Illustration of how learning rules are assigned to connections. Image taken from [19].

3.2 Weight Agnostic Neural Networks

Gaier and Ha [20] present their Weight Agnostic Neural Networks (WANNs) approach where they question the importance of weight parameters of a neural network in comparison to the network architecture in regards to the learning capabilities of a network. In their work, they explore to what extent a network architecture alone, without learning any weight parameters, is able to solve a given task. To this end, they sample a single weight parameter from a uniform random distribution that is shared between every single connection in the network. They demonstrated that they were able to find minimal network architectures that could solve several reinforcement learning tasks using this approach.

3.3 Polyworld

Some of the key driving forces in natural evolution for many species are cooperation and competition. Certain species are absolutely dependent on each other and achieve much more together with other individuals in flocks or packs than they would alone. Competition both inter- and cross-species serves as pressure pushing the most fit individuals to figure out the best way to survive. For these reasons, there are good arguments to be made that multi-agent environments are promising when searching for AGI.

One of the first approaches using multi-agent environments is the PolyWorld ecological simulator by Yaeger [21]. PolyWorld is a simulated environment where agents forage for randomly generated food. These agents have artificially evolved neural network controllers that utilizes Hebbian learning. The agents can interact both with the environment and each other by eating, moving, fighting, mating, change their field of view and communicate by changing the brightness of their bodies. Their results showed that the behavior that emerged from the agents in PolyWorld showed similarities to behavior seen in natural occurring environments.

3.4 Projective Simulation for AI

One of intelligence's many manifestations is creativity, a phenomenon where something new and valuable is formed. Creativity comes from being able to imagine or project something that has never happened or existed, and extrapolate previous limited knowledge to something new. In other words, creativity allows intelligent beings to imagine unprecedented scenarios, and to relate previously experienced scenarios with future conceivable scenarios. For example, applying music theory and musical inspiration from other artists into the process of constructing a completely new song is creativity in motion.

Briegel and De las Cuevas [22] present a scheme of information processing for intelligent agents which allows for an element of creativity as described above, allowing agents to adapt based on experience. They call their central feature *Projective Simulation* (PS), which allows agents to project themselves into possible future scenarios based on previous experience. PS uses a memory system which they call *Episodic and Compositional Memory* (ECM) that serves as the basis that allows agents to simulate possible future actions before they decide on their actual action. In broad strokes, ECMs are stochastic networks of *clips*, which represent previous experience. The ECM is constantly updated in three different ways:

1. By updating the transition probabilities between existing clips;
2. By creating new clips based on experience (i.e. new input);
3. By creating new clips from existing ones.

PS then happens by doing a random walk through the ECM, which when processed influences the action an agent makes together with perceived input as illustrated in Figure 3.2.

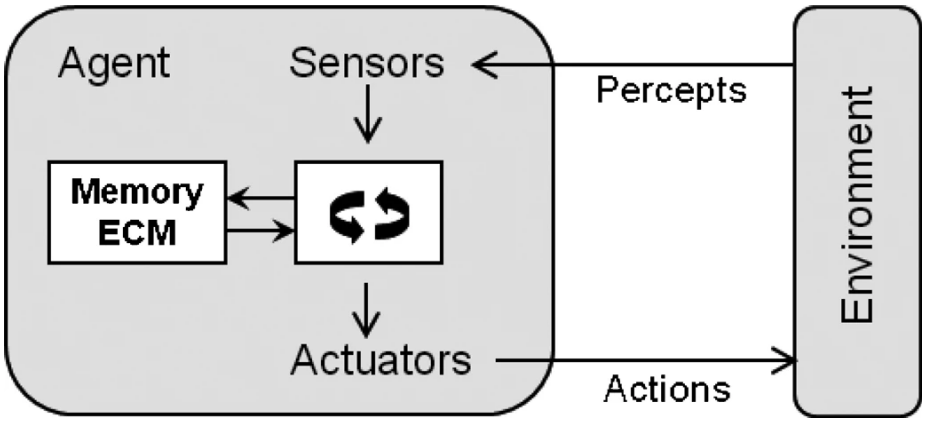


Figure 3.2: Illustration of the information processing flow in an agent using PS. Illustration taken from [22].

3.5 Neural MMO

Suarez et al. [23] present an AI environment inspired by the video game genre *Massively Multiplayer Online Role-Playing Games* (MMORPGs or simply MMOs), aiming to simulate the setting of a massive number of organisms competing for limited resources. The environment pressures the agents to learn and adopt robust combat and navigation strategies in order to survive in the presence of a large number of other agents attempting to do the same.

The results from their experiments show that population size magnified the development of behavior needed to survive due to increased exploration. Agents trained in a large population outclassed agents trained in smaller populations when pitted against each other. In their approach, agents could also have shared or unshared weights. They found that agents with unshared weights more so than agents with shared weights developed policies that naturally diverged to fill different niches in order to avoid competition, and thus they found that niche development was magnified by increasing the number of agents with unshared weights in the population.

3.6 A Brain-Inspired Framework for Evolutionary Artificial General Intelligence

Nadji-Tehrani and Eslami [24] present *A Brain-Inspired Framework for Evolutionary Artificial General Intelligence* (FEAGI), a framework inspired by the evolution of the human brain, which utilizes neuroevolution, both excitatory and inhibitory spiking neurons, neuroplasticity and neuronal/synaptic prun-

ing[1]. They present a proof of concept which they claim demonstrates how a simplified model of the human visual cortex is capable of character recognition.

3.7 Unsupervised Learning of Digit Recognition Using Spike-Timing-Dependent Plasticity

Diehl and Cook [26] present an approach to digit recognition using SNNs with conductance based synapses, STDP for weight changes, lateral inhibition (an excited neuron has the ability to reduce the activity of its neighbors) and adaptive spiking thresholds. They did not use a teaching signal or labelled data in their approach. Their architecture was able to achieve a 95% accuracy on the MNIST benchmark data set. Since no domain-specific knowledge was used when training the architectures, they argue that this points towards the general applicability of the resulting networks.

3.8 Social Learning vs Self-teaching in a Multi-agent Neural Network System

Le et al. [27] present a study where they compare the effect of social learning with the effect of individual learning. They propose a neural architecture called a 'self-taught neural network' which allows an agent to learn by itself without any external supervision. This network architecture operates with two modules, both of which are neural networks: an action module and a reinforcement module, which have the same sets of inputs, but differing hidden and output neurons. The action module takes sensory information as inputs and produces reinforcement outputs in order to guide the action of the agent. The goal of the reinforcement network is to provide reinforcement signals to guide the behavior of each agent.

They simulated a multi-agent system where agents need to develop adaptive behavior in order to compete with each other in order to survive. The results from their experiments showed that the evolved self-taught behavior was the most effective in their simulated environment.

[1]Synaptic pruning is a process happening in the brain of mammals (including humans) where synapses are gradually being eliminated between early childhood and puberty [25].

Chapter 4

Neuroevolution of Artificial General Intelligence

This chapter contains a detailed explanation of the approaches that are used in this thesis, which make up the NAGI framework. The first section explains the concept behind the framework itself, while the subsequent sections explain its components in detail.

4.1 Framework concept

Neuroevolution of Artificial General Intelligence (NAGI), proposed by Pontes-Filho and Nichele [18], is a framework that brings together approaches from AI, evolutionary robotics and artificial life [28]. The authors of NAGI envision that the framework can lead research a step in the right direction for the emergence of AGI in its simplest form. The main concept and inspiration behind the framework stems from the long-lasting natural evolution of general intelligence found in biological organisms.

The learning paradigm in focus is self-supervised learning through embodiment, which is also inspired from how biological organisms do most of their learning. This is done by simulating an agent interacting with an environment that is constantly changing. The hypothesis is that by using this approach, it will lead to the emergence of simple models that have general, problem independent learning capabilities, and that are able to adapt in changes to their environment without any supervision from the environment or otherwise.

The following is a condensed summary of the framework concept. An agent is equipped with a SNN as its control unit. The SNN topologies are evolved by an EA starting from a minimalist structure which gets increasingly bigger and more complex as generations go by. The agent is placed in a mutable environ-

ment where the rules of interaction are constantly changing. The mutability of an environment is important in order for the agents to be able to develop generalizing capabilities (or 'learn how to learn') and solve the survival problem for multiple environments rather than just learning how to survive optimally in a single static environment. Over time, agents that are consistently able to survive in changing environments should be able to survive in environments never seen before, even by their ancestors. Agents survive for longer if they perform the correct actions, which is determined by some logic dependant on the modeled problem. The same action can be correct and incorrect at different points in time because of the mutable nature of the environment. Agents have access to the environment through an 'interface' of sensory inputs. The environment provides rewards and punishment to the agent through these senses. Fitness of each agent is measured by how long they survived in an environment, so the goal for each agent is to survive for as long as possible.

4.2 Spiking Neural Networks as Control Units

This section explains how to encode input for SNNs and some spiking neuron models that were used in the framework. SNNs are explained in Section 2.2.2.

4.2.1 Data Representation and Input/Output Encoding

In classic ANNs, input and output usually come in the form of integers or floating point values. In SNNs, however, the data flowing in and out of a network are encoded as a sequence of spikes with an associated firing rate, i.e. a frequency, or the number of spikes per second. The firing rate usually has a minimum and a maximum value, and signals can be encoded to either continuous firing rates in between these, or binary values. The range of the firing rates can either be encoded explicitly, such as values in the range $[0Hz, 100Hz]$, or simplified such as a real number in the range $[0, 1]$. From these firing rates, spike trains (sequences of spikes over a time interval) are generated which serve as input (also called *stimulus*) to the network. The spikes can be sampled from different probability distributions such as Poisson, Normal or Uniform distributions, resulting in different inter-spike interval patterns.

In the approach used in this thesis, we used binary values for the firing rates. Binary data values such as 0 and 1 were encoded into low and high frequencies respectively. We also chose to use *One Hot Encoding* for the inputs. A one-hot is a group of bits where the only legal combinations of values are those where a single bit has the value 1, and the rest have the value 0 [29, p. 129]. An example is shown in Table 4.1.

Binary	One-Hot	Firing Rate
0	01	(low, high)
1	10	(high, low)

Table 4.1: How binary encoded data values coincides with one-hot encoded values, which in turn translates into a tuple of firing rates for SNN input.

4.2.2 Network Architecture

Each SNN has a fixed number of input nodes and output neurons, and an arbitrary number of hidden neurons. Hidden neurons can be both excitatory or inhibitory, while output neurons are always excitatory. Cycles are permitted while duplicate connections between two neurons are prohibited. Neurons may also be connected to themselves.

4.2.3 Spiking Neuron Models

There are many different models available for spiking neurons that differ in complexity and computational requirements. In this section we will explain two models that were explored for use in NAGI: the biologically plausible Izhikevich Neuron Model and the simpler Integrate and Fire Neuron Model.

Izhikevich Neuron Model

The first neuron model that was considered for the framework is the Simple Model of Spiking Neurons presented by Izhikevich [5]. The model is claimed to be as computationally efficient as the integrate and fire model, and as biologically plausible as the Hodgkin-Huxley model [30].

Mathematically the model operates with two variables u and v:

- v denotes the membrane potential of the neuron;
- u denotes the membrane recovery of the neuron.

In addition, it uses four parameters a, b, c and d that affects how the value of these variables are manipulated:

- a decides the time scale for the recovery variable u. A smaller value for a results in a slower recovery for u;
- b decides the sensitivity of the recovery variable u with regards to the fluctuations of the membrane potential v below the firing threshold. A bigger value for b makes the recovery variable u follow the fluctuations of v more strongly;

- c decides the reset value for the membrane potential v following an output spike;
- d decides the reset value for the recovery variable u following an output spike.

Different combinations of values for a, b, c and d result in neurons with varying firing patterns, some of which can be seen in Figure 4.1. To update v and u,

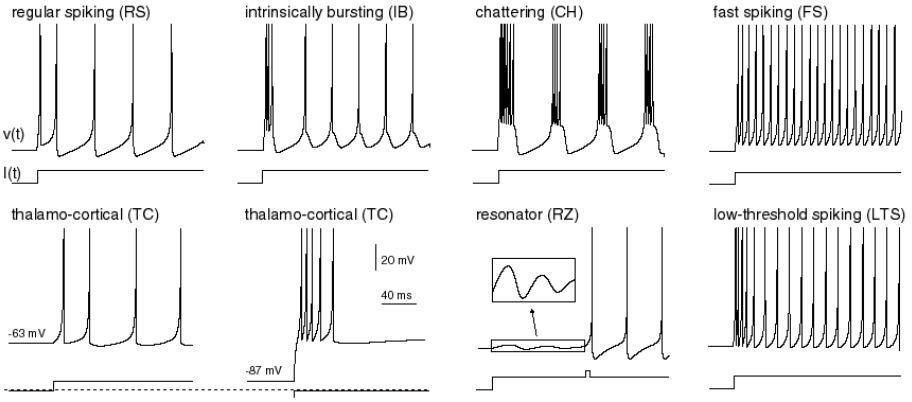


Figure 4.1: Different firing patterns resulting from different combinations of values for a, b, c and d. Electronic version of the figure and reproduction permissions are freely available at `www.izhikevich.com`.

the equations for the derivatives with regards to the time t are given by

$$v' = 0.04v^2 + 5v + 140 - u + I \tag{4.1}$$
$$u' = a(bv - u) \tag{4.2}$$

where I is the current going through the neuron at that point in time. We can then then express the changes in these variables, Δv and Δu, with regards to a time step size Δt with the following equations:

$$\Delta v(\Delta t) = \Delta t(0.04v^2 + 5v + 140 - u + I) \tag{4.3}$$
$$\Delta u(\Delta t) = \Delta t a(bv - u) \tag{4.4}$$

Given input values $\{x_1, x_2, ..., x_n\}$ and the weights of the connections the inputs are passed through $\{w_1, w_2, ..., w_n\}$ at time step $t + 1$, the current I is updated by

$$I = b + \sum_{i=1}^{n} w_i x_i \tag{4.5}$$

where b is the bias current into the neuron.

v and u are reset to the following after the membrane potential has exceeded the firing threshold v_{th}:

$$\text{if } v > v_{th}, \text{ then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{4.6}$$

Integrate and Fire Neuron Model

The Integrate and Fire (IF) Neuron Model is one of the first neuron models that was explored and is still used somewhat frequently today because of its computational simplicity. The idea behind the model is to represent a neuron with a simple electric circuit consisting of a resistor and a capacitor connected in parallel to each other [31], as illustrated in Figure 4.2.

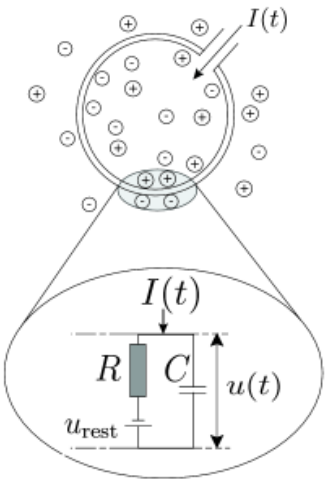


Figure 4.2: Illustration of an IF neuron modelled after an electronic circuit. Image taken from [31].

A neuron is represented in time by the equation

$$I(t) = C_m \frac{dv(t)}{dt} \tag{4.7}$$

which is the derivative with regards to time of the law of capacitance $q = CV$, where q is the electronic charge, C is the capacitance and V is the voltage. When the membrane potential v exceeds the membrane threshold v_{th}, a spike is released and the membrane potential returns to the resting membrane potential v_{rest}:

$$\text{if } v > v_{th}, \text{ then } v \leftarrow v_{rest} \tag{4.8}$$

Simplified Integrate and Fire Neuron Model

We can introduce a simplified IF model which mimics the behavior of the IF model while abstracting away its electric circuit nature. A neuron's membrane potential v is increased directly by its inputs (similar to the current in the Izhikevich Model) and decays over time by a factor λ_{decay}. We can then express the change in membrane potential Δv with regards to a time step Δt by the

equation

$$\Delta v(\Delta t) = \sum_{i=1}^{n} w_i x_i - \Delta t \lambda_{decay} v \tag{4.9}$$

Like the IF model, if the membrane potential v exceeds the membrane threshold v_{th}, a spike is released and the membrane potential returns to the resting membrane potential v_{rest} as expressed by Equation 4.8.

4.2.4 Homeostasis

Homeostasis in biological terms is a steady equilibrium of physical and chemical conditions in a living system. We want to achieve something similar for our neural networks. Neurons can have inhomogenous inputs, which could lead to very different firing rates. For example, the firing rate of a neuron with five incoming connections may dominate the firing rate of a neuron that only has one incoming connection. It is desirable that all neurons have approximately equal firing rates [32]. In order to homogenize the firing rates of the neurons in a network, the net membrane threshold v_{th}^* is given by

$$v_{th}^* = \min(v_{th} + \Theta, \sum_{i=1}^{n} w_i) \tag{4.10}$$

where Θ is increased every time a neuron fires and decays exponentially. Each neuron has an individual Θ. This way, a neuron firing very frequently will get an increasingly large membrane threshold and by consequence a lower firing rate, while a neuron with weak incoming weights will get an increased firing rate.

4.3 Spike Timing Dependent Plasticity

Adjustment of the weights of the connections going into a neuron happens on every input and output spike to and from a neuron through STDP. This is done by keeping track of the time elapsed since the last output spike, as well as the time elapsed since each input spike for each incoming connection within a time frame, called the STDP time window, which is usually set to be around $\pm 40ms - 50ms$. The difference between pre and post synaptic spikes, or the relative timing between them, denoted by Δt is given by the equation

$$\Delta t(t_{out}, t_{in}) = t_{out} - t_{in} \tag{4.11}$$

where t_{out} is the timing of the output spike and t_{in} is the timing of the input spike.

The synaptic weight change Δw is calculated in accordance to one of the four Hebbian learning rules mentioned in Section 2.2.2. An illustration of the curves corresponding to each learning rule can be seen in Figure 2.3. The functions for

each of the four learning rules are given by the equations below

$$\Delta w(\Delta t) = \begin{cases} A_+ e^{\frac{-\Delta t}{\tau_+}} & \Delta t > 0 \\ -A_- e^{\frac{\Delta t}{\tau_-}} & \Delta t < 0 \\ 0 & \Delta t = 0 \end{cases} \quad \text{Asymmetric Hebbian} \quad (4.12)$$

$$\Delta w(\Delta t) = \begin{cases} -A_+ e^{\frac{-\Delta t}{\tau_+}} & \Delta t > 0 \\ A_- e^{\frac{\Delta t}{\tau_-}} & \Delta t < 0 \\ 0 & \Delta t = 0 \end{cases} \quad \text{Asymmetric Anti-Hebbian} \quad (4.13)$$

$$\Delta w(\Delta t) = \begin{cases} A_+ g(\Delta t) & g(\Delta t) > 0 \\ A_- g(\Delta t) & g(\Delta t) < 0 \\ 0 & g(\Delta t) = 0 \end{cases} \quad \text{Symmetric Hebbian} \quad (4.14)$$

$$\Delta w(\Delta t) = \begin{cases} -A_+ g(\Delta t) & g(\Delta t) > 0 \\ -A_- g(\Delta t) & g(\Delta t) < 0 \\ 0 & g(\Delta t) = 0 \end{cases} \quad \text{Symmetric Anti-Hebbian} \quad (4.15)$$

where $g(\Delta t)$ is a Difference of Gaussian function given by

$$g(\Delta t) = \frac{1}{\sigma_+\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{\Delta t}{\sigma_+})^2} - \frac{1}{\sigma_-\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{\Delta t}{\sigma_-})^2} \quad (4.16)$$

A_+ and A_- are parameters that affect the height of the curve, τ_+ and τ_+ are parameters that affect the width or steepness of the curve of the Asymmetric Hebbian functions and σ_+ and $\sigma-$ are the standard deviations for the Gaussian functions used in the Symmetric Hebbian functions. It is also required that $\sigma_- > \sigma_+$. By using the graphing tool Desmos (`www.desmos.com`) we manually found fitting ranges for each of these parameters, which can be seen in Table 4.2 and Table 4.3.

Symmetric	
A_+	$[1.0, 10.6]$
A_-	$[1.0, 44.0]$
σ_+	$[3.5, 10.0]$
σ_-	$[13.5, 20.0]$

Table 4.2: Symmetric STDP parameter ranges.

Asymmetric	
A_+	$[0.1, 1.0]$
A_-	$[0.1, 1.0]$
τ_+	$[1.0, 10.0]$
τ_-	$[1.0, 10.0]$

Table 4.3: Asymmetric STDP parameter ranges.

Weights can take values in a range $[w_{min}, w_{max}]$, and every neuron has a weight budget w_{budget} it must follow. What this means is that if the sum of a neuron's incoming weights exceed w_{budget} after STDP has been applied, they are

normalized to w_{budget}, given by

$$\text{if } \sum_{i=1}^{n} w_i > w_{budget}, \text{ then } w_i = \frac{w_i w_{budget}}{\sum_{i=1}^{n} w_i}. \tag{4.17}$$

4.4 Modified NEAT

The framework uses an EA to evolve the SNNs for agents which is a modified version of NEAT, explained in Section 2.4.1. Most of the modifications have been made to accommodate for the fact that we are using SNNs in place of regular ANNs. This section will explain all the modifications to the EA. Any part of the EA not mentioned in this section functions exactly the same as in NEAT.

4.4.1 Genome

At the top level, the genome looks a lot like the genome in NEAT. It contains a collection of node genes and a collection of connection genes. It's on the gene level where the modifications to the genome are made, which we will examine in the following sections. A full example genome is illustrated in Figure 4.3.

Node Genes

Like in NEAT, the node genes come in three types: input nodes (or sensor nodes, as they are called in NEAT), hidden nodes and output nodes. Depending on the type of the node gene, it will have a different collection of *loci*[1].

The input nodes serve the same purpose as they do in regular NEAT. They do not actually represent neurons, they simply receive input signals and distribute them along the network. As such, they have not been modified at all. Both the hidden nodes and the output nodes represent spiking neurons. They both have three additional loci: a learning rule, a collection of learning rule parameters and a bias.

The learning rule is one of the four Hebbian Learning rules seen in Figure 2.3. The collection of learning rule parameters contain four parameters that decides the shape of curve corresponding to the learning rule. They are different for symmetric and asymmetric learning rules, with the symmetric parameters being $\{A_+,\ A_-,\ \sigma_+, \sigma_-\}$ and the asymmetric parameters being $\{A_+,\ A_-,\ \tau_+, \tau_-\}$. Refer to Section 4.3 for an explanation of each parameter. The bias is a Boolean value that determines whether it should have a constant bias signal, analogous to the background noise of a neuron.

[1] In EA terms, a value within a gene is also called a *locus* (plural *loci*).

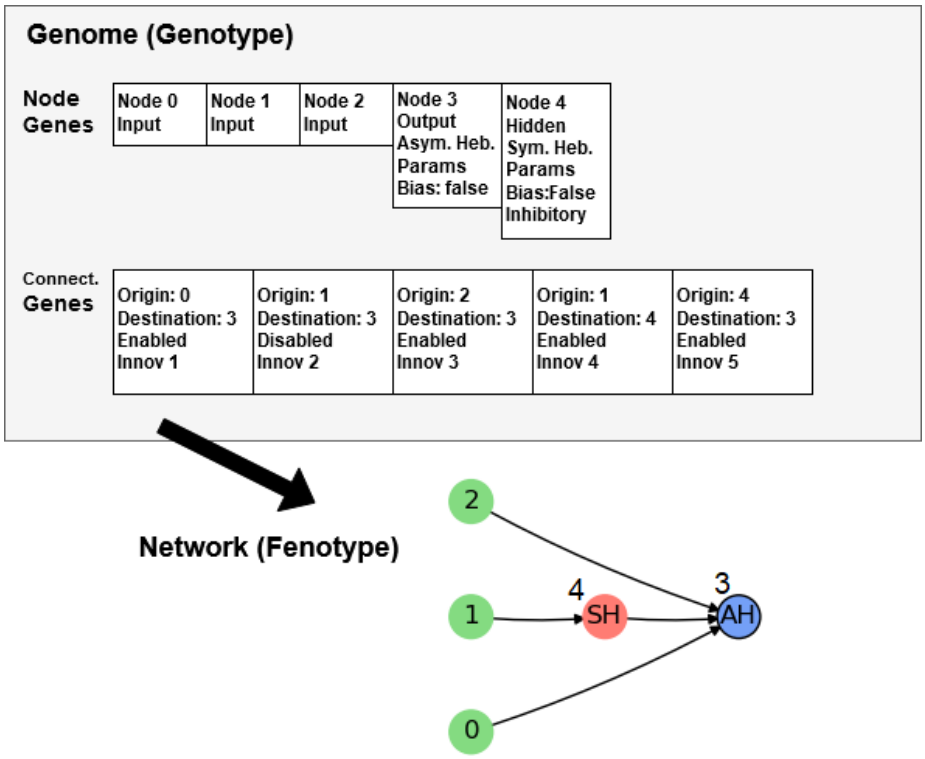


Figure 4.3: An example genotype to phenotype mapping in NAGI.

In addition, the hidden node genes have a locus with a Boolean value that determines if it represents an inhibitory or excitatory neuron. This locus is not included in the output node genes because they are always excitatory.

Connection Genes

The connection genes in our modified NEAT are very similar to the ones found in regular NEAT. One important principle in the NAGI framework is that we don't want individuals inheriting information about the environment from their predecessors. Because of this, the locus containing the value for the weight are removed from the genome. Weights are instead assigned random values during initialization when the genome is decoded into a SNN. We also chose to rename two loci in the connection genes: the 'in' and 'out' node identifiers have been renamed to 'origin' and 'destination' respectively to avoid confusion. A connection points *from* the origin node *to* the destination node.

4.4.2 Initializing Additional Loci

The following is a description of how each of the loci introduced to the genome in NAGI are initialized:

- **Excitatory/Inhibitory:** A neuron is initialized either as a inhibitory or excitatory neuron. The probabilities for the two outcomes can be different;
- **Bias:** A neuron is initialized either with or without a bias. The probabilities for the two outcomes can be different;
- **Learning Rule:** A neuron is initialized with one of four Hebbian learning rules described in Section 4.3. The probabilites for each outcome can be different. In the approach used in this thesis, inhibitory neurons had a higher probability of being initialized with the two Anti-Hebbian rules, and excitatory neurons had a higher probability for the two 'regular' Hebbian rules;
- **Learning Rule Parameters:** Each parameter is initialized by sampling from a uniform distribution within the ranges listed in Table 4.2 and Table 4.3.

4.4.3 Mutating Additional Loci

The following is a description of how each of the loci introduced to the genome in NAGI are mutated when a node mutates:

- **Excitatory/Inhibitory:** There is a chance with a predetermined probability that it changes from excitatory to inhibitory and vice versa;
- **Bias:** There is a chance with a predetermined probability that it gains a bias (if it lacks one) or loses its bias (if it has one);
- **Learning Rule:** There is a chance with a predetermined probability that the learning rule is changed to one of the three other learning rules. If the learning rule changes from a symmetric learning rule to an asymmetric learning rule or vice versa, the learning rule parameters are also re-initialized as described in Section 4.4.2;
- **Learning Rule Parameters:** The learning rule parameters can mutate in two ways, each with a predetermined probability:
  - By incrementing or decrementing each parameter p by sampling from a normal distribution with $\mu = 0$ and $\sigma^2 = M(p)$ and adding the sampled value to the current parameter value. $M(p)$ is the mutate scale for the given parameter and is given by the equation

$$M(p) = 0.2(p_{max} - p_{min}) \tag{4.18}$$

where p_{max} and p_{min} are the maximum and minimum values the parameter can have, given by the ranges in Table 4.2 and Table 4.3. The mutate scales for each parameter can be found in Table 4.4 and Table 4.5;

– By fully re-initializing each parameter as described in Section 4.4.2.

Symmetric	
$M(A_+)$	1.92
$M(A_-)$	8.6
$M(\sigma_+)$	1.3
$M(\sigma_-)$	1.3

Table 4.4: Symmetric learning rule parameter mutate scales.

Asymmetric	
$M(A_+)$	0.18
$M(A_-)$	0.18
$M(\tau_+)$	1.3
$M(\tau_-)$	1.3

Table 4.5: Asymmetric learning rule parameter mutate scales.

4.4.4 Initial Population Topologies

At the start of the algorithm, every individual in the population consists of only input and output nodes. There's a few things to consider when initializing their topology. One approach is to initialize every network densely connected[2], but that would mean that all genomes are equal initially, greatly narrowing the search and also limiting the possible topologies, especially minimal ones. On the other hand, if you initialize them completely randomly, there's a good chance that many individuals in the population will have 'dead' nodes (nodes that are not connected to the rest of the network), as illustrated in Figure 4.4.

As a balanced measure, we opted for the following approach when initializing connections in the initial genomes: first, each output node is connected to one input node. After that, every remaining possible connection has a chance of also being included in the genome with a predetermined probability $I_{connection}$. This ensures that there is a path going *to* every output node and most likely *from* every input node, which makes for 'healthier' initial topologies while keeping a varied population.

[2]In a densely connected network, also called a *Dense Neural Network*, every single node in a layer is connected to every single node in the next layer.

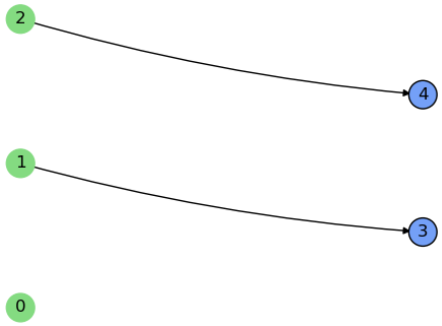


Figure 4.4: A network with a dead input node (0).

4.4.5 Fitness Function

The fitness of an individual is measured through a simulation of agent-environment interaction which is explained in Section 4.5.3. Performance is based on the lifetime t of an agent in an environment. We make use of a fitness function that measures performance based on an agent's lifetime t and is normalized for values in the range $[0, 1]$ using the maximum possible lifetime L_{max} and minimum possible lifetime L_{min}. The fitness function f is given by

$$f(t) = \frac{t - L_{min}}{L_{max} - L_{min}}. \tag{4.19}$$

4.5 Agent-environment Simulation

In this section we describe the different parts and actors that go into play during simulation of agent-environment interactions.

4.5.1 Self-Supervised Learning Through Embodiment

Our approach suggests that agents are able to learn self-supervised through embodiment, i.e. interacting with and sensing the environment with their bodies. Since local learning rules are tasked with adjusting the weights, and by consequence the behavior of the network, the learning approach is therefore within the realm of self-supervision. However, simulated agents don't have senses or tangible bodies. Self-supervised learning is only plausible when interaction between an agent and the environment is mutually reactive (also referred to as an agent interacting in a *reactive environment*), meaning that the environment affects the agent and vice versa, for example the sensory-motor system of self-driven cars driving on roads in the real world. Non-reactive environments do not react to anything the agent does. An example is an object detector where the environment only provides visual information with no other interaction.

To achieve self-supervised learning through embodiment for non-reactive environments, we propose Virtual Embodied Learning (VEL), a method for emulating a virtual reactive environment. VEL introduces a sensory-motor system, providing the agent with environmental rewards and penalties, as illustrated in Figure 4.5. Additionally, VEL keeps track of information pertaining to the internal state of the agent, like health and/or hunger.

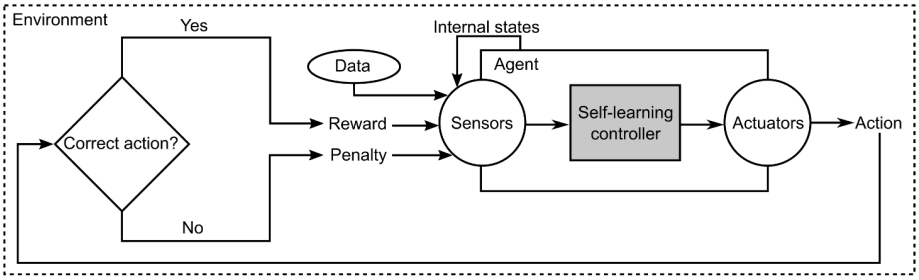


Figure 4.5: Illustration of information flow and mechanisms allowing for self-supervised learning through embodiment by use of VEL. Image taken from [18].

4.5.2 The Components of the Agent

The agent can be broken down into four main components, all of which interact with each other in a feedback loop:

- **Self-learning controller:** A SNN as described in Section 2.2.2 and Section 4.2;
- **Sensors:** The sensors provide input to the controller. The inputs can further be broken into two types: the input samples from the environment and the reward or penalty from the sensory-motor system;
- **Actuators:** The actuators keep count the number of spikes that has occurred within a time window in the output nodes of the controller. Different actuators represent different actions the agent can make. The current action of the agent is determined by the actuator with the highest current spike count. In the case of tied spike counts between actuators, the action will be chosen from the actuator that previously had the highest spike count;
- **Internal States:** In the work conducted in this thesis, this is simply the agent's health points, which were not used as sensory input.

All of the different components of an agent is illustrated in Figure 4.5.

4.5.3 Flow of Agent-environment Simulation

The simulation starts with creating an agent from a genome evolved from our EA, as well an environment. On creation, the environment creates an input order and a state order. The order of the inputs can be any permutation of the possible input types, but the order always loops after one full sequence of the possible inputs have been presented to the agent. Similar to the input order, the environment state order is random and loops once every possible state has been active.

Once everything is set up, the agent is presented with input one at a time. Interaction is then simulated for a predetermined number of time steps with a predetermined step size, while the environment gives reward or penalty signals and deals damage to the agent depending on the agents actuators. After a predetermined number of inputs have been presented to and consumed by the agent, the environment mutates into a different state.

The agent continuously loses health points (or receives damage) from living in the environment. The damage an agent takes depends on the value of its actuators and the environment state. In simple terms, if the agent is making the correct decision, it takes minimal damage, while taking maximal damage if making the wrong decision. In addition, if the agent is *confident* in its decision (if the spike count in one actuator is considerably larger than the other), the damage is either reduced or amplified based on whether the action for that actuator is correct or not. First, two partitions are calculated: a correct partition p_c, and an incorrect partition p_i, given by

$$p_c(s_c, s_i) = \begin{cases} \frac{\max(0,\min(s_c,s_t)) - \max(0,\min(s_c,s_t)) + s_t}{2s_t} & s_c + s_i \leq 2s_t \\ \frac{s_c}{s_c+s_i} & s_c + s_i > 2s_t \end{cases} \tag{4.20}$$

$$p_i(s_c, s_i) = 1 - p_c(s_c - p_i) \tag{4.21}$$

where s_c and s_i are the spike count for the correct and incorrect actions and s_t is the minimum 'target' number of spikes. The purpose of s_t is to avoid assigning a too high or low fitness to agents that fire few spikes through their outputs. The agent takes damage at every time step given by

$$d_(s_c, s_i) = d_c p_c(s_c, s_i) + d_i p_i(s_c, s_i) \tag{4.22}$$

where d_c and d_i are values for the damage an agent would take for a fully correct or incorrect damage respectively. This is iterated until the agent runs out of health points and 'dies'. At the end of simulation, the fitness of the agent is calculated from the fitness function expressed in Equation 4.19.

4.5.4 Mutable Environment

We're looking to exploit and assert the self-learning and generalizing properties of the evolving SNNs, which serve as controllers in agents tasked with surviving

in their environment for as long as possible. We therefore introduce a mutable environment, where the rules of survival are constantly changing.

In the real world, biological organisms inherit survival mechanisms that allow them to adapt to the environment, like animals with camouflaging bodies that mimic their habitat. Certain insects, like stick-bugs have inherited bodies that makes them difficult to make out visually as long as they stay in their habitat [33]. Some organisms have taken it a step further, allowing them to adapt to changes in the environment. Certain octopi, fish, frogs and chameleons are able to change the colour and/or texture of their bodies to blend in with their surroundings (or for communication) [34][35], and furred mammals like the arctic fox change the density and/or color of their coat depending on the season to keep them from freezing or overheating, or to camouflage themselves in snow [36].

In the following sections, we propose two types of mutable environment inspired by such examples from nature. They both make use of binary inputs, but differ in the number of input signals per input sample.

Food Foraging

We first propose a simple environment for food foraging simulation. The environment provides the agent with two types of input, or food: black and white. The agent can interact with food in one of two ways, by eating it or by avoiding it. The food can either be toxic or healthy, and whether a color of food is toxic or healthy is dependent on the state of the environment. For example, white food can start out being healthy, and the agent should eat it. But once the environment mutates, it can suddenly make white food toxic, and now the agent should avoid it. Figure 4.6 is a simple illustration of this type of environment. The food is encoded in a way so that the agent is able to distinguish between them, but the agent cannot know which food is healthy and which is toxic at any one time. The agent can only figure this out by interacting with the food. An incorrect action is defined by eating a toxic food, or avoiding a healthy food, while a correct action is defined by eating a healthy food or avoiding a toxic food. If the agent makes an incorrect action, it receives a penalty signal (representing pain, revulsion or hunger) and if it makes a correct action, it receives a reward signal. The environment has four possible states which describe which color of food is currently healthy: black, white, both or none. The correct action for each state is shown by Table 4.6.

Food Truth Table				
Input \ Healthy	Black	White	None	Both
Black	Eat	Avoid	Avoid	Eat
White	Avoid	Eat	Avoid	Eat

Table 4.6: Truth table showing the correct action for each combination of input food color and healthy food.

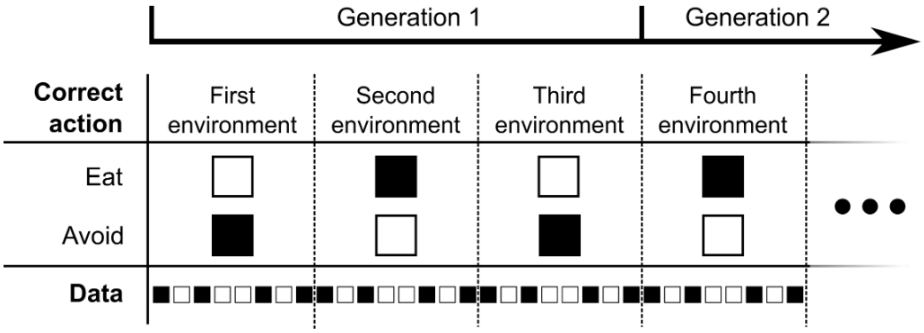


Figure 4.6: Simple illustration of a food foraging type environment using binary encoding. Image taken from [18].

Logic Gates

In this environment, the mutable environment state is a two-input logic gate. The environment provides the agent with two binary inputs of 0's and 1's. The agent's task is to predict the correct output for the current logic gate given the current input. Similarly to the food foraging environment, it receives a reward signal if it's currently predicting the correct output, and a penalty signal if it's currently predicting the wrong output.

In order to measure the generalizing properties of agents, we propose the use of two slightly different environments: a training environment, which is used in calculating the fitness while running the EA, and a test environment which has a fully disjoint set of possible environmental states. A full overview of the logic gates found in both the training and the test environments, as well as exhaustive truth values for all input and output combinations, are found in Table 4.7 and Table 4.8.

Training Logic Gate Truth Table									
Input		A	B	NOT A	NOT B	ONLY 0	ONLY 1	XOR	XNOR
A	B								
0	0	0	0	1	1	0	1	0	1
0	1	0	1	1	0	0	1	1	0
1	0	1	0	0	1	0	1	1	0
1	1	1	1	0	0	0	1	0	1

Table 4.7: Truth table showing the correct output for each training logic gate.

Testing Logic Gate Truth Table					
Input		AND	NAND	OR	NOR
A	B				
0	0	0	1	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	0	1	0

Table 4.8: Truth table showing the correct output for each testing logic gate.

Chapter 5

Implementation of the NAGI Framework

In this chapter we briefly discuss choices related to the implementation of the code, as well as giving credits to existing code that influenced the implementation. The code is available at `https://github.com/krolse/neat-nagi-python`. The implementation is part of the Socrates Project [37] on GitHub, and future development will be made in a fork of the original implementation which will be made available at `https://github.com/SocratesNFR/neat-nagi-python` at a later point.

5.1 Language

When deciding on the language for the implementation, there were two main deciding factors. It made a lot of sense to choose a language that was familiar to both the author and the supervisors, and we wanted the implementation to be written in a language that's accessible and relevant to the scientific AI community. We decided on writing the implementation in Python 3 [38], because it is widely used in AI computing, having access to excellent AI frameworks such as *PyTorch* [39] and *TensorFlow* [40], as well as optimization packages such as *multiprocessing* and *NumPy*.

5.2 Coding Practice

Reading and writing code is a great way to learn the inner workings of a subject, such as SNNs and neuroevolution. However, it can be quite challenging to comprehend an implementation of a complex subject, depending on how the

code is written. This is especially true with dynamically typed languages like Python where it can be confusing what data structures are being used in the code. We wanted the implementation to be easy to read and follow so that it may serve as a comprehensible resource later for anyone looking to learn or implement any of the concepts used in NAGI. To achieve this we emphasized structure and readability over pure performance during implementation. We made sure that variables, arguments and functions had semantically descriptive names across the entire implementation. We also made frequent use of type hints, a feature introduced in the 3.5 version of Python 3 that allows you to specify type hints of any argument or variable. While type hints doesn't make Python a statically typed language, it is valuable for readability and linting purposes. Below is a short excerpt of code from the implementation that illustrates these practices.

```
def get_nth_top_genome(n: int, measure: str, results: Dict[int, Dict]):
    individuals = [(generation_number, genome_key)
                   for (generation_number, value) in results.items()
                   for genome_key in value['population'].genomes.keys()]

    if measure == 'fitness':
        individuals.sort(key=lambda x:
            results[x[0]]['fitnesses'][x[1]], reverse=True)
    elif measure == 'accuracy':
        individuals.sort(key=lambda x:
             results[x[0]]['accuracies'][x[1]], reverse=True)
    elif measure == 'end_of_sample_accuracy':
        individuals.sort(
            key=lambda x:
                (results[x[0]]['end_of_sample_accuracies'][x[1]],
                 results[x[0]]['accuracies'][x[1]]),
            reverse=True)
    else:
        raise Exception("Error: Faulty measure.")
    generation, genome_id = individuals[n]
    return results[generation]['population'].genomes[genome_id]
```

5.3 Credits

While much of the code in our implementation is original, we drew inspiration from three repositories in certain areas of the code. Much of the code related to SNNs, as well as some of the code for modified NEAT was inspired by the code found in the GitHub repository *neat-python*, authored by McIntyre et al. [41]. The code for modified NEAT is also inspired by code from Sean Wellecks compact NEAT-implementation on the GitHub Gist *neat.py* [42]. The code for visualizing neural network topologies is an expansion of the code related to visualizing neural networks in the GitHub repository *brain-tokyo-workshop* authored by Gaier and Ha [43].

5.4 Implementation Storyline

Before the implementation of the framework could start, we had to divide it into smaller tasks. First of all, we divided it into three main components: i) SNNs, ii) Modified NEAT, and iii) agent-environment simulation.

We decided to start with implementing the code for SNNs first, as this would give a better idea of which additional features that was going to be needed for the modified NEAT. The first thing we did was to implement code for SNNs with the Izhikevich neuron model, using *neat-python* by McIntyre et al. [41] as a starting point. Once an early iteration of SNNs was implemented, we went on to implement modified NEAT, using both *neat-python* and *neat.py* by Wellecks [42] for inspiriation. After this, we implemented STDP functionality for the SNNs, at which point we figured the parameters for the Hebbian learning rules could be added to the genome and be evolved by the EA.

When most of the SNNs and modified NEAT was implemented, we started writing tests to verify different parts of the code. In order to do this properly, we needed some way of visualizing both network behavior and topology. We implemented functionality for visualizing neuron behavior, weight modification (STDP) and visualizing neural network, the latter of which we used *brain-tokyo-workshop* by Gaier and Ha [20] as a starting point.

We encountered a challenge with visualizing our networks because we allow the networks to create cycles of connections between neurons, which makes it difficult to decide which layer a neuron belongs to. We solved this issue by making use of NEAT's innovation numbers:

1. Make a copy of the network, which is only used for calculating neuron positions;
2. Identify all simple cycles in the network;
3. For each simple cycle[1] in the copy, remove the connection with the largest innovation number, if it still remains (it is possible that a single connection can close multiple simple cycles);
4. Calculate node positions (layers) using the modified copy;
5. Visualize the original genome using the positions from the modified copy.

After visualization scripts were in place, we implemented the components needed for simulation. We also made visualization plots for the network activity during simulation, but we found that it was too complicated to analyze the activity of the Izhikevich neurons. After some consideration, we decided to use the simplified IF neuron model instead as it would simplify the analysis and presentation

[1] A *cycle* or a *circuit* is a path of connections through a graph where the first and final nodes are repeated. A *simple cycle* or a *simple circuit* is a cycle where the first and final nodes are the *only* nodes repeated, meaning there is no smaller cycle contained within the cycle [44, p. 164].

of a simulation. We re-used the implementation from the Izhikevich neuron model and modified it to accommodate for the differences in the two models.

After this followed multiple iterations of parameter tweaking, bug-fixing and testing until the implementation was performing as intended. We streamlined the user experience of the framework by making use of the operative systems file browsing. Finally, we wrote scripts for saving data structures, extracting data, and visualizing many different data plots that show activity during simulation and statistics from EA, which will be presented in Chapter 7.

Chapter 6

Experiments

This chapter details the experiments and hypotheses that were defined in order to answer the research questions of the thesis.

In order to answer **Research Question 1**, we define:

- **Hypothesis 1:** The evolved controllers with randomly initialized weights will be able to learn multiple decision boundaries by using evolved local learning rules.

The following hypothesis was used to answer **Research Question 2**:

- **Hypothesis 2:** The evolved controllers display general and problem-independent properties by being able to perform in new environments.

The following hypothesis was used to answer **Research Question 3**:

- **Hypothesis 3:** The evolved controllers are able to adjust the course of their actions based on sensory feedback.

6.1 Experiment 1: Food Foraging (Single Input)

The goal of this experiment was to test Hypothesis 1 and 3. To do this, we used the modified NEAT to evolve SNN controllers for agents with a single one-hot encoded input sensor and a single one-hot encoded reward/penalty sensor for a total of 4 input nodes, and 2 output nodes representing each possible action. Their fitness scores were measured by simulation in a food foraging type environment, as described in Section 4.5.4.

6.1.1 Expectations

The results from this experiment were expected to show that the agents were able to perform the correct actions in the changing environment, and that they would adjust their behavior from the sensory feedback.

6.2 Experiment 2: Logic Gates (Dual Input)

The goal of this experiment was to test Hypothesis 2, in addition to Hypothesis 1 and 3. To do this, this experiment includes two types of environments: a training environment for measuring the fitness of the population, and a test environment that is never encountered during training. In this experiment we used modified NEAT to evolve SNN controllers for agents with two one-hot encoded input sensors and a single one-hot encoded reward-penalty sensor for a total of 6 input nodes, and 2 output nodes representing each possible prediction. Their fitness scores were measured by simulation in a logic gate type environment, as described in Section 4.5.4, and their performance was tested on an unseen environment to measure generalizing and adaptive properties.

6.2.1 Expectations

From this experiment, the results were expected to show that the evolved controllers could perform comparably well in new environments that were not encountered during training.

6.3 Explanation of Metrics

Three different metrics are measured in the experiments: i) fitness, ii) accuracy, and iii) end-of-sample accuracy. The fitness of an agent is explained in Section 4.4.5. The accuracy A of an agent is the number of time steps where the agent was currently selecting the correct action $t_{correct}$ divided by the total number of time steps t_{total}, given by

$$A = \frac{t_{correct}}{t_{total}}. \tag{6.1}$$

The end-of-sample accuracy A_{eos} of an agent is similar to the accuracy, but instead of regarding the state of the agent at every time step, we only regard the state of the agent at the end of each input sample, given by

$$A_{eos} = \frac{s_{correct}}{s_{total}} \tag{6.2}$$

where $s_{correct}$ is the number of correct actions at the end of a completed sample and s_{total} is the total number of completed samples.

A fourth metric that we didn't explicitly measure in the experiments, but is important to define because of the discussion of the results in Chapter 7, is *confidence*. Confidence was briefly mentioned when explaining how damage is dealt in Section 4.5.3. The confidence C of an agent's action is given by

$$C = \frac{s_c}{s_c + s_i} \tag{6.3}$$

where s_c is the spike count for the correct action actuator and s_i is the spike count for the incorrect action actuator. In simple terms, a large spike count for the correct action and a low spike count for the incorrect action yields a high confidence.

6.4 Experimental Setup

This section contains tables with values and explanations of every configurable hyperparameter that were used in the experiments. The Simplified Integrate and Fire Neuron Model (described in Section 4.2.3) was used for both experiments.

Spiking Neural Networks			
Parameter	Explanation	Value	
		Experiment 1	Experiment 2
V_{spike}	The voltage of the spike signals from both the inputs and the spiking neurons.	1mV	1mV
v_{th}	The membrane potential threshold.	1mV	1mV
b	The value of the constant bias voltage of a neuron (if it has a bias).	1.0e-3mV	1.0e-3mV
Θ_{incr}	How much the threshold Θ of a neuron is incremented each time it fires an output spike, as discussed in Section 4.2.4.	0.2mV	0.2mV
Θ_{decay}	The decay rate of the threshold Θ per time step, as discussed in Section 4.2.4.	1.0e-3	1.0e-3
f_{high}	The high frequency for the one-hot encoded data, as discussed in Section 4.2.1.	50Hz	50Hz
f_{low}	The low frequency for the one-hot encoded data, as discussed in Section 4.2.1.	5Hz	5Hz

Table 6.1: Experimental setup for hyperparameters related to SNNs.

Weight Adjustment (STDP)			
Parameter	Explanation	Value	
		Experiment 1	Experiment 2
w_{budget}	The weight budget of each neuron, as explained in Section 4.3.	5	5
w_{max}	The maximal possible value of a connection weight.	1	1
w_{min}	The minimum possible value of a connection weight.	0	0
t_{window}	The time window where STDP happens, as explained in Section 4.3.	$\pm 40ms$	$\pm 40ms$

Table 6.2: Experimental setup for hyperparameters related to STDP.

NEAT Mutation Rates			
Parameter	Explanation	Value	
		Experiment 1	Experiment 2
M_{enable}	The probability of mutating the enabled/disabled gene.	0.01	0.01
M_{node}	The probability of mutating by adding a node.	0.1	0.1
$M_{connection}$	The probability of mutating by adding a connection.	0.1	0.1
$M_{inhibitory}$	The probability of mutating the inhibitory/excitatory gene.	0.1	0.1
M_{rule}	The probability of mutating by changing the learning rule.	0.1	0.1
M_{params}	The probability of mutating by perturbing the learning rule parameters.	0.1	0.1
M_{reinit}	The probability of mutating by fully reinitializing the learning rule parameters.	0.02	0.02
M_{bias}	The probability of mutating the bias gene.	0.1	0.1

Table 6.3: Experimental setup for hyperparameters related to mutation in modified NEAT.

NEAT (Miscellaneous)			
Parameter	Explanation	Value	
		Expmt. 1	Expmt. 2
pop_{size}	The number of individuals in a population.	100	100
n_{gen}	The number of generations the algorithm is ran for.	500	1000
I_{bias}	The probability of a neuron being initializied with a bias.	0.2	0.2
$I_{excitatory}$	The probability of a neuron being initializied with as an excitatory neuron.	0.7	0.7
$I_{connection}$	The probability of additional connections being included in a genome in the initial population, as explained in Section 4.4.4	0.7	0.7
$p_{disabled}$	The predetermined disabled rate for connections during recombination, as explained in Section 2.4.1.	0.7	0.5
b_{dist}	The learning rule distribution bias, or the probability that excitatory and inhibitory neurons are initialized with Hebbian and Anti-Hebbian learning rules respectively.	0.7	0.7
E	The excess connection coefficient used in calculating the distance δ between two individuals.	1	1
D	The disjoint connection coefficient used in calculating the distance δ between two individuals.	1	1
δ_{th}	The distance threshold for deciding if two individuals belong to the same species.	0.7	1
m_{cutoff}	The mating cutoff percentage. Only the m_{cutoff} top individuals of a species are able to reproduce.	0.2	0.2
s_{min}	The minimum size of a species.	2	2
$s_{protection}$	The number of generations a species is protected from extinction.	30	30
$s_{stagnation}$	The number of generations after which a species is considered stagnant if it has not improved its average fitness.	20	20
pop_{min}	The minimum number of species in a population.	4	4
$elitism$	The percentage of top individuals from a species that is kept in the population between generations.	0.1	0.1

Table 6.4: Experimental setup for miscellaneous hyperparameters related to modified NEAT.

Simulation			
Parameter	Explanation	Value	
		Experiment 1	Experiment 2
Δt	The time step size used when advancing the SNNs each time step during simulation.	0.1ms	0.1ms
t_{sample}	The simulation time for each input sample.	1s	1s
$t_{actuator}$	The time window of the actuators.	0.25s	0.25s
n_{input}	The number of input samples in a max duration simulation.	40	32
n_{flip}	Decides how often the environment mutates (after every n_{flip}th input sample).	4	4
s_t	The minimum 'target' number of spikes, as discussed in Section 4.5.3.	3	3
d_c	The damage taken from executing a correct action.	1	1
d_i	The damage taken from executing an incorrect action.	2	2

Table 6.5: Experimental setup for hyperparameters related to simulation.

Chapter 7

Results

In this chapter we present figures illustrating the results from the experiments described in Chapter 6.

7.1 How to Read the Results

There is a lot of information in the result figures. In this section we explain how to read them.

7.1.1 NEAT Monitoring

Figure 7.1, Figure 7.2, Figure 7.3, Figure 7.9, Figure 7.10 and Figure 7.11 show statistics over the NEAT population in both experiments. They show different metrics but are read the same way. The blue dots represent the fitness of a single individual in the population, the yellow line shows the average of the given metric in the population and the green line shows the maximum of the given metric in the population. The plots from Experiment 2 also have a gold line, which is the result of a simulation in a test environment of the top individual in that metric category (the individual on the green line).

Figure 7.4 and Figure 7.12 are stack plots that show the number of individuals in each species of the population at each generation.

7.1.2 Simulation Figures

In Figure 7.6, Figure 7.7, Figure 7.8, Figure 7.14, Figure 7.15 and Figure 7.16 there are certain elements that are common among them:

- **White Background Regions:** The regions with white background indicate blocks of time steps where the agent was currently deciding on a correct action and receiving a pleasure/reward signal.
- **Red Background Regions:** The regions with red background indicate blocks of time steps during simulation where the agent was currently deciding on an incorrect action and receiving a pain/penalty signal.
- **Vertical Grey Perforated Lines:** These lines indicate a point where a new input sample is encountered.
- **Vertical Black Lines:** These lines indicate a point where the environment mutates.
- **Bottom Table:** This table shows the input (top row) and the environment state (bottom row) for that block of the simulation.

7.1.3 Membrane Potential Figures

Figure 7.6 and Figure 7.14 show the membrane activity of each neuron in an agent's SNN during simulation. The numbers in parenthesis on the left hand side of the figure is the neuron identifier for that line lane. The green line represents the neuron's membrane potential and the blue line represents the neuron's membrane threshold. Each time the blue line's value increases, the neuron fires an output spike.

7.1.4 Weight Figures

Figure 7.7 and Figure 7.15 show the value of the weight of each connection in the agent's SNN during simulation. The numbers in parenthesis on the left hand side of the figure indicates the connection for that lane and is read as '(origin node, destination node)'. The y-axis ranges from [0, 1] for all lanes in the weight figures.

7.1.5 Actuator History Figures

Figure 7.8 and Figure 7.16 show the spike count of the agent's actuator during simulation. In Figure 7.8 from Experiment 1, the green line represents the spike count of the 'eat' actuator while the blue line represents the spike count of the 'avoid' actuator. In Figure 7.16 from Experiment 2, the green line represents the spike count of the '1' actuator, while the blue line represents the spike count of the '0' actuator.

7.2 Experiment 1

This section presents the results from Experiment 1.

7.2.1 NEAT Monitoring

Figure 7.1, Figure 7.2 and Figure 7.3 illustrate fitness, accuracy and end-of-sample accuracy statistics for the population during the run of the EA. Figure 7.4 illustrates the distribution of species in the population.

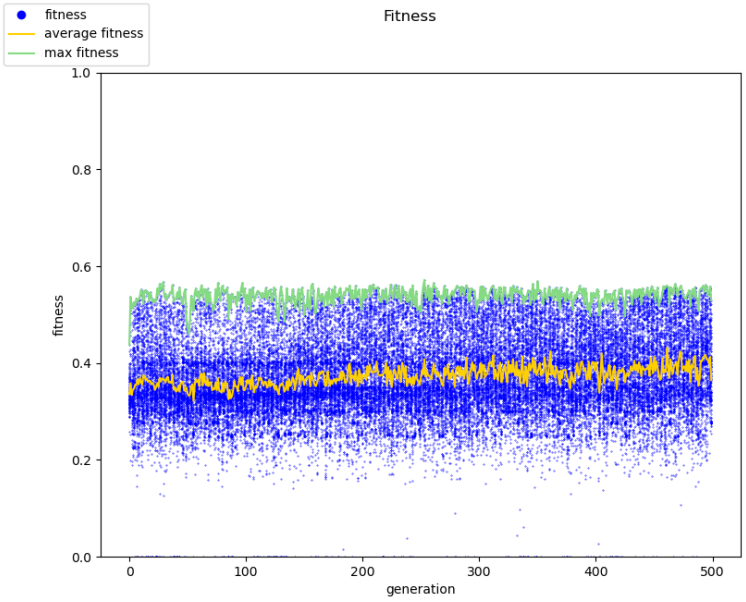


Figure 7.1: Visualization of the fitness statistics from NEAT in Experiment 1. See Section 7.1.1 for a full explanation.

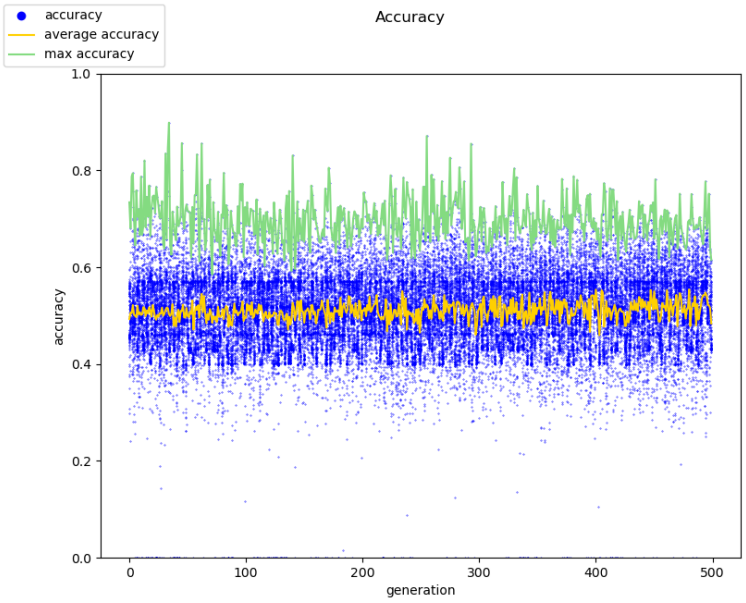


Figure 7.2: Visualization of the accuracy statistics from NEAT in Experiment 1. See Section 7.1.1 for a full explanation.

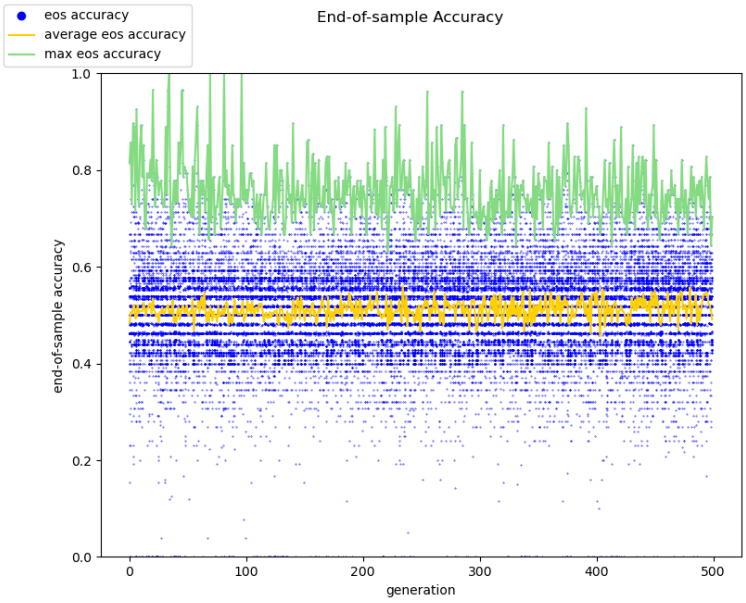


Figure 7.3: Visualization of the end-of-sample accuracy statistics from NEAT in Experiment 1. See Section 7.1.1 for a full explanation.

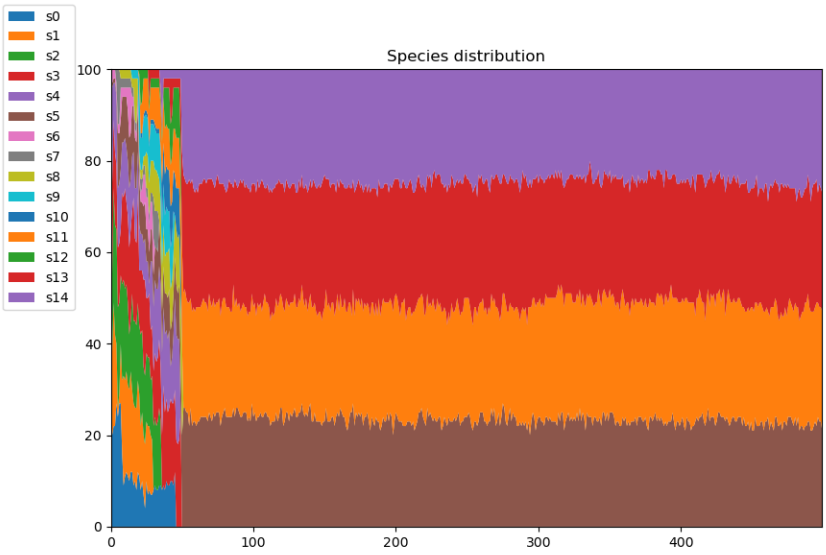


Figure 7.4: Visualization of the distribution of species in the NEAT population in Experiment 1.

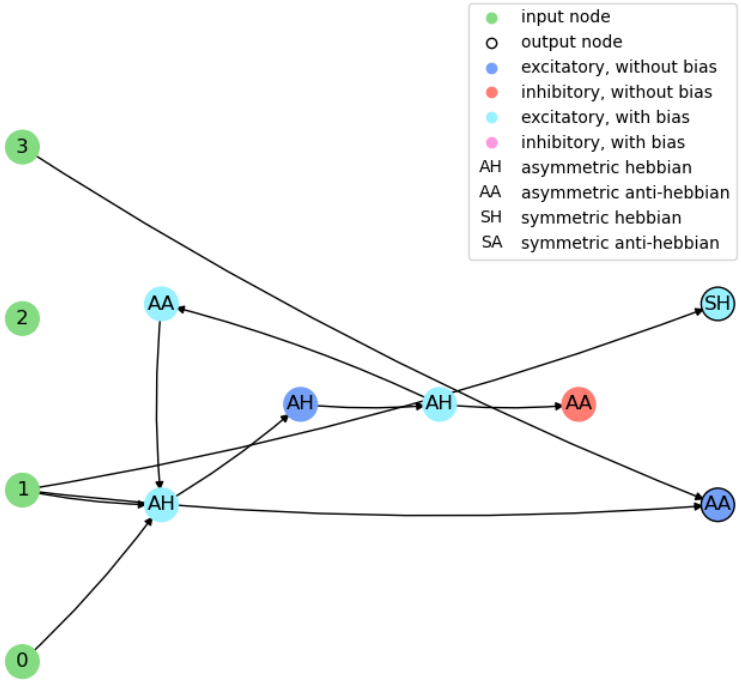
7.2.2 Simulation of High Performing Agent

After termination of the EA, a high performing agent was selected for validation simulation. For this experiment, the chosen agent was the agent with the highest accuracy in any generation, which was the agent with the highest accuracy in generation 34, as can be seen in Figure 7.2. Five validation simulations were conducted on this agent (all with different randomly initialized weights), the results of which can be seen in Table 7.1.

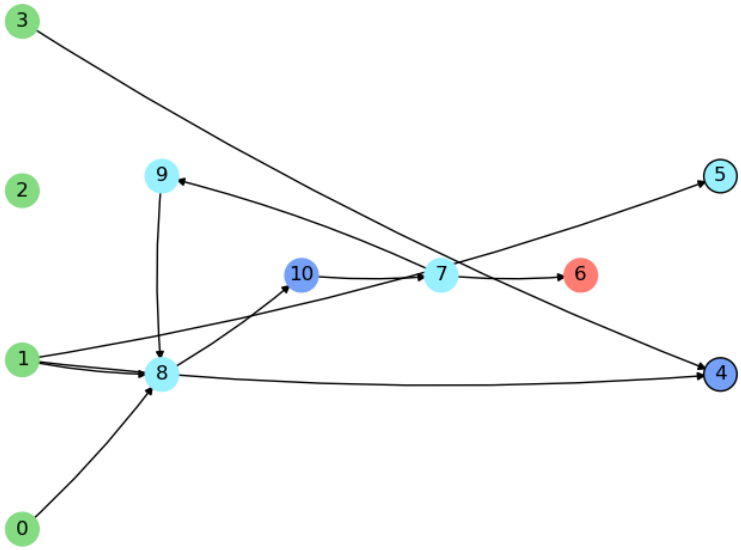
Experiment 1 Validation Simulations					
#	Fitness	Accuracy	Eos. Acc.	Input Order	Environment Order
1	0.356	89.5%	96.3%	white, black	both, none, white, black
2	0.362	90.6%	100%	white, black	white, none, both, black
3	0.361	91.3%	100%	black, white	white, both, none, black
4	0.348	85.4%	92.3%	white, black	white, black, both, none
5	0.359	89.2%	100.0%	black, white	both, white, black, none
Avg.	0.357	89.2%	97.7%	n/a	

Table 7.1: Validation simulations of the chosen agent from Experiment 1.

Figure 7.5 illustrates the topology of the chosen high-performing agent's SNN. Figure 7.6 illustrates the membrane activity of each neuron during simulation 1. Figure 7.7 illustrates the adjustment of each weight in the agent's SNN during simulation 1. Figure 7.8 illustrates the actuator history of the agent during simulation 1.



(a) Learning rules.



(b) Numeric identifiers.

Figure 7.5: Illustration of the network topology of the chosen agent. The one-hot encoded input sample goes into node 0 and 1, and the one-hot encoded reward/penalty signal goes into node 2 and 3. Node 4 is the output for the 'eat' actuator and node 5 is the output for the 'avoid' actuator.

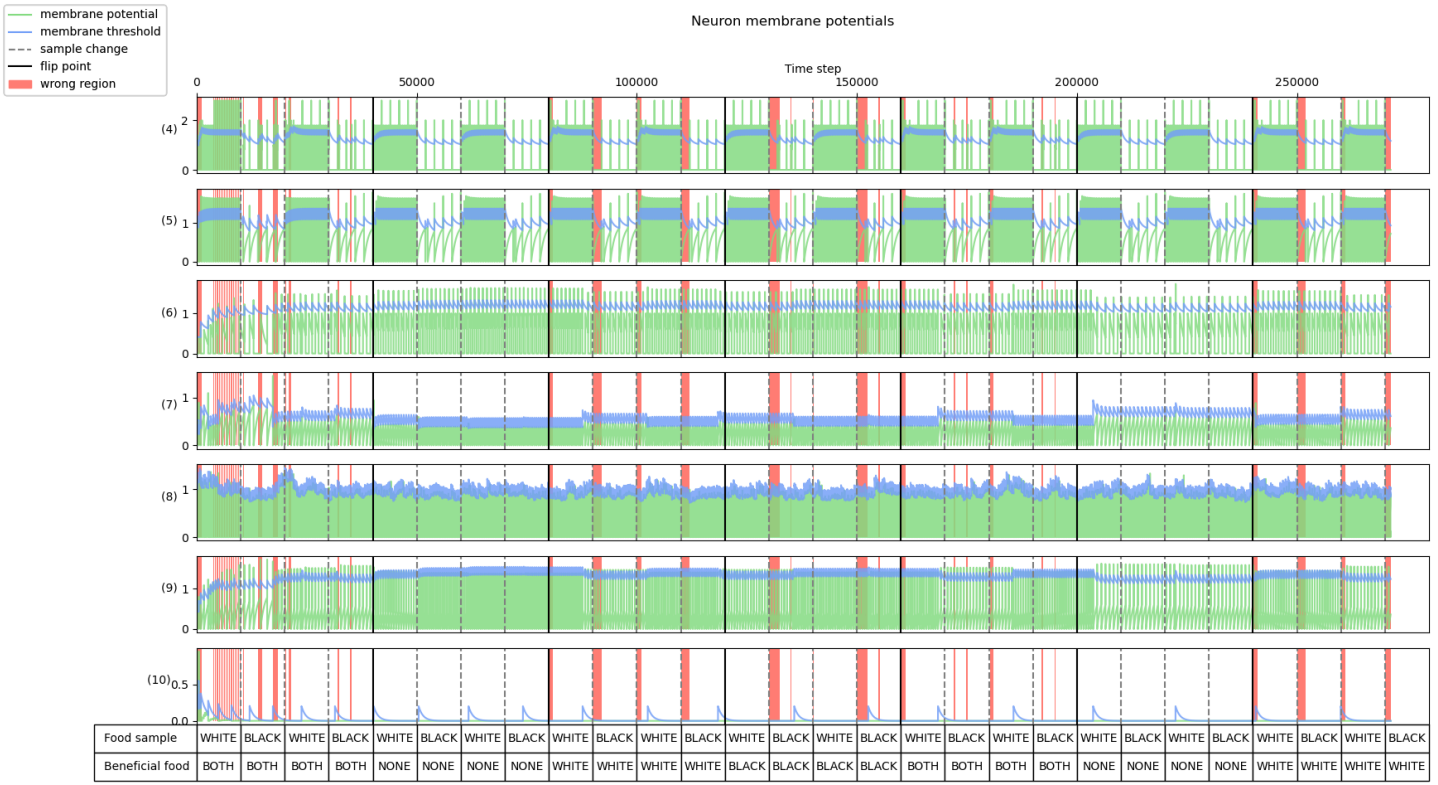


Figure 7.6: Visualization of the membrane potential of every neuron in the agent's SNN during validation simulation 1 for Experiment 1. See Section 7.1.2 and Section 7.1.3 for a full explanation.

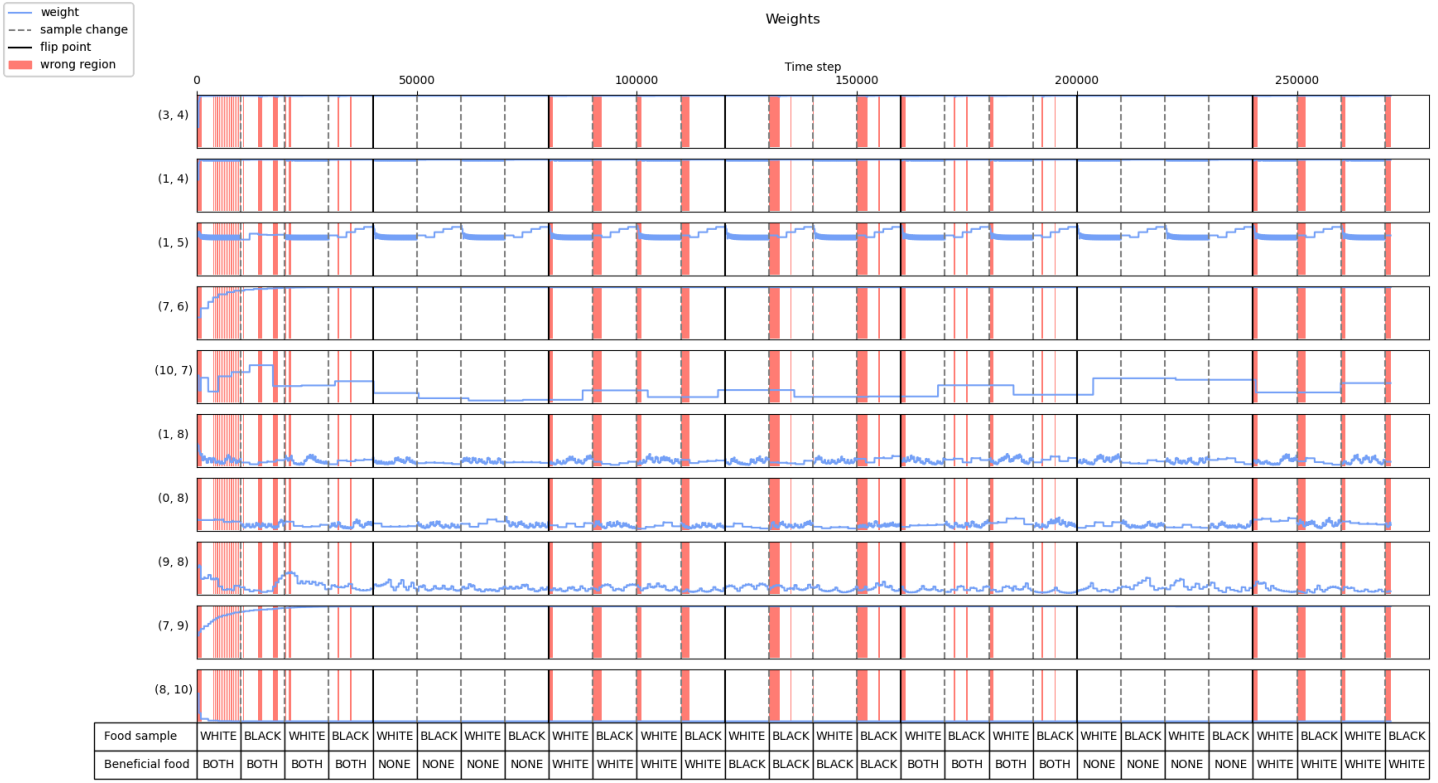


Figure 7.7: Visualization of the weight of every connection in the agent's SNN during validation simulation 1 for Experiment 1. See Section 7.1.2 and Section 7.1.4 for a full explanation.

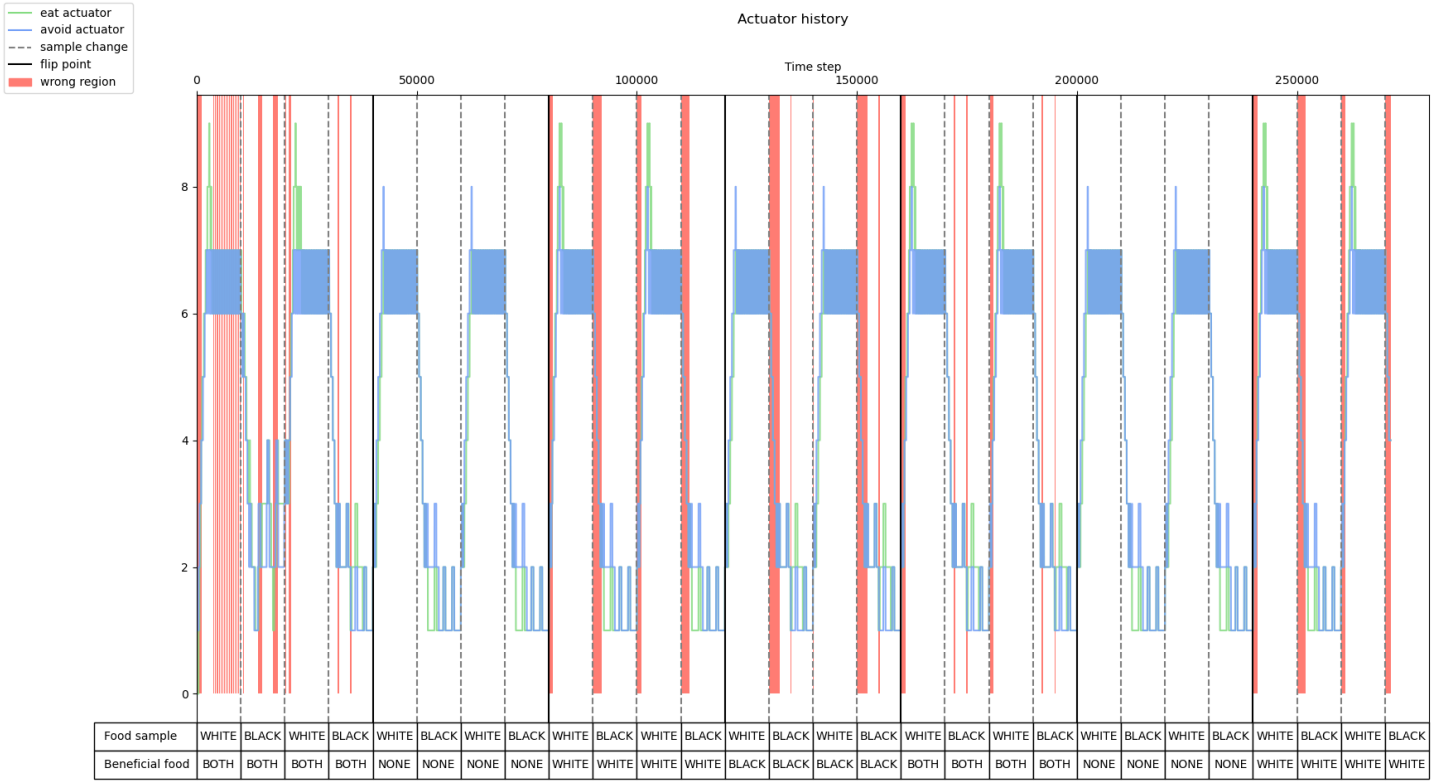


Figure 7.8: Visualization of the spike count of the agent's actuators during validation simulation 1 for Experiment 1. See Section 7.1.2 and Section 7.1.5 for a full explanation.

7.3 Experiment 2

This section presents the results from Experiment 2.

7.3.1 NEAT Monitoring

Figure 7.9, Figure 7.10 and Figure 7.11 illustrate fitness, accuracy and end-of-sample statistics for the population during the run of the EA. Figure 7.12 illustrates the distribution of species in the population.

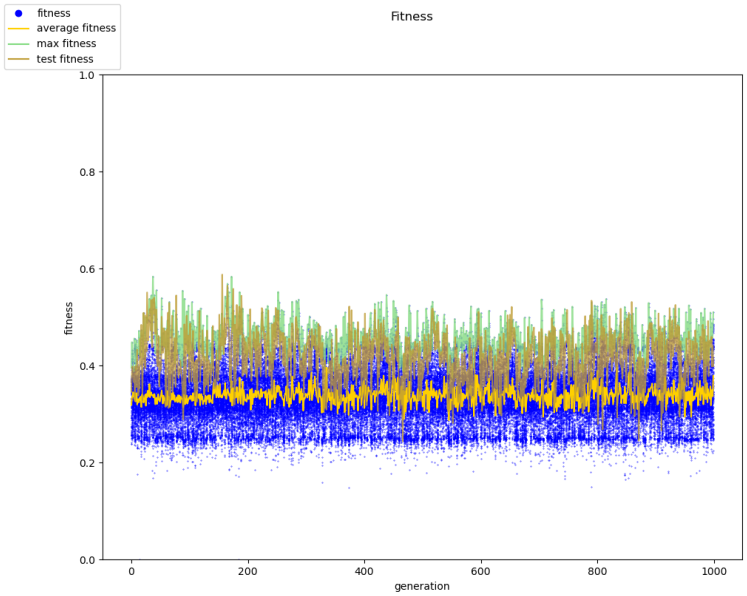


Figure 7.9: Visualization of the fitness statistics from NEAT in Experiment 2. See Section 7.1.1 for a full explanation.

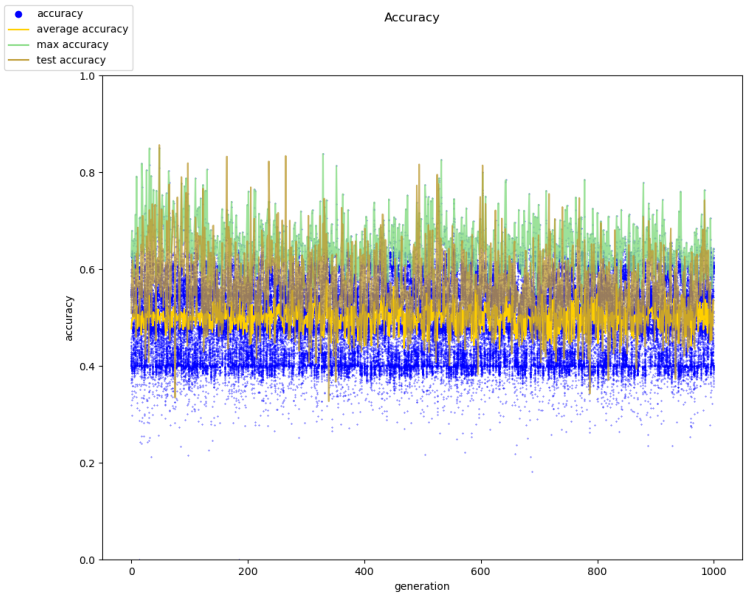


Figure 7.10: Visualization of the accuracy statistics from NEAT in Experiment 2. See Section 7.1.1 for a full explanation.

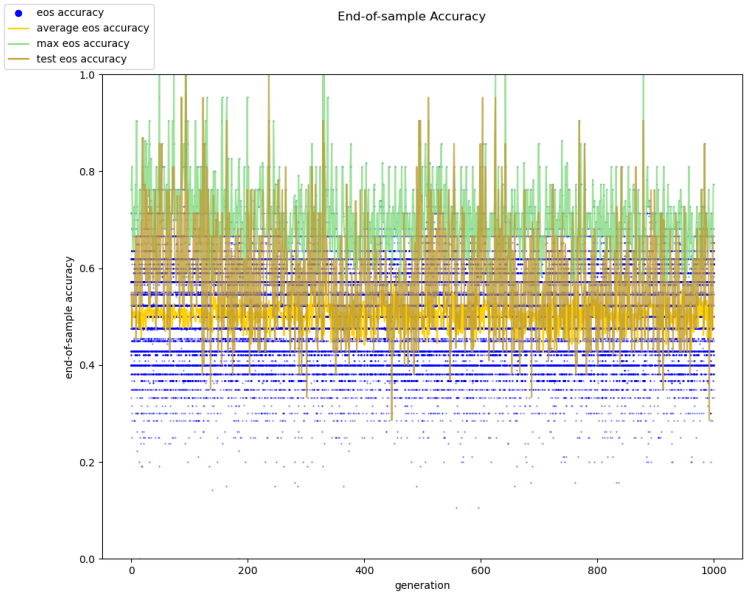


Figure 7.11: Visualization of the end-of-sample accuracy statistics from NEAT in Experiment 2. See Section 7.1.1 for a full explanation.

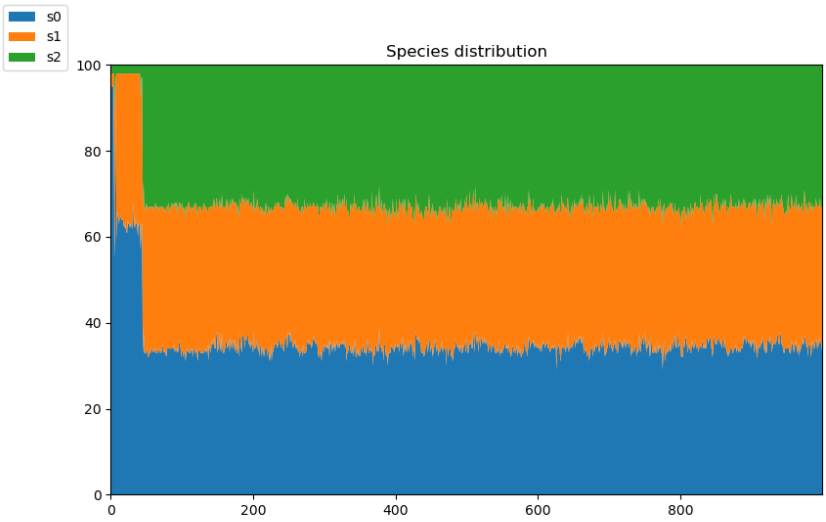


Figure 7.12: Visualization of the distribution of species in the NEAT population in Experiment 2.

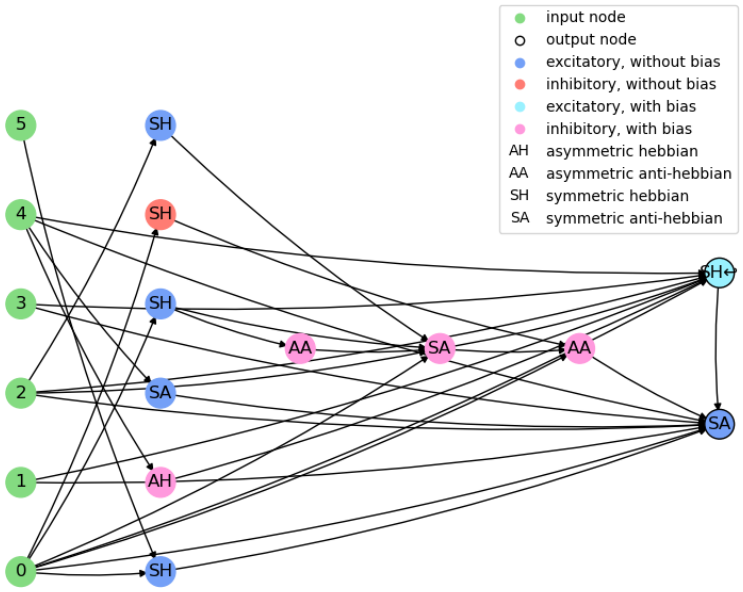
7.3.2 Simulation of High Performing Agent

After termination of the EA, a high performing agent was selected for validation simulation. For this experiment, we chose an agent that demonstrated a high end-of-sample accuracy in both the training environment and the test environment. The chosen agent is the agent from generation 86 with the highest end-of-sample accuracy, as can be seen in Figure 7.11. Five validation simulations were conducted on this agent (all with different randomly initialized weights), the results of which can be seen in Table 7.2.

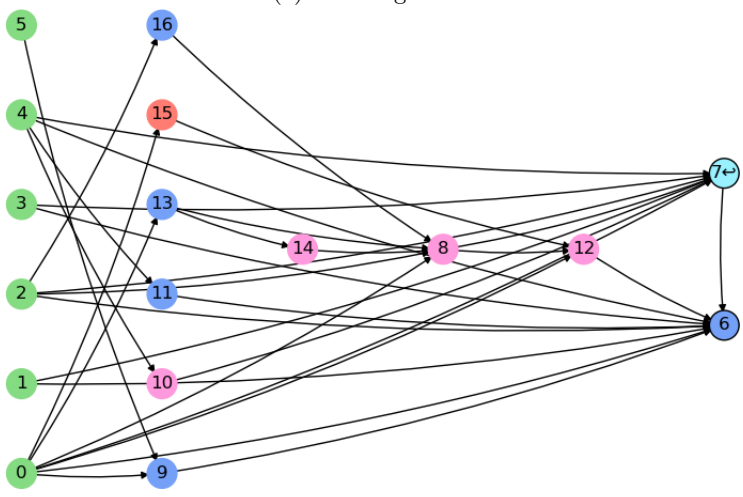
Experiment 2 Validation Simulations					
#	Fitness	Accuracy	Eos. Acc.	Input Order (A, B)	Environment Order
1	0.333	82.9%	95.2%	(1, 1), (0, 1), (0, 0), (1, 0)	NAND, AND, NOR, OR
2	0.333	80.5%	95.2%	(0, 0), (0, 1), (1, 1), (1, 0)	NOR, NAND, OR, AND
3	0.330	75.9%	90.5%	(1, 1), (0, 1), (1, 0), (0, 0)	NOR, NAND, AND, OR
4	0.333	86.2%	100%	(1, 1), (0, 1), (0, 0), (1, 0)	NAND, OR, AND, NOR
5	0.332	77.6%	90.5%	(1, 1), (0, 0), (0, 1), (1, 0)	OR, AND, NOR, NAND
Avg.	0.332	80.6%	94.3%	n/a	

Table 7.2: Validation simulations of the chosen agent from Experiment 2.

Figure 7.13 illustrates the topology of the chosen high-performing agent's SNN. Figure 7.14 illustrates the membrane activity of each neuron during simulation 1. Figure 7.15 illustrates the adjustment of each weight in the agent's SNN during simulation 1. Figure 7.16 illustrates the actuator history of the agent during simulation 1.



(a) Learning rules.



(b) Numeric identifiers.

Figure 7.13: Illustration of the network topology of the chosen agent. The one-hot encoded input sample 'A' goes into node 0 and 1, the one-hot encoded input sample 'B' goes into node 2 and 3, and the one-hot encoded reward/penalty signal goes into node 4 and 5. Node 6 is the output for the '0' actuator and node 7 is the output for the '1' actuator.

Figure 7.14: Visualization of the membrane potential of every neuron in the agent's SNN during validation simulation 1 for Experiment 2. See Section 7.1.2 and Section 7.1.3 for a full explanation.

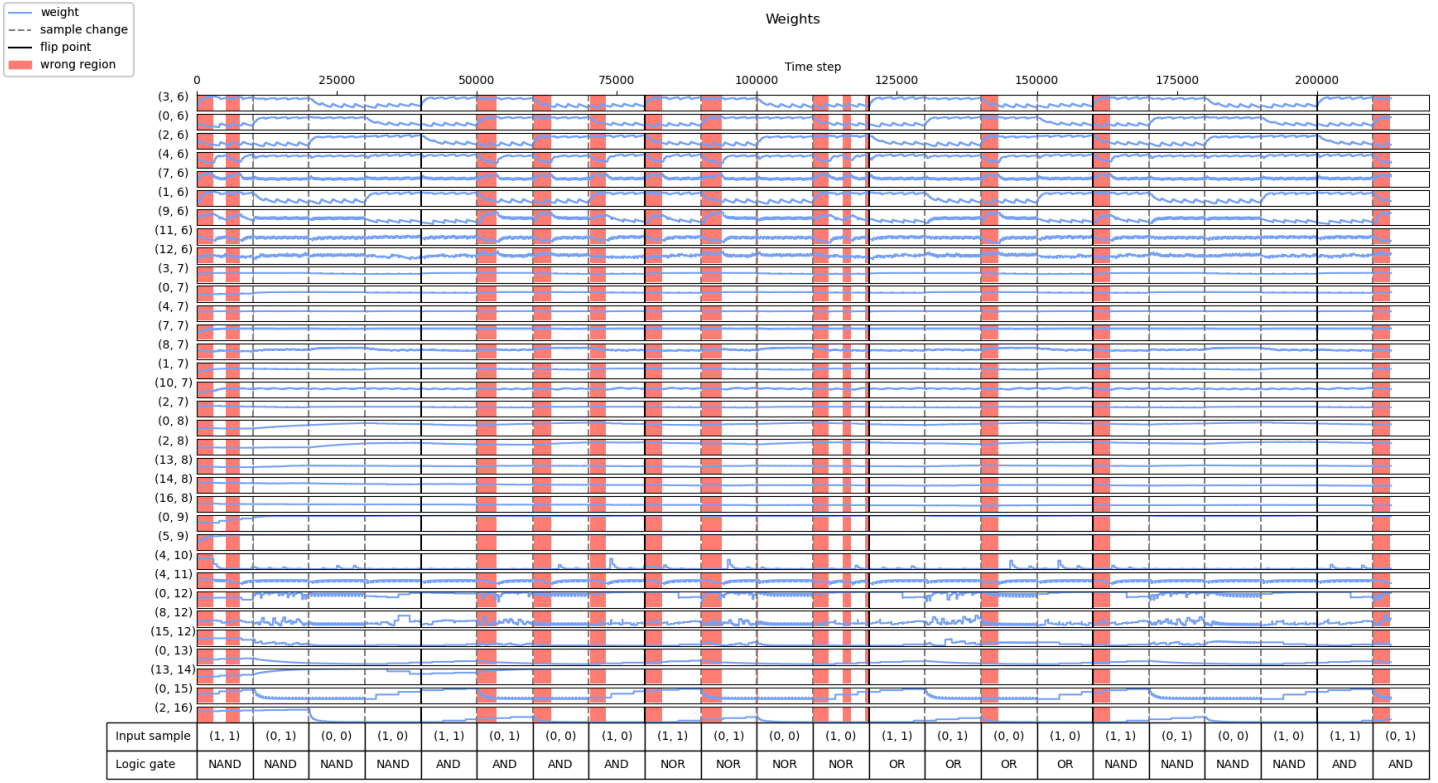


Figure 7.15: Visualization of the weight of every connection in the agent's SNN during validation simulation 1 for Experiment 2. See Section 7.1.2 and Section 7.1.4 for a full explanation.

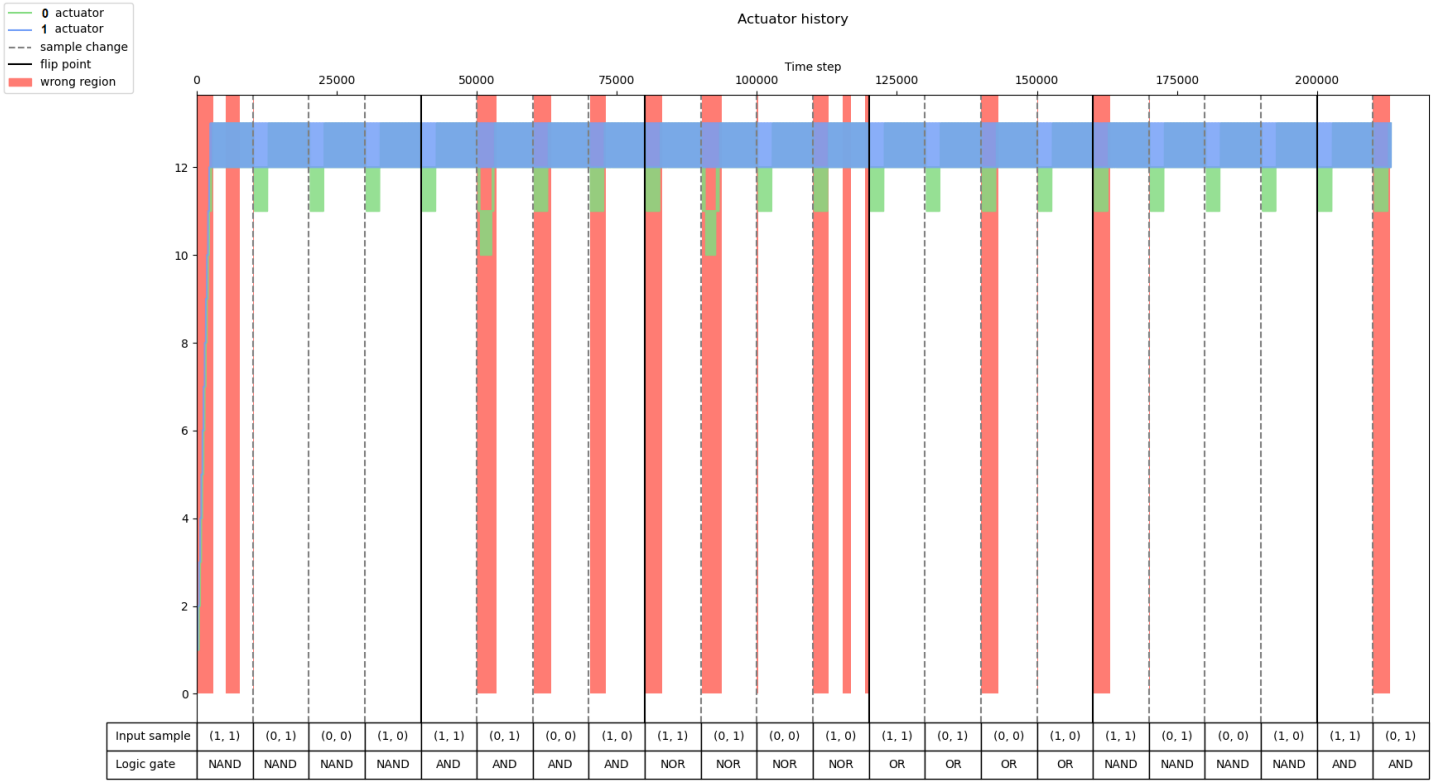


Figure 7.16: Visualization of the spike count of the agent's actuators during validation simulation 1 for Experiment 2. See Section 7.1.2 and Section 7.1.5 for a full explanation.

Chapter 8

Evaluation

In this chapter we evaluate the results presented in Chapter 7.

8.1 Non-increasing Fitness in the Population

In both experiments, the fitness distribution of the population stays roughly the same through all generations, as shown by Figure 7.1 and Figure 7.9. The expected and desired result would be an increase in the fitness distribution as generations go by.

A reason for the flat fitness curve could be that the fitness is measured by an agents lifetime, which in turn is increased by optimizing two properties: accuracy and confidence. Agents that displayed a high accuracy tended to employ a strategy of having actuators with a spike count close to each other (either tied or within 1 spike of each other), which can be observed in Figure 7.8 and Figure 7.16. This way, the actuators can very quickly overtake one another, allowing the agent to quickly switch to the correct action once a new input sample is encountered or the environment changes. Of course, this strategy leads to a low confidence, implying that these two properties are directly competing with each other. The end-of-sample accuracy, on the other hand, does not compete with confidence to the same degree as accuracy because it doesn't regard the state of the actuators until the end of an input sample and could be a better candidate as a metric for simultaneously optimizing the confidence.

8.2 Fluctuating Fitness and Accuracy in the Population

In Experiment 2, the maximum fitness, as well as the test fitness seems to fluctuate considerably (however, the test fitness seems to loosely correlate with the maximum training fitness). This is also true for accuracy and end-of-sample accuracy in both experiments, though the average distribution stays fairly consistent. This may be because of the different environments encountered in each generation, meaning that the environment in some generations is 'easier' to survive in because of the input and environment order of that simulation. Another reason for this could be that in our approach, the individuals that were kept in the population due to elitism were also subject to mutation. By keeping the top individuals in the population unchanged, it could lead to a more stable performance in the upper tier of the population.

8.3 Speciation of the Population

In both experiments, the species approximately equalize in size after some generations. In Experiment one there is a mass extinction event between generation 30 and 50, after which the remaining four species approximately equalize in size. In Experiment 2, there are only three species in the initial population with no additional species being developed. Experiment 2 concluded before Experiment 1, so we lowered the species distance threshold δ_{th} in order to increase the initial number of species in Experiment 1, as shown in Table 6.4.

8.4 Validation Simulations

The results of the validation simulation found in Table 7.1 and Table 7.2 show that the chosen agents for each experiment were able to consistently achieve a high accuracy and end-of-sample accuracy in different environments, even though their weights were initialized randomly each time. In Experiment 1, the agent was able to get an average of 89.2% accuracy and a 97.7% end-of-sample accuracy across five validation simulations. In Experiment 2, the agent was able to get an average of 80.6% accuracy and a 94.3% end-of-sample accuracy across five validation simulations in environments that were never encountered during training. Interestingly, these highly accurate agents achieve a relatively low fitness score which implies little to no correlation between our measure of fitness and their accuracy.

From looking at any of Figure 7.6, Figure 7.7, Figure 7.8, Figure 7.14, Figure 7.15 or Figure 7.16, we can see that the agents are able to change their behavior in response to sensory feedback. After a change in input sample or environment mutation, there usually occurs a red region where the agent is receiving a pain

signal, after which it changes action resulting in a white region and a pleasure signal for the remainder of that input sample.

8.5 Answering the Research Questions

In this section we relate the results to the research questions and discuss if the hypotheses were proven or disproven.

8.5.1 Research Question 1

Research Question 1 asks if it is feasible to evolve controllers that were able to keep learning in mutable environment in a weight agnostic manner. The results show that the chosen agents were indeed able to get a very high accuracy with randomly initialized weights. We can see from Figure 7.7 and Figure 7.15 that the weights in the network are adjusted in different ways at different points during simulation, depending on both the type of input and the sensory feedback. However, the flat fitness progression of the population means that it was not necessarily an evolutionary process that gave this result, but rather a random search that was able to find a suitable controller for achieving high accuracy. In any case, we argue that Hypothesis 1 was proven because the agents did not inherit any weights and displayed highly accurate behavior.

8.5.2 Research Question 2

Research Question 2 asks if the controllers display general, problem-independent learning capabilities by being able to perform in never before seen environments. The results from experiment 2 shows that an agent was able to achieve a high accuracy in the testing environment which it had not seen during training. Because of this, we argue that Hypothesis 2 was proven in terms of general, problem-independent learning capabilities.

8.5.3 Research Question 3

Research Question 3 asks if the agents were able to learn by themselves with only sensory feedback from their interaction with the environment. The results show that the agents were indeed quick to adjust their actions when receiving a penalty signal until they instead were receiving a reward signal. Therefore we argue that Hypothesis 3 was proven.

Chapter 9

Further Research

This chapter outlines some of the ideas and findings that appeared during the research process, but that were not implemented due to time constraints or other reasons.

9.1 Inverted Pendulum / Pole Cart Balancing

A third experiment was initially intended to be conducted in the thesis, but was scoped out due to time constraints: the Inverted Pendulum / Pole Cart Balancing problem, which is a classic benchmark problem in testing controllers. Gaier and Ha [20] were able to solve this problem in their approach, which is referenced in as a related work in Section 3.2. It is suited for floating point encoding for input, in contrast to the experiments conducted in this thesis which both used binary encoding. Figure 9.1 shows an example of such an environment with floating point encoded input. It would be valuable to measure if the controllers emerging from NAGI are able to handle this class of problems. To add a mutable property to this environment, one could for example flip the orientation of the angle and/or the angular velocity, the directions the cart moves in or the gravitational force.

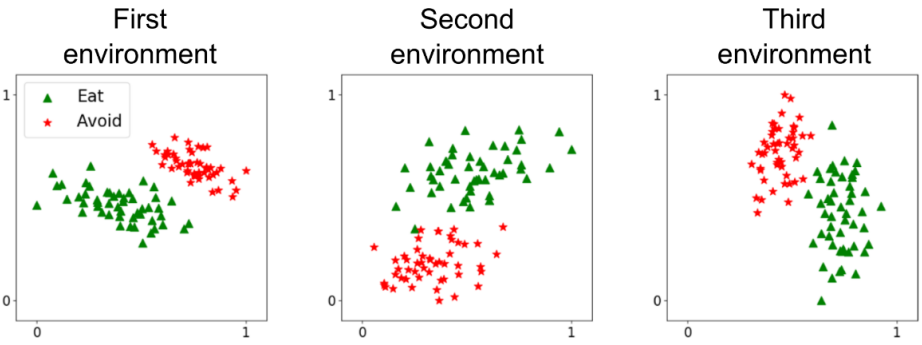


Figure 9.1: Illustration of a simple two-dimensional environment using floating point encoding. Image taken from [18].

9.2 Experimenting with fitness functions

One of the bigger challenges related to implementing the framework was the design of a proper way of measuring the fitness of a solution. As it stands, a solutions fitness is measured as the normalized lifetime of the solution in an agent-environment simulation, and the fitness function can be tweaked by changing the rules of how much damage is dealt to an agent given the state of the current input, the environment and the agent actuators.

Something that could be explored to keep networks from growing too big is to penalize a network by incorporating the number of hidden neurons into the damage calculation. A suggestion could look something like

$$D^* = 1.01^n D, \tag{9.1}$$

where D^* is the real damage, D is some arbitrary damage and n is the number of hidden neurons.

Another thing that can be explored is to combine the accuracy of a network into the fitness measure. We found that some networks can have a low fitness, but a good accuracy. Since a high accuracy is a desirable trait in solutions, there could be some merit to utilize this in fitness calculation.

9.3 Optimizing implementation

The implementation of the framework works reasonably well as a proof of concept. However, simulation of SNNs take much longer time to run than with classic ANNs. In addition to this fact, it's common knowledge that interpreted languages such as Python, without using optimization libraries such as *NumPy* is a lot slower at run time compared to a compiled low level language such as C++. This, combined with the reality that simulations of SNNs lasts a lot longer than simulations of regular ANNs, results in simulations taking a

very long time to run, and limits certain hyperparameters that affect run time such as population size and number of generations for the genetic algorithm. To illustrate this point, the 150th generation of a run of the algorithm with a population size of 200 took 46 minutes to complete with multiprocessing on a research server with an Intel® Xeon® W-2123 Processor with a clock speed of 3.60GHz, 4 cores and 8 threads, as well as 32GBs of RAM. It would certainly be interesting to see the results of an optimized implementation that can handle bigger networks, bigger populations and more generations in a reasonable time.

9.4 Distance metric

The distance metric used in the implementation is identical to the distance metric used by NEAT, which only takes into consideration excess and disjoint connection genes in the genome. However, in NEAT all neurons are equal. Neurons in NAGI differ by multiple properties. They can be either excitatory or inhibitory, they can have four different local learning rules with different parameters and they may either have a bias or not.

Let us first consider the same connection present in two models evolved from NEAT. This connection will behave very similarly in both models, only differing by the associated weight because the origin neuron and the destination neuron behave the same in both models and as such we can consider them a measure of likeness.

Now let us consider the same case for two models emerging from NAGI. The connection is present in both models, but what if the destination node in the first model is excitatory and the destination node in the second model is inhibitory? Now these connections serve very different purposes. In the first model, the connection is contributing to reinforcing the membrane potential in the neurons pointed to by the connections from destination neuron, but in the second model the connection is doing the opposite. The same reasoning can be applied for differing learning rules, as two neurons with different learning rules will adapt the values of incoming connection weights in a different manner, even though the connection is identical topologically.

There seems to be some merit to take into consideration these attributes of the neurons when calculating the distance between two genomes. There could also be some merit to exploring an approach to neuroevolution that doesn't utilize crossover at all, which would neutralize this challenge.

9.5 Izhikevich Neuron Model

The Izhikevich neuron model (discussed in Section 4.2.3) is more biologically plausible and produces richer firing patterns than the IF neuron model. It also

has multiple parameter configurations resulting in neurons with different firing patterns, which could be integrated into NAGI's neuroevolution by adding a gene containing the parameters to the genome. This would increase the dimensionality of the search landscape, which could be kept minimal by limiting the possible hyperparameter configurations to a predetermined set that produce neurons with distinct and unique firing patterns. Using the Izhikevich neuron model in NAGI could lead to more sophisticated neural networks.

9.6 Input encoding

In the work of this thesis, inputs were encoded into spike trains with a certain frequency, also called a firing rate. These spikes were uniformly distributed over the entire signal duration, leading to regular spike intervals. But observations in the human cortex show us that spike intervals occurring in the brain are irregular, meaning that a Poisson distribution of spike signals across a given time interval would be more biologically plausible [45]. This would ever so slightly affect how the agents make their decisions and how they learn.

9.7 No Overlap of Actuator Counting Between Input Samples

In the work of this thesis, the actuators counting the spikes occurring in the outputs of controllers within a time window overlapped between input samples, meaning that once a new input sample is presented to an agent, the actuators still count the spikes that are still within the time window from the last input sample. This means that the last input sample may affect how the agent reacts to the next input sample, at least in the beginning. Instead, one could explore an approach where there is no such overlap. Once a new sample is encountered by the agent, it "forgets" about the previous one and the sliding time window simply starts at the time step where the new sample is encountered. Figure 9.2 illustrates the difference in these two approaches.

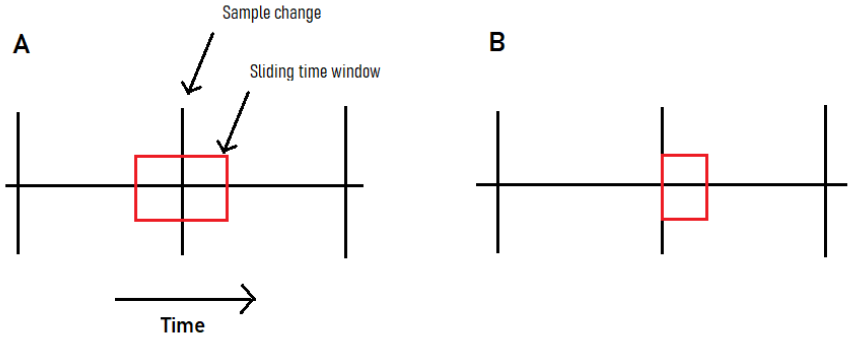


Figure 9.2: Illustration of the difference in sliding window behavior with and without overlap at the exact same time step (A with overlap, B without overlap).

9.8 Simulation of Multiple Environments Each Generation

In our approach, environments were initiated with a random order for both inputs and environment states. Since simulation is terminated once the agent dies, it means that an agent will experience certain combinations of inputs and environment states more than others in a given simulation. As a consequence, agents have a chance get 'lucky' in some generations. They may be put in an environment that starts out with an environmental state that they can easily solve and as a consequence get a boost in fitness for that generation. In order to reduce this fluctuation in fitness and to generalize even further, we suggest simulating each agent in multiple environments each generation. The fitness measure would be the mean of the fitness from the simulation of each of the environments. Using this approach, one could either use different randomly initialized weights for each environment, or use the same weight initialization for each environment.

9.9 Increasingly Complex Environments

In our approach, the complexity of the environments stayed the same throughout the generations of the EA. One approach that could be explored is to increase the complexity of the environment throughout evolution, for example by creating new possible environmental states and input types, or through some other measure like procedurally generated environments. This could facilitate the development of adaptation and complexity of the agents in the later generations.

Chapter 10

Conclusion

The main goal of this thesis was to explore how using EAs on SNNs could evolve controllers for agents that were capable of self-learning throughout their lifetime by interacting with mutable environments through simulation. A weight-agnostic neuroevolution technique based on NEAT, but modified for SNNs was used to evolve controllers. Local learning rules and STDP were the mechanisms used to adjust weights in order for learning to happen. All of these approaches came together in the the first implementation of the NAGI framework. Experiments were conducted in order to measure properties related to AGI, such as self-learning, adaptation and generalization.

The results from the experiments showed that agents emerging from the framework were able to consistently achieve a high accuracy of beneficial actions in validation simulations with constantly changing environments, even being able to generalize by achieving a high accuracy in new environments that were never encountered during training. The agents showed signs of self-adaptation through sensory experiences by changing their course of action when exposed to the emulated pain. It was found that optimizing the speed of decision making comes with a trade-off of lower confidence in the decision making, and designing a proper fitness measure that optimizes both accuracy and confidence at the same time proved challenging.

The results showed that it is possible to use SNN architectures and STDP for weight adjustment to create controllers with AGI at a very basic level, but also that considerable care must be taken when designing a neuroevolution technique in order to evolve them. The results suggest that more research should be spent on designing a neuroevolution technique that properly guides the evolution of controllers towards AGI, as well as researching agent-environment interactions with more complex data encoding.

Bibliography

[1] *Decision Boundaries*. https://www.cs.princeton.edu/courses/archive/fall08/cos436/Duda/PR_simp/bndrys.htm. Accessed: 2020-06-14.

[2] Pei Wang and Ben Goertzel. "Introduction: Aspects of Artificial General Intelligence". In: *AGI*. 2006.

[3] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. 2nd ed. Chapman and Hall/CRC, 2014. ISBN: 978-1-4665-8328-3.

[4] Tavanaei et al. "Deep Learning in Spiking Neural Networks". In: (2018). URL: https://arxiv.org/abs/1804.08150.

[5] E. M. Izhikevich. "Simple model of spiking neurons". In: *IEEE Transactions on Neural Networks* 14.6 (Sept. 2003), pp. 1569–1572. ISSN: 1045-9227. DOI: 10.1109/TNN.2003.820440.

[6] M.-M. Mesulam. "Acetylcholine Neurotransmission in CNS". In: *Encyclopedia of Neuroscience*. Ed. by Larry R. Squire. Oxford: Academic Press, 2009, pp. 1–4. ISBN: 978-0-08-045046-9. DOI: https://doi.org/10.1016/B978-008045046-9.00680-X. URL: http://www.sciencedirect.com/science/article/pii/B978008045046900680X.

[7] Yi Li et al. "Activity-Dependent Synaptic Plasticity of a Chalcogenide Electronic Synapse for Neuromorphic Systems". In: *Nature* (2014). URL: https://doi.org/10.1038/srep04906.

[8] Ian J. Goodfellow et al. "Generative Adversarial Nets". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 2672–2680. URL: http://dl.acm.org/citation.cfm?id=2969033.2969125.

[9] John R. Doyle. "Survey of Time Preference, Delay Discounting Models". In: *Judgment and Decision Making* 8 (Apr. 2012). DOI: 10.2139/ssrn.1685861.

[10] Linda Smith and Michael Gasser. "The Development of Embodied Cognition: Six Lessons from Babies". In: *Artificial Life* 11.1-2 (2005), pp. 13–29. DOI: 10.1162/1064546053278973. URL: https://doi.org/10.1162/1064546053278973.

[11] Anthony M Zador. "A critique of pure learning and what artificial neural networks can learn from animal brains". In: *Nature communications* 10.1 (2019), pp. 1–7.

[12] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. 2nd ed. Springer-Verlag Berlin Heidelberg, 2015. ISBN: 978-3-662-44874-8.

[13] Scott M. Thede. "An Introduction to Genetic Algorithms". In: *J. Comput. Sci. Coll.* 20.1 (Oct. 2004), pp. 115–123. ISSN: 1937-4771. URL: http://dl.acm.org/citation.cfm?id=1040231.1040247.

[14] Christian Blum and Andrea Roli. "Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison". In: *ACM Comput. Surv.* 35 (Jan. 2001), pp. 268–308. DOI: 10.1145/937503.937505.

[15] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. London: Murray, 1859.

[16] Xin Yao. "Evolving artificial neural networks". In: *Proceedings of the IEEE* 87.9 (Sept. 1999), pp. 1423–1447. ISSN: 0018-9219. DOI: 10.1109/5.784219.

[17] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks Through Augmenting Topologies". In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. URL: http://nn.cs.utexas.edu/?stanley:ec02.

[18] Sidney Pontes-Filho and Stefano Nichele. "Towards a framework for the evolution of artificial general intelligence". In: *CoRR* abs/1903.10410 (2019). arXiv: 1903.10410. URL: http://arxiv.org/abs/1903.10410.

[19] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. "Evolving Adaptive Neural Networks with and Without Adaptive Synapses". In: *Proceedings of the 2003 Congress on Evolutionary Computation*. Piscataway, NJ: IEEE, 2003. URL: http://nn.cs.utexas.edu/?stanley:cec03.

[20] Adam Gaier and David Ha. "Weight Agnostic Neural Networks". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 5364–5378. URL: http://papers.nips.cc/paper/8777-weight-agnostic-neural-networks.pdf.

[21] Larry Yaeger. "Computational Genetics, Physiology, Metabolism, Neural Systems, Learning, Vision, and Behavior or PolyWorld: Life in a New Context". In: (Mar. 1995).

[22] Briegel et al. "Projective Simulation for Artificial Intelligence". In: *Scientific Reports* (2012). URL: https://doi.org/10.1038/srep00400.

[23] Joseph Suarez et al. "Neural MMO: A Massively Multiagent Game Environment for Training and Evaluating Intelligent Agents". In: (Mar. 2019). URL: https://arxiv.org/abs/1903.00784.

[24] M. Nadji-Tehrani and A. Eslami. "A Brain-Inspired Framework for Evolutionary Artificial General Intelligence". In: *IEEE Transactions on Neural Networks and Learning Systems* (2020), pp. 1–15. DOI: 10.1109/TNNLS.2020.2965567. URL: www.feagi.org.

[25] Gal Chechik, Isaac Meilijson, and Eytan Ruppin. "Synaptic Pruning in Development: A Computational Account". In: *Neural Computation* 10.7 (1998), pp. 1759–1777. DOI: 10.1162/089976698300017124. eprint: https://doi.org/10.1162/089976698300017124. URL: https://doi.org/10.1162/089976698300017124.

[26] Peter U Diehl and Matthew Cook. "Unsupervised learning of digit recognition using spike-timing-dependent plasticity". In: *Frontiers in computational neuroscience* 9 (2015), p. 99.

[27] Nam Le, Anthony Brabazon, and Michael O'Neill. "Social Learning vs Self-teaching in a Multi-agent Neural Network System". In: *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*. Springer. 2020, pp. 354–368.

[28] Stephane Doncieux et al. "Evolutionary Robotics: What, Why, and Where to". In: *Frontiers in Robotics and AI* 2 (2015), p. 4. ISSN: 2296-9144. DOI: 10.3389/frobt.2015.00004. URL: https://www.frontiersin.org/article/10.3389/frobt.2015.00004.

[29] David Harris and Sarah Harris. *Digital design and computer architecture*. 2nd ed. Morgan Kaufmann, 2012. ISBN: 978-0-12-394424-5.

[30] A. L. HODGKIN and A. F. HUXLEY. "A quantitative description of membrane current and its application to conduction and excitation in nerve". eng. In: *The Journal of physiology* 117.4 (Aug. 1952). PMC1392413[pmcid], pp. 500–544. ISSN: 0022-3751. DOI: 10.1113/jphysiol.1952.sp004764. URL: https://doi.org/10.1113/jphysiol.1952.sp004764.

[31] Wulfram Gerstner et al. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, 2014. ISBN: 1107635195.

[32] Peter Diehl and Matthew Cook. "Unsupervised learning of digit recognition using spike-timing-dependent plasticity". In: *Frontiers in Computational Neuroscience* 9 (2015), p. 99. ISSN: 1662-5188. DOI: 10.3389/fncom.2015.00099. URL: https://www.frontiersin.org/article/10.3389/fncom.2015.00099.

[33] Simcha Lev-Yadun et al. "Plant coloration undermines herbivorous insect camouflage". In: *BioEssays* 26.10 (2004), pp. 1126–1130. DOI: 10.1002/bies.20112. URL: https://doi.org/10.1002/bies.20112.

[34] Hugh Bamford Cott. *Adaptive Coloration in Animals*. John Wiley & Sons Inc, 1940.

[35] Peter Forbes. *Dazzled and Deceived: Mimicry and Camouflage*. Yale University Press, 2011. ISBN: 978-0-300-17896-8.

[36] Pål Prestrud. "Adaptations by the Arctic Fox (Alopex lagopus) to the Polar Winte". In: *ARCTIC* 44.2 (Jan. 1991). DOI: 10.14430/arctic1529. URL: https://doi.org/10.14430/arctic1529.

[37] *SocratesNFR*. URL: https://github.com/SocratesNFR (visited on 06/15/2020).

[38] *Python*. URL: https://www.python.org (visited on 06/15/2020).

[39] *PyTorch*. URL: https://pytorch.org (visited on 06/15/2020).

[40] *TensorFlow*. URL: https://www.tensorflow.org (visited on 06/15/2020).

[41] Alan McIntyre et al. *neat-python*. https://github.com/CodeReclaimers/neat-python.

[42] Sean Welleck. *neat.py*. https://wellecks.wordpress.com/.

[43] Adam Gaier and David Ha. “Weight Agnostic Neural Networks”. In: (2019). https://weightagnostic.github.io. eprint: arXiv:1906.04358. URL: https://weightagnostic.github.io.

[44] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications*. CRC Press, 2005. ISBN: 9781584885054.

[45] David Heeger. “Poisson Model of Spike Generation”. In: (Oct. 2000). URL: https://www.researchgate.net/publication/2807507_Poisson_Model_of_Spike_Generation.