Sara Rambjør

Searching for models of Artificial Neurons using Cartesian Genetic Programming

Master's thesis in Datateknologi/Computer Science (MTDT)
Supervisor: Gunnar Tufte
June 2022

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Sara Rambjør

Searching for models of Artificial Neurons using Cartesian Genetic Programming

Master's thesis in Datateknologi/Computer Science (MTDT)
Supervisor: Gunnar Tufte
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Most contemporary connectionist approaches to AI use an Aritifical Neural Network (ANN) approach which is similar to Rosenblatt's Perceptron neuron model. This is true for contemporary ANNs tuned using backpropagation gradient descent, and for most Neuroevolutionary approaches. Although other artificial neuron models such as spiking neurons exist, they are often less computationally efficient. In neuron model design there has typically been a performance-biological accuracy trade off. This thesis does not attempt to solve this dichotomy, but rather to take a step to the side, and apply the biological principles of evolution to neuron model design, in the hopes that evolved models can be efficient due to being based on the computers actual computational primitives while also exhibiting more advanced behaviour than Rosenblatt-type neuron models. Such a development could be viewed as part of a wider trend in Aritficial Intelligence where leveraging computational search tends to outperform human design.

In this work neuron models are evolved by first defining an incompletely specified abstract model, and then using evolutionary search in the form of a variant of Cartesian Genetic Programming (CGP) to evolve complete specifications. The incomplete neuron specification defines what neurons can do rigidly, but not when, by dividing neuron behaviour into separate actions and using CGP to evolve the control programs for these actions. The thesis primarily focuses on the One-Pole Balancing control problem as a test problem.

The results show that the evolved neurons are able to form network structures which perform better than taking random actions in the one-pole balancing problem. Results for different iterations of the software is presented as a design case study, discussing primarily causes for observed program behavior and verifying these hypotheses through changes in the next iteration. Although the developed algorithm has several flaws, such as not maintaining population diversity within runs and some core design features not being useful in practice, the results still indicate that further research into the evolution of lifetime behaviour neurons would yield further results.

Sammendrag

Nåtidens konneksjonistiske Kunstig Intelligensmetoder pleier å bruke *Aritifial Neural Network* (ANN) metoder, med neuron-modeller som ligner på Rosenblatt's Perceptron. Dette gjelder både for ANNs trent med gradientnedstigning (gradient descent) og bakoverpropagasjon (backpropagation), og mange neuroevolusjonære metoder. Det finnes andre neuronmodeller som *spiking neurons*, men de trenger ofte mer datakraft for tilsvarende resultat (dersom tilsvarende resultat er mulig for neuronmodellen). I neuronmodelldesign har det tradisjonelt vært nødvendig å vektlegge utregningseffektivitet opp mot biologisk nøyaktighet. Denne masteroppgaven prøver ikke å løse dette vektleggingsproblemet, men heller å ta et skritt til siden og se på designproblemet fra en annen vinkel. Istedetfor å direkte ta inspirasjon fra biologiske neuroner, så undersøker denne oppgaven om man kan ta inspirasjon fra biologisk evolusjon og utvikle fungerende neuronmodeller. Slike utviklede neuronmodeller kan kanskje både være effektive, da de kan utvikles til å ta nytte av effektive operasjoner på datamaskiner, og ha mer avansert oppførsel enn Perceptron-lignende modeller. Dette kan ses på som en del av en større trend innen Kunstig Intelligens der datamaskinsøk ofte produserer bedre resultat enn direkte menneskelig design.

I denne oppgaven er neuronmodeller utviklet ved å først definere enn inkomplett spesifikasjon (en abstract modell), og å deretter bruke evolusjonært søk til å finne komplette spesifikasjoner ved å bruke *Cartesian Genetic Programming* (CGP). Dette gjøres ved å definere hvilke handlinger neuroner kan gjøre, og deretter bruke CGP til å utvikle kontrollprogram for å styre neuronene. Oppgaven undersøker hovedsakelig metoden ved å teste den på *One-Pole Balancing*-kontrollproblemet.

Det vises at utviklede neuroner kan forme nettverkstrukturer som implementer løsninger på One-Pole Balancing som er bedre enn å ta tilfeldige handlinger. Forskjellige iterasjoner av programvaren undersøkes, og presenteres for til dels å få fram viktige hensyn med design av neuronevolusjonære algoritmer, samt å bruke endringer til algoritmen til å teste og forstå den bedre. Selv om algoritmen har flere nedsider, for eksempel at den ikke beholder diversitet i befolkningen og at evolusjon viser at noen av designelementene ikke er nyttige, så viser resultatene likevel at videre forskning på evolusjon av livstidsoppførselsneuroner kan gi videre forskningsresultater.

Preface

Special thanks to my thesis supervisor, Gunnar Tufte, for feedback, guidance, and interesting discussions.

I would also like to thank the following authors. During my thesis work I've read several books on the topic of the emergence of intelligence, which helped awaken my interest in the topic, specify my project after the initial research phase in cooperation with Tufte, and provided me with a conceptual framework to work from. In particular I would like to highlight *Intelligence Emerging: Adaptivity and Search in Evolving Neural Systems* by Keith Downing Downing (2015), *Emergence: From Chaos to Order* by John H. Holland John H. Holland (1998), and *The Self-Assembling Brain: How Neural Networks Grow Smarter* by Peter Robin Hiesinger Hiesinger (2021). In particular, Downing and Hiesinger argue that a greater focus on emergent behavior produced by the interactions of local behavior may be necessary for advanced intelligent behavior, which is the basic assumption behind the focus on neuron models in this thesis.

Contents

Chapter 1

Introduction

Most contemporary Artificial Neural Network (ANN) approaches use neuron models which are similar to Perceptrons (Rosenblatt (1958)), making most ANNs Multiple-Layer Perceptron (MLP) models. Differences to the original Perceptron include varied activation functions, recurrence, batch normalization, different network structures such as LSTMs, training algorithms and much more (Goodfellow et al. (2016)), but the basic model for individual neurons is similar to Perceptrons despite differences. For convenience these similar approaches are all refereed to as MLPs. Figure 1.1 shows the neuron model referred to in this work when referring to MLPs and Perceptrons from this point on. MLPs are well suited to training using backpropagation gradient descent (Goodfellow et al. (2016)) which forms the basis for contemporary Deep Learning. Neuroevolutionary approaches use evolutionary algorithms to create ANNs, and here too the MLP model is common (ex. Stanley and Miikkulainen (2002), Jackobi (1995)). Other neuron models do exist and do see some use, for example, Elbrecht and Schuman (2020) details a neuroevolutionary approach based on a spiking neuron model. MLPs are dominant in research and engineering due to being less computationally expensive than other models (ex. Spiking Neuron models, see Downing (2015) for other examples), as well as being compatible with GPU calculation, allowing MLPs to achieve great performance on many problems given enough tunable parameters, data, and training time. However, MLPs do have several drawbacks, such as a lack of explainability (Xu et al. (2019))[1], weakness in multi-task learning and lifelong learning (Parisi et al. (2019) and Crawshaw (2020) - there are reasons to think that some very modern architectures may support multi-task learning, see Reed et al. (2022), but these still have the drawback of needing to be very large), as well as requiring large amounts of parameters, data, and training time.

The central problem of neuron model design is that more biologically plausible models are less computationally efficient, and that it is not clear how bio-

[1] Essentially, Deep Learning use backpropagation to tune a network, learning a series of functions which minimizes the problem error, however as these functions are just a large series of matrices it is difficult to understand why or how the functions work in isolation and together. Conceptually, Deep Neural Networks don't need to have a reason for why something works. It just needs to work.

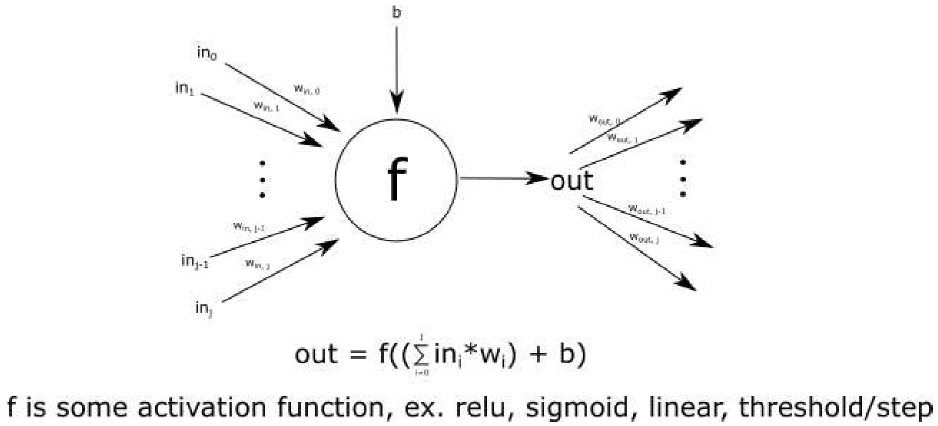


Figure 1.1: Depicts the neuron model referred to in this work as a perceptron. Arranged in some directed graph, may or may not be recurrent.

logically plausible neuron models "need to be" (Downing (2015)). Evolutionary algorithms may be beneficial in neuron model design. Evolved neurons may be more computationally efficient by permitting only efficient computational operations, in contrast with simulating real physical processes which may be computationally expensive. It may also be possible to find neuron models which are novel and unexpected and difficult for humans to design by hand, as the novelty factor is one of the advantages of evolutionary methods (Lehman et al. (2018), Miikkulainen (2021)). By searching for neuron models which can be more complex than MLPs, for example by having internal state and greater signal dimensionality, it may be possible to find models with desirable properties such as a greater ability to perform lifelong learning or multitask learning. Such models would be useful from an engineering perspective, and the methods may also provide insight into the emergence of cooperative intelligent behavior.

This work uses a method dubbed Neuron Model Search (NMS) which defines an abstract model of a neuron by partially defining how neurons work, and then using an evolutionary process to evolve complete specifications. The specific algorithm developed in this work is called NMS-LOC[2], and defines which actions a neuron can do, and then uses a variant of Genetic Programming (GP) called Cartesian Genetic Programming (CGP) to search for control programs to control the actions. The method makes no distinction between training time and runtime. As will be discussed in greater detail in the theory section there has been a limited amount of research into searching for neuron models, especially combined with lifelong learning, making it interesting to observe if such an approach can work. A crucial concept in NMS-LOC and neuroevolutionary approaches is the concept of network growth. In general, neurevolution works by gradually changing the neural network structures of the evolved solution into better and better neural

[2] For NMS-LOCal

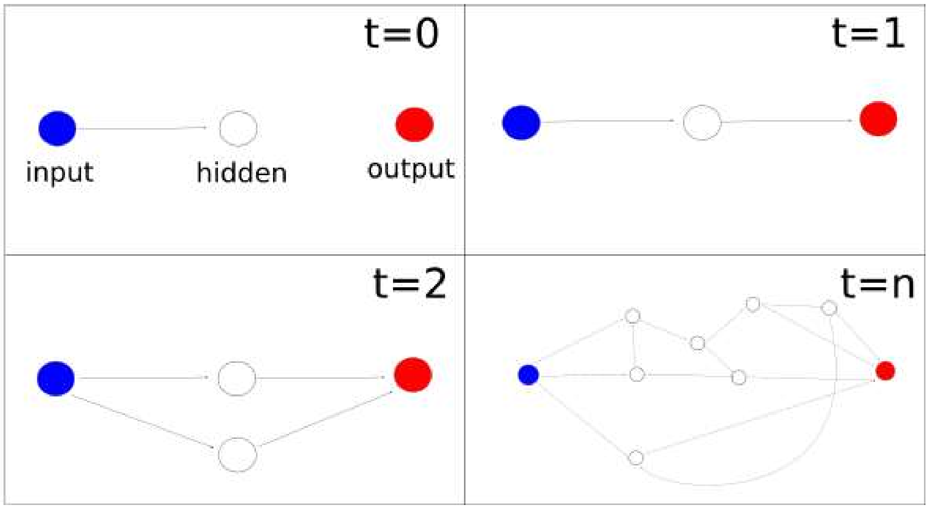


Figure 1.2: Illustrates the development of a neural network. (t=0): The initial network structure. (t=1): A connection is added to the output. (t=2): A new neuron is added to the network. It could for example be generated in response to the existing neuron not being able to handle a training sample (i.e. a target input-output mapping). (t=n) Through many iterations complicated network structures can develop. Usually the decoding process ends at some defined point.

networks. They do so by evaluating the fitness of the evolved networks against a problem domain and selecting the best found solutions to base further search of. Each solution consists of a genotype which is mutated and mapped to a phenotype (a neural network) using a mapping function. In general, this mapping function always uses the genotype data, but may be more complex and also use data from the environment, random seeds, and step-by-step growth or development of the phenotype using feedback from previous steps. Figure 1.2 shows an example of how a phenotype may develop step by step using a complex mapping function. The genotype size defines the search space size of the evolutionary search, while the mapping function defines the set of phenotypes which may be found during the search, and the evaluation of these phenotypes determines the fitness of different genotypes.

The combined runtime and training-time approach means that NMS-LOC uses training data not only to evaluate the fitness of an evolved solution, but also as input data during the development of the solution. Therefore, single NMS-LOC genotypes can map to several different solution phenotypes, and a single genotype could in principle produce phenotypes with good performance in several problem domains (see 2.2.2). Such developmental approaches are in general interesting for NMS as it may be possible to find genotypes which can produce good-performing phenotypes in problem domains it has not been trained on. In other words, it may be possible to use NMS approaches to search for learning algorithms, rather than specific solutions, which differs from deep learning and most neuroevolutionary

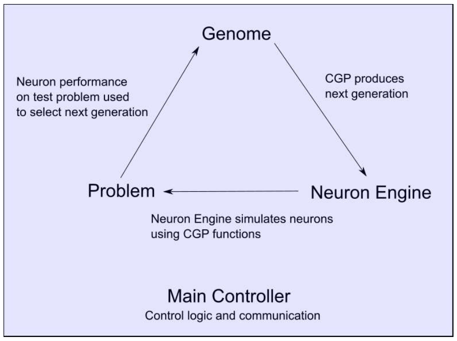

Genome
Neuron performance on test problem used to select next generation
CGP produces next generation
Problem
Neuron Engine
Neuron Engine simulates neurons using CGP functions
Main Controller
Control logic and communication

Figure 1.3: The overall logic of NMS. A CGP genome produces children, which are evaluated in a neuron simulation engine on a specific problem domain. By selecting children with a lower error as the next generation better models are evolved.

approach which typically attempt to optimize a single phenotype for a defined problem or set of problems. However, this work focuses on using NMS-LOC on the one-pole balancing problem specifically due to scope limitations.

Figure 1.3 shows the overall design of the NMS-LOC system, where a Genome is used to make a Phenotype (Neurons and Neural Network) which attempts to solve a problem, and a fitness function defined over the problem domain is used to continuously select better genotypes. The functionality of the system is discussed in more details in Chapters 3 and 4. Additionally, the code for the version of the NMS algorithm used in this paper is available at Github [3]. The thesis presents the design of NMS-LOC at a high level, while implementation details are presented by the codebase itself.

In order to explain how NMS-LOC solutions work, the produced phenotypes and statistical data is logged. Analyzing genotype logs was investigated, but due to some bugs and the quickly very large complexity of doing so it was determined to primarily use phenotypes and statistical data for analysis.

During research it became clear that it would be interesting to analyze the impact of using randomness in the NMS-LOC algorithm, and of using different CGP node functions (See Chapter 2 and 3). This thesis investigates the following research questions:

1. Can NMS-LOC produce better than random phenotypes for the one-pole balancing problem?
2. How does using randomness and different CGP node functions impact NMS-LOC performance on the one-pole balancing problem?
3. Which design lessons can be learned from NMS-LOC for potential future NMS systems?

[3] `https://github.com/SaraRambjoer/CGP_Neuron_Masters/tree/IRIS`

1.1 Note on preliminary work

During the autumn semester of 2021 work began on NMS-LOC in the form of a "preliminary thesis" ("Prosjektoppgave") (Rambjør (2021)) which is done at the Computer Science department at NTNU as a preparation for the master's thesis. It is common for the preliminary thesis to be about the same topic as the master's, this is also the case for this work. Therefore, please note the following: Firstly, the theory chapter (Chapter 2 and the method design chapter (Chapter 3) are based on equivalent sections in the preliminary thesis. However, for the master's thesis the sections have been updated to contain more references after additional literature search, more discussion and new sections, improved writing, and updates due to system changes. In general the master's thesis builds upon the work in the preliminary thesis, trying to extend and further the ideas and research done. The preliminary work concluded that the version of NMS-LOC developed at that point could find solutions in a simple problem domain by evolving non-environment responsive programs which output some determined sequence, and that NMS-LOC was only able to find solutions consisting of one hidden/evolved neuron networks (plus input and output neurons). The preliminary thesis is included in Appendix F.

1.2 Why Search?

This thesis uses evolutionary search to find neuron models. What grounds are there for (i) concluding that NMS can find useful neuron models (ii) that NMS may produce novel models of neurons?

Chapter 2 gives a more thorough presentation of evolutionary methods in AI, but it is clear that evolutionary methods are often suitable for solving a variety of problem domains (ex. see Stanley, Clune et al. (2019) and Del Ser et al. (2019)). The success of other evolutionary approaches signal that the hypothesises (i) and (ii) are at the very least worth investigating. Further, Julian F. Miller (2021) and Julian F. Miller et al. (2019) successfully evolve neurons using a similar approach.

Further, a central issue in the design of artificial neurons is weighing biological accuracy against computational performance, that is, selecting the appropriate level of abstraction (Downing (2015)). A level of abstraction that is too high may end up being insufficient for intelligence equivalent to that found in biological neural networks, and a level of abstraction that is too low may make computation too slow for any application with contemporary computers. Another issue is that human knowledge of biological neurons, brains and intelligence is incomplete, which means that implementing biologically inspired neuron models is to some degree educated guesswork. NMS reduces the need for educated guesswork, as it may be easier to define a search space that contains good neuron models than it is to define a good neuron model directly. Sutton (2019) argues that methods which leverage computational search have historically outperformed human design within AI. Due to the historical success of computational search it is reasonable to consider that computational search may outperform human design of neuron models as well.

Finally, consider that pursuing biological accuracy in neuron model design is not necessarily the optimal approach. Biological evolution has evolved neurons and neural networks which are capable of intelligent behaviour through efficient computation using a biological substrate; namely, the chemical interactions of carbon-based molecules as defined by the laws of physics. This biological computation substrate is inherently different than the one used in contemporary computers. It is likely that evolution has found ways to utilize the computational properties of the biological substrate as this would provide an evolutionary advantage. A biological neuron can utilize the results of complex molecular interactions to compute. Replicating these interactions in computers is complex and time intensive as molecular interactions must be simulated instead of being inherent in the computational substrate, vastly increasing the expense, and that is assuming the chemical process is currently understood. As such one may expect a perfect simulation of the brain which uses the same computational primitives as the universe (i.e. the laws of physics) to be very difficult to develop on a computer. In a similar vein, getting Windows to run on cells and organic molecules is far more difficult than using transistors. This is because the computational substrates are different, and the two "programs" are designed for their respective substrate. As such one may expect that computational processes which are capable of producing intelligent behavior on a contemporary computer and in a biological structure to be different, because the computational intelligences have to leverage different properties and primitives for the computation. Therefore, intelligence in computers should be based on the available computational primitives; namely, the CPU's instruction set, GPUs, and use of various parallelization techniques - or by using other primitives, i.e. a virtual machine, which entails a low computational overhead.

To emphasise this point, consider the success of Deep Learning in recent years. This was in large part made possible by an increase in computational capacity, such as cluster computing, and through utilizing GPUs for matrix computations. In other words, Deep Learning's success is possible because it uses an appropriate computational substrate for it's algorithm. However, single-processor CPUs can in principle run Deep Learning algorithms, but because of the different properties of the substrate it is not efficient and therefore not feasible. In other terms, just because a computational system is Turing complete does not mean that the computational properties of the system makes it suited to a certain type of computation. This is relevant both in terms of computational efficiency and ease of development. For example, the elementary cellular automata rule 110 is Turing complete (Cook (2004)), but would be horribly unsuited to practical development.

Luckily, search provides an alternative to biological modeling. An evolutionary search process will tend to select for efficient solutions, because these solutions will tend to be more fit as long as there is some pressure on computation time in the evolutionary process. Taking inspiration from biology when designing search spaces for NMS may be desirable, but doing so still provides computational search the opportunity to find solutions which would be difficult to design by hand. Because the evolutionary pressure imposed on efficient computation differs in the biological substrate and the contemporary computer substrate it is possible that evolutionary algorithms can find solutions which would be sub-optimal in the biological environment, but which are high-performing in the computer environment. However, evolutionary algorithms are themselves inspired by biology, and human knowledge of evolution is also incomplete, which entails similar challenges to designing evolutionary algorithms as to designing neurons directly. As such it is likely prudent to investigate both approaches, but currently the field has focused mostly on using human designed neuron models.

Chapter 2

Theoretical Background

This section gives a presentation of the core concepts of evolutionary computation and neuroevolution and provides examples of relevant approaches. Evolutionary algorithms work by searching over genotypes, mapping them onto phenotypes whose fitness are evaluated and used for selection. Core issues in evolutionary algorithms is the genotype representation, the genotype to phenotype mapping, the fitness function, and how selection and reproduction should work. New genotypes are produced using the current best genotype(s) using mutation operators which makes some type of change in the genotype, or crossover operators which combine the genotype of two or more parents (Goldberg and John Henry Holland (1988); Eiben and Smith (2015)).

2.1 Literature Search Methodology

The initial inspiration for the thesis is the work of Julian Francis Miller, in particular Julian F. Miller et al. (2019). As such, the article was used as a starting point for literature search. Reviewing the citations in the article was used to gather further references forming an initial start point for literature search.

Further, Downing (2015) was recommended to me by my supervisor Gunnar Tufte when discussing relevant literature, and articles mentioned in Downing further added to the literature search pool. In general keeping a look out for interesting articles mentioned in the similar work sections of read articles helped grow the pool of relevant literature.

Similarly, the preparatory class TDT04 - Advanced Bio-inspired Methods focused on many articles from different parts of bio-inspired AI, some of which were relevant to this thesis. Further, the class lecturer Pauline Haddow recommended the book Eiben and Smith (2015) to me when I requested an introductory book to bio-inspired AI, and the textbook was very useful to gain an overview of the field.

Finally, literature search was conducted using Google Scholar, searching for relevant keywords, such as "CGP", "Neuroevolution", "Evolving neurons", "CGP artificial neural networks" and more.

Articles were examined by first reading over the abstract and introduction, and potentially skimming other parts, to determine if the articles held relevant information. If so, the articles were read thoroughly. This way it was possible to both consider an larger amount of articles, and read the relevant ones properly. Citations in read articles were used to find new references, a sort of "branches of trees" approach.

2.2 Introduction to Neuroevolution

Neuroevolution is a subfield of evolutionary algorithms focused on producing artificial neural networks. For a comprehensive overview see Downing (2015) or Floreano et al. (2008). For a review of the state of the art of neuroevolution see Stanley, Clune et al. (2019). Early work in neuroevolution often focused on evolving weight parameters in a fixed topology, such as in Whitley et al. (1993). Later, work began on Topology and Weight Evolved Artificial Neural Networks (TWEANNs) in which both weights and topology is evolved. Earlier systems typically used direct genotype representations, where the topology and weights are directly encoded in the genotype (such as in Stanley and Miikkulainen (2002)), while later works moved on to using indirect genotype representation where a sophisticated mapping function describes how a phenotype can be produced from the genotype. The most well-known direct encoding approach may be NEAT (Stanley and Miikkulainen (2002)), and it is perhaps an endorsement of indirect approaches that NEAT too moved on to indirect encodings in the form of HyperNEAT (Stanley, D'Ambrosio et al. (2009)). NMS-LOC can be viewed as using a form of indirect encoding, specifically a developmental approach. See section 2.2.2 for a more in depth discussion of direct, indirect and developmental encodings.

Indirect genotypes allow for smaller genotypes relative to network size which reduces the search space at the cost of introducing more human design in the mapping function. A smaller genotype is advantageous as it allows for evolving large networks without having equally large genomes. Grammar-based approaches are an example of indirect encodings, which work by applying rules in a formal grammar to produce a neural network (Cangelosi et al. (1994)). Some indirect encoding approaches use artificial chemical systems, such as Genomic Regulatory Networks (GRNs) (Jackobi (1995); Eggenberger (1997)). In GRNs chemical concentrations in and around cells define which genes should be active, which then defines which chemicals should be produced. The chemicals also define cell behavior, such as how cells should migrate and connect, and how strong connection weights should be in the neural network phenotype produced by mapping from the chemical neuron simulation. Among the mentioned approaches GRNs are most similar to NMS-LOC, but there are two crucial distinctions. First, a distinction between training time and runtime is common in neuroevolution, where neuroevolutionary methods typically use a specific algorithm to produce an ANN, which then acts as a normal MLP. However, in biological neural networks there is not such a strict distinction between learning time and runtime (ex. neurogenesis in adults Zhao et al. (2008), or the simple facts that children can exhibit intelligent behaviour while their brains are developing, and that adults can still learn), therefore moving away from this separation could produce advances in lifetime learning and transfer learning. Secondarily, the aforementioned approaches only use training data to evaluate the fitness of genotypes. Biological neural structures grow by reacting and interacting with their surroundings, not through a one-to-one mapping from the genotype. For both reasons NMS has no separation between training/development time and runtime, using the same neuron model for both.

As mentioned, Neural Evolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen (2002)) is one of the most well known TWEANN algorithms. NEAT outperformed contemporary systems at its release and solved the competing conventions problem which made crossover in TWEANNs difficult because different networks could be interpreting the output of subnetworks differently. Among other improvements NEAT introduced historical markers or tags in the genome to denote which evolutionary changes produced which sections. These tags where also used to divide the population into several species in order to maintain diversity, which is a common design issue in evolutionary algorithms conceptually similar to the problem of exploration versus exploitation in reinforcement learning (Eiben and Smith (2015)). NEAT was further developed in HyperNEAT where NEAT is used to evolve a computational network which given neuron coordinates outputs the weights between the neurons (Stanley, D'Ambrosio et al. (2009)) which made it possible to grow large networks displaying modularization and regularity. Like other neuroevolutionary approaches NEAT and HyperNEAT makes a distinction between Artificial Embryogeny (Stanley and Miikkulainen (2003)) (otherwise known as neurogenesis or "the developmental phase") and runtime behavior. NEAT also advocated beginning the evolutionary search from a minimally simple structure, which is employed in NMS-LOC by defining the initial network structure to be input neurons, output neurons and a single unconnected neuron controlled by the evolved functions. NEAT and HyperNEAT are relevant as alternative methods to NMS within the MLP-paragdim, as well as having speciation which is not used in NMS-LOC.

Figure 2.1 illustrates how the phenotypes produced by NEAT can look over several iterations. The figure is not meant to capture technical details of NEAT, but just to highlight how the algorithm works conceptually by gradually making the phenotypes more complex. Notice how similar this graph is to the general case illustration of phenotype development in Figure 1.2. The difference is just that NEAT changes the phenotype based on mutation and crossover-based search from each phenotype iteration, while a developmental process executes a developmental program which may not involve fitness evaluation and often does not use evolutionary search, although it could in principle. One of NEATs innovations was an emphasis on starting search from a minimal structure, and has since become a common feature in Neuroevolution.

Some hybrid approaches attempt to combine backpropagation gradient descent with evolutionary search. For example, Suganuma et al. (2002) searches for

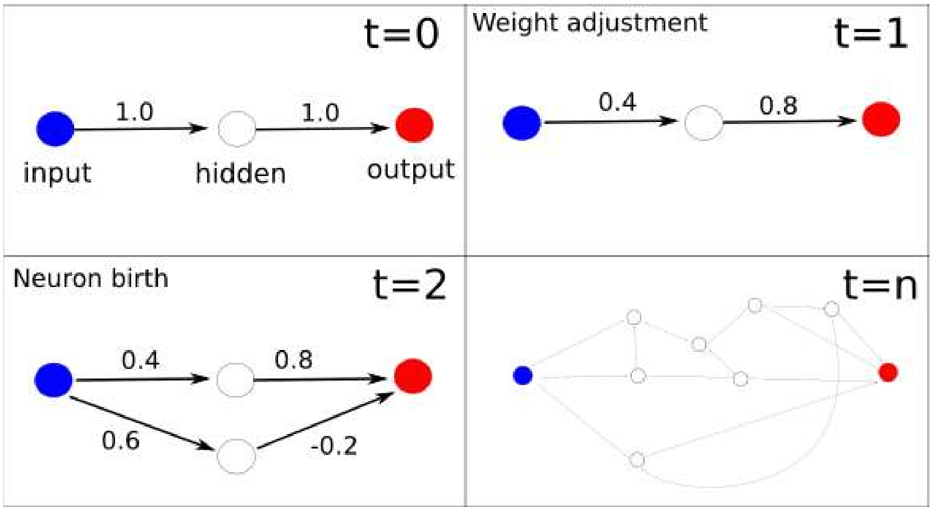


Figure 2.1: Illustration of the evolution of ANN using NEAT. Illustration meant to convey concept of evolving neurons and weights, not technical details of the algorithm.

CNN architectures using CGP, and then uses backpropagation to tune the network weights and evaluate the fitness of the solution. For other hybrid approaches see Stanley, Clune et al. (2019). Approaches like Suganuma et al. (2002) can also be viewed as Baldwinian evolution approaches (Eiben & Smith, 2015), where mutation operators are applied during genotype decoding which does not have any impact on the genotype. The alternative to such approaches are Lamarckian evolution approaches where genome processing operators alters the genome. NMS-LOC is a strictly Baldwininan approach, without explicit changes of genomes during runtime nor any epigenetic processes.

2.2.1 Developmental Approaches

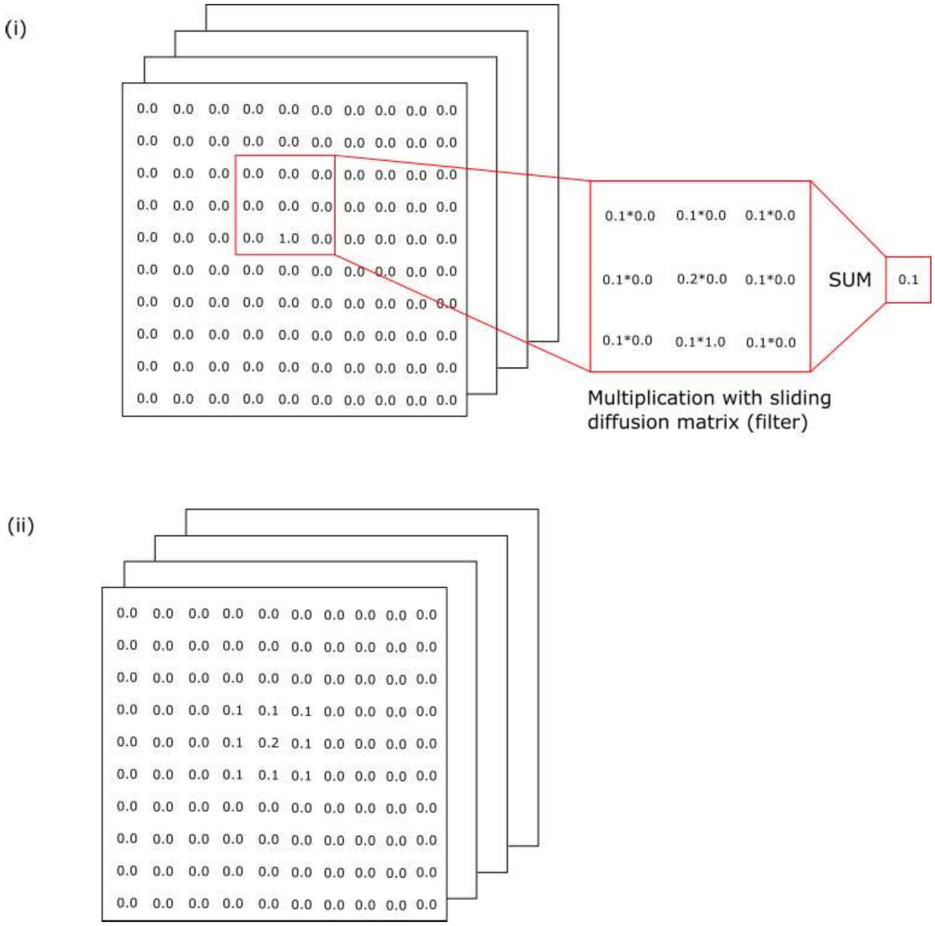
Developmental approaches can be thought of as a subset of indirect encodings. Specifically, developmental encodings are encodings which require the execution of a computational process as encoded by the genotype to produce a phenotype. The difference from indirect encodings is that it is not clear which phenotype a genotype will map to before completing the computational process, nor exactly which differences a change will cause. The intent is that developmental encodings can allow for heavily compressed genotypes, because the produced phenotype is produced by an extensive computational process, which also means that a small change in the genotype may cause large changes to the phenotype due to it's compound effects. See 2.2.2 for a longer discussion of indirect and direct encodings.

NMS-LOC has a lot in common with other developmental approaches which produce a phenotype through interaction with training data. One difference is that developmental approaches sometimes map the developed structure onto a MLP (ex. Julian F. Miller et al. (2019)). However, this is not always the case, and NMS has more in common with developmental approaches which do not map the solution to an MLP. For example, Astor and Adami (2000) outline an approach based on GRNs and chemical gradients where input neurons emit chemicals depending on their state, eventually producing a multi-cell phenotype from a GRN genotype. Despite the similarities to Astor & Adami's work, NMS-LOC attempts to move away from the use of chemical gradients as to not need to simulate chemicals to make computation more efficient, and to have a greater focus on CGP-program centric models, in the hopes that this would increase interpretability and explainability, and to rely more on "computational primitives" as argued in section 1.2. Instead, NMS-LOC focuses on the concept of neurons sending signals. However, an advantage of chemical gradient based approaches could be that the chemical gradients provide a type of signaling mechanism that may be easier to evolve than signal sending. In principle a neuron program can evolve which sends signals around performing an equivalent function to chemical gradients, but if gradient simulation is necessary an evolutionary overhead is introduced which may make the model less efficient than just simulating chemical gradients. In particular, due to the the availability of high performance GPUs, chemical gradient based models may be more feasible as chemical diffusion could be simulated using GPUs through the use of something similar to convolutional filters (see Figure 2.2). Therefore, the use of chemical gradients should not be dismissed in future research, even though this paper investigates an alternative approach.

However, even using GPUs the computational cost of chemical diffusion simulation can be expensive. If simulating neurons in a 3D-grid consisting of one hundred by one hundred by one hundred positions, then each timestep would involve updating one million position parameters per chemical.

Several other authors have used CGP to evolve neural networks. The Cartesian Genetic Programming Artificial Neural Network (CGPANN) approach extends CGP-graphs to include weights on directed links, and then uses standard CGP-techniques to evolve networks (See Turner and Julian F. Miller (2013); M. M. Khan, Gul M. Khan et al. (2010); M. M. Khan, Ahmad et al. (2013); Gul Muhammad Khan (2018); N. Khan and Gul Muhammad Khan (2021)). Figure 2.3 illustrates a CGPANN graph, showing how CGPANN graphs are just normal CGP graphs with the addition of weights which may be tuned using gradient descent. CGP is discussed in more detail later, but as a contrast consider Figure 3.1 which illustrates a CGP program which implements a one-bit fulladder. The disadvantage of using CGPANNs is that tuning may be computationally expensive, and that it may be more more difficult to interpret the graphs. The most relevant approach to NMS-LOC is research into CGP Developmental Networks (CGPDN), which use CGP to evolve functions for use in a developmental process.

Julian F. Miller et al. (2019) presents a CGPDN consisting of two CGP programs: One simulating a neuron soma, and one simulating a dendrite. Using internal state variables and hyperparameter defined increments and action thresholds a one-dimensional network is grown. The development is done by running their



(i)

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

0.1*0.0	0.1*0.0	0.1*0.0
0.1*0.0	0.2*0.0	0.1*0.0
0.1*0.0	0.1*1.0	0.1*0.0

(ii)

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.1	0.1	0.1	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.1	0.2	0.1	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.1	0.1	0.1	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 2.2: Illustrates chemical diffusion using matrix computation. Each chemical layer in a 3D matrix has a magnitude preserving convolutional filter simulating chemical diffusion. (i) The state of one chemical layer. (ii) The same chemical layer after one diffusion. The convolution would have to be zero-padded to maintain dimensionality and should likely either maintain the magnitude of chemicals in the system, or reduce it on diffusion.

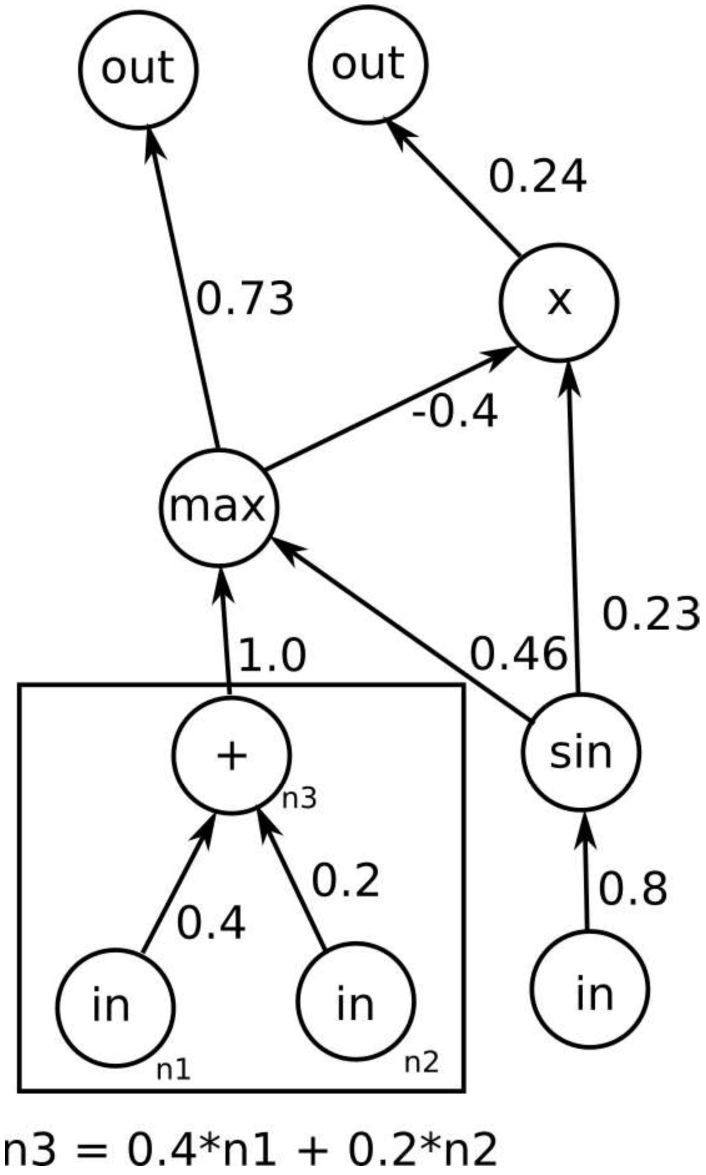


Figure 2.3: An illustrated example of a CGPANN graph.

soma and dendrite programs a given number of times, or until fitness decreases in extracted ANNs. This makes Julian F. Miller et al. (2019) approach an indirect encoding approach which is sensitive to the genotype and fitness landscape of the problem, but not the specific problem instances encountered. Similarly to other neuroevolutionary methods it also makes a distinction between the development phase of the network and the runtime network.

There are several key differences between Julian F. Miller et al. (2019) approach and NMS-LOC. NMS-LOC is three-dimensional, incorporates the problem domain in the developmental phase by making the neurons interact with problem domain input and output, does not practice early stopping on fitness decreases and does not make a distinction between development and runtime, that is, it does not extract an ANN from the developed network. Julian F. Miller et al. (2019) noted that networks generated by their approach were often small, which may be because it uses an assumption of dendrite lengths being half of the neuron position which may make it difficult to grow modular components and grow the network as the one-dimensional half-distance from origo assumption may not be able to take advantage of spatial data. Moving to two or three-dimensional spaces may make it possible to use better geometric heuristics.

Further, Julian F. Miller et al. (2019) relies on a set of pre-defined value types, such as neuron health, which ties into the neuron control behavior in an explicit fashion. The neuron model used in NMS-LOC instead uses a variable amount of variables with no specific semantic meaning, functionally similarly to registers. This was done to investigate if evolving CGP neuron programs would still be viable in a more loosely defined model. Another approach which is similar to Miller et. al.'s is Gul Muhammad Khan (2018), which also uses CGP to evolve neuron control programs. Khan uses semantically defined state variables such as health and resistance. The advantage of the approach taken in Khan and in Miller et. al. is that it allows for freezing the developed network. That is, it is possible to extract a static network, either by mapping the structure to an ANN, or freezing system-changing variables like Health. The advantage is that this approach makes it possible to extract stable networks which can run quickly in contemporary AI frameworks, which is not the case for NMS-LOC. Miller recently published further work focusing on their developmental neuron model applied to a two-dimensional space (Julian F. Miller (2021)). Julian F. Miller (2021) also argues that the prevalence of storing information as synaptic weights is not necessarily accurate and is not necessarily the best approach. However, the use of alternatives neuron models for actual computation in connectionist networks is not investigated in Millers work. The similarities between Miller's work and NMS-LOC is as such primarily conceptual, and both approaches could be considered NMS systems.

Figure 2.4 shows the algorithm used in Julian F. Miller et al. (2019) at a conceptual level. A phenotype is developed over several iterations by running evolved neuron soma and dendrite programs. Using the neurons and the dendrites internal state variables MLP ANNs are extracted, and each MLP ANN solves a different problem (classification problems in Julian F. Miller et al. (2019)). Notice how al-

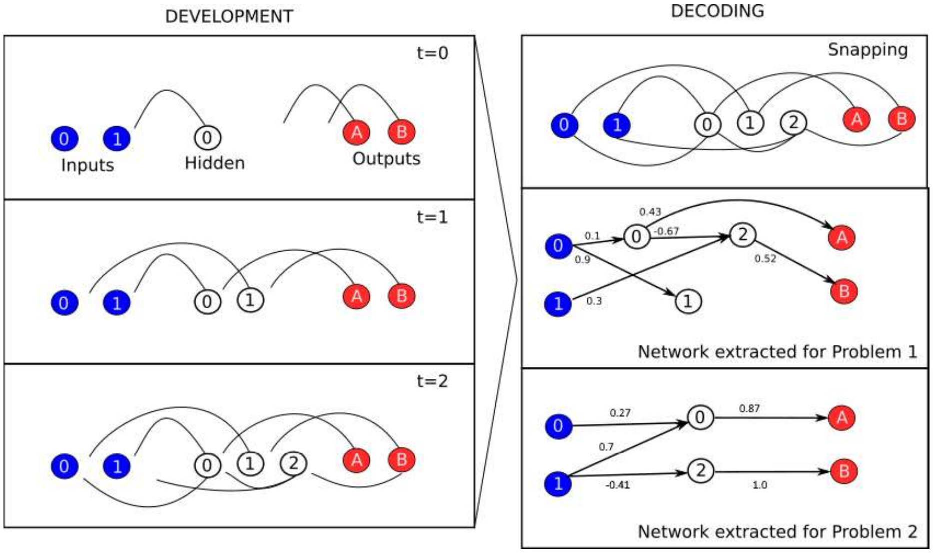


Figure 2.4: An illustrated example of the algorithm used in Julian F. Miller et al. (2019) at a conceptual level. First, a phenotype is developed, then several ANNs which solve different problems can be extracted from that phenotype. Figure is adapted from Julian F. Miller et al. (2019).

though the specifics of this algorithm differ greatly from NEAT it is clear that they both are based on the same conceptual ideas of evolution and producing phenotypes, and both approaches begin by starting at a minimal phenotype.

Another interesting area of existing research is the use of CGP in artificial life simulations. One can view the problem of evolving neurons as trying to evolve a information-processing creature, making these research areas conceptually similar. Rothermich and J. Miller (2003) uses CGP to evolve cells capable of moving towards energy sources in a simulated environment, and analyzes the strategies used in the evolved phenotypes. In Rothermich, Wang et al. (2003) the same methodology is used to allocate a parallel "cell-based" system for allocating computational resources between databases. One interesting aspect of the method in Rothermich, Wang et al. (2003) is that it allows for CGP node functions to perform actions on the cell level directly, instead of interpreting CGP output as control signals. It is outside the scope of the work to compare these approaches more in detail, but it is possible that the action-node-function approach could have advantages such as making it easier to evolve if-else type control structures in the node programs.

An important sub-field of artificial life is the study of using Cellular Automata (CA) to construct patterns through the application of local rules on "cellular"/local organisms/programs. Öztürkeri and Johnson (2011) gives a survey of historical research into this subfield, and presents a solution which uses CGP to evolve the local programs. Öztürkeri and Johnson (2011) generates patterns using a two-dimensional cellular automata, meaning that the geometry/geography of the simulated world is divided into a two-dimensional grid. Each cell can access it's own n-bit state along with the n-bit state of it's neighbours. The evolved CGP function then outputs the new internal state of the cell and the new internal state of it's neighbours. To avoid conflicts this is done in a deterministic order where cells only updates neighbouring cells which comes after itself in the order. Unlike many other approaches for evolving patterns, this approach does not use chemical diffusion simulation. The authors claim that the state overwriting mechanism is an sufficient signaling mechanism to permit organizing the pattern through the application of local behavioural rules. Similarly, NMS-LOC uses a signaling mechanism instead of chemical diffusion. The state updating mechanisms in NMS-LOC and Öztürkeri and Johnson (2011) still have several differences, such as NMS-LOC lack of an explicitly pre-determined execution order and permitting recurrence, but both approaches still attempt to find computational signaling mechanisms that do not depend on chemical diffusion.

In conclusion; NMS-LOC (and NMS) is a neuroevolution method inspired by several other neuroevolutionary algorithms but attempts to be more general. In a sense most neuroevolutionary approaches can be viewed as algorithms for finding a specific MLP for a specific problem, while NMS-LOC instead seeks for a controller for a neuron "robot", which interacts with other neuron robots to solve one or more problems requiring learning (or equivalently, NMS-LOC constitutes an agent-based approach to neural network intelligence (Laubenbacher et al. (2013))). By not extracting MLPs from the developed networks, by not separating the development and runtime phase and by not using a one-to-one genotype-phenotype mapping and by searching for "neuron robot controllers" (i.e., neuron models) NMS-LOC may be able to find novel solutions. This comes at the cost of potentially reduced computational efficiency, and the potential of neuron models being unstable and degenerating which cannot occur with standard ANNs extracted from developmental networks.

2.2.2 Conceptual difference between developmental systems and other evolutionary approaches

In direct encodings the genome can be directly mapped to the resulting phenotype. In effect the genotype space can be a complete one-to-one mapping from the genotype domain to the phenotype domain. Evolutionary search therefore directly navigates the search space of the phenotype applied to the problem domain. Figure 2.5 illustrates how direct encoding mapping schemes work. In the figure a genotype of a given size maps to a set of phenotypes which may be as large or smaller to the set of possible genotypes. The size of the search space is limited by the size of the genome, and will be unable to create neural networks of a given size.

Indirect encodings are by definition compressed encodings which require a decoding process before being mapped to phenotype space. It is possible that the

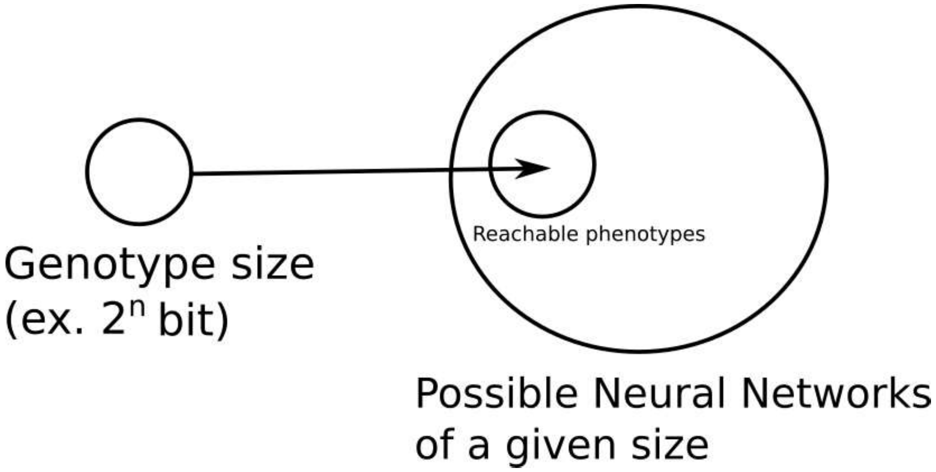


Figure 2.5: Illustrating a deterministic mapping from genotype to phenotype space.

indirect encoding forms a complete map, that is, every phenotype (of a given size) can be produced by the possible genotypes, but this is not an requirement. The indirect encoding can in principle also be a many-to-one encoding from genotype space to phenotype space, but does not need to be. As mentioned indirect encodings are used because the compressed representation allows for searching for larger phenotypes. In effect, the compression/decompression mapping applies a transformation to the target search space. This is beneficial under at least two circumstances: One, if the used evolutionary algorithm is better suited to navigating the transformed search space. Two, the transformation biases the search space, either by dropping some poor solutions from the search space entirely, or by making the evolution of high-performing phenotypes statistically more likely, for example by increasing the percentage of genotypes which map to such phenotypes. Figure 2.5 also illustrates the indirect encoding mapping scheme. The amount of possible phenotypes does not increase by using indirect encoding, but by using an indirect encoding scheme one can reach a different set of phenotypes. For example, the phenotype space may contain more complex phenotypes (i.e. large neural networks).

Developmental encodings are a subset of indirect encodings, but can have some key differences from non-developmental indirect encodings. Developmental encodings may focus on defining the behavior of sub-components, which together produce a phenotype through their interactions during the decoding process - often referred to as emergence. If the decoding process is deterministic and does not take any input from the environment, developmental encodings are just a particular approach to indirect encodings. However, if the developmental encodings take input from the environment during the decoding process the encodings have different properties from direct encodings and other indirect encodings. By taking environmental input in account during the developmental process the mapping from genotype space to phenotype space is no longer just a function of the genotype, but also the environment. Instead of one genotype mapping to a specific phenotype, genotypes can now map to a set of phenotypes. Other Indirect and all direct encodings search for phenotypes, but input-sensitive developmental approaches search for information processing programs which produce phenotypes. In effect the developmental system can produce an information structure using the information of the genotype as well as the information from the input. This is interesting as it means that a single genotype can in principle map to phenotypes which perform well under different circumstances - for example, if the problem domain is dynamic or changing. Further, it is interesting because it means that the genotype in principle needs to encode less information (or at least less information about the problem domain), because additional information will be provided by the environment during the developmental process. Allegorically, a direct encoding searches for a bitmap image, an indirect encoding searches for an jpeg image, and developmental encodings search for image compression/decompression algorithms, or alternatively, direct encodings searches for a specific solution, indirect encodings searches for a compressed solution, and developmental encodings search for compressed solutions which may use information from the environment, i.e. programs capable of some degree of learning or adaptation. From an engineering perspective this could entail several desirable properties, such as robustness of genotypes during a change in the problem domain, or the application of a single genotype to different problem domains. It is also clearly how advanced biological organisms such as humans develop. Therefore, environment sensitive developmental encodings are used in NMS-LOC (A similar argument is made in Hintze et al. (2020) and the above is primarily a rephrasing of common arguments in favor of indirect and developmental encodings such as Julian Francis Miller (2003) and Eggenberger (1997)).

Direct and indirect encodings can be considered mapping functions of the type Phenotype = mapping(Genotype). Developmental encodings on the other hand can be more complex, and can in the most complex case be functions of the type $Phenotype_t = mapping(Genotype, Phenotype_{t-1}, environment_t, feedback_{t-1})$. This both enables the use of different types of mapping functions which ideally can bias the search to more favourable phenotypes and can also make it possible to reach a larger amount of phenotypes, either by using environmental data such as testing the phenotype on problems and getting fitness feedback or using random seeds. Figure 2.6 illustrates how mapping functions which use more inputs than just the genotype and previous timestep phenotypes can reach a larger set of phenotypes, where some may be more probable than others. Note that direct and indirect encodings as described by Figure 2.5 can also contain some phenotypes which are unlikely to be actually reached in the search. Developmental encodings are in practice a type of indirect encodings when they use mapping functions of the type $Phenotype_t = mapping(Genotype, Phenotype_{t-1})$, as such functions are in practice incremental deterministic decodings.

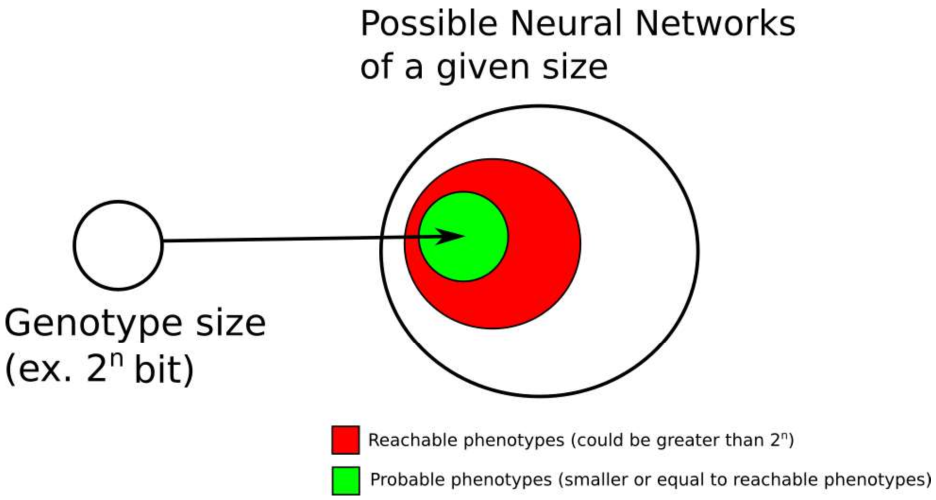

Possible Neural Networks
of a given size
Genotype size
(ex. 2^n bit)
🟥 Reachable phenotypes (could be greater than 2^n)
🟩 Probable phenotypes (smaller or equal to reachable phenotypes)

Figure 2.6: Mapping the genotype to phenotype when using mapping functions with other inputs than just the genotype and phenotypes at previous timesteps.

2.2.3 Biological Inspiration

Although NMS-LOC does not intent to be biologically accurate, some aspects of biological organisms and neurons still helped inspire the design.

- Dendrite/axon lengths in human brains follow a power law distribution (Downing (2015)). This means that Cartesian distance in three-dimensional spaces can be a useful heuristic in algorithms when searching for possible neuron connections. As such neurons search for connections to other neurons by trying to find a target distance based on sampling a power law distribution in NMS-LOC.
- Vertebrae bodies and brains develop modular components through the activation of homeobox gene-variants through the use of chemical markers. A neuron model could potentially achieve similar homeobox modularization by allowing it to switch between different versions of the neuron functions. This is often done through the creation of chemical gradients in the body, but as Stanley and Miikkulainen (2003) point out computer programs can access the coordinates of a cell directly, and therefore CGP functions in NMS-LOC is given the x, y, and z coordinates of the relevant neuron as input. As an approximation of homeobox gene variants the CGP genome can contain variants of each function, with a master control function which can select between the variants, such that different functions can be used in different circumstances.
- The theory of facilitated variation (Gerhart and Kirschner (2007)) posits that evolutionary variation in the modern pos-pre-Cambrian era is primarily done through searching over different combinations of core processes, which are building blocks construed of genomes which are robust to evol-

utionary mutation, environmental change and recombination. The theory states that evolutionary adaptivity is increased by searching over different ways of combining the core processes, as mutations affecting the connections between core processess are more likely to produce useful changes in the phenotype than other gene mutations. To approximate core processes NMS-LOC allows the genome to extract sub-graphs from CGP-graphs, and collapsing the sub-graph into a single CGP node during crossover. Ideally this would make NMS-LOC capable of finding useful functions similarly to core procesess.

- Biological neurons have a complex state, both represented through chemicals secreted throughout the local area as well as chemical buildup within the neuron. John H. Holland (1998) discusses how buildup of "fatigue chemicals" can cause neural networks to switch between activation patterns, and as such "switch focus". To allow for complex states within neurons NMS-LOC has a configurable hyperparameter dictating an amount of neuron state variables (and axon-dendrite state variables) which can be read and written to by the neuron (or axon-dendrites) functions.

2.3 A general model of connectionism

Connectionism is a paradgimn within cognitive neuroscience which focuses on the study of the mind modelled by interconnected neurons. Historically, connectionism has been related to neural network research, as computational neural networks provide a way to determine the capabilities of connectionist models. However, although neural networks have been used as models in connectionism, this does not mean that connectionists hold the view that the neuron model used is biologically plausible, rather just that the abstraction is inspired by biology and useful for connectionist distributed processing (Waskan (n.d.); Buckner and Garson (2019); Downing (2015); Minsky (1991); Hiesinger (2021)).

No matter which neuron model used all connectionist models have in common that they model intelligence or computation through the use of several interconnected neurons, or more abstractly, interconnected computational components. Therefore, it would be beneficial to have a metaphorical or conceptual framework to describe the computational components used in connectionist models in general, here dubbed the General Connectionist Model (GCM). To some degree models must always be compared through listing specific differences, such as whether the systems use backpropagation or a Hebbian learning rule, but viewing these different systems as a specification of an abstract connectionist model may help illuminate their relationships, provide a language to discuss the differences, and inspire changes and new designs.

In particular, the use of an abstract connectionist model can be useful in the design of NMS systems. As will be discussed connectionist models can be described as a set of constraints upon the GCM. NMS systems may therefore be defined by imposing constraints with degrees of freedom, that is, incompletely specified constraints which are specified through evolutionary search. In effect defining a NMS system is equivalent to defining a set of connectionist models and then using evolutionary search to select a specific model from that set.

2.3.1 General Connectionist Model

At a high level the GCM describes an abstract system built up of an infinite amount of computational units which may recieve, transmit, and process signals between themselves. The concrete specification follows, and is made with the intent to be general enough to represent any connectionist software as a set of constraints.

The GCM consists of an infinite amount of Turing machines with infinite memory (in the sense of being some type of computer which can compute any computable function), with the addition of permitting the Turing machines to transmit and receive signals between themselves. There is no concept of distance, sending a signal between any pair takes the same amount of time, and the signal may contain any amount of information. Further specification of computing substrate may be represented as additional constraints. The GCM Turing machines never degrade, break down, or have errors unless further constraints say so. Each computational unit may implement it's own program, but may also use the same program as other units. The objective of the GCM is to minimize the time spent on computing the output, as such it is not desirable to use a single Turing machine to compute the output, but rather to spread it across other parallel units i.e. a connectionist mode of computation. Other performance measures such as energy efficiency, robustness in face of component degradation, memory efficiency, and ease of developing useful programs can be desirable properties of connectionist models, but are modelled as additional constraints imposed on the GCM.

Finally, the GCM needs a way to handle input and output. For generality every computational unit is given every input, and the neuron program determines which neurons act upon which inputs. Similarly, every computational unit may transmit one or more of the outputs of the GCM.

Constraints upon the GCM should be defined as a constraint upon the individual computational neurons whenever possible. I.e. instead of defining the update of network weights using the efficient matrix operations, one should instead define the functions computed in each neuron, as if backpropagation was computed in a distributed manner, because this is what happens to each computational unit on a conceptual level. Of course, actual implementation of models defined in the GCM may be implemented differently.

Although the following discussion will be focused on using the GCM for neuron models in relation to NMS, the GCM is intended to be general enough to describe other forms of distributed computation. For example, distributing computations over a server or several servers, multi-core processors or implementing cellular automata (ex. Cook (2004)) could all be modelled through imposing a set of constraints upon the GCM due to it's generality.

2.3.2 Hebbian learning as a set of constraints upon GCM

The GCM by itself is not implementable. It is abstract, and requires impossibilities such as having an infinite amount of Turing machines with infinite memory. The intended use for the GCM is instead as a framework for defining other connectionist models in order to facilitate a language allowing comparison and possibly invention. To illustrate use of the GCM a description of a Hebbian-type connectionist model follows:

In the following set of constraints, a neuron refers to a computational unit in GCM parlance.

- Neurons may only send signals to neurons they have an explicit pre-determined connection to.
- Neurons may only have outgoing connections to non-ancestor neurons - in other words, no recurrence.
- Only a finite set of neurons are used.
- Neurons only store the learning rate, the connection weights of incoming connections in memory, in addition to whatever the neurons need to couple incoming signals with the correct connection weight, the last incoming inputs and whatever it needs to connect these to the connection weights and it's own last output.
- There are three types of neurons, defined by implementing different neuron programs: Input neurons, which transmits a specific input signal unchanged/unprocessed to neurons it has connections to. Hidden neurons, which take a weighted sum of inputs, compute the sine of the sum, and transmits the signal to neurons it has connections to. Hidden neurons may only do this when all their inputs are ready. Output neurons function identically to hidden neurons, except that output neurons do not transmit any output to other neurons, instead giving it as program output.
- After each training sample hidden neuron and output neurons are given a signal. Using the contents of memory, the neurons update their weights according to the formula:

$$\begin{aligned} weight_i = (1 - learningrate) * weight_i + \\ learningrate * (abs(incomingOutput_i - outgoingOutput_i)) \end{aligned}$$

2.3.3 Using the GCM

To show how GCM can be used to define neurons for use in NMS systems the previous sections Hebbian learning is changed to include the use of evolveable programs. The NMS system described in this section is meant to be illustrative, and not as a suggestion for further research:

- Neurons are given coordinates in a two-dimensional space. Neurons do not need to have unique coordinates.

- Each neuron is given n internal state values it can write to (referred to as neuron internal states - although the neurons strictly speaking also store incoming weights etc.). The neuron internal states have default values used at initialization.
- Neurons may form new connections to non-ancestor neurons. This is done by defining a computable program. Defining the weight-path to a neuron as the product of all weights along a path between the neurons, the computable program is given the maximum of its weight paths to the target neuron, the average weight path, the coordinates of the target neuron, it's own coordinates as input, and it's own writable internal states. The program outputs a value, if it is greater than one, a connection is formed - additionally it outputs how much it's internal state variables should change. (In order to facilitate neurons getting this information neurons are allowed to send signals back-stream for the explicit purpose of exchanging information for this program). After each training sample a random subset of neurons runs this program for a random subset of target neurons.
- Neurons may spawn new neurons. This is defined by an evolved program. The program is given it's internal state variables and coordinates, and outputs a value defining whether or not a neuron should be spawned, the coordinates the spawned neuron should be spawned at, and it's own internal state. In terms of the GCM this is done by just selecting one of the infinite amounts of unused neurons.
- An evolved program which updates internal state is ran after updating the neurons weights - is given the average weight, average input value, and own output value and coordinates as input. Outputs weight change.
- At the very end of each training sample each neuron runs a program which determines if it dies. Input are coordinates and neuron internal states. On death all incoming and outcoming connections are removed, no further activity is performed in the neuron.

The Hebbian learning system has been extended to allow the neural network to grow and shrink. Using an error measure over the outputs of the neural network over a given amount of iterations makes it is possible to search for suitable evolved programs.

2.3.4 Other neural models as sets of constraints on the GCM

It is outside of the scope of this thesis to give a detailed account of many contemporary neural network methods as sub-methods as GCM. However, as it is possible a brief note is included to highlight a key difference between NMS and contemporary methods (ex. Deep learning, NEAT).

In view of previous discussion it is possible to define a clear and specific definition of NMS systems. An NMS system is any system which attempts to solve problems in an connectionist manner, and which constitutes a GCM subset of a size greater than one, meaning that some part of the neuron (or computational unit's) specification is done by evolutionary search. For example, evolving activation functions for a specific MLP structure would qualify. This is a loose definition, but there is no particular benefit to having a very strict definition and debating at length what is and what isn't an NMS system. As such NMS systems do not need strictly need to use developmental encodings, per definition. However, the properties of developmental systems, such as emerging through the interaction of sub-components, seems to make NMS and developmental encodings a desirable combination.

Stanley and Miikkulainen (2003) provides a way to classify neuroevolutionary systems. Using their taxonomy several characteristics of NMS systems can be identified. In the following the word "program" is used, but strictly speaking the evolved specification does not need to be a program; but "program" is a convenient shorthand for "some evolved aspect of neuron behaviour".

1. Cell Fate (death, birth, topology, weights, position, internal states, runtime behaviour or similar etc.) is determined by emergent behaviour resulting from the execution of one or more evolved neuron programs. Not every aspect of Cell Fate needs to be directly determined by evolved programs, but at least one evolved program exists whose execution in some way effects some aspect of Cell Fate.
2. Targeting may be evolved directly, partially or indirectly affected by evolved programs, but does not necessarily have to be.
3. Heterochrony is most likely affected by the evolved programs. Heterochrony is the timing of developmental events, and in NMS systems there may exist programs which in some way affect the development, such that future developmental events is affected by the result of previous, making the developmental process an adaptive process determined by running the "overall program". However, strictly speaking an NMS system could only evolve runtime/program solving behaviour, such as only evolving which function neurons compute over their inputs.
4. Canalization (robustness to genomic mutation) is probably desirable, but is a property of the specific system used.
5. Complexification, the ability to produce more advanced phenotypic behaviour through making the genome gradually more complex is also an implementation detail and is not necessary, although it may be desirable in some systems.

2.4 NMS Search Space Size

Another way to view the GCM in relation to NMS system design is as a way to reason about the size of the search space. The more behaviour that needs to be evolved, the larger the search space. The following discussion of search space size assumes that CGP is used to evolve the programs. This is not a requirement for NMS, but is done as it applies to the NMS-LOC. An equation for a loose upper bound of search space size for a CGP-program with N inputs, M outputs, a maximum of Z active nodes, and Q different CGP node functions and a maximal node function arity of T is shown in equation (2.1).

$$2^{\frac{Z*T!}{2}} * \frac{Z!}{(Z-M)!} * Q * Z * Z * T! * \frac{N!}{(N-T)!} \tag{2.1}$$

A proof is shown in the Appendix (Section A).

Further, the NMS system used in this thesis permits modular CGP node functions to contain other modular CGP node functions, theoretically permitting an infinite amount of genotypes. The intent behind allowing modular CGP functions is for useful components to evolve which could make further evolution simpler, inspired by core processes in facilitated variation theory (Gerhart and Kirschner (2007); Downing (2015)). In the ideal case these core processes could be combined into new components, which themselves could form core processes. As such the intent is that modular CGP makes the search space easier to navigate, at the cost of increasing the size of the search space. In principle for a high-recursive-depth modular function to survive it needs to correlate with an increase in fitness, although it is possible that it may be used or persist due to assosiation with another change leading to improved fitness, corresponding to the phenomenon of hitchiking in Schema Theory (Eiben and Smith (2015)). As such recursion depth of CGP modular functions and the amount of CGP modular functions should be monitored to ensure that issues do not occur in practice, which could be indicated by the use of many high recursive-depth modular functions in poor-fitness programs. Such issues could be alleviated by introducing a max recursion depth for modular CGP functions, in which case a theoretical upper bound of the search space could also be determined[1].

Further, the use of modular CGP makes the T parameter difficult to set, as although the may only use 'basic' CGP node functions of some arity T, the CGP modules may still evolve to have a larger arity than T (unless it is deliberately capped). Additionally, the Q parameter does not account for the creation of CGP node types. As such the upper bound is better suited for non-modular CGP programs, or if modules are rare or not present in the genotypes in practice. However, Equation 2.1 may still give a decent estimate as long as high-arity modular CGP functions are not common in the genotype, even if it is not the actual upper bound for the modular CGP case. Experimentally NMS-LOC does not use modular CGP functions, even though it is permitted, making the upper bound in practice accurate, and thus a reasonable measure to discuss the size of the search spaces in various configurations of NMS-LOC.

Additionally, the NMS system used in this thesis uses several evolved programs in its evolved neuron models, several of which contain several inputs and outputs depending on the set hyperparameters. If O is the set of input arities to each program, I is the set of output arities of each program, there are P different programs

[1] As an implementation detail there is an effective maximal recursion depth in NMS-LOC constrained by the maximal depth of the Python callstack.

and the function f(o, i) denotes equation (2.1) with N=o and M=i then equation (2.2) shows the total size of the search space.

$$\prod_{p=i}^{P} f(O_p, I_p) \tag{2.2}$$

Clearly, the genotypic search space in NMS can be extremely large. Note however that the phenotypic space may be far smaller, as it it is possible that many genotypes map to the same or very similar phenotype(s). It is clear that NMS systems should focus on developing and using algorithms which are able to efficiently navigate the search space and on optimization in order to handle the search space size. There are several issues which can occur in terms of the navigability of the search space: First, it may simply be too large, in which case evolvable program inputs and size can be adjusted, or an evolvable program could be dropped in favor of a hardcoded function. Secondly, it is possible that few genotypes map to meaningful phenotypes. In other words, the search space may consist of many local optima or large search space plateaus or flat ridges, making it difficult to navigate due to a lack of an available gradient (see Figure 2.7 for an illustration of search spaces). Although CGP's neutral drift means that it can escape local optima given sufficient runtime, it still needs a navigable search space, as otherwise one might as well use random search. If this is a problem the neuron design could be reevaluated in order to make it easier or more likely to produce phenotypes with meaningful behaviour, for example by replacing an evolveable program with an hardcoded function. The third possible problem is that the search space does not contain a good solution or contains very few good solutions, in which case a reevaluation of the neuron design is necessary. The first and second problem are made more critical by the fact that developing a phenotype from a genotype may be an computationally expensive operation in NMS systems.

2.4.1 Search spaces in the experiments

Based on the configuration files for the experiments in the thesis the above formulas can be used to calculate the search space size of the NMS-LOC problems. In short, the search spaces are large due to combinatorial explosion, but no larger than other AI problems. The config files are shown in Appendix C, and the corresponding search space sizes are shown in Table **??**. These search spaces are large, but search spaces of most AI problems is large due to combinatorial explosion, what matters is that the applied algorithm is able to navigate the search space correctly. For example, an MLP with one million weights and one thousand possible values per weight gives one thousand to the power of one million, or, $10^{100000000}$ different possible permutations, which is far larger than the NMS search space sizes. The nature of these search spaces are different, as the MLP fitness landscape is continuous and differentiable (or approximately so, in the case of some activation functions like ReLU), while CGP search spaces are made up of discrete points. However, due to CGP's neutral drift and inactive nodes large parts of the

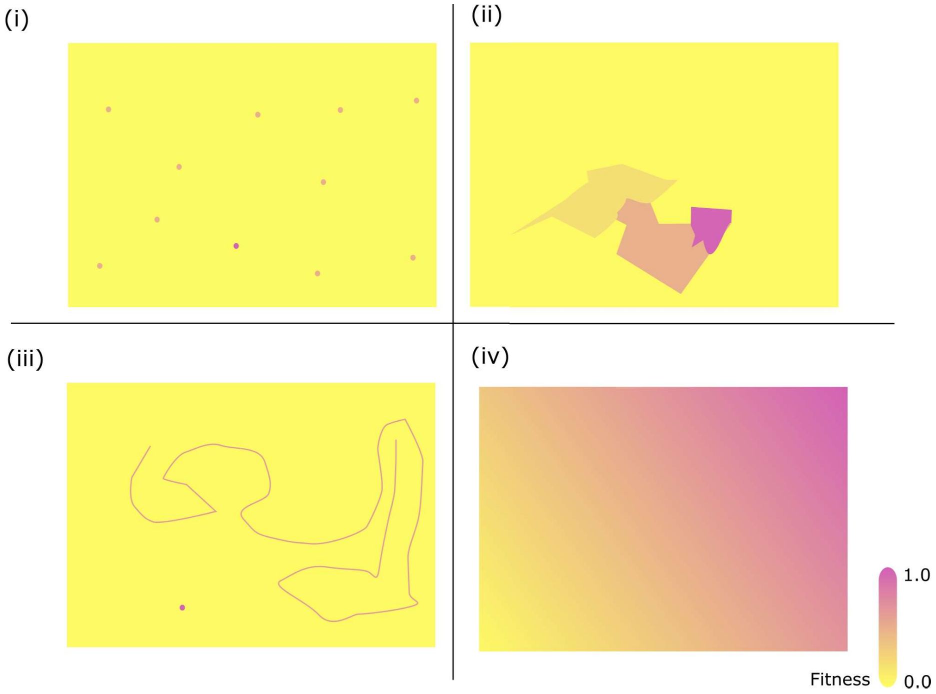


Figure 2.7: Illustrated search spaces.(i) Many local optima make it difficult to find the global optima. (ii) The search space is divided into several plateaus. Although the plateus are connected, movement within each plateau is random, making it difficult to reach the optimum. (iii) Movement in the search landscape will likely move along the long ridge, making it difficult to reach the global optimum. The issue is not that the system moves along the ridge, but that the ridge does not lead anywhere in the search space. (iv) An idealized smooth gradient, making it easy for evolutionary search to find the global optimum.

Config file	*Search space size*	*Search space size limiting to one homeobox function variant per function*
Config 2	$4.8 * 10^{352}$	$4.8 * 10^{352}$
Config 3	$1.0 * 10^{1063}$	$3.0 * 10^{366}$
Config 4	$1.6 * 10^{1408}$	$6.4 * 10^{481}$

Table 2.1: NMS-LOC search spaces sizes by config file.

search space will effectively be compiled to the same genotype. Further, although the theoretical amount of programs can be large, the amount of programs which is likely to be considered by a given search algorithm can be smaller. Similarly to discrete state games such as chess there may exist states which are technically legal, but which would never occur in practice.

As such, it is reasonable to expect that it is possible for NMS-LOC to find good CGP programs, but that a suitable search algorithm is used, as entirely random search would probably struggle in the face of the large search space size. Experimentally it is clear that NMS-LOC is able to find good programs through a variant of the CGP evolutionary algorithm.

2.5 Used problems

This section presents the problem domains used in experiments in this thesis.

One pole balancing: The one pole balancing problem is a classic control problem, where the task is to balance a pole placed on top of a cart at an angle by moving the cart to the left or to the right by applying a force to the cart at each timestep. The pole starts at randomized angles. As long as the pole position is no more than 15 degrees from vertical and the cart does not move more than 2.4 units from the center position a reward of 1 is given each timestep. When this is no longer the case, the problem instance has failed. This version of the one pole balancing problem uses the CartPole-v0 implementation in Brockman et al. (2016), which is based on the specification in 'Neuronlike adaptive elements that can solve difficult learning control problems' (1983). In NMS-LOC each evaluation against the one pole balancing problem consists of one hundred timesteps, where reaching a fail state counts as a reward of 0, and failing to output an action counts as a reward of 0. The fitness of a genotype is 1 - (Reward/100), making the objective fitness minimization (aka error minimization).

IRIS flower dataset: The IRIS flower dataset is a classic tabular data classification problem. The task is to classify Iris flowers into species based on the measured length and width of petals and sepals. There are three species, and the dataset contains 150 samples. At the start of an evolutionary run the dataset is split into one part containing 120 samples - the training data - and one containing 30 samples - the validation data. A phenotype is given the training set in a randomized order to develop and evaluate it's fitness twice, and is afterwards then additionally evaluated on the validation data to track validation fitness over time. Only the fitness from the evaluating the training set is used in the evolutionary algorithm, as such, the validation data has no impact on the evolutionary run. The version of the IRIS flower dataset used is maintained by Dua and Graff (2017), and was originally made by Fischer (1936).

Chapter 3

Methodology & Design

The following details the design of NMS-LOC at a high level. Not every technical detail is presented, but the overall system functionality should be clear from the following discussion. The system concludes with a discussion of possible issues with the design.

3.1 NMS-LOC specification

The following section describes the design of NMS-LOC, first presenting the variant of CGP used to control the neurons, then presenting the neuron engine (or the "neuron environment" or "neuron world") in which the neurons act and are simulated.

3.1.1 The modfified CGP system

In this section the NMS systems use of CGP is discussed, introducing CGP briefly as well as describing the modifications and choices made for this specific system.

Introduction to CGP

Genetic Programming (GP) is a subfield of evolutionary algorithms which searches for computer programs by searching over syntax parse trees (Willis et al. (1997)). CGP is a variant of GP where programs are represented as directed acyclic graphs. CGP genomes define nodes by defining which function (hence node function) the node executes over it's input(s), and which nodes each node gets input from. Additionally, the genome specifies which nodes are output nodes. Input nodes are added as a part of decoding the genome (Julian F. Miller (2020)). Several variants and extensions of CGP exists, of note for this work are modular CGP-functions, where other CGP-programs can be used as node functions (Walker and Julian F. Miller (2004)). For an example of a CGP program see Figure 3.1, which shows an CGP full adder. Modular CGP works by allowing a single node to compute another CGP-program, such as an full adder.

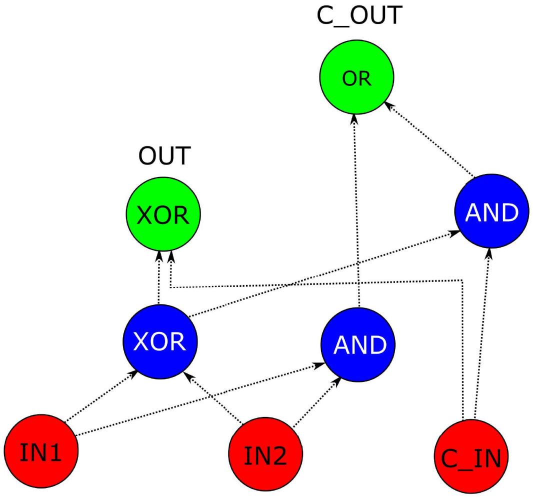


Figure 3.1: Example of a CGP function equivalent to a 1-bit full adder. Inputs are sent form bottom up, red nodes are inputs, green are outputs, blue neither.

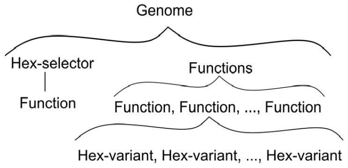

Genome
Hex-selector
Function
Functions
Function, Function, ..., Function
Hex-variant, Hex-variant, ..., Hex-variant

Figure 3.2: Shows the structure of the NMS genome.

By default CGP does not use crossover operators. Default CGP uses mutation operators which can add nodes up to a predefined limit, change edge connections up to a maximum arity of the node or change node type. CGP genomes can contain inactive nodes which are either not fully connected to input or output nodes, and mutation over these nodes serves as neutral drift facilitating evolutionary exploration (Julian F. Miller (2020)). The standard evolution strategy consists of maintaining one parent genotype and producing a given number of offspring and selecting the offspring with the highest fitness that is higher or equal to the parent to facilitate neutral drift through mutations in the inactive nodes (A (1+λ) evolutionary strategy) (Julian F. Miller (2020); Eiben and Smith (2015)). However, by maintaining only a single population member it may be difficult to explore a wide area of the design space. Therefore, the variant of CGP used in NMS-LOC introduces a crossover operator to facilitate the use of larger populations.

Details on CGP variant used in NMS-LOC

In NMS-LOC the CGP genome is split in two main parts, as illustrated in Figure 3.2. One part, the Functions part, defines a set of functions, each of which is a control program for one of the things neurons and axon-dendrites can do in the neuron engine. Each function has a configurable amount of variants, called hex-variants. The other part of the genome, the Hex-selector, is itself a CGP program. However, the hex-selector has a special function. When run it outputs which hex-variant of the functions should be used. This is inspired by homeobox genes in vertebrae biology, where chemical markers in the body can dictate that different parts of an organism's genome should be used, i.e. whether to make an arm or a leg, an hippocampus or a frontal lobe (Downing (2015)). The intent is that different homeobox variants can be used in different circumstances, for example in different regions of the neuron engine. To facilitate this functions are given the position of the neuron as input, inspired by the suggestion in Stanley and Miikkulainen (2003) to let evolutionary programs access coordinates directly instead of simulating chemical gradients.

An alternative to having several functions for every neuron action would be to have a single master-control program controlling all possible input and output actions. The choice to have several programs is inspired by Walker and Julian F. Miller (2004) which successfully solved a circuit design problem by separating the problem into n-subproblems where n is the desired amount of circuit outputs. Unlike Walker and Julian F. Miller (2004) the function programs have several outputs, and each genotype has a shared fitness instead of an individual fitness for each function as the fitness of each function is dependent on their relation to the other functions. The intent of using a design which has several functions (see the Appendix B) instead of one unified neuron-controller program is to simplify the search space, and to make the functionality of found programs easier to interpret. The design decision is also inspired by biology, specifically the theory of facilitated variation (Gerhart and Kirschner (2007)). Facilitated variation states that evolutionary variation in the post-pre-Cambrian era is primarily done through searching over combinations of gene blocks called core processes, which are gene blocks that are stable and robust to being combined with other blocks in many ways. The theory postulates that robust evolutionary variation can be achieved through evolving different ways for these core processes to interact and inter-regulate, rather than mutating the processes themselves. Each function can be viewed as a core process which has a specific functionality and is only weakly linked to the others through the neurons internal states and the effects each function has on the neurons behavior, which the implemented crossover operations attempt to take advantage of.

It should be noted that the aforementioned CGPANN approaches sometimes split the genome up in chromosomes, but do not contain a concept of homeobox variants. An interesting approach in N. Khan and Gul Muhammad Khan (2021) is the use of an ensemble approach to evolution (Eiben and Smith (2015)), where

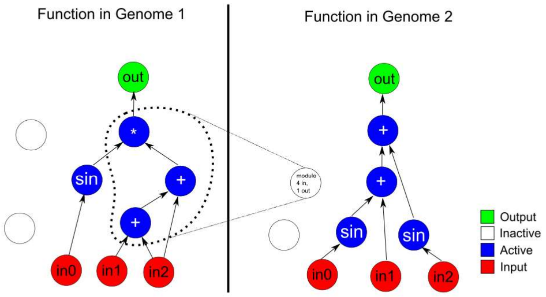


Figure 3.3: Illustrates how crossover works at the CGP-function level. A part of one parent genomes active nodes is made into a node function, and set as the node function executed by an inactive node in a copy of the other parent genome. This inactive node may then become active through mutation.

each chromosome is evolved separately with separate fitness values (similar to Walker and Julian F. Miller (2004)), achieved by averaging the fitnesses of solutions in which the chromosome appears. Such an approach could be useful for NMS-LOC, but has not been investigated due to scope limitations.

Crossover is implemented at two levels, one of which is also inspired by facilitated variation. Functions and their homeobox variants can be swapped with a given probability, and within a homeobox variant may be copied over to the neighbouring homeobox variant of the same function with a given probability. The other level of crossover is at the CGP level, where functions can be crossovered by extracting sub-graphs of one function and using it as a modular function in an inactive node in the other function, as shown in Figure 3.3. Modular CGP functions therefore act as core processes, potentially accelerating evolution and allowing greater complexity by finding good modules. Pairs of genomes are selected as crossover pairs randomly, and each genome can only participating in one pairing. Each pairing produces a number of offspring which together with other offspring are evaluated for selecting the next generation.

After crossover mutation is applied. CGP mutation in this work is slightly different from the standard CGP-mutation operator which can change and add edges or changing the type of a node. For one, the existing modular CGP-functions in a genome is added to the set of types a node can mutate to. Secondly, mutation probabilities are scaled depending on the input arity of a function: The node type mutation chance is scaled by dividing by input count, such that large-input functions are not more likely to be discarded than low-input functions due to inactivity. Secondly, the link mutation chance is scaled by dividing by input count such that large-input node functions can change inputs at the same rate. Finally, output index change chance is scaled by input arity of node pointed to by the index for the same reason.

Modular CGP-programs were introduced in Walker and Julian F. Miller (2004), but unlike in Walker & Miller modules can contain other modules in NMS-LOC, and cannot be expanded and changed after creation. Modules creation in NMS-LOC is based on Kaufmann and Platzner (2008); specifically modules are created from cones (i.e., beginning with a node and picking nodes connected to it such that there is always a path from any node to the first picked node). Kaufmann and Platzner (2008) also investigated using crossover, but found it tended to lead to increased computational cost - in case this applies to the NMS-LOC the code supports populations as low as 2, minimizing this overhead, and could be customized to single-population as in regular CGP relatively easily by making a single genotype pair with itself.

Each pair of parents create two or four children - four if using an optional setting which produces two extra children without using crossover, two otherwise. If a child has equal or higher fitness than its parent, then it replaces the parent. This is done instead of selecting the highest fitness genotypes overall for the next generation to reduce the speed at which dominant genotypes take over the population, in the hopes of preserving more diversity in the search. A child of equal fitness replaces its parent to facilitate neutral drift, as this allows traversing a plateau in the fitness landscape. To make it possible for good genotypes to spread in the population an explicit replacement mechanism is implemented: Genomes are split around the average fitness. A random genome from the best split then replaces a random genome from the worst split.

Mutation rates can be configured using hyperparameters. Finding a better child genotype multiplies the mutation rate by a configurable factor, and failure to find a neutral or better child genotype, that is, being stuck in the state landscape, multiplies the mutation rate with a configurable number. Additionally, neutral child genotype mutation rates suffer a slight decay rate. This ensures that as mutation rate goes low, the mutation rate does not get stuck due to NMS-LOC producing the same or functionally similar genotypes. A mutation rate limit is included as a configurable hyperparameter, and when the mutation rate is lower than this hypermutation is triggered which sets the mutation rate to a far higher value in order to escape local minima (Eiben and Smith (2015)).

3.1.2 The Neuron Model

The neuron engine consists of a three-dimensional Cartesian grid. Each position in the grid can contain several neurons which saves computational resources for checking and handling neuron collisions. Similarly, there is no concept of dendrite collisions. This is done to save computational resources on detecting and handling collisions and to take advantage of the fact that the artificial neurons are not actually constrained by physical space.

Axons and dendrite are unified in axon-dendrites, which each are connected to a neuron, and can be connected to another axon-dendrite or be a free axon-dendrite. The distinction is done based on whether signals are being sent forwards from dendrite to axon, or backwards through axon to the dendrite, and depending on the direction different programs are used for processing and transmitting signals. A unified axon-dendrite model was selected to reduce the size of the search space. Hidden neurons (that is, evolved neurons (and their axon-dendrites)) can perform the actions defined in Appendix A. Input and Output neurons can perform no actions, but are always considered as having free dendrites, such that the genotype-controlled neurons can connect to them. When a neuron seeks a connection to an axon-dendrite it samples a power-law distribution to determine the target distance in order to favor shorter connections as in the human brain (Downing (2015)). This assumption is made to avoid simulating axon-dendrite movement while also trying to maintain some similarity to biological structures. To simplify distance calculations the overall grid is divided into sub-grids, and candidate connections are gathered by investigating the sub-grids which have a gridwise distance close to the sampled distance.

When initializing the neuron grid when evaluating a genotype the neuron grid contains a single genotype-controlled (i.e. hidden) neuron that is connected to the input nodes[1], as well as input and output nodes. For each problem instance the neuron engine allows up to a given amount of neuron functions to execute, and then stops the neuron engine to avoid infinite loops and select for quicker programs. When given the next problem instance the grown network is maintained, such that the growing neuron structure can develop.

The neuron simulation engine maintains an action queue, which determines which neuron function should be run next. Each action in the queue has a timestamp, and the lowest timestamp in the queue is always selected as the next action. Within a single timestep there is a FIFO queue to determine the order of same-timestep actions. The timestamp system ensures that actions are executed in a temporally sensible manner and gives each neuron equal access to computational resources.

Signals sent between neurons and axon-dendrites can be multi-dimensional, i.e., several floats can be sent in one signal. The number of floats per signal is a configurable hyperparameter. Likewise, the number of internal state variables is a configurable hyperparameter. These two design decisions are inspired by biological neurons, which maintain complex chemical states internally and in their local area (Lovinger (2008); John H. Holland (1998); Downing (2015)). Further, there is no reason to assume that one-dimensional signals as done in MLP is necessarily optimal.

A complete list of neuron and axon-dendrite functions is given in Table **??** and Table **??** in Appendix D. Most functions are given internal state variables and neuron position as input, and some are given signals as input.

Figure 3.4 illustrates how an NMS-LOC phenotype can develop. The figure shows the initial NMS-LOC phenotype structure with a hidden neuron go through several timesteps upon receiving an input signal. The figure illustrates how the learned program may grow a simple neural network structure, signal sending is not illustrated. Notice again how this process is conceptually similar to neural network growth/development in general, and for example NEAT and Miller's 1D

[1] Earlier versions of NMS-LOC started with the hidden neuron not being connected to anything, but this was changed, as further discussed in Chapter 5

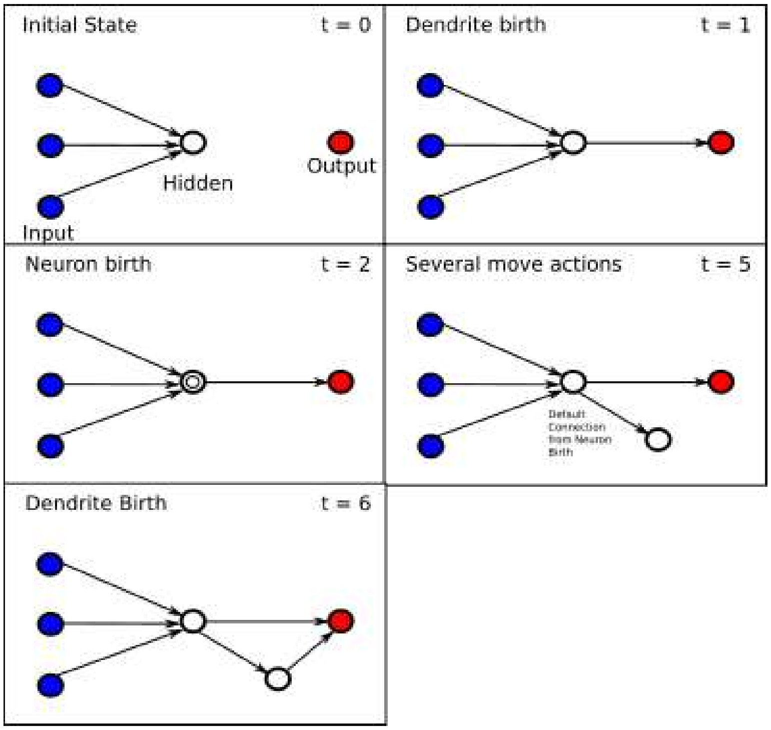


Figure 3.4: Shows how a NMS-LOC phenotype may develop over several timesteps to produce a simple neural network.

CGPDNN, even though the actual algorithms differ.

3.1.3 CGP, Neuron Engine and randomness

NMS-LOC allows for the use of randomness (through sampling psuedo-random number generators). This is always used in the "evolution" part of NMS-LOC, and various amounts of randomness in the "execution" part is investigated and discussed in Chapter 5.

Randomness in the execution of CGP functions may make it possible to find smoother gradients. When the neuron engine determines if an action should be taken based on output from a CGP function it may do so semi-randomly. A value greater than 1 means always do the action, a value lower than 0 means never do the action, and a value in the range [0, 1] denotes the probability of doing the action. This allows for non-deterministic programs which could not be evolved without allowing for randomness, and may make it possible to have smooth gradients in the state space by adjusting probabilities gradually. It is also investigated if using Gaussian sampling as a CGP node function where the inputs is the mean and standard deviance of the Gaussian is beneficial. Such sampling may provide similar advantages to the above, additionally programs which are evolved to handle noise in the form of Gauss sampling may also be more robust to unexpected input. In terms of Stanley and Miikkulainen (2003)'s taxonomy for artificial embryogeny using randomness may increase canalization. Initial states of the one-pole balancing problem are also randomly sampled.

The disadvantage with using randomness is that there is no deterministic mapping between genotypes and phenotypes. This means that the phenotypes, and therefore the fitness of the genotypes, can vary between runs. This could be handled by running each evaluation many times, effectively doing a Monte Carlo estimation of the genotype fitness. However, running many evaluations could quickly become prohibitively expensive, especially if the genotype space contains many bad genotypes. Effectively the fitness of a genotype is a random variable, and the issues introduced by making fitness a random variable are discussed at length in Chapter 5. The following part of this section describes the design choices made to accommodate randomness in NMS-LOC.

The approach taken in this thesis is to only perform multiple evaluations on the most promising genotypes. In standard CGP the current genotype in the (1+lambda) evolutionary strategy is only evaluated once, namely when it is a child genotype of the previous parent genotype. In this version of CGP, each of the several parent genotypes are evaluated every generation. Their fitness is updated by calculating the average fitness across every evaluation. That way genotypes which seem promising, that is, have the highest fitness, are evaluated many times and as such are given a more accurate estimate.

Still, two potential issues can occur. One issue that can occur is that a child genotype is "lucky", and it's first fitness evaluation is far higher than the actual average fitness given enough evaluations. In this case the child will take the population slot from another genotype, which may be higher performing on average. Between children competing for the same population slot, we can expect that by definition in the average case the average best child genotype will win out, even if only one phenotype-genotype mapping is performed. The larger issue is if a on-average worse child happens to win out over a on-average better parent, effectively leading to loss of information through gradient ascent against a minimization optimization goal. To circumvent this issue a second list of "all-time" historic best genotypes is maintained, of length equal to the population. The historic best genotypes do not produce children, and are not evaluated. However, on each population iteration the current population is evaluated against the historic best, and the n-best genotypes where n is population size from the current population and the historic best become the new population. The genotypes swapped out from the population are maintained in the historic best list. This helps reduce the probability and severity of information loss between children and parents, because when a child replaces a parent genotype in the population, the parent takes a slot in the historic best list if it's fitness is great enough. To prevent a singular good genotype from dominating the historic best list and therefore the population, a check is made to ensure that each genotype ID can only exist in the historic best once. To ensure that a genotype family does not dominate the historic bests completely (effectively reducing variance and preventing any speciation in the population), a list of genotype parent ID history is maintained up to depth 3[2], s.t. a genotype can not be added to historic bests if it has the same parent up to depth 3 as another genotype in the historic bests - in this case it may only replace the existing genome in historic bests. In genetic-programming parlance the wrongful child-parent replacement would make the evolutionary strategy non-elitist: That is, good parent genotypes are discarded in favour of worse children, which may lead to a loss of information (Eiben and Smith (2015)). There is in general some debate on whether or not elitism is necessary or desirable in general, but for CGP specifically elitism is a core assumption (Julian F. Miller (2020)).

For the same reason when a genome is replaced by another due to the replacement mechanism, the replaced genome is added to the historic best list. Of course, it is possible that a on-average worse genome is lucky and replaces a on-average better genome in the historic list, but the historic list at least mitigates the issue to some degree.

The other issue that can occur is that a child genotype is "unlucky", and randomly has a statistically unlikely poor performance. This is not as critical, because if this causes it to perform worse than a parent or other child it does not cause gradient ascent. At worst it could lead to situation where the search converges or gets stuck at a worse local minimum. This issue is therefore less severe, and is not mitigated against, instead it is accepted as a risk of using randomness. Chapter 5 also discusses how making the fitness of a genotype into a random variable can introduce noise which may make the gradient disappear.

3.2 Potential Issues

This section presents possible issues of NMS-LOC which were not considered critical for the purposes of this thesis, and suggest possible remedies to facilitate further development if it should be desirable.

3.2.1 Comparing solutions

When using CGP for NMS it may at some point be desirable to compare solutions in some meaningful semantic sense. For example, if evaluating a genome takes a long time, it would be sensible to check that a child solution is actually behaviourally different than the parent. Further, in order to control the population in evolutionary programming it is often important to consider the diversity of the population, that is, how different population members are (Eiben and Smith (2015)). This can be used to tune hyperparameters, or to produce more advanced population control mechanisms like speciation (Eiben and Smith (2015); Stanley and Miikkulainen (2002)). A naive approach would be to compare genetic strings directly, but this may be misleading. Depending on the specifics of the CGP genetic

[2] Later increased to 6.

string representation, a CGP genome could be shuffled and still produce the same solution directly, or be shuffled with appropriate changes to indexes to produce the same solution. In other words, a single CGP program may have several different genetic representations - as such similarity at the level of genetic strings does not necessarily correlate to the similarity of the programs. A simple improvement could be to compare the set of active nodes instead, as suggested in Goldman and Punch (2013). In Goldman and Punch's terminology the NMS system handles duplicates by skipping evaluation, instead using the parent genomes fitness. In theory one could do graph comparison, but this can be a relatively expensive operation, and is made more complex as two CGP functions may have the same or similar hex variants but in a different order, and because it is difficult to estimate how "big" the impact a genotype-space difference would have in phenotype-space. On the other hand, CGP function sizes may be relatively small (ex. in NMS-LOC experiments max size was set to 50 nodes, and some of those will be inactive) making it possible that even inefficient graph-comparison algorithm could be significantly quicker than developing a phenotype (See Wills and Meyer (2020) for an overview of graph comparison algorithms).

Goldman and Punch (2013) set solution makes it possible to detect some duplicates, but it may still be desirable to compare how similar non-identical solutions are. A percentage wise set comparison could give some view on how similar the genotypes are, but would not be able to compare how similar the produced phenotypes are. A single change to one of the active genes in a genome can produce drastically different behaviour. As such, it would be desirable to instead compare the phenotypic similarity of the programs.

To compare the phenotypic similarity of CGP programs I raise the suggestion of using a input-output tests, a sort of black-box analysis. By giving the programs which should be compared the same set of input, and observing how the output of the programs differ, one has data one can use to say something about how similar the programs function. The disadvantage is that for this comparison to be objective, one would have to input every possible input value. However, if the input values are representative enough of realistic program input input-output comparison may provide meaningful data about the similarity of phenotypic behaviour, perhaps by observing typical inputs to functions in the parent genotypes when they are evaluated, and using these as testing criteria for the child genotype. Other alternatives could be observing how sensitive programs are to variation of single variables. One possible optimization could be to determine the "points of divergence" between the CGP graphs, that is, the earliest points at which the CGP graphs have different active genes, and then performing input-output analysis only on the inputs which feed into points of divergence. Such testing might be useful as a heuristic similarity measure.

NMS-LOC tracks population variance for statistical purposes through a naive method of tokenizing genome ID's and calculating the Shannon entropy of the distribution. This is not a true measure of the variance of the population, as it only reflects the variance in ID space, and not in genotype or phenotype space.

However, if the ID Shannon entropy is low then singular IDs are more prevalent in the population, and the genotype and phenotype variety is necessarily lower, such that the measure can detect if NMS-LOC literally gets stuck on the same genotype in many or all population slots.

3.2.2 The signal-sending assumption

In the NMS-LOC it is assumed that neuron behavior can be evolved which is able to send communication signals of several types between neurons. As such the neuron signals may have several dimensions. The explicit signal-sending mechanism is intended as a simpler computational approach than simulating chemical diffusion.

However, it is possible that signal-sending behavior is difficult to evolve. Although the search space may in principle include signal-sending behavior, a problem may still emerge if the search spaces navigability relies on behavior which is only present in a small subset of the search space. Signal-sending behavior is relatively difficult to evolve, as it requires several things:

1. The ability to form neuron networks
2. The ability to have meaningful connections in these networks
3. The ability to transmit signals of any type
4. Evolving control programs which interpret signals and transmit signals in a meaningful semantic manner

In particular, as will be discussed in Chapter 5, the design of NMS-LOC was unsuited for sending signals backwards through connections, and as sending a signal from a hidden neuron to another involves 6 or 7 CGP program evaluations it is also too time intensive to send many control signals between neurons in the allotted computation time.

Following are suggested changes to NMS-LOC which could alleviate the potential signaling issue, but investigating these are outside of the scope of this work:

- Simplifying axon-dendrites such that they always transmit signals without running CGP programs to reduce the CGP programs used per neuron-to-neuron transmission from 6-7 to 2-3.
- Hardcoding neurons to only process signals when their magnitude is greater than some set or evolved constant, or special state variable. Incoming signals may be accumulated dimension-wise.
- Introduce a shared state for neurons in the same sub-grid as an abstraction of local chemicals. May diffuse to nearby grids.
- Instead of sending signals to specific neurons, neurons may send signals to all other neurons in the same sub-grid with strength inversely proportional to the distance between the neurons. Sub-grids should have some overlap, for example 25% between adjacent grids, effectively sharing corners with adjacent sub-grids. Could involve splitting the sending of control signals or "chemical signals" using this method and sending "normal" signals using axon-dendrite connections.

- Originally the intent was for NMS-LOC to be able to send signals backwards through axon-dendrites. However, this was done by sharing functions for sending backwards and forwards, which could make it difficult to evolve meaningful behavior. Additionally, there was not enough functionality for generating the control signals which were to be sent backwards, and the CGP function call amount per signal sent neuron-to-neuron also made this approach untenable. With other changes it could be reintroduced, but care should be made to ensure that there is enough opportunity to generate backwards signals, that backwards signals are handled differently either by giving functions inputs denoting backwards/forwards or through evolving another set of functions, and that signaling is not too time intensive.

Chapter 4

System Documentation

In this section the implemented CGP system is discussed. A high-level overview of design is given in Chapter 3, but the following chapter discusses more in detail the structure of the code base, software implementation, and development process. As such this section discusses NMS-LOC from a software development perspective, and presents choices made, challenges encountered, and advice for the development of other NMS systems.

4.1 Tech Stack

4.1.1 Code Structure

The following section provides a high-level overview of the structure of the codebase.

NMS-LOC's implementation is divided into four main parts. One part contains the control code for evolution and statistics gathering. This part calls the Neuron Engine, which controls the simulation of neurons during the evaluation of a genotype. The Neuron Engine is provided with a problem object which should implement a interface. The Neuron Engine is also provided with a Genome object. The Genome part of the code functions as a container for CGP code, and handles all non-inner-CGP specific mutation and crossover code. The CGP part of the codebase contains the implementation of CGP.

Additionally, there exists several scripts for processing statistics and doing the Monte-Carlo simulations used in this thesis.

4.1.2 Programming Language

NMS-LOC was implemented using the Python programming language[1]. Python was chosen because it is generally regarded as being a programming language suited to quick prototyping and development. Further, I have used Python extensively in the past, and as such using Python prevents the introduction of de-

[1] `https://www.python.org/`

velopment overhead in the form of learning a new programming language. The disadvantage of using Python is that as an interpreted language it is slower than other alternatives, such as C++[2]. However, lower-level compiled languages such as C++ are fast when the software is written well, and learning to write C++ code which is both functional and efficient constitutes a large development overhead in addition to Python being generally easier to write code in. As such Python was chosen for NMS-LOC.

The use of Cython[3] was investigated. Cython is a C++ compiler which compiles code written in Python to C++ by extending the Python language with optional language constructs such as strict typing to aid in the conversion. However, using Cython ultimately provided little time improvement. It was attempted to optimize Python code by converting it to Cython, however, due to the codebases heavy reliance on Python objects this provided little improvement, as much of the code could not be effectively converted. Taking advantage of Cython would therefore require an extensive code rewrite, which was not desirable at that point of the project.

4.1.3 Cloud Computing

Evaluating NMS-LOC requires some computational power. As NMS-LOC is written in Python and in an Object Oriented manner for ease of implementation it is not hyper-optimized or lightning fast. Additionally, to evaluating and comparing different settings for NMS-LOC requires running several runs with each setting to gather statistics. In order to achieve this a cloud computing platform was used to run separate evolution runs in parallel. The platform used is NTNU's computational cluster IDUN[4], whose setup is described in Själander et al. (2019).

Without access to IDUN the analysis in this thesis would necessarily have been more limited, as it would have made it necessary to be more "economical" with running NMS-LOC. Even so, one possible criticism of this work is that NMS-LOC configuration files (hence configs) are tested using five to ten evolutionary runs per experiment per config. Although doing a hundred or a thousand runs for each config would provide more statistically significant results, it is not possible with the available computational resources.

4.1.4 Advice for future tech stacks

Using a lower-level programming language could make it possible to write faster code, but this requires a development team that is experienced with a lower-level language, which was not available for this thesis project. Further, higher level languages typically lend themselves to quicker prototyping by being simpler to write and debug. Compromises could be prototyping the NMS system in a higher-level language and then porting it, or writing some parts of the software such as

[2] `https://www.cplusplus.com/`

[3] `https://cython.org/`

[4] `https://www.hpc.ntnu.no/`

the neuron engine or the CGP engine in a lower-level language and then calling it from a higher-level language as a library.

Computational clusters such as cloud computing should be used if available, unless the implemented NMS software is very efficient.

It may be desirable to consider how parallelization can be used during the design phase. For example, if one can design a system which is able to offload a large degree of the computation to a GPU or other Single Instruction Multiple Data computational systems it may provide greatly increased efficiency. Sometimes leveraging efficiency for computational search can be more useful than clever design (as argued by Sutton (2019)).

4.1.5 Advice for handling bugs and errors

The NMS software package is a relatively large software package. The interaction and behavior of the software is by design complex, and there is a great amount of different possible inputs the the various functions in the code. As such there is also a great possibility for bugs in the code.

During the development of the code it was decided to not use unit testing or test-driven development[5] [6] paradigms. Although test driven development can help detect bugs quickly, it also introduces a overhead in writing and re-writing test upon changing the code. As the NMS code-base is experimental it has been through many changes and iterations, and rigorous unit testing would therefore require rigorous test-rewriting which may have constituted an overhead larger than the time saved.

Instead, the codebase uses exceptions and warnings to detect and handle incorrect behavior in select parts of the code. The advantage of this approach is that due to the large amount of possible program states it would be very difficult to reach 100% data flow coverage using unit testing anyway. Instead, the continuous checking for invalid inputs/states in runtime effectively tests the codebase over the effective input space to the various functions. The disadvantage is that there are some errors which could have been detected using unit testing by testing for expected edge cases/errors. The disadvantage is that error checking introduces a runtime overhead, and that thrown errors in runtime introduces overhead in the form of cancelled runs.

Therefore, a better testing paradigm for the development of NMS and NMS-like systems could be to combine partial unit-test coverage in sections of the codebase deemed unlikely to go through changes, in order to detect errors in these parts as quickly as possible, in particular by testing edge cases. Additionally, by raising exceptions and warnings at various points of the code the debugging of errors is greatly simplified, as when errors occur during the programs runtime the source of the error is far clearer from a deliberately raised exception than waiting

[5] https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/

[6] https://www.agilealliance.org/glossary/tdd/

for the Python runtime to crash due to invalid use of base library functions caused by earlier unexpected and undetected errors. In larger development teams (than just one person) pair-programming or code-review processes could also help detect and correct bugs.

Finally, NMS systems should support gathering and presenting statistics about runs of the system. First of all, gathering statistical knowledge makes it possible to discuss the system in an analytically meaningful way. NMS systems may also be difficult to analyze without gathering statistics, as it is not necessarily easy to understand or explain program behavior otherwise without spending great deals of time analyzing various program traces. Finally, gathering statistics helps detect bugs and issues in the software, which is particularly important as NMS systems may need to go through several design iterations. An overview of the gathered statistics is shown in Appendix E.

For example, in NMS-LOC a bug in the code prevented the system from descending the gradient even when it was able to find better solutions, but by gathering statistics it was easy to identify and correct this problem. Figure 4.1 shows the relevant statistical graphs. First, the fitness graph makes it is clear that the system is not able to improve the fitness of the population. Further, the ID Shannon entropy graph shows that a single genotype is able to overtake the entire population over time. Then, the graph showing the percentage of child genotypes which are a neutral or a better change from a parent genotype shows that the systems should actually be improving it's fitness, or at having population slots overtaken by child genotypes. Finally, the rightmost graph shows that population slots are not being overtaken as they should be according to the previous graph. As such it was clear that the systems error was in the child-parent swapping mechanism.

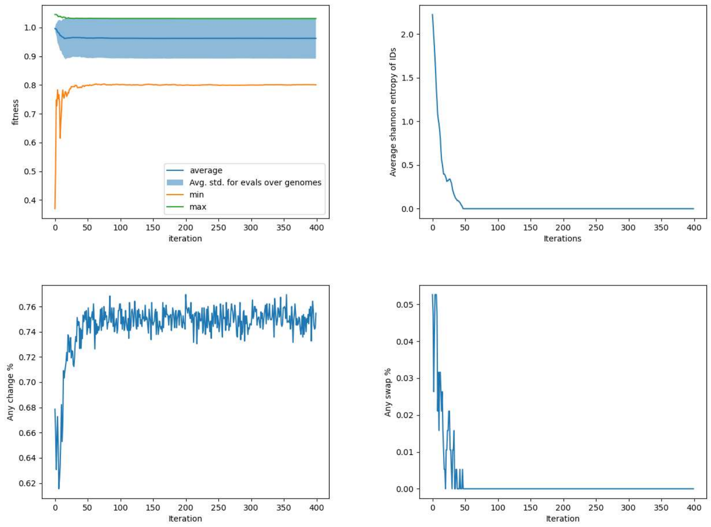


Figure 4.1: From top left to bottom right: Fitness over time, Shannon entropy of distribution of ID's in population, chance of a child genotype being an improvement or neutral change, percentage of times a child genotype takes over a population slot.

Chapter 5

Experiments and Discussion

The following section documents a series of experiments done using NMS-LOC by presenting the experiment, and discussing a selection of the statistical results depending on the intent of the experiment. The discussions are then used to change aspects of NMS-LOC's design if it seems that parts of the design are not functional, compare different settings, and discuss properties of NMS-LOC. As such some changes to the system are done during the course of the experiments. The experiments are presented in chronological order and the changes made likewise. Further, the statistical logging functionality of NMS-LOC is developed as experiments are conducted, as the results of experiments made it clear if new aspects of the system would be desirable to investigate using statistical logging.

The following section uses gathered statistics from NMS-LOC runs. It should be clear what each statistic means from the discussion, but for a complete reference see Appendix E. Note two things regarding statistics: First, not every statistic is presented or discussed for each run; for the sake of brevity only statistics which are deemed interesting for the discussion is presented. Secondly, note that not every statistic is gathered for each run as the statistics gathering is developed along with the experiments as it becomes evident that gathering other types of data is desirable; as such some data which may seem desirable in an Experiment may also be missing simply because the functionality was not implemented at that time.

Please note that in the following experiments the optimization objective is minimizing fitness, i.e. getting the fitness score as low as possible, where 0 is an perfect performance. Note also that some experimental runs are cancelled due to computation time exceeding the allocated computation time on IDUN (maximal 170 hours per job with default user privileges). Cancelled runs could have things in common which cause them to take longer to compute, but there is no evidence to conclude that this is the case. It may also just be due to random chance and particularities of IDUN scheduling and computation speeds of different cores on the cluster. Usually 5 runs are scheduled per config, but the experiment are sometimes rerun to gather more statistics, giving a total of 10 started runs per config.

5.1 Experiment 1 - Growing networks capable of sending signals

Early on in the thesis work NMS-LOC was evaluated for its ability to solve the one-pole balancing problem. However, due to an error in the fitness function, the genomes were given the highest possible fitness score for each timestep in the problem domain where the genomes phenotype successfully sent an output signal, regardless of if the pole fell during that timestep or not (see Equation 5.1 and 5.2). However, this was to some degree a happy accident: The problem in experiment 1 is effectively if NMS-LOC is able to form a connection to an output node and transmit a signal, which can be viewed as the most simple problem possible. As such these results are suitable for discussing the properties of NMS-LOC in the most basic case.

$$fitness_t(action) = \begin{cases} 0, & \text{if output signal is not None.} \\ 1, & \text{otherwise.} \end{cases} \tag{5.1}$$

$$\Sigma^{t \in \text{timestep}}(\text{fitness}_t) \tag{5.2}$$

The results show that NMS-LOC is capable of connecting the initial hidden neuron to the output neuron, and transmitting a signal. In 32 out of 32 runs NMS-LOC was able to evolve a genotype that could transmit a signal, and spread that genotype throughout the population. It should be noted that the ancestor check for historic bests as discussed in Section 3.1.3 was not enabled during these runs due to a bug. The used CGP node functions are listed in table **??** in the Appendix, and the config files are listed in Appendix C. For this experiment ten runs were done for each of the four used config files.

Figure 5.1 shows the fitness over all runs for the NMS-LOC system, and the fitness improvement means that NMS-LOC was able to learn to form connections to the output node, and successfully send signals to the output. Further, the figure shows that on average NMS-LOC is able to follow a gradient in the state space of this simple problem. Even the worst genome with the highest fitness was able to descend the gradient over time. One of the advantages of CGP is that it's neutral drift make it possible to make large steps in the state space when a block of inactive nodes is made active (Julian F. Miller (2020)), which may explain why even the worst-case across all runs got better over time. The developmental encoding used in NMS-LOC may also have helped, as small changes could potentially compound to larger differences in the final phenotype. Either way, the results show that NMS-LOC is able to find a gradient and learn to send signals to the output node.

It may seem strange that the minimal fitness sometimes increases in the fitness graph in Figure 5.1. This is likely due to the use of randomness in NMS-LOC, which means that a genome may have a statistically unlikely good fitness on it's first evaluation. On subsequent evaluations the fitness of the genome is the average of its finesses from each run, and as such the minimum fitness may appear to increase, while it really is just being made more accurate in terms of the actual

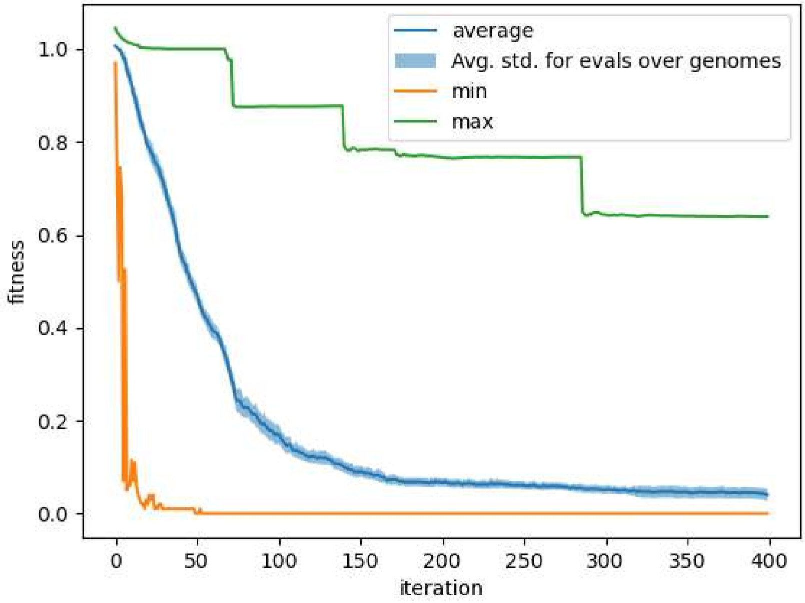


Figure 5.1: Average fitness across runs for all configs (as shown in Appendix C).

"underlying" fitness. Looking into similar fitness graphs for individual evolutionary runs shows similar noise in the fitness over time.

Figure 5.2 graphs the neuron count in the phenotypes over all runs and shows that the evolved NMS-LOC phenotypes have a varied amount of hidden neurons. The high standard deviance means that NMS-LOC was able to find good phenotypes with few and many neurons. One could expect that this fits well with the amount of unique neurons connected to output neurons graphed in Figure 5.3 and the amount of unique neurons connected to input neurons graphed in Figure 5.4, because the figures also have high standard deviation, which could be interpreted as the high neuron count phenotypes also having a lot of unique connections. However, the correlation matrix in Figure 5.5 shows the correlation between the tracked statistics (see Appendix E) and the correlation between neuron count and unique input/output connections tells a different story: In actuality, neuron amount is weakly negatively correlated with the amount of unique output connections, and has approximately zero correlation with the amount of unique input connections.

The correlation matrix also shows that all activation functions help minimize the fitness. Further, it seems like SINU is somewhat different than the other activation functions, as it is strongly correlated with a minimization of maximal fitness, while others are more strongly correlated with minimizing minimum fitness. This is likely because the other activation functions have a positive correlation, mean-

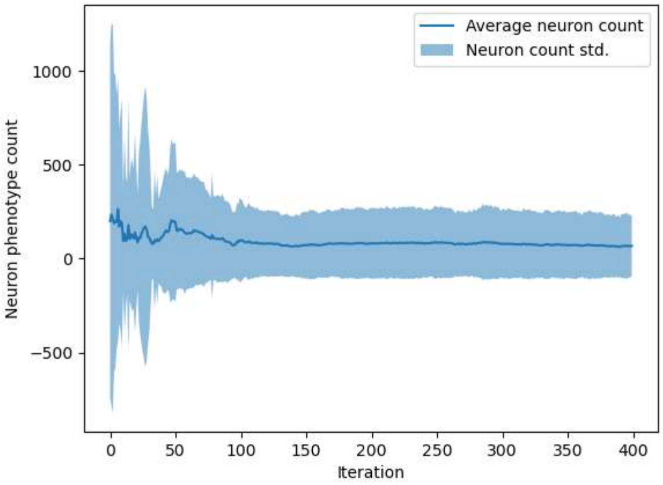


Figure 5.2: Neuron count in phenotypes across all runs for all configs

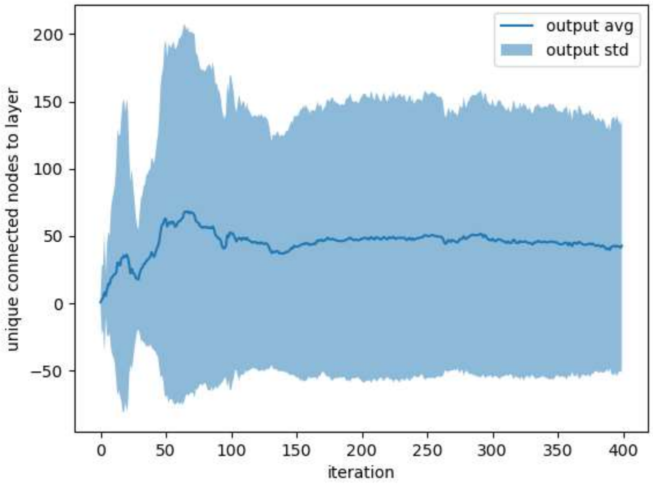


Figure 5.3: Unique hidden neurons connected to output neuron across all runs for all configs

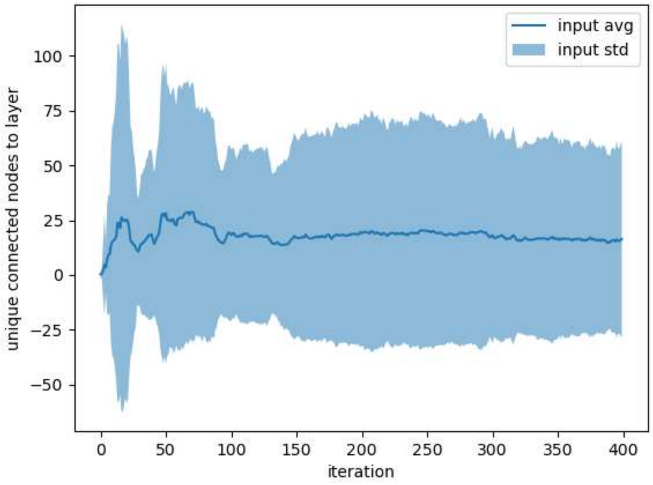


Figure 5.4: Unique hidden neurons connected to input neurons across all runs for all configs

ing that they are often used together, while SINU has a negative correlation with every other activation function than itself. This means that the other activation functions have a correlation with stochastic behavior caused by GAUSS, while SINU is usually used in more deterministic genotypes. Stochastic behavior can be expected to both make the minimum fitness lower, and the max higher, as compared to deterministic chromosomes, due to the introduction of variance. This matches with SINU also greatly reducing the standard deviance of fitness.

Figure 5.6 shows the average amount of connections from hidden neurons to the output neuron. Interestingly, this number is very high. This seems strange, because the output neuron simply outputs the last received value at the end of the processing period for each sample. An hypothesis for why this may occur is that it is an effect of the use of randomness. When using GAUSS and due to the inherent randomness of the neuron engine there can exist for some genomes a probability that a signal is successfully transferred. One way to ensure that a signal gets through is therefore to form many connections, and attempting to send the signal many times. Further evidence supporting this hypothesis could be gathered by removing randomness, and observing if the output connectivity decreases. Certainly, this high of an output connectivity is problematic, as it would require more actions than the neuron engine is alloted per sample to transmit a signal over all of these connections. When the action queue of the neuron engine is initialized, every neurons action controller is added to the queue. This could make it possible for NMS-LOC to learn solutions which do not react to input signals, and instead relies on the action controllers to establish connections and generate signals. Indeed, experimental data from later experiments also suggests that this is the case

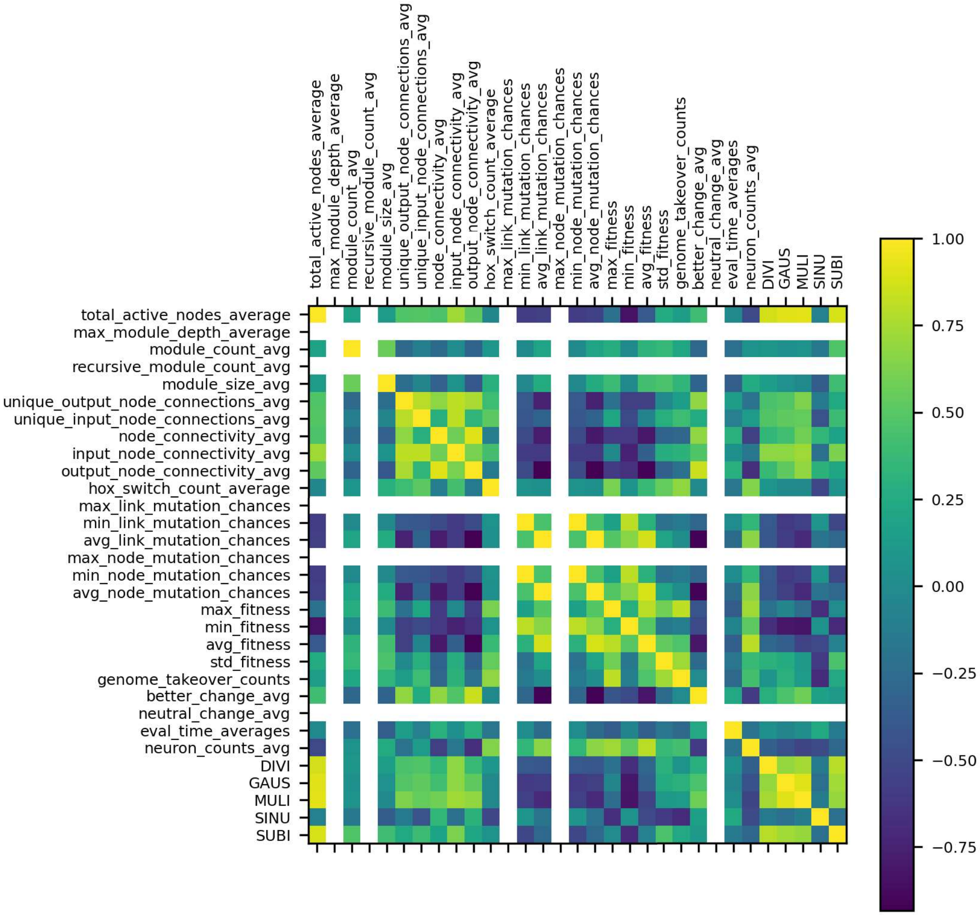


Figure 5.5: Correlation Matrix of tracked statistics for all runs for all config files

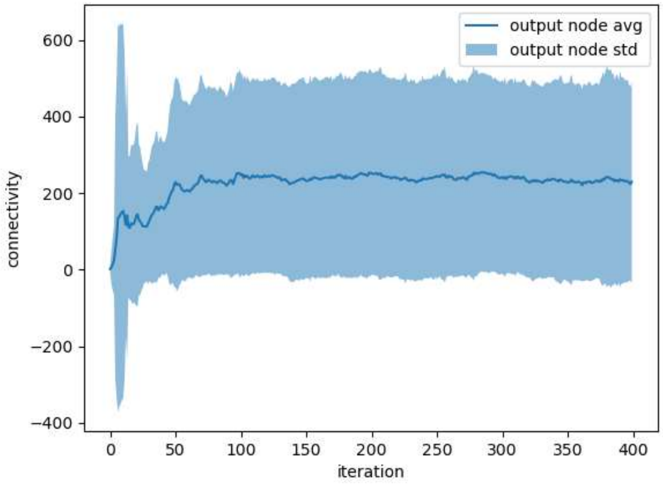


Figure 5.6: Average amount of hidden neuron connections to output neuron for all runs for all configs.

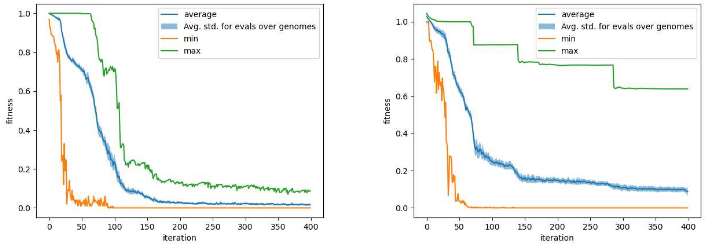


Figure 5.7: Fitness over time for the most complex settings without additional penalty terms (See Section C.1) (left) vs with additional penalty terms (right)

here.

By comparing the fitness graphs of running NMS-LOC with and without additional penalty terms it is possible to get a gauge on whether or not smoothened gradients can have a positive effect. As shown in Figure 5.7 the fitness graphs are mostly similar, with the exception of the maximal fitness. It seems that the smoothened gradient actually deteriorated the worst-case convergence rate.

Investigating the use of switching between homeobox functions reveals that this functionality is mostly used in the most complex genotype, without gradient smoothing. The amount of homeobox functions switches are graphed in Figure 5.8. It is not clear why gradient smoothing should have a negative effect on use of homeobox functions, other than that it perhaps biases the search towards simpler genotypes. A possible explanation for why more complex variants of NMS-LOC use

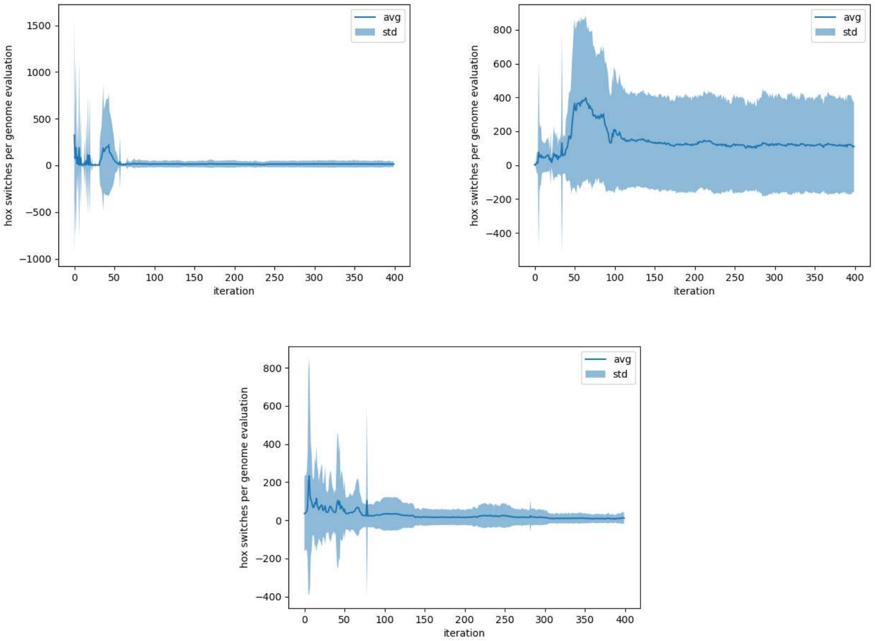


Figure 5.8: Compares the use of switching between homeobox function variants. Config 3 (left) vs Config 4 (middle) vs Config 4 Smoothend Gradient (right). NMS-LOC with Config 4 tends to switch more between homeobox variants.

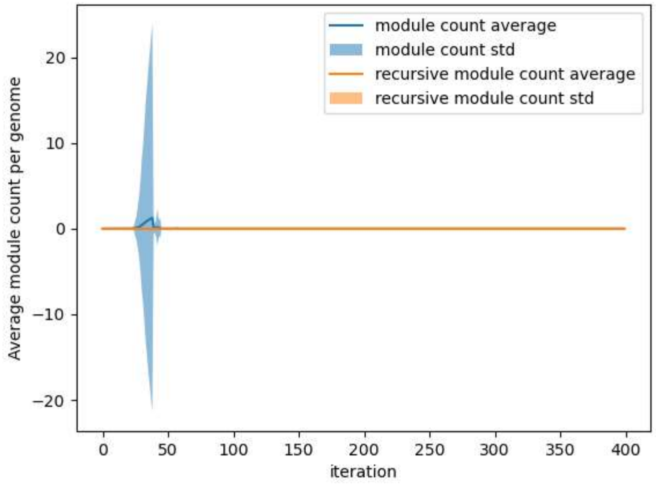


Figure 5.9: Shows the average use of modules in CGP genomes over all runs for all configs. Modules are barely used at all.

homeobox function variants more often is that the more complex variant of NMS-LOC use more internal states and a higher signal dimensionality, which means that there is an larger amount of functions which can be computed, which could be making switching between different functions more useful. Note that config 2 does not contain homeobox variants, and as such never switches between homeobox function variants.

Another interesting result is that the found genotypes do not use modular CGP node functions. Figure 5.9 graphs the average amount of modules and shows that modules are rarely used, and are not used in the high fitness functions at all. One possible reason could be an program error, which put a size requirement on on module size (to avoid a large amount of one-module modules, which happened in preliminary test runs), but also generated suggested modules of an smaller size, which may have made it unlikely to generate a valid module which would get evaluated. This is fixed for further experiments. Another reason could be that the used CGP node functions do not lend themselves well to modular CGP programs. It is also possible that modular CGP is not very useful for this problem, i.e., there is no need for advanced modular functionality.

In conclusion the results show that NMS-LOC is able to form neural networks, and transmit signals of some description. However, due to the simplicity of the problem domain it is not possible to conclude if NMS-LOC in this variant is suited for more advanced problem solving. For example, it is possible that NMS-LOC networks are not able to process information, but instead can only transmit some more or less meaningless signal to the output neurons. The fact that NMS-LOC forms a very large amount of connections to the output neurons suggests that

Iterations/Fitness	*0.01*	*0.03*	*0.05*	*0.08*	*0.10*	*0.12*	*0.14*	*0.18*	*0.25*	*0.50*	*0.75*
1	1.24	3.72	6.20	9.92	12.40	14.88	17.36	22.32	31.00	62.00	93.00
2	1.23	3.69	6.15	9.84	12.30	14.76	17.22	22.14	30.75	61.50	92.25
3	1.22	3.66	6.10	9.76	12.20	14.64	17.08	21.96	30.50	61.00	91.50
5	1.20	3.60	6.00	9.60	12.00	14.40	16.80	21.60	30.00	60.00	90.00
10	1.15	3.45	5.75	9.20	11.50	13.80	16.10	20.70	28.75	57.50	86.25
20	1.05	3.15	5.25	8.40	10.50	12.60	14.70	18.90	26.25	52.50	78.75
50	0.75	2.25	3.75	6.00	7.50	9.00	10.50	13.50	18.75	37.50	56.25

Table 5.1: The inner numbers in the table detail how many times the pole falls (i.e. problem resets) based on how many iterations it takes NMS-LOC to start sending signals and what fitness level the genotype achieves.

this may be the case, due to the use of randomness making the result of actions highly uncertain. If so, it an possible improvement to NMS-LOC could be to move away from randomness - if doing so, it may be desirable to use the CGP node functions used by Julian F. Miller (2021) for the simple reason that they have been shown to be functional in an CGPDNN. Other than that, the results seem to suggest that smoothened gradients may make search worse, and that the most complex version of NMS-LOC (Config 4) may be able to find more interesting behaviors, such as using homeobox variants. The next experiment conducted on the One-Pole Balancing problem domain will help provide more conclusive results regarding these issues.

5.2 Experiment 2

The following section details Experiment 2. It is divided into three subsections: Preamble, which discusses changes from Experiment 1 and how the fitness scores should be interpreted, and in Experimental Run 1 & 2. Experimental Run 2 is a repeat of Experiment Run 1 due to a a found bug, however, the analysis and discussion in Experimental Run 1 still applies and to a large degree concurs with the results in Experimental Run 2. As such Experimental Run 2 can be read as a continuation of Experimental Run 1.

5.2.1 Preamble

Experiment 2 is almost identical to Experiment 1. The fitness measure is fixed, and the statistical logging functionality is further developed based on which areas seemed to warrant more investigation. Experiment 1 was useful for continuing the development of statistical logging, as analyzing the results revealed which parts of NMS-LOC required more data to analyze. Further, the signal sending behaviour of NMS-LOC is tweaked. Previously, it has been possible for neurons to send signals backwards through connections, however, this was removed. Although the capability of sending signals backwards might be useful, the implementation had several issues:

- NMS-LOC used the same evolved programs to process information moving forwards and backwards. Although this might produce useful behaviour, separate programs for signal directional should necessarily contain more useful genotypes in the search space due to containing every identical program set and all others.
- NMS-LOC functions had limited functionality for generating signals to send backwards.
- Sending signals backwards uses the limited amount of actions NMS-LOC has to process training samples, which could make it harder to find useful solutions. Currently, sending a signal takes 6-7 actions and is a relatively expensive operation time-wise.

For these reasons sending signals backwards through connections was disabled entirely. Potential fixes are discussed in Section 3.2, but would in short require extensive design revisions, increasing the search space, or more computational power than available. Therefore, it was deemed prudent to investigate if NMS-LOC functions without backward signaling. Do note that NMS-LOC can still send signals backwards through recurrent connections.

$$fitness_t(action) = \begin{cases} 0, & \text{if output signal is not None and pole does not fall on timestep.} \\ 1, & \text{otherwise.} \end{cases} \tag{5.3}$$

$$\Sigma^{t \in \text{timestep}}(\text{fitness}_t) \tag{5.4}$$

Experiment 2 uses the fitness measure shown in Equations 5.3 and 5.4. The objective is still minimization. To provide a better intuitive understanding of the fitness measure, consider Table 5.1. The table maps how many iterations (i.e. problem samples) an NMS-LOC genotype needs to start sending signals and the achieved fitness to how many times the pole balancing problem resets (assuming runs of 100 problem samples, as used in the configs). Taking a random action results in a fitness of about 0.102 (found by Monte-Carlo estimation), which can be used as a reference for the achieved fitness values; that is, if a genotype has an fitness adjusted for the iterations it uses to establish a signal sending network lower than 0.102 it implements a policy that is better than taking random actions, meaning that the NMS-LOC phenotype implements some meaningful behavior. It should be noted that the establishment time is actually the amount of one-pole balancing timesteps which do not get an output from the NMS-LOC network, which doesn't strictly speaking have to happen only during the establishment of the network, as the time of these events are not tracked. However, the comparison still holds to some degree, as an adjusted fitness lower than 0.102 indicates that successful processing of a sample implements a better-than-random policy, especially if the non-adjusted fitness is also lower than 0.102.

Value	0.092	0.082	0.052	0.022	0.0
Probability	≈ 0.33%	≈ 0%	≈ 0%	≈ 0%	≈ 0%

Table 5.2: The probability of getting a fitness equal or lower to value when sampling a random policy four times. Estimated using Monte Carlo estimation.

However, it should be noted that the standard deviation of the random policy is approximately 0.0038 (Estimated by Monte Carlo estimation). Because genotypes are evaluated for single runs, it is possible that seemingly high-performing genotypes could simply be implementing random or random-like policies. Because the deviation for random policies is high, and because each genotype in NMS-LOC has four children, the system may present genotypes which implement random policies as having fitnessess lower than 0.102, simply because as the amount of samples taken from a Gaussian with mean 0.102 and standard deviation 0.0038 increases it becomes likely that some of the samples will score lower than 0.102. However, this effect would only become noticeable as the amount of children per genotype is high enough, which may not be the case in NMS-LOC, which only produces four children per pair of genotypes. To check if this could influence experimental results, Monte Carlo estimation is done to check the probability of consistently getting lower scores for child genotypes of genotypes implementing the random policy (assuming child genotypes also implement the random policy). The results are shown in Table 5.2, and indicate that it is highly unlikely that this effect should consistently lower the fitnesses of the population to values much lower than 0.102 if the produced phenotypes simply implement the random policy. It is possible that NMS-LOC could find other policies with higher standard deviance when sampling the fitness random variable, in which case the effect or similar effects could be impactful, but in this case NMS-LOC still finds policies which are different than the standard random policy.

In the following discussion reference to statistics will be made using normal distributions. These are calculated by taking the statistical values for the last 50 timesteps and assuming these are approximately normally distributed, which is reasonable under the Central Limit Theorem, especially as these numbers will mostly be used to present the results in a more abbreviated manner than using graphs for each thing discussed.

5.2.2 Experimental Run 1

To start with the average performance of NMS-LOC over all configs will be discussed. 38 runs out of 40 started runs were successfully completed. Their fitness graphs are shown in Figure 5.10, which show that NMS-LOC is able to find gradients and converge to low fitness values, both the average fitness and the highest fitness over all runs at each timestep decrease over time. The minimal fitness decreases quite quickly. As seen on the rightmost figure, the average corrected fitness is lower than 0.102, meaning that on average NMS-LOC is able to find solutions

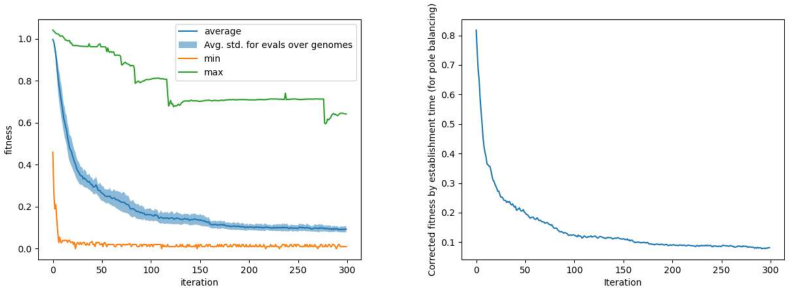


Figure 5.10: Shows the average fitness over all configs for Experiment 2. To the left are unadjusted fitness stats, to the right is the average fitness adjusted for establishment time.

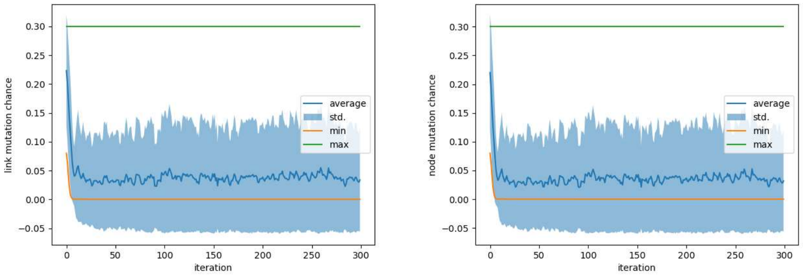


Figure 5.11: The average, min and max, respectively, link mutation chance and node mutation chance for each timestep for all configs combined.

slightly better than a random policy.

Figure 5.11 shows the link and node mutation rate over time for NMS-LOC over the different config files. These figures can be used to evaluate the mutation rate adaptation scheme. As expected some of the genotypes have the maximal hypermutation rate at 30%. On average it seems that the mutation rate balances around 5% for link mutation and 2.5% for node mutation, due to link mutation being initialized at twice the value of node mutation. At minimum the mutation rate goes low, but is still over the hypermutation cut-off at 0.01% node mutation. This could mean that a low but good mutation rate is found, but it may also mean that the mutation rate is sufficiently low that NMS-LOC is producing child genotypes identical to the parent, in which case the mutation rate should remain approximately constant with some fluctuations due to sampling randomness. To remedy this potential issue, NMS-LOC is changed to include a small decay factor is added to the mutation rate if the genotype has the same fitness as the parent, such that it should eventually enter hypermutation.

Figure 5.10 graphs average fitness over time and shows that the fitness of genotypes tend to converge to a low value, and then "jitter" around that value, with little further improvement. As pointed out in the discussion of Experiment 1 it could be that this is an unfortunate effect of using randomness. Figure 5.13 graphs the percentage of child genomes which have a neutral or better fitness than at least one of their parent genomes shows that it is far more likely to produce a child genotype which is better than the parent than one that is neutral. Due to the usage of randomness the fitness of a genotype is itself a random variable, and due to children being evaluated using a single sampling of the genotype fitness random variable the observed sample fitness will sometimes be an outlier on the side of lower fitness than the actual unobserved mean. As fitness improves the potential improvements in fitness that can be made from a step in the state space may also become small, for example, if each step can improve fitness by 1% in the optimal case, the absolute value of the improvement will converge to zero. If however the variance of the sampled genotype fitness random variables remains constant or scales down more slowly, then it may be possible that the gradient is not observable due to the noise of the genotype fitness random variables.

To determine if the gradient could be disappearing a Monte Carlo simulation is run. The simulation is run for different standard deviances, different sizes of better steps and worse steps in the fitness landscape assuming the parent genotype has a fitness of 0.09[1], and for different percentages of better child genotypes. Some simplifying assumptions are made. For one, it is assumed that good and bad solutions are distributed using a normal variable with a mean equal to 0.09 +/- the step size. As the observed amount of child genotypes with finesses sampled lower than the parent tends to be around 20% (see Figure 5.13), different percentages of children with a better mean then the parent are evaluated for percentages lower than 20%. Finally, it assumes that the remaining child genotypes are split fifty-fifty between neutral (i.e. same fitness) and worse than the parent. When referring to child genotypes as better/neutral / worse it is done in reference to the underlying actual mean of the fitness random variable, which is not observed. The simulation then tests how "obscured" the gradient becomes under different parameters. In effect it may be possible that the gradient descent gets stuck in a "noise swamp"[2], where it can not improve even though there is a gradient present in the fitness landscape.

The results of the simulation are shown in Tables **??**, **??**, **??**, **??**, and **??**. These tables show two interesting things. First, the actual chance of choosing a better child is lower than in the deterministic case across the board, and gets progressively lower as the standard deviation increases. Secondly, there is a significant

[1] 0.09 is chosen as it is the average fitness observed in Experiment 2 Run 1 in the last 50 iterations. Note also that the observed standard deviance was 0.003, higher standard deviance are tested to examine the concept of "noise swamps", and because the standard variance of the average fitness may be different than the standard variance of the genotype fitness random variables. Samples are capped to a minimum value of 0.0, and it is necessary for the better child to have a strictly lower fitness to count as being chosen correctly.

[2] I wouldn't be surprised if similar analysis or concepts have been discussed before, but I do not believe that I have personally encountered discussion of "Noise Swamps" or equivalent. In either case it was desirable to get some numerical results specific to the parameters in NMS-LOC.

Fitness improvement, std	9.00E-06	9.00E-05	9.00E-04	9.00E-03	0.02
3.00E-03	7.60E-04 / 0.50 / 0.50	1.44E-03 / 0.51 / 0.49	1.64E-03 / 0.58 / 0.42	3.68E-03 / 0.92 / 0.08	4.00E-03 / 0.93 / 0.07
0.05	1.44E-03 / 0.50 / 0.50	1.04E-03 / 0.49 / 0.51	7.60E-04 / 0.51 / 0.49	1.36E-03 / 0.54 / 0.46	2.24E-03 / 0.62 / 0.38
0.10	7.60E-04 / 0.46 / 0.54	9.20E-04 / 0.46 / 0.53	1.04E-03 / 0.46 / 0.54	1.04E-03 / 0.49 / 0.51	1.16E-03 / 0.53 / 0.47
0.50	3.20E-04 / 0.33 / 0.67	4.00E-04 / 0.32 / 0.67	3.20E-04 / 0.33 / 0.67	5.20E-04 / 0.33 / 0.67	5.20E-04 / 0.34 / 0.66
1.00	3.60E-04 / 0.30 / 0.70	2.80E-04 / 0.31 / 0.69	3.60E-04 / 0.30 / 0.70	2.40E-04 / 0.31 / 0.69	3.20E-04 / 0.31 / 0.69
1.50	2.80E-04 / 0.30 / 0.70	3.20E-04 / 0.30 / 0.70	4.00E-04 / 0.30 / 0.70	1.20E-04 / 0.30 / 0.70	4.40E-04 / 0.30 / 0.70
2.00	2.80E-04 / 0.30 / 0.70	1.60E-04 / 0.30 / 0.70	1.20E-04 / 0.30 / 0.70	4.80E-04 / 0.30 / 0.70	6.00E-04 / 0.30 / 0.69
5.00	1.60E-04 / 0.28 / 0.72	4.40E-04 / 0.29 / 0.71	2.40E-04 / 0.29 / 0.71	1.60E-04 / 0.29 / 0.71	2.80E-04 / 0.29 / 0.71
10.00	3.20E-04 / 0.28 / 0.72	3.60E-04 / 0.29 / 0.71	1.60E-04 / 0.29 / 0.71	2.80E-04 / 0.29 / 0.71	4.00E-04 / 0.29 / 0.71

Table 5.3: Percentage of respectively actually good steps / neutral steps / worse steps in state space for different variances and step sizes if 0.001 of the child genotypes are better than the parent. For reference, the actual percentage of the time a deterministic gradient descent algorithm could descend the gradient is the amount of children times the chance of a child being better (0.004)

Fitness improvement, std	9.00E-06	9.00E-05	9.00E-04	9.00E-03	0.02
3.00E-03	0.01 / 0.49 / 0.50	9.92E-03 / 0.51 / 0.48	0.02 / 0.57 / 0.42	0.04 / 0.89 / 0.07	0.04 / 0.90 / 0.06
0.05	8.88E-03 / 0.49 / 0.50	9.68E-03 / 0.49 / 0.50	9.80E-03 / 0.49 / 0.50	0.01 / 0.54 / 0.45	0.02 / 0.61 / 0.38
0.10	9.44E-03 / 0.46 / 0.54	8.48E-03 / 0.46 / 0.53	8.84E-03 / 0.46 / 0.53	9.84E-03 / 0.48 / 0.51	0.01 / 0.52 / 0.47
0.50	4.72E-03 / 0.33 / 0.67	4.72E-03 / 0.33 / 0.66	4.48E-03 / 0.33 / 0.67	3.56E-03 / 0.34 / 0.66	4.16E-03 / 0.34 / 0.65
1.00	3.52E-03 / 0.31 / 0.69	3.40E-03 / 0.31 / 0.69	3.24E-03 / 0.31 / 0.69	3.80E-03 / 0.31 / 0.69	3.00E-03 / 0.31 / 0.68
1.50	3.80E-03 / 0.30 / 0.70	3.04E-03 / 0.30 / 0.69	3.28E-03 / 0.30 / 0.69	3.48E-03 / 0.31 / 0.69	3.92E-03 / 0.30 / 0.69
2.00	3.44E-03 / 0.29 / 0.70	4.16E-03 / 0.30 / 0.70	3.36E-03 / 0.29 / 0.71	3.04E-03 / 0.30 / 0.70	3.56E-03 / 0.30 / 0.70
5.00	2.76E-03 / 0.29 / 0.71	2.88E-03 / 0.29 / 0.71	3.20E-03 / 0.29 / 0.71	2.92E-03 / 0.29 / 0.71	3.84E-03 / 0.30 / 0.70
10.00	3.32E-03 / 0.28 / 0.71	3.48E-03 / 0.29 / 0.71	2.80E-03 / 0.29 / 0.71	3.96E-03 / 0.29 / 0.71	3.28E-03 / 0.29 / 0.71

Table 5.4: Percentage of respectively actually good steps / neutral steps / worse steps in state space for different variances and step sizes if 0.01 of the child genotypes are better than the parent. For reference, the actual percentage of the time a deterministic gradient descent algorithm could descend the gradient is the amount of children times the chance of a child being better (0.04)

Fitness improvement, std	9.00E-06	9.00E-05	9.00E-04	9.00E-03	0.02
3.00E-03	0.05 / 0.47 / 0.47	0.05 / 0.48 / 0.47	0.08 / 0.53 / 0.39	0.18 / 0.75 / 0.06	0.19 / 0.76 / 0.05
0.05	0.05 / 0.48 / 0.47	0.05 / 0.47 / 0.48	0.05 / 0.48 / 0.47	0.06 / 0.52 / 0.42	0.09 / 0.56 / 0.35
0.10	0.04 / 0.44 / 0.52	0.04 / 0.44 / 0.52	0.04 / 0.44 / 0.51	0.05 / 0.46 / 0.49	0.06 / 0.50 / 0.44
0.50	0.02 / 0.33 / 0.65	0.02 / 0.33 / 0.65	0.02 / 0.33 / 0.65	0.02 / 0.33 / 0.64	0.02 / 0.34 / 0.63
1.00	0.02 / 0.31 / 0.67	0.02 / 0.31 / 0.67	0.02 / 0.31 / 0.67	0.02 / 0.31 / 0.67	0.02 / 0.32 / 0.66
1.50	0.02 / 0.30 / 0.68	0.02 / 0.31 / 0.68	0.02 / 0.30 / 0.68	0.02 / 0.30 / 0.68	0.02 / 0.31 / 0.67
2.00	0.02 / 0.30 / 0.68	0.02 / 0.30 / 0.68	0.02 / 0.30 / 0.68	0.02 / 0.30 / 0.68	0.02 / 0.31 / 0.67
5.00	0.02 / 0.29 / 0.69	0.02 / 0.30 / 0.69	0.02 / 0.30 / 0.69	0.02 / 0.30 / 0.68	0.02 / 0.30 / 0.69
10.00	0.02 / 0.29 / 0.69	0.02 / 0.29 / 0.69	0.02 / 0.30 / 0.68	0.02 / 0.29 / 0.69	0.02 / 0.29 / 0.69

Table 5.5: Percentage of respectively actually good steps / neutral steps / worse steps in state space for different variances and step sizes if 0.05 of the child genotypes are better than the parent. For reference, the actual percentage of the time a deterministic gradient descent algorithm could descend the gradient is the amount of children times the chance of a child being better (0.2)

Fitness improvement, std	9.00E-06	9.00E-05	9.00E-04	9.00E-03	0.02
3.00E-03	0.10 / 0.45 / 0.45	0.11 / 0.45 / 0.44	0.14 / 0.50 / 0.36	0.34 / 0.61 / 0.05	0.34 / 0.62 / 0.04
0.05	0.10 / 0.45 / 0.45	0.10 / 0.45 / 0.45	0.10 / 0.45 / 0.45	0.13 / 0.48 / 0.39	0.16 / 0.51 / 0.32
0.10	0.09 / 0.42 / 0.49	0.09 / 0.42 / 0.49	0.09 / 0.42 / 0.49	0.10 / 0.44 / 0.47	0.11 / 0.47 / 0.42
0.50	0.05 / 0.33 / 0.63	0.04 / 0.34 / 0.62	0.05 / 0.33 / 0.62	0.05 / 0.33 / 0.62	0.05 / 0.35 / 0.60
1.00	0.04 / 0.32 / 0.64	0.04 / 0.31 / 0.65	0.04 / 0.31 / 0.65	0.04 / 0.32 / 0.64	0.04 / 0.32 / 0.64
1.50	0.04 / 0.31 / 0.65	0.04 / 0.31 / 0.65	0.04 / 0.31 / 0.65	0.04 / 0.31 / 0.65	0.04 / 0.32 / 0.64
2.00	0.04 / 0.30 / 0.66	0.04 / 0.31 / 0.65	0.04 / 0.30 / 0.66	0.04 / 0.31 / 0.66	0.04 / 0.31 / 0.65
5.00	0.04 / 0.30 / 0.66	0.03 / 0.30 / 0.67	0.04 / 0.30 / 0.67	0.04 / 0.31 / 0.66	0.04 / 0.31 / 0.66
10.00	0.04 / 0.29 / 0.67	0.03 / 0.30 / 0.67	0.04 / 0.30 / 0.66	0.03 / 0.30 / 0.66	0.03 / 0.30 / 0.67

Table 5.6: Percentage of respectively actually good steps / neutral steps / worse steps in state space for different variances and step sizes if 0.1 of the child genotypes are better than the parent. For reference, the actual percentage of the time a deterministic gradient descent algorithm could descend the gradient is the amount of children times the chance of a child being better (0.4)

Fitness improvement, std	9.00E-06	9.00E-05	9.00E-04	9.00E-03	0.02
3.00E-03	0.20 / 0.40 / 0.40	0.21 / 0.41 / 0.39	0.28 / 0.41 / 0.30	0.58 / 0.39 / 0.03	0.60 / 0.38 / 0.02
0.05	0.20 / 0.40 / 0.40	0.20 / 0.40 / 0.40	0.20 / 0.40 / 0.40	0.24 / 0.42 / 0.34	0.31 / 0.43 / 0.26
0.10	0.17 / 0.39 / 0.44	0.18 / 0.38 / 0.44	0.18 / 0.38 / 0.44	0.20 / 0.40 / 0.40	0.23 / 0.41 / 0.36
0.50	0.10 / 0.33 / 0.57	0.10 / 0.33 / 0.57	0.10 / 0.33 / 0.57	0.10 / 0.33 / 0.57	0.10 / 0.35 / 0.55
1.00	0.09 / 0.32 / 0.59	0.09 / 0.32 / 0.60	0.09 / 0.32 / 0.59	0.09 / 0.32 / 0.60	0.10 / 0.32 / 0.58
1.50	0.09 / 0.32 / 0.60	0.09 / 0.31 / 0.60	0.09 / 0.32 / 0.60	0.09 / 0.31 / 0.60	0.09 / 0.32 / 0.59
2.00	0.08 / 0.32 / 0.60	0.08 / 0.31 / 0.61	0.09 / 0.31 / 0.60	0.09 / 0.31 / 0.60	0.08 / 0.31 / 0.60
5.00	0.08 / 0.31 / 0.61	0.09 / 0.31 / 0.61	0.08 / 0.31 / 0.61	0.08 / 0.31 / 0.61	0.08 / 0.31 / 0.61
10.00	0.08 / 0.30 / 0.62	0.08 / 0.31 / 0.62	0.08 / 0.31 / 0.61	0.08 / 0.31 / 0.61	0.08 / 0.30 / 0.61

Table 5.7: Percentage of respectively actually good steps/neutral steps/worse steps in state space for different variances and step sizes if 0.2 of the child genotypes are better than the parent. For reference, the actual percentage of the time a deterministic gradient descent algorithm could descend the gradient is the amount of children times the chance of a child being better (0.8)

chance of picking a child genotype which is actually worse which increases with standard deviation. This is more impactful than that the chance of picking good children is lower than in the deterministic case, as in the deterministic case one would never perform gradient ascent accidentally, but based on the Tables there seems to be a significant chance of doing so in random NMS-LOC, and it seems that NMS-LOC may indeed get stuck in "Noise Swamps". This begs the question: How does NMS-LOC manage to decrease fitness at all, and why does not fitness start increasing when Noise Swamps are encountered?

First, although Noise Swamps would interfere with NMS-LOC, the entire state space is likely not an Noise Swamp. The tables show that for low standard deviance areas in the state space the chance of finding the good children is within the same order of magnitude as in the deterministic case, at least while the percentage-wise fitness improvement per step is 1% or better, as one may expect at the beginning of the search. Further, although bad children may be accidentally picked, NMS-LOC has mechanisms to deal with this. NMS-LOC reevaluates genotypes fitness when they are a part of the parent population, meaning that the observed fitness should eventually converge to the actual fitness of the genotype. Further, NMS-LOCs historic best tracking ensures that even if gradient ascent is performed, the information loss is minimal, because the discarded genotypes can be recovered. Indeed, Figure 5.12 graphs the use of historic best swaps and shows that NMS-LOC uses the historic best feature frequently as the fitness converges. One possible explanation for NMS-LOC stalling in the fitness descent may therefore be that when it finds a sufficiently good genotype, it gets stuck in a Noise Swamp. The Historic Bests tracking ensures that it is able to keep the good genotypes, but because the chance of finding new better children is significantly reduced the search gets stuck - In reality, these effects may interact to produce an even worse situation, where several fitness evaluation steps are "wasted" on getting an accurate estimate of worse-mean genotypes, making it even less likely to find a better-mean genotype. Other explanations are also possible, for example, the high-performing genotypes that NMS-LOC finds might simply be local minima which are difficult to escape from in any case. These explanations are also not mutually exclusive; local minima would make it less likely to find children which are better, but the Noise Swamps due to the use of randomness also make it less likely that better children are correctly identified. To further investigate the impact of randomness later experiments will compare versions of NMS-LOC which allow for varying degrees of randomness. It should also be noted that the use of randomness is not necessarily negative; some use of randomness may make the fitness landscape space "more continuous" and "smoother" by allowing gradual transitions by adjusting probabilities between the points which would exist in an equivalent deterministic case, which could make it easier to find gradients up to the point that the search gets stuck in a noise swamp. Randomness also allows for programs which are not possible in the strictly deterministic case.

On average NMS-LOC solutions consist of multiple neurons connected in some network structure. The average amount of neurons in a phenotype was 270.84 ± 13.15, where each neuron had an average of 36.45 ± 2.13 connections. Input neurons typically had 35.7 ± 0.87 connections, while output neurons had an larger average of 296.97 ± 7.25 connections. This may seem promising in the sense that it indicates that NMS-LOC creates intricate networks, but considering that the maximal amount of actions NMS-LOC has to process each sample is 125 the high connection count is worrying. Depending on the network topology it is possible that NMS-LOC phenotypes simply can not transmit a signal from the input to the output in the allotted amount of signals. However, as the timesteps of one-pole

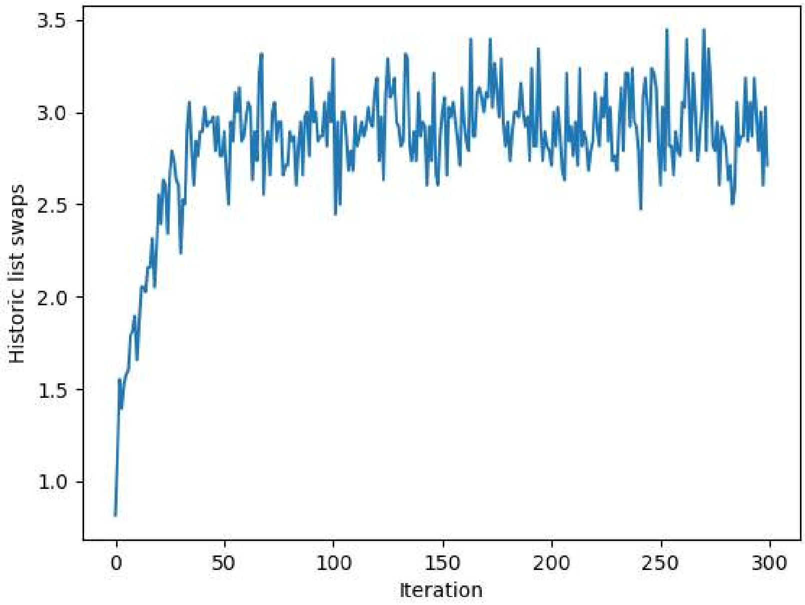


Figure 5.12: Shows the average amount of historic best swaps done over time.

balancing tends to have a direct correlation with the previous it may also not be necessary to react to the current state input in order to output something reasonable. However, with only 135 actions per sample, and with the connectivity of the output neuron being higher than 135, it should in most cases not be possible to for NMS-LOC to transmit a signal at all if the action queue's FIFO order is preserved. Checking if this is the case reveals that a bug permitted some actions to be added to the top of the queue under rare circumstances.

5.2.3 Experimental Run 2

Fixing this error still results in good fitness, giving a minimal fitness of 0.0094 ± 0.0042 before correcting for setup time, and an average fitness across all runs for all configs of 0.0646 ± 0.00174 before correcting for setup time - meaning that in the best case NMS-LOC still performs far better than the random policy, and on average also does so. In Experimental Run 2 26 out of a total of 40 runs were completed. The fitness numbers are better than before the error fixing (0.0963 ± 0.0151 before and 0.0.0646 ± 0.0115 after). It should be noted that this may also be in part due to the change in the mutation rate adaptivity described earlier, as it should in principle help with escaping local minima (in practice this lead to mutation rates for individual runs sometimes spiking, i.e. entering hypermutation and then either exiting hypermutation after finding an gradient or being replaced

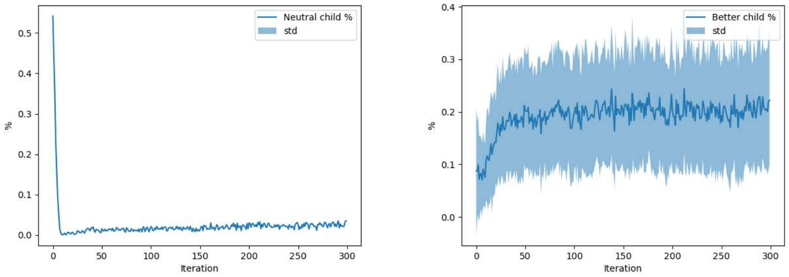


Figure 5.13: The percentage of child genotypes which are respectively as good as the parent or better than the parent.

by another genome). Interestingly however, the neuron connectivity and neuron amount is higher after the bugfix, which likely means that although it is no longer possible to "sneak ahead" in the action queue, it is still possible to use the call to the neuron controller to send signals without necessarily interfacing with inputs. Another significant difference is that after the bugfix there is a on average decrease of approximately 100 active CGP nodes per genome, meaning that the programs that are found are fall smaller. Otherwise the stats are similar, indicating that the preceding discussion of "Noise Swamps" and the impact of randomness is equally applicable. Notably, the changed mutation rate adaption scheme and the bugfix did not change the "fitness wobbling" at convergence. Based on the fact that NMS-LOC tends to converge using all the tested config files further investigation of the smoothened gradients is discarded. Because NMS-LOC is able to descend the gradient without smoothened gradients it is unnecessary to bias the search.

Fixing this bug did not entirely prevent the existence of these programs. The neuron engine sends a special type of signal on death which skips some of the steps which a signal normally would have to go through in the action queue, which might make it possible to send signals rapidly enough even if there is a large amount of neurons and output connections. As long as calls to the neurons action controller is added to the action queue on initialization, such programs can be possible (especially as actions of the same timestep were added first in the queue for that timestep, instead of in a proper FIFO queue per timestep as intended). In future experiments the death signals will still be present, but neuron action controllers won't be added on initialization, and the action queue order is fixed, more on this later.

In the bugfix-version of Experiment 2 data was also gathered logging samples of the evolved programs and produced phenotypes at the end of evolution. To analyze the phenotypes they are categorized according to whether they have none, (<1), single (1), low (<20), medium (< 100) or high neuron counts, input connectivity, hidden connectivity, and output connectivity. It was observed that in general each evolutionary run produced a single phenotype "type" based on the categories above, which may indicate that NMS-LOC is not good at preserving diversity in the population, likely due to the explicit population takeover feature and the implicit population takeover produced by the historical list. Although historical list replacements checks that there aren't common ancestors to a certain depth it is possible that this depth is exceeded quickly, and this check is also not applied to the current population, meaning that genotype "species" can spread by getting put into the historic best list, then into the population, then back into the list, and so on. To attempt to remedy this and preserve diversity better the historic best checks are changed to check for common ancestors up to and including depth 6 from a previous 3, and by also checking the current population.

However, the distinct genotype split between each runs makes it convenient to group the runs by produced phenotype. That makes it possible to examine common statistics for the different types of phenotypes. Note that there can of course exist other important characteristics of phenotypes not captured or represented by this clustering method, but this grouping method is still chosen as it is easy to observe and do using the gathered statistics and captures important and semantically meaningful properties of the phenotypes. For the purposes of this discussion a phenotype is considered interesting if it could be possible for it to be reactive to input signals (as to some degree any high-performing phenotype is interesting otherwise - in general all phenotypes which perform well are equally valid solutions and show that NMS-LOC works, but under the view of using NMS-LOC to find connectionist computational structures these solutions are not necessarily as interesting).

Phenotypes[3] with either high output or high input are unlikely to be interesting, as they likely rely on the neuron engine's action controllers instead of actual input values, along with death signals and relying on the incorrect LIFO-queue for same-timestep actions. Empirically these phenotypes do not have more neuron deaths than on average (87.5 ± 4.0 vs. 213.2 ± 7.3), but they do have more neuron births (2173.6 ± 139.5 vs 842.6 ± 32.6), axon-dendrite deaths (743.3 ± 26.5 vs. 422.2 ± 11.2) and axon-dendrite births (835.8 ± 9.4 vs. 481.4 ± 8.2). It is possible that the birth-functions are more useful than the death-functions for sending signals in time; both neuron creation and axon creation provides axon-dendrites with the opportunity to connect to other axon-dendrites, which could potentially be the ones for the output node, and axon-dendrite birth specifically can generate signals. On the other hand calling functions for creating neurons and axon-dendrites more is necessary to create high-neuron and high-connection phenotypes in the first place, so these statistical differences could also be related to that bias instead of how the programs work.

In general, phenotypes which rely exclusively on signal generation without regard for input signals can be considered another example of how evolutionary AI can find solutions which exploit the simulation engine in unexpected ways (Lehman et al. (2018)).

Phenotypes with no input connectivity are necessarily not interesting as it is

[3] Please note that in following phenotype graph visualizations neurons which are at the same position are randomly spread out around that point to improve the visualization.

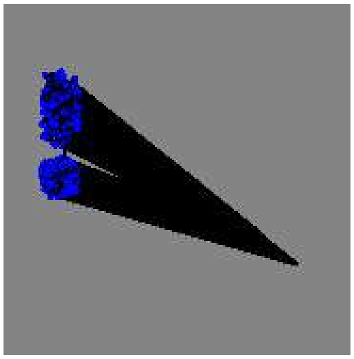


Figure 5.14: An example of a neuron phenotype with high neurons, high output, and high input. Color key (used in all phenotype graphs): Blue - hidden neuron, green - output neuron, red - input neurons, black line - axon-dendrite connection.

not possible for them to consider input signals. They perhaps work similarly to high-neuron high-connectivity phenotypes, in that high-neuron and high-connectivity phenotypes may also be using the signal-generating functions instead of actual signal input.

High neuron phenotypes with low or medium input and output connectivity, and a low hidden neuron connectivity can be interesting. In such phenotypes it can be possible to send signals from input to output for each sample, as neurons could for example "filter" through which neurons fire, effectively doing some form of "if/else" control structure. Another interesting class of policies which could be implemented in such phenotypes would be policies which pick an action for problem instance t+n using data from problem instance t.

High output phenotypes typically look something like Figure 5.14. A large amount of neurons are gathered in close proximity, and most are connected to the output neuron. Alternatively, they may look like Figure 5.15 which depicts a phenotype with just a single neuron - here the single neuron has formed many connections to the output neuron.

Phenotypes with low or medium neurons, input, output and hidden connectivity are likely to be interesting, as they are likely able to process and transmit outputs based on the input signal for the same problem sample. Singular neuron phenotypes can also be interesting and easy to analyze, but ultimately phenotypes with multiple neurons are more interesting from the perspective of connectionist computing, and because one neuron has a limited computational power due to the CGP program size limit. However, a phenotype belonging to either of these classes does not prove that it does not use action engine signal generation instead of processing input signals, only that it might be possible for the phenotype to transmit signals from input neurons to output neurons for the same problem instance. It should also be noted that all phenotype variants call the various CGP functions for processing and transmitting signals, but it is not explicitly tracked

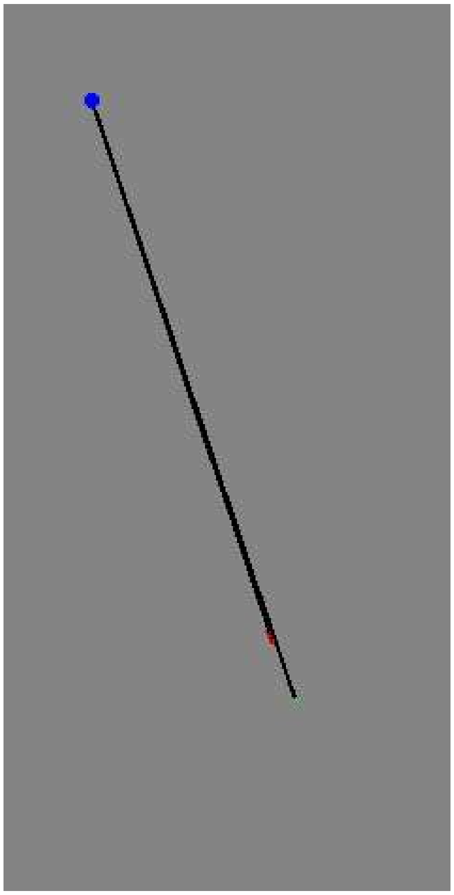


Figure 5.15: An example of a neuron phenotype with only one neuron. This specific phenotype has many connections to the output neuron.

where these signals originate[4] nor how far they reach before the neuron engine has used all the allotted actions for the timestep.

Figure 5.16 shows several examples of low or medium neurons, input, output and hidden connectivity phenotypes. Note that when a neuron creates a new neuron, they automatically have a connection, such that programs which do not properly utilize inter-hidden neuron connections might still have them. Another reason why seemingly reasonable connection structures may not actually utilize the connections meaningfully is that the programs may just be searching for connections to the output neuron specifically, and sometimes accidentally forms connections with other neurons instead, without necessarily using these connections meaningfully.

It was attempted to investigate the logged CGP-Programs. However, a logic bug in the CGP engine made it so that information was lost, making it impossible to reproduce the programs accurately. The bug essentially permitted the output nodes to not be in the set of nodes which are given sufficient input to produce a output, which worked by defaulting output values to 0, while also making it possible to produce genotypes which contain such output nodes. Possible other impacts of this logic error could be making the use of CGP modules less useful, and making it more difficult to evolve deeper/larger CGP programs.

However, some conclusions can still be drawn from the logs. The evolved CGP programs often used division and subtraction to create constant numbers, by dividing a number by itself it could produce the number 1 reliably, and by subtracting a number from itself it could produce the number 0, which are meaningful in NMS-LOC because 0 means "definitely do not do" something, while 1 means "definitely do something". It was also observed that the effective genotype variance in each population was small or nonexistent, likely due to the population replacement mechanism and the use of historic lists, such that it will be interesting to see if diversity is more maintained with the changes to these mechanisms described above. Additionally, the common ancestor check is added to the population replacement mechanism, which could help increase diversity further.

The results of Experiment 2 shows that NMS-LOC is able to find solutions to the one-pole balancing problem which are better than taking entirely random actions. The use of randomness in NMS-LOC may cause evolution to get stuck in "noise swamps", making it difficult to continue improving after a certain point. Due to some bugs and design choices it was possible for NMS-LOC to generate signals and transmit them to the output node without considering the input from the input node. Although such solutions may be functional, they are not as "interesting" as programs which can be reactive to input data. For this reason NMS-LOC is changed to not add neuron controller calls to the action queue of the engine at the start of each problem instance, and at initialization of a phenotype the single hidden neuron is initialized with connections to the input nodes. NMS-LOC was

[4] This would be difficult to do as previously received signals which were not transmitted may in principle be part of the information sent when a received signal triggers sending a signal, and because the logs for the programs would grow extremely large.

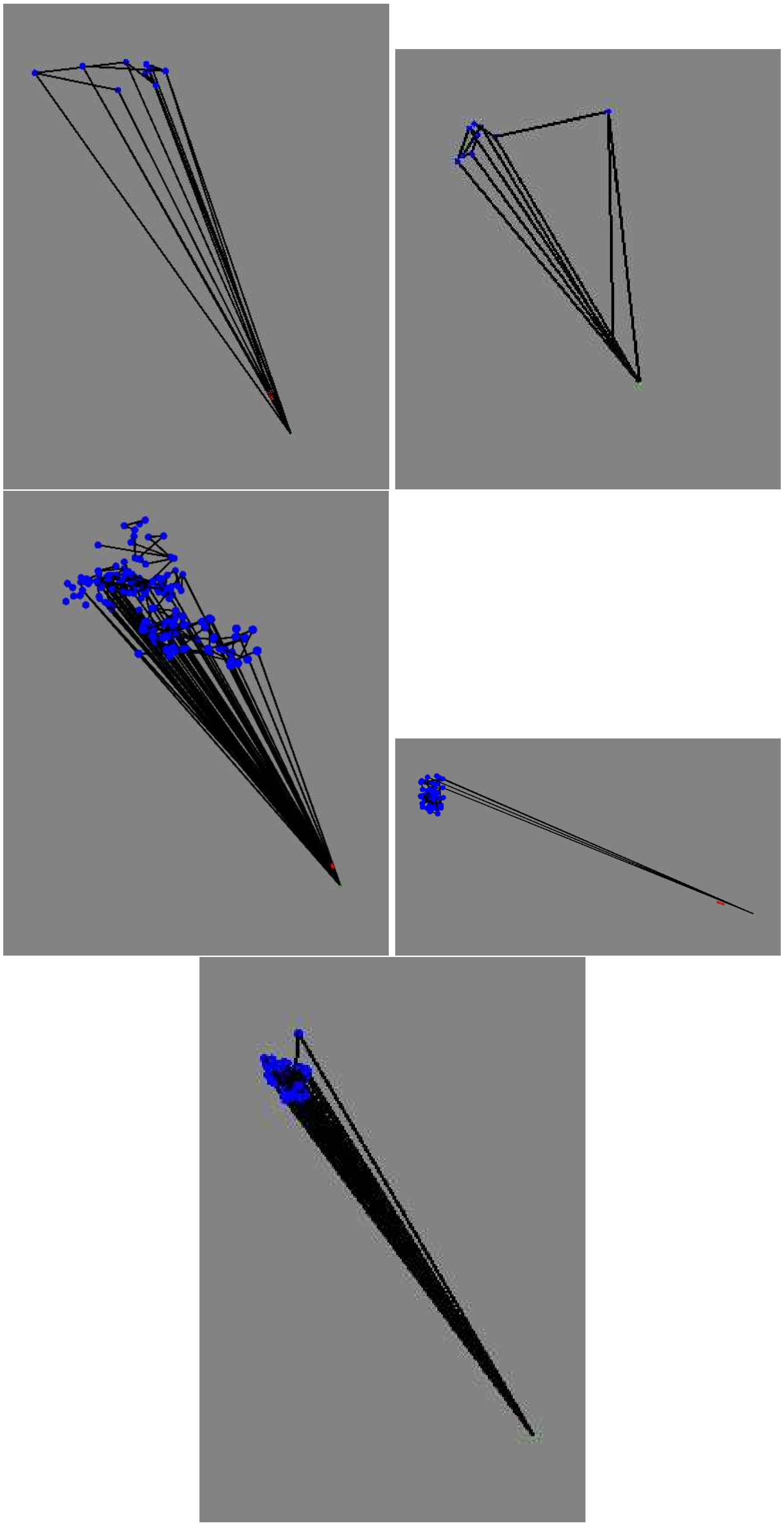


Figure 5.16: Several examples of phenotypes which seem to form interesting structures.

able to create several types of phenotypes, but some of them may have relied on generating signals, and not on transmitting them from the input node. Overall the results show that NMS-LOC is capable of evolving solutions to the One-Pole Balancing problem, but that the algorithm could be improved in several areas.

5.3 Experiment 3

Experiment 3 investigates NMS-LOC further. First, the fixes from the previous experiment may improve diversity and performance, and will fix the remaining issues with logging genotypes properly, allowing for more analysis. The experiment will only use the input signals as the initial actions in the action queue instead of calling every neurons action controller also, which is expected to produce more phenotypes with low amounts of neurons and connections. Further, the impact of randomness will also be investigated. Specifically, another set of CGP node functions are implemented, the ones used in Julian F. Miller (2021), which should be suitable as they are used successfully in Miller's experiments which also control neuron models using CGP. These will be tested alone in a deterministic neuron engine (where >1 means action, <1 means no action) (MILLER-DET), in a stochastic neuron engine (where the range <0, 1> denotes the probability of taking an action) (MILLER-RAND) and in a stochastic neuron engine with the addition of Gauss sampling as a node function (MILLER-GAUSS). These results will be compared with running the changed version of NMS-LOC with the same node functions and the stochastic neuron engine as in experiments 1 and 2 (RAND). Finally, the locations of output neurons, the initial hidden neuron, and input neurons are changed, the expected result is that the output phenotypes might look less cluttered. However, this location change does make input and output neurons more geometrically distanced, which is information the search could use in found solutions.

Due to computational limitations the iterations per evolutionary run is lowered down to 200, as this version of NMS-LOC tended to take longer to run. In total one 120 runs were started, ten for each config version, and 86 of these runs finished successfully. The rest were cancelled due to taking longer than 70 hours to compute, which is the default limit in the IDUN computational cluster. It is possible that the cancelled runs could have something in common with each other, but on there is also no evidence to conclude that this is the case.

Otherwise, the config files are identical to the ones shown in Appendix C. The four variants with various use of Miller functions and randomness as described above were each run for each config, giving a total of 12 config files.

This experiment will help answer the following questions:

- Does removing neuron action controller calls from the initial action queue mean that there are less high-neuron high-connectivity phenotypes at the end of the search?
- Is randomness/stochasticity necessary to find a gradient?

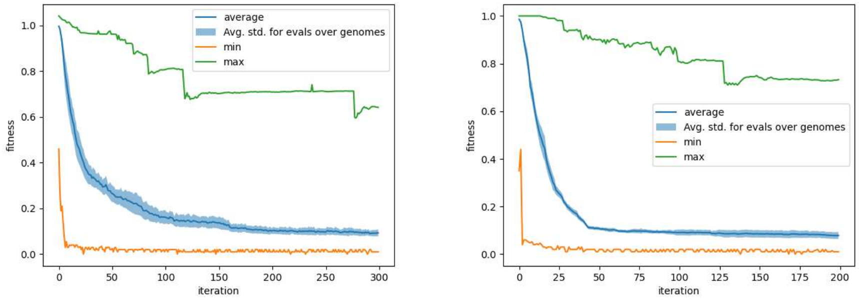


Figure 5.17: The unified fitness graph for Experiment 2 Experimental Run 2 (left) vs. the unified fitness for Experiment 3 RAND (right)

- Does removing randomness/stochasticity help with finding a gradient in low-fitness parts of the state space, potentially because the search no longer gets stuck in Noise Swamps?
- Does Julian F. Miller (2021) set of CGP functions work in NMS-LOC, and are they perhaps better suited for creating modular CGP functions?

Fitness Reference

Due to the decreased amount of training iterations per run it should be expected that fitness is on average higher, regardless of the impact of the other changes made to NMS-LOC. By comparing the RAND runs with the first two hundred iterations of Experiment 2 Experimental Run 2 it is possible to compare the new NMS-LOC version with the old, and establish a reference for the other runs. Figure 5.17 shows that the changes made seem to improve NMS-LOC's convergence rate. Statistically the 50 last iterations (150-200) of RAND had an average fitness of $\approx 0.0828 \pm 0.0150$ while the last of Experiment 2 had $\approx 0.0646 \pm 0.0115$. As such it seems that the two versions of NMS-LOC are comparable, only that Experiment 2 had another 100 iterations to further improve the fitness and that Experiment 2 may be slightly better on average. Both are better than a random policy on average. The minimum fitness[5] for RAND was ≈ 0.0123, while Experiment 2 was ≈ 0.009, and are comparable similarly to the average fitness.

Fitness across the different runs

NMS-LOC is able to find a gradient for each of the four different run types. For RAND, MILLER-RAND and MILLER-GAUSS $\approx$ 19% of children per generation is better than one of their parent genotypes. MILLER-DET has $\approx$ 15%, which is lower,

[5] Note that this is the average minimum fitness for the last 50 iterations for all aggregated over all the relevant runs. This is not the same as the minimum in the last iteration of the run, but by taking an average the measure is less susceptible to randomly good fitness random variable samplings while still giving a measure of the "best case".

but still enough to follow a gradient. It is likely that some of the difference is due to randomness giving a too optimistic estimate for some fitness random variables, causing the better child statistic to be slightly inflated. However, comparing the fitness's suggests that randomness is also useful for faster convergence to lower fitness values. Due to the limited run length it is not possible to conclude that MILLER-DET would not eventually converge to the same fitness values (or better) as the other run types. MILLER-DET had an average fitness of $\approx 0.165 \pm 0.002$ which is worse than random, and a minimum fitness of 0.009 ± 0.004, which is significantly better than random, showing that deterministic programs for NMS-LOC can also do well on the one-pole balancing problem. Numbers for MILLER-RAND is $\approx 0.103 \pm 0.04$ average, which is slightly worse than random, and $\approx 0.0103 \pm 0.002$ at minimum, which is better than random and comparable to MILLER-DET. RAND as mentioned has an average fitness of $\approx 0.082 \pm 0.015$ and a minimum of $\approx 0.012 \pm 0.005$, both better than random. However, MILLER-GAUSS does the best with an average fitness of $\approx 0.054 \pm 0.003$ and a minimum fitness of zero, as in perfect performance. The low standard deviance in fitness may indicate that there is little variation in the produced phenotypes for each evolutionary run, as the standard deviance is the average of the standard deviances for each individual run.

MILLER-GAUSS only found a "perfect" solution[6] once, however. It is possible that the low score is as such partially caused by this potential statistical outlier. Additionally, unlike the other versions MILLER-GAUSS did not have any statistical outliers with unusually high fitness. It is possible that this could be due to random chance, but it does seem to indicate that MILLER-GAUSS performs better than the other versions. If so, two conclusions can be drawn. Using Julian F. Miller (2021)'s functions with the addition of Gaussian sampling in a stochastic engine is better than the node functions shown in Appendix B. Additionally, using Miller's functions with the addition of Gauss sampling is better than just using the node functions in a deterministic and in a stochastic engine. It makes sense that Miller's functions work better, as they are a more extensive set of functions and should make it possible to evolve a larger class of functions, and allow larger variance of programs in low-active node programs. The hypothesis behind allowing for randomness was that this might help find gradients, which in view of these results may be true. However, it could also just be that the addition of Gauss sampling opens up for a larger class of functions in CGP programs, as they permit for stochasticity in the programs which is otherwise not included in Miller's function set. Although these results are not entire conclusive they do suggest that MILLER-GAUSS is the preferred version of NMS-LOC for the one-pole balancing problem.

MILLER-DET is clearly different from the other versions in that while all ver-

[6] That is, a perfect score for the encountered problem instances. However, as the minimum fitness is taken as the average over 50 runs and is still zero with no variance, it seems clear that the solution is able to solve the one-pole balancing problem for most problem instances, as the initial problem instance is random each time.

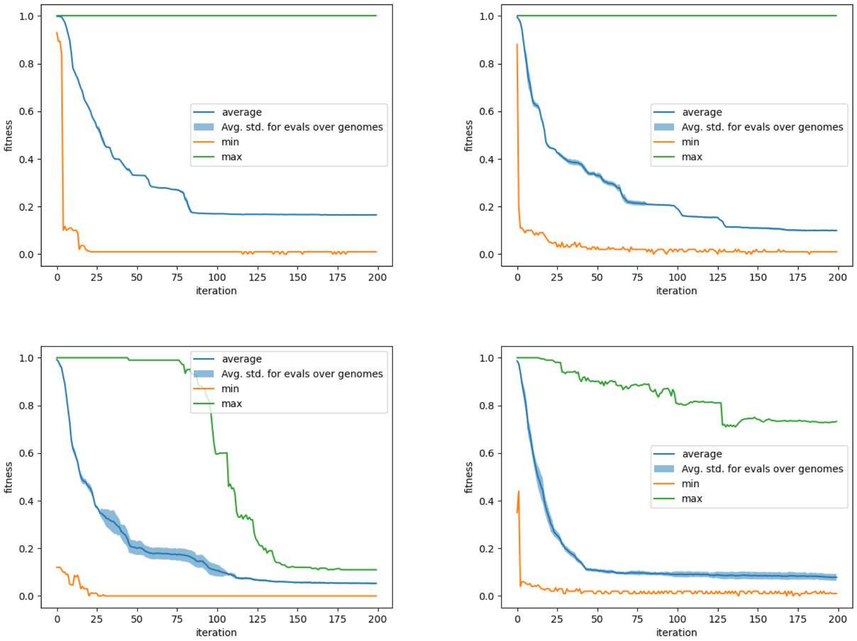


Figure 5.18: The fitness graphs of the different versions of NMS-LOC. In order from upper left and clockwise: MILLER-DET, MILLER-RAND, MILLER-GAUSS, RAND

sions consider using modular CGP functions, MILLER-DET is the only one to actually use these in the evolved programs. However, this still only happened in two of the runs. It is possible that the determinism makes it easier to evolve modules which behave "consistently", but that is a hypothesis at best. Statistically, in the run with the most used modules, the logs suggest that the modules used the tanh, sub, and rmux functions, as they are far more common than the other node functions in this run. Due to a logging bug the modules are unfortunately not logged[7]. Notably, the average fitness of this run was 1.0, meaning that no meaningful behavior was found. This lead to a "failure state" in NMS-LOC's design, which favors trying to spread CGP modules under the assumption that they are useful, but as there is in this case no found gradient they spread widely due to this bias. As such the logging bug ultimately did not matter, as the modules did no useful computation anyway. The other run had an average fitness of $\approx 0.041 \pm 0.005$, suggesting that the found modules may have been useful, which shows that although NMS-LOC rarely ends up using CGP modules it is still possible for it to do so, meaning that CGP modules can be useful, which means that the system could potentially be tweaked or changed such that it can take more advantage of CGP modules.

Figure 5.18 shows the fitness graphs for the different versions. It shows that

[7] A bug which was difficult to notice previously as modules were simply not being used.

while all versions can do large jumps in fitness, the versions which permit randomness may have an easier time of successfully making smaller steps in the fitness landscape, which would indicate that the use of randomness can help the search find an gradient. The fact that the the versions which use Gauss sampling have a worst case performance better than 1.0 (no meaningful behavior, i.e. no outputs sent) may also suggest that randomness helps provide an gradient. Otherwise it is clear that the best-case performance is somewhat comparable across the versions, which concurs with the statistics presented earlier (with the exception of the 0.0 potential outlier). Overall it seems reasonably fair to conclude that MILLER-GAUSS is the preferable version, while noting that the available evidence is not enough to conclusively say that it must be better than the other versions.

Another interesting fact is that the best-case runs (min) seem to have converged or basically converged by iteration 30 to 50. This suggest that an alternative strategy for finding good solutions would be to only run NMS-LOC evolution for 30 to 50 timesteps, checking if the solution is sufficiently good, and if not, starting the search again from a random point in the state space. For example, if the fitness is not better than random by run thirty, it might be a good choice to discard the evolutionary search and restart. However, this observation only holds for the one pole balancing problem. It is possible that other problem domains will require more evolutionary iterations to find a good solution.

Finally, we can note that there are not grounds to comment on the validity of the hypothesis about noise swamps. Due to the lower amount of iterations it is less clear how the search progresses when it starts to converge. What we can say for sure, however, is that it is possible for NMS-LOC to avoid noise swamps when using Miller's function with the addition of Gaussian sampling, the fitness graph for MILLER-GAUSS in Figure 5.18 clearly shows how the minimal fitness program is stable at a fitness of 0.0. The results also show that even in the deterministic case, in MILLER-DET, there is still some noise as can be told from the "wiggle" in the fitness graph. This noise is a result of the problem instance sampling still being stochastic/random in MILLER-DET, meaning that there is still a source of some noise.

Impact on Phenotype types

The changes made to NMS-LOC significantly reduced the amount of high-neuron high-connectivity phenotypes. Out of 86 phenotypes, 60 ($\approx$ 70%) had only a single neuron on average, 10 ($\approx$ 0.12%) had 2 to 20, 6 ($\approx$ 0.07%) had between 21 and 100, and 10 ($\approx$ 0.12%) had more than one hundred neurons. In the second run of experiment 2, 4 ($\approx$ 15%) out of 26 phenotypes had a single neuron, 9 ($\approx$ 36%) had a 2 to 20, 5 ($\approx$ 0.19%) had 21 to 100, and 8 ($\approx$ 0.31%) had more than 100. The results show that single-neuron phenotypes are far more common after the changes. It is interesting to observe that some high-neuron phenotypes still exist, and some have better-than-random performance, even though the phenotype control flow must start from the input neurons. MILLER-DET had

$\approx$ 84% single neuron phenotypes, MILLER-RAND had $\approx$ 54%, MILLER-GAUSS had $\approx$ 73%, and RAND had $\approx$ 70%. As mentioned, the functions from Julian F. Miller (2021) are designed for the range of real numbers in the range [-1, 1], and are not designed for stochastic systems, which may have made it more difficult for MILLER-RAND to evolve programs which have exactly one neuron without the addition. Instead, MILLER-RAND had more low ($\approx$ 0.23%) and medium ($\approx$ 0.09%) neuron phenotypes, but due to the relatively low sample sizes (respectively, 19 MILLER-DET, 22 MILLER-RAND, 22 MILLER-GAUSS, and 23 RAND) this may be a random difference.

MILLER-DET, MILLER-RAND, and MILLER-GAUSS primarily exhibited two types of network structures in the final phenotypes. The one type is single-neuron structures with connections to all inputs and the output neuron, as shown in Figure 5.19. The other phenotype type is also a single-neuron structure, but contains several more neurons which are not connected to anything, some sort of unpurged remnants from the phenotypes development. The difference in phenotype types mentioned above is therefore just that some of the runs randomly had more of these remnants than others. Such remnants are shown in Figure 5.20. It should be noted that although in the problem solving phase of the neuron engine the initial actions in the action queue are only signal transmissions from the input neuron, in the reward phase the reward program for each neuron in the simulation is called, which means that the neurons could still purge themselves even if they are not connected to anything, i.e. it is not impossible to "clean up" in such phenotypes, but it's possible that doing so doesn't provide evolutionary advantages. It is also possible that they might reconnect to the network in the reward phase. Finally, some phenotypes contained many neurons, and in these it is not so easy to tell if the entire network structure does anything, or if there are just more unpurged remnants and unpurged connections. Examples are shown in Figure 5.21. In high neuron phenotypes there are typically more connections to the output neuron than to the input neuron, which may indicate that some of the neurons are not doing anything, and that actual computation is done in a small part of the network structure, probably single neurons connected to both input and output neurons, or the neuron connected to the input neuron and a few other neurons, at least one of which is connected to the output.

Even low-neuron phenotypes can have a high amount of connections to the output nodes. This varies across the phenotypes, likely both few and many connections can be useful. Having many connections to the output neuron may be useful for several reasons. It is possible that each separate axon-dendrite reacts to a different input, making up collectively some "if-else" type of structure. In versions of NMS-LOC which permit stochasticity in the engine (and CGP programs) having multiple axon-dendrite connections to the output neuron may be a way of handling uncertainty. Another possible explanation is that axon-dendrite birth could be used to generate signals, and that old axon-dendrites just aren't purged completely.

The prevalence of single-neuron phenotypes or effectively single-neuron phen-

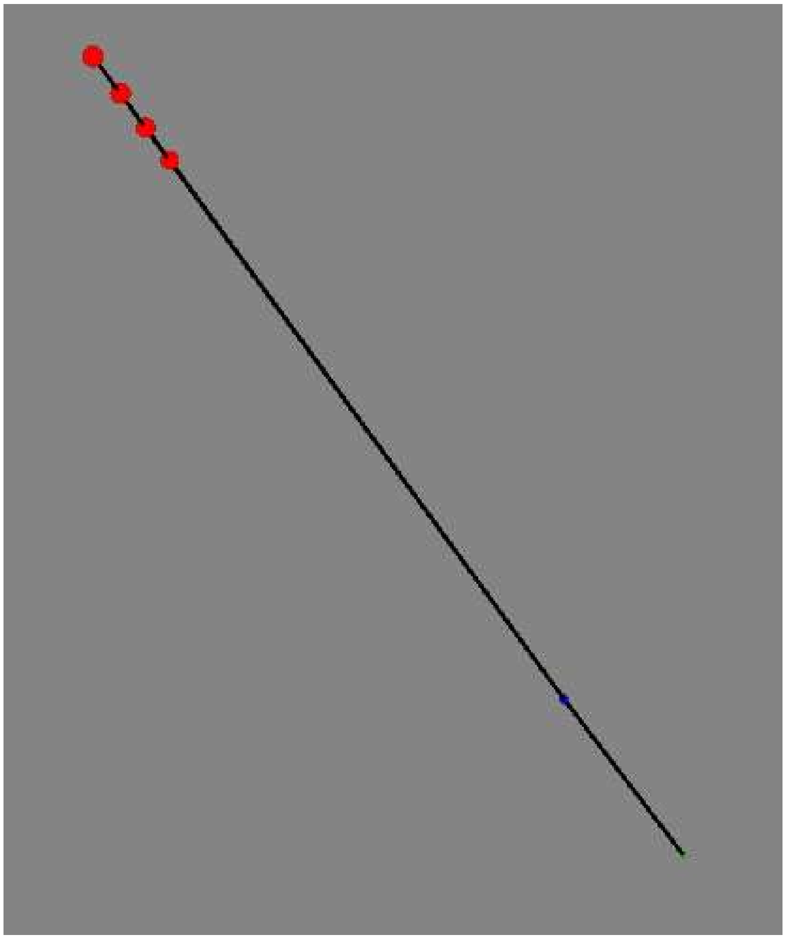


Figure 5.19: A phenotype displaying the usual case for single neuron phenotypes. The 0.0 (perfect) phenotype had this structure.

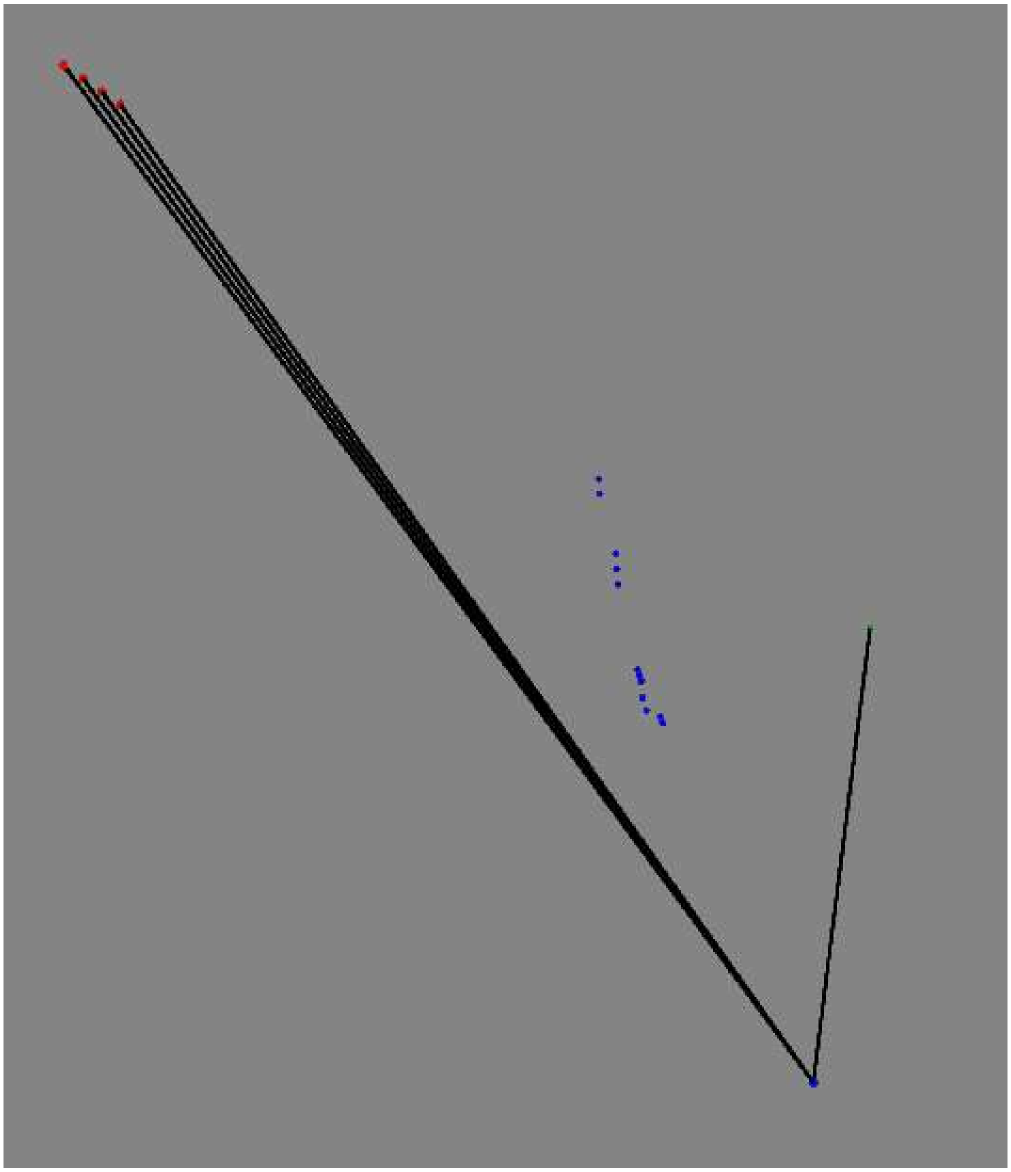


Figure 5.20: A phenotype displaying a fake multi-neuron structure. As can be seen, most of the neurons are disconnected, and the phenotype has only one active neuron. The example shows how the neuron locations have moved over time. Hypothetically, adjusting the position of a single neuron could be a type of learning, as neuron position is given as program input in many of the CGP programs, although this is not necessarily how these phenotypes work.

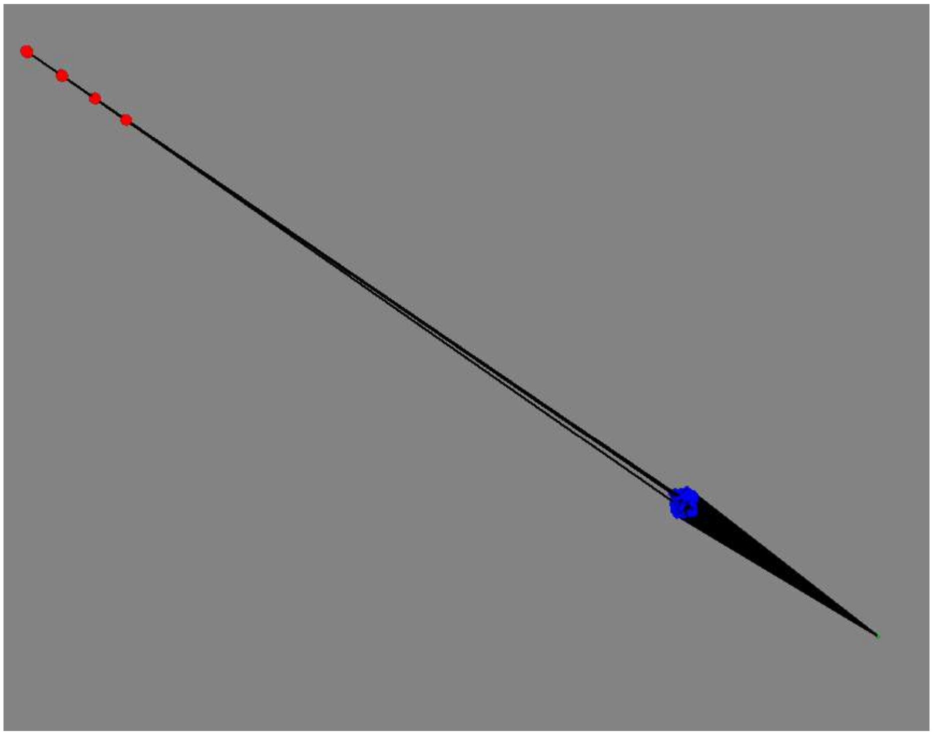


Figure 5.21: A phenotype displaying a multi-neuron phenotype. Are all neurons involved in computation? Just subsets? Only a few?

otypes can be explained by the fact that having a single neuron is sufficient for NMS-LOC to do well on the one-pole balancing problem, which follows from single-neuron phenotypes having low fitness values across the different versions. Evolution may then tend to find these solutions if they are easier to find than multi-neuron solutions - after all, it is simpler to just process and transmit a signal to the output neuron, than to do so while also creating a more complicated network structure.

RAND contains more actual multi-neuron phenotypes than the other versions. RAND seems to both create high-neuron phenotypes which likely share similarities with high-neuron phenotypes (see Figure 5.22 in the other versions - albeit seeming to spread out more in geometric space - but also phenotypes with a low amount of neurons which seem to exist in some type of network structure (see Figure 5.23. A possible explanation for this is that the other versions have more CGP node functions, allowing a larger class of functions to be implemented in the CGP programs. Because RAND may be restricted to smaller class of CGP programs, it is possible that relying on more distributed computation scheme is more often useful. Hopefully, as RAND seems to be able to form network structures, it is possible that the other versions of NMS-LOC can also form network structures should they be applied to a sufficiently difficult problem.

5.3.1 Looking into CGP functions

There are two problems with interpreting CGP functions, which makes it not possible to do so. First of all, the bug causing data incompleteness prevailed. Secondly, the CGP programs are not friendly to human interpretation, although it may be possible to understand. Figure 5.24 shows the problem, namely that individual programs can be very complex. Additionally, there are 17 functions, and for some variants of NMS-LOC each of these can have three hex variants. Finally, these 51 programs interact in complex ways which there is no trace of, as such a trace would be very large.

5.4 NMS-LOC and the IRIS Flower Classification Problem

NMS-LOC was adapted to the IRIS flower dataset (Fischer (1936)). The results are slightly promising, but detailed experiments and analysis is outside of the scope of this work, and the limited amount of experimentation with IRIS where honestly mostly done to see if it would work at all, but as the very limited results are still somewhat interesting they are documented and discussed. Essentially, several runs were attempted, but likely due to the problem complexity NMS-LOC was too inefficient to compute many iterations, and low iteration runs seemed to struggle to produce useful phenotypes. However, doing a 100-iteration run and changing the config file to increase the CGP program size to 400 managed to produce useful phenotypes. The increased maximal function size would mean that a larger class of functions can be evolved, and an larger amount of inactive nodes

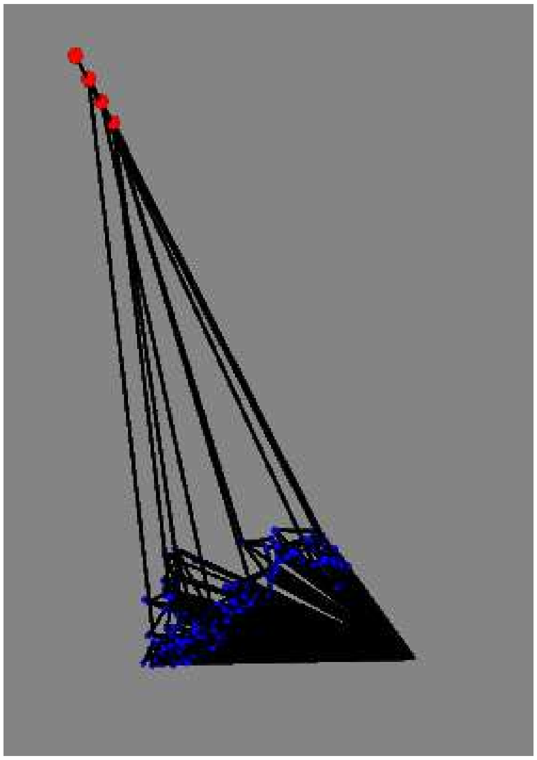


Figure 5.22: A high neuron phenotype evolved using the RAND version of NMS-LOC.

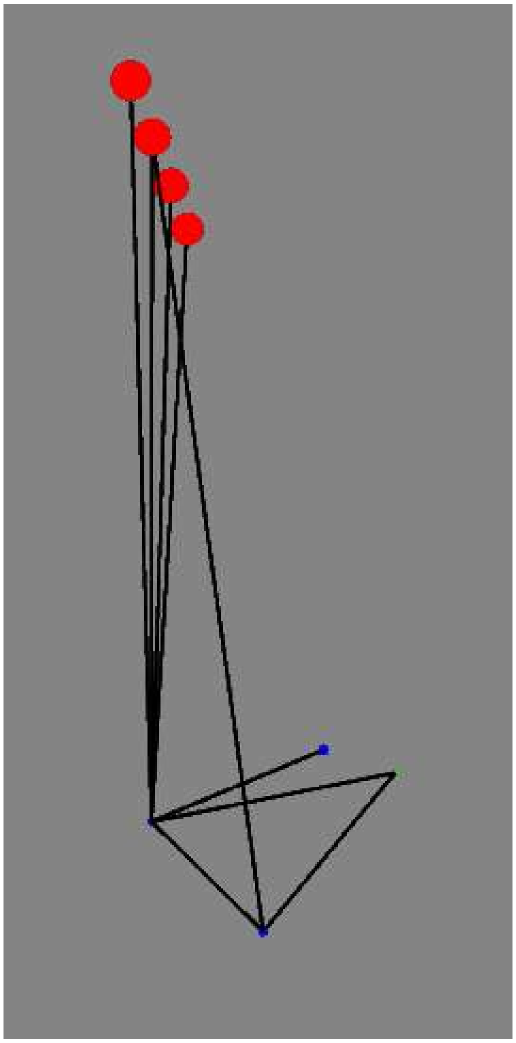


Figure 5.23: A low neuron phenotype which seems to have a network structure evolved using the RAND version of NMS-LOC.

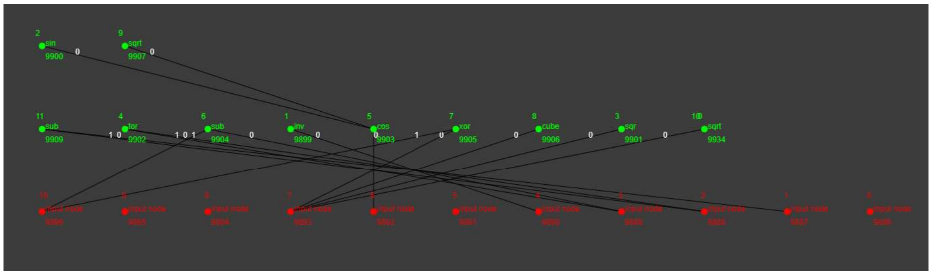


Figure 5.24: The figure shows a single CGP function, which consists of 24 nodes in a complicated network structure.

could potentially be beneficial for CGP neutral drift. The run uses Miller functions, a stochastic neuron engine, and the GAUSS node type.

Before describing these further a couple of notes should be made about the IRIS problem domain. Unlike One-Pole balancing, IRIS is a classification problem. Implementation wise IRIS contains three output neurons in NMS-LOC, while One-Pole only has one. IRIS having multiple output neurons and NMS-LOC being able to still evolve meaningful phenotypes mean that NMS-LOC can find solutions for multiple-output neuron problems.

It should also be made clear that only one experimental run was done after the change to CGP program size, meaning that the results should be read more as a particular case or example and not as representative of average performance on the problem domain. In particular, each evolutionary run begins with splitting the IRIS dataset into a validation set and a training set constant for the duration of the run, and it is possible that this particular split was favorable, but no log was made of the split so there are no grounds to say anything conclusive.

Figure 5.25 shows the fitness graph for the IRIS classification problem with both training set and validation set fitness over time. In this run there were only two population slots, so the max, min, and average value is essentially the same. The graph shows that NMS-LOC was able to evolve a phenotype with a training set Mean-Squared Error (MSE) lower than 0.5, indicating that it classifies a little more than 30% of the training set correctly. Taking into consideration that the fitness score is evaluated from the entire evolution from the minimal network structure of a single hidden neuron connected to the input neurons this score is not too bad considering that this is a one-off test and not the use case NMS-LOC has been tested and developed for. The MSE correcting for the amount of times no output is sent to the output neuron (which may be the amount of iterations before a connection to the output neuron is established, but is not necessarily only establishment iterations) gives a MSE of about 0.4 which means that about 37% of the training data samples are classified correctly after establishment. The IRIS dataset is balanced, meaning that each classification category has an equal amount of samples, but this does not need to be true for the training data/validation data split in the NMS-LOC implementation. However, a random policy which simply picks each class with equal probability should be right about 0.33% of the time on average and get an MSE of about 0.44. This means that the evolved phenotype is about as good as outputing a random classification, which is not particularly impressive. It does not mean that the evolved phenotype implements the random policy, however, only that the implemented program is about as good as the random policy. It is possible that longer evolutionary runs or runs with more populations lots could produce better phenotypes, but there is no reason to conclude that this is true. Note also that the phenotypes are only exposed to one pass through of the training dataset, e.g. 120 training data samples. In a way, the results are more impressive considering how few training passes are done to produce the phenotypes, and increasing the amount of training data passes could potentially improve fitness significantly.

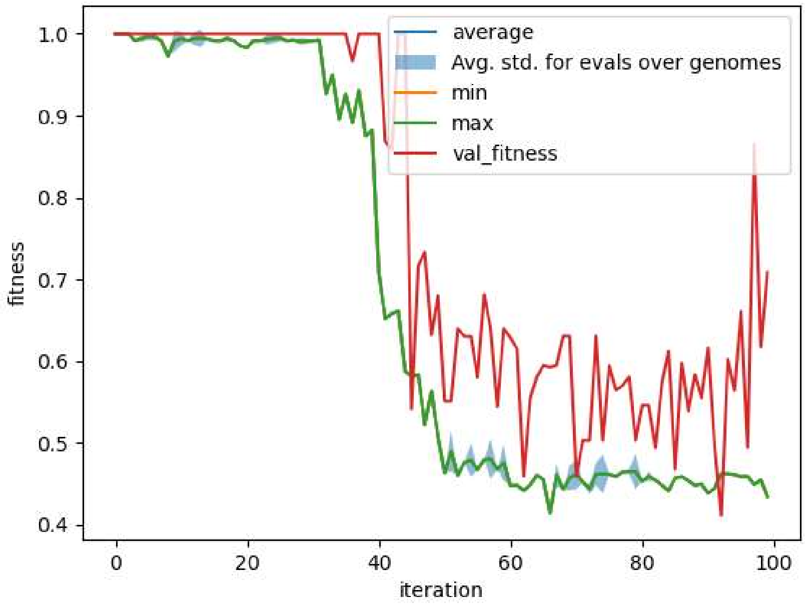


Figure 5.25: The fitness graph for the IRIS classification problem

Figure 5.25 shows that the validation set fitness is worse than the training set fitness and has a greater variance over time. In general one can expect validation set fitness to be worse than training set fitness on classification problems. The higher variance might is likely caused by two factors: One, the used version of NMS-LOC is random, but the genotype is only trained to be robust for the training set, and the random permutations may have a greater impact on validation set performance. Two, genotype state space steps which are neutral or good in the training set may be negative in the validation set. Note also that the validation set is smaller, which may produce higher variance. Also, be aware that the validation set evaluation is done on the phenotype produced after the training set evaluation, and the evolutionary algorithm is only aware of the training set fitness.

Figure 5.26 shows a typical phenotype at the end of evolution. These phenotypes have a large amount of neurons ($\approx 1591 \pm 204$). There are no grounds to make a conclusion about how the phenotype works, but based on the statistics some observations and hypothesises can be made. The hidden neurons have few connections on average ($\approx 2.75 \pm 0.05$), and the input neurons also have few connections on average ($\approx 14.4 \pm 3.30$). This may explain how the neuron network is able to work despite having so many neurons. Because the amount of connections to the input are low, it is possible that most neurons are filtered out at an early stage. As the network does not appear to be very deep, at least from the graph although it is hard to tell for sure and since average network depth calculation is not implemented (and could be infinite if there is a recursive loop) it can't be concluded that the network does not contain long paths, but there must at least exist some paths which are shallow enough and filtered enough for signals to get through to the output neurons from the input neurons for some cases. A hypothesis is that the neuron network implements some type of "if/else"-like decomposition of the input data. Another hypothesis is that the neuron network simply sends a signal to a random output node and happens to develop a large network structure as a computational artefact, which would fit with the approximately random-policy level performance on the training data. However, if the network truly implemented a random policy, one could expect it to also perform at random level on the validation data, which it does not. As such it is probable that the network structure is in some way reactive to the input data and the returned error score given after each training sample, i.e. in some way reactive to the training data. Another interesting observation is that the hox switch count and the neuron count have a good-as perfect positive correlation. A hypothesis is that each hox version of the CGP functions (of which there are three) implements a program in some way suitable or correlated for each problem domain class. However, it should be noted that there is a default hox selection check at neuron creation which is not logged as a statistic, and that it is perhaps more likely that the hox selection program is in the neuron which gives birth to another neuron after neuron birth, in which case another possible hypothesis is that the hox functions are related to how many steps away the neuron is to the output neuron.

Ultimately, the results show that NMS-LOC can find a gradient in problems which have multiple output neurons. The results are not particularly impressive in regards to NMS-LOCs capability of solving classification problems in general or IRIS specifically as the produced phenotypes are not much better than random. However, as only one evolutionary run is analyzed there are the results may not be indicative of the average case performance on IRIS with the same settings. Possible changes which may improve NMS-LOC performance on the IRIS problem are more iterations per evolutionary run- More problem samples per genome evaluation/phenotype development would allow NMS-LOC to have more time to learn and have more of the evaluations percentage-wise be done with developed phenotypes which may improve recorded performance.

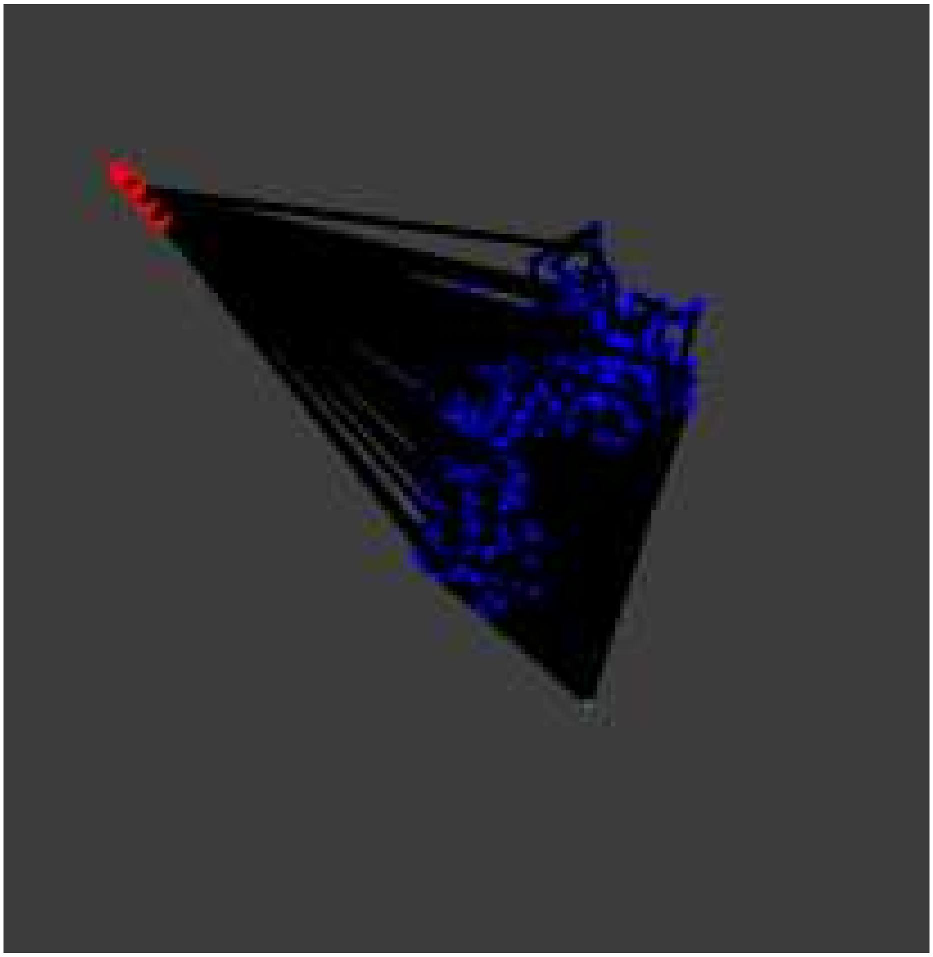


Figure 5.26: An NMS-LOC phenotype produced for the IRIS problem

Chapter 6

Conclusion

The results on the One-Pole Balancing problem show that Neuron Model Search is conclusive proof that NMS-methods can at solve some problems (at least the One-Pole Balancing problem). More specifically, systems which evolve neurons and do not extract ANNs, i.e. learning lifetime behaviour, work to some degree. However, despite showing that NMS-LOC is able to solve a problem domain it remains to investigate which other properties NMS systems may have, which could be interesting areas to investigate in further work. For example:

- Can a single NMS genotype create phenotypes which solve several or even many problem domains, when each phenotype is evolved in interaction with that problem domain?
- Can NMS phenotypes adapt to dynamic problem domains?
- Can single NMS genotypes create phenotypes which solves several problems at once?
- Can an NMS genotype with good performance on a set of training problems get good performance on an non-overlapping set of validation problems?
- Which other problem domains can NMS approaches be shown capable of solving?
- How do found solutions work?

Personally, I find the potential of evolving neurons which can solve several problem domains interesting. If a neuron which implements a novel way for neural structures to learn it could potentially be extracted from the evolutionary context and be re-implemented in an optimized fashion and potentially be efficient enough for engineering use. Additionally, a hypothetical discovery of new learning rules would be interesting by itself.

Despite being able to solve the One-Pole Balancing problem there are several likely problems with the NMS-LOC algorithm. There is good reason to believe that the population diversity within an evolutionary search becomes small as time goes on, based on how similar the produced phenotypes by different genotypes in the search are. This is likely due to the explicit population replacement mechanism and the implicit population replacement produced by swapping in and out of the historic best list. Possible changes to fix could be removing the explicit population replacement and separating the historic best into several list, such that a genome can only be swapped out for one of it's own ancestors. Overall there has been much research into population management in bio-inspired AI, for an introduction see Eiben and Smith (2015).

A core design idea in NMS-LOC was using modular CGP functions as core processes, but in practice the algorithm struggled to find useful modules. Although it is perfectly fine by itself that modular CGP was not useful for this particular problem with this particular algorithm, I will argue that designing NMS systems which successfully use components analogous to core processes would be beneficial, based on how important they seem to be in nature (Gerhart and Kirschner (2007)). Successful use of such processes would per definition provide increased ability for canalization and complexification in terms of Stanley and Miikkulainen (2003)'s taxonomy. Similar arguments are made in Downing (2015), there with focus on the emergence of behaviour as a result of complex interactions at several levels of abstraction. Finally, spreading and creating modules should be restricted such that modules are only created in better-than-worst-case phenotypes to prevent neutral drift from creating genotypes with large amounts of recursive modules which implement no useful behaviour .

One potential alternate system could be to, instead of evolving neuron behaviour, directly evolve parallellizable CGP part-functions. A set of "input" CGP functions are called at the beginning of processing input, and these may output an output tag which is matched against other functions in the function pool, along with passing along signals and potentially writing to state registers or chemical state. By extending self-modifying CGP (Julian F. Miller (2020)) to be able to produce new functions during the execution of a genotype the resulting program could have several desirable traits: One, it could evolve re-usable components (i.e. core processes). Two, it could modify itself. And three, it would process information at both a symbolic level (the definition of each CGP program), and a sub-symbolic level (the signals and states in the system).

Other changes could be to change evolution of hex-variants to a coevolutionary approach where each hex variant is evolved separately, and their fitness is the average fitness of put-together complete genotypes. Another potential change could be to include some neuron actions on the CGP program level as node functions, such as: "If output is >= 1.0, then Neuron dies." or "Move neuron in the x-direction equal to input.", which could reduce the amount of genotypes that need to be evolved as several could be abstracted away as node functions. In fact, all neuron actions could be implemented as node functions of a specific arity. This could be combined with co-evolving several "control programs", which may call on each other. Such a change might be beneficial as it further limits the impact of human design: In NMS-LOC the way the different programs interact in terms of which might call which with what inputs is pre-defined, but in principle this could also be left to evolutionary search.

One notable issue in the design of NMS-LOC is that sending a signal between two neurons requires 5 to 6 calls to CGP programs. This makes sending signals a relatively expensive operation, putting an effective limit to the complexity of produced network structures, and making the system slower than alternatives with faster signaling. In general it would be beneficial to produce more optimized NMS systems in the future, especially as a proof-of-concept system now has been shown to work in one problem domain.

Overall, the success of NMS-LOC on the one-pole balancing problem warrants further investigation into if other neuron-evolving systems could work better. Based on the aforementioned design issues with NMS-LOC, it is the authors opinion that there likely exist other and better systems which evolve lifetime behaviour neurons. In a sense the existence of biological neurons is conclusive proof that it is possible to evolve lifetime behaviour neurons, at least on some computational substrates. The question is just if we can do if far faster than biological evolution on our computational substrates. To do so, it will likely be necessary to go through several designs in research, create more optimized systems, and continue to take inspiration from the biological neurons and evolutionary systems found in nature to some degree.

There is not sufficient data to determine if using randomness in the simulation engine and in CGP functions is beneficial or detrimental. What can be said with certainty is that NMS-LOC was able to find good solutions to the one-pole balancing problem when using randomness, and that there is a slight indication that using Julian F. Miller (2021)'s node functions in addition to simulation engine randomness and adding Gauss as a node function may preferable as default settings for NMS-LOC. The advantages of using randomness is that it opens up for a new class of functions, and may lend itself to increased robustness and otherwise make the search landscape easier to navigate. The disadvantage is that it can make solutions harder to interpret and understand, and that it makes fitness evaluation into a random variable sampling rather than a deterministic evaluation. This means that there is a need to handle potential information loss from incorrect estimations, and a need to run several fitness evaluations for the same genotype.

Finally, the single result on the IRIS problem show that NMS-LOC can find a gradient in problems with multiple output neurons, but did not evolve better-than-random phenotypes. However, the limited amount of computational resources used, such as having only a hundred evolutionary iterations and just one experiment means that there is no data to comment on if NMS-LOC could perform better on IRIS with more computation. The IRIS example also highlights possible system configuration changes which may be important to some problems, such as the CGP function maximal node size, and the amount of passes through the training set.

Ultimately, the conducted experiments show conclusively that NMS systems can work on at least one problem. In principle as long as neuron models which can solve the target problem exist within the search space of an NMS system they can be reached. In practice the system must be designed such that fitness landscape has gradients which can be navigated by a suitable evolutionary algorithm. Additionally, NMS software must be sufficiently efficient from a computational perspective, especially if NMS systems are to be applied to more complex problem domains.

Bibliography

Downing, Keith L. (2015). *Intelligence Emerging: Adaptivity and Search in Evolving Neural Systems*. The MIT Press.

Holland, John H. (1998). *Emergence: From Chaos To Order*.

Hiesinger, P. Robin (2021). *The Self-Assembling Brain: How Neural Networks Grow Smarter*. Princeton University Press.

Rosenblatt, Frank (1958). 'The perceptron: a probabilistic model for information storage and organization in the brain.' In: *Psychological review* 65 6, pp. 386–408.

Goodfellow, Ian, Yoshua Bengio and Aaron Courville (2016). *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press.

Stanley, Kenneth O. and Risto Miikkulainen (2002). 'Evolving Neural Networks through Augmenting Topologies'. In: *Evolutionary Computation* 10, pp. 99–127.

Jackobi, Nick (1995). 'Harnessing Morphogenesis'. In: *International Conference on Informatino Processing in Cells and Tissues*, pp. 29–41.

Elbrecht, Daniel and Catherine Schuman (2020). 'Neuroevolution of spiking neural networks using compositional pattern producing networks'. In: *International Conference on Neuromorphic Systems 2020*, pp. 1–5.

Xu, Feiyu, Hans Uszkoreit, Yangzhou Du, Wei Fan, Dongyan Zhao and Jun Zhu (2019). 'Explainable AI: A Brief Survey on History, Research Areas, Approaches and Challenges'. In: *Natural Language Processing and Chinese Computing*. Ed. by Jie Tang, Min-Yen Kan, Dongyan Zhao, Sujian Li and Hongying Zan. Cham: Springer International Publishing, pp. 563–574. ISBN: 978-3-030-32236-6.

Parisi, German I., Ronald Kemker, Jose L. Part, Christopher Kanan and Stefan Wermter (2019). 'Continual lifelong learning with neural networks: A review'. In: *Neural Networks* 113, pp. 54–71. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/j.neunet.2019.01.012`. URL: `https://www.sciencedirect.com/science/article/pii/S0893608019300231`.

Crawshaw, Michael (2020). 'Multi-Task Learning with Deep Neural Networks: A Survey'. In: *CoRR* abs/2009.09796. arXiv: `2009.09796`. URL: `https://arxiv.org/abs/2009.09796`.

Reed, Scott, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar and Nando de Freitas (2022). *A Generalist Agent*. DOI: 10.48550/ARXIV.2205.06175. URL: https://arxiv.org/abs/2205.06175.

Lehman, Joel, Jeff Clune, Dusan Misevic, Christoph Adami, Julie Beaulieu, Peter Bentley, Samuel Bernard, Guillaume Beslon, David Bryson, Nick Cheney, Antoine Cully, Stephane Donciuex, Fred Dyer, Kai Olav Ellefsen, Robert Feldt, Stephan Fischer, Stephanie Forrest, Antoine Frénoy, Christian Gagneé and Jason Yosinksi (Mar. 2018). 'The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities'. In: *Artificial life* 26, pp. 274–306. DOI: 10.1162/artl_a_00319.

Miikkulainen, Risto (2021). 'Creative AI Through Evolutionary Computation: Principles and Examples'. In: *SN Computer Science* 2, pp. 163–170.

Rambjør, Jon Oddvar (2021). 'Cartesian Genetic Programming for searching for novel Neuron Models'. In.

Stanley, Kenneth O., Jeff Clune, Joel Lehman and Risto Miikkulainen (2019). 'Designing neural networks through neuroevolution'. In: *Nature Machine Intelligence* 1, pp. 24–35.

Del Ser, Javier, Eneko Osaba, Daniel Molina, Xin-She Yang, Sancho Salcedo-Sanz, David Camacho, Swagatam Das, Ponnuthurai N. Suganthan, Carlos A. Coello Coello and Francisco Herrera (2019). 'Bio-inspired computation: Where we stand and what's next'. In: *Swarm and Evolutionary Computation* 48, pp. 220–250. ISSN: 2210-6502. DOI: https://doi.org/10.1016/j.swevo.2019.04.008. URL: https://www.sciencedirect.com/science/article/pii/S2210650218310277.

Miller, Julian F. (Nov. 2021). 'IMPROBED: Multiple Problem-Solving Brain via Evovled Developmental Programs'. In: *Artificial Life*, pp. 1–36.

Miller, Julian F., Dennis G. Wilson and Sylvain Cussat-Blanch (2019). 'Evolving Developmental Programs That Build Neural Networks for Solving Multiple Problems'. In: *Genetic Programming Theory and Practice*, pp. 137–178.

Sutton, Richard (2019). 'The Bitter Lesson'. In: URL: http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

Cook, Matthew (2004). 'Universality in Elementary Cellular Automata'. In: *Complex Syst.* 15.

Goldberg, David E and John Henry Holland (1988). *Genetic algorithms and machine learning*. Kluwer Academic Publishers-Plenum Publishers; Kluwer Academic Publishers ...

Eiben, A.E. and James E. Smith (2015). *Introduction to Evolutionary Computing*.

Floreano, Dario, Peter Dürr and Claudio Mattiussi (2008). 'Neuroevolution: from architectures to learning'. In: *Evolutionary Intelligence* 1, pp. 47–62.

Whitley, Darrell, Stephen Dominic, Rajarshi Das and Charles W. Anderson (1993). 'Genetic Reinforcement Learning for Neurocontorl problems'. In: *Machine Learning* 13.

Stanley, Kenneth O., David B. D’Ambrosio and Jason Gauci (2009). ‘A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks’. In: *Artificial Life* 15, pp. 185–212.

Cangelosi, Angelo, Domenico Parisi and Stefano Nolfi (1994). ‘Cell division and migration in a ’genotype’ for neural networks (Cell division and migration in neural networks)’. In: *Network Computation in Neural Systems* 5.

Eggenberger, Peter (1997). ‘Creation of Neural Networks Based on Developmental and Evolutionary Principles’. In: *Artificial Neural Networks - ICANN’97*, pp. 337–342.

Zhao, Chunmei, Wei Deng and Fred H. Gage (2008). ‘Mechanisms and Functional Implications of Adult Neurogenesis’. In: *Cell* 132.4, pp. 645–660. ISSN: 0092-8674. DOI: `https://doi.org/10.1016/j.cell.2008.01.033`. URL: `%5Curl%7Bhttps://www.sciencedirect.com/science/article/pii/S0092867408001347%7D`.

Stanley, Kenneth O. and Risto Miikkulainen (2003). ‘A taxonomy for artificial embryogeny’. In: *Artificial Life* 9, pp. 93–130.

Suganuma, Masanori, Masayuki Kobayashi, Shinichi Shirakawa and Omoharu Nagao (2002). ‘Evolution of Deep Convolutional Neural Networks Using Cartesian Genetic Programming’. In: *Evolutionary Computation* 28, pp. 141–163.

Astor, J.C. and Chrstoph Adami (2000). ‘A Developmental Model for the Evolution of Artificial Neural Networks’. In: *Artificial Life* 6, pp. 189–218.

Turner, Andrew J. and Julian F. Miller (2013). ‘Cartesian Genetic Programming encoded Artificial Neural Networks: A Comparison using Three Benchmarks’. In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, pp. 1005–1012.

Khan, Maryam M., Gul M. Khan and Julian F. Miller (2010). ‘Evolution of Neural Networks using Cartesian Genetic Programming’. In: *IEEE Congress on Evolutionary Computation*.

Khan, Maryam M., Arbab M. Ahmad, GUl M. Khan and Julian F. Miller (2013). ‘Fast learning neural networks using Cartesian Genetic Programming’. In: *Neurocomputing* 121.

Khan, Gul Muhammad (2018). ‘Structure and Operation of Cartesian Genetic Programming Developmental Network (CGPDN) Model’. In: *Evolution of Artificial Neural Development: In search of learning genes*. Cham: Springer International Publishing, pp. 57–82. ISBN: 978-3-319-67466-7. DOI: `10.1007/978-3-319-67466-7_5`. URL: `https://doi.org/10.1007/978-3-319-67466-7_5`.

Khan, Nadia and Gul Muhammad Khan (June 2021). ‘Multi-chromosomal CGP-evolved RNN for signal reconstruction’. In: DOI: `10.1007/s00521-021-05953-4(`.

Rothermich, Joseph and Julian Miller (July 2003). ‘Studying the Emergence of Multicellularity with Cartesian Genetic Programming’. In.

Rothermich, Joseph, Fang Wang and Julian Miller (Jan. 2003). ‘Adaptivity in cell based optimization for information ecosystems’. In: pp. 490–497. DOI: `10.1109/CEC.2003.1299615`.

Öztürkeri, Can and Colin Johnson (Jan. 2011). 'Evolution of Self-Assembling Patterns in Cellular Automata Using Development.' In: *J. Cellular Automata* 6, pp. 257–300.

Laubenbacher, Reinhard, Franziska Hinkelmann and Matt Oremland (2013). 'Chapter 5 - Agent-Based Models and Optimal Control in Biology: A Discrete Approach'. In: *Mathematical Concepts and Methods in Modern Biology*. Ed. by Raina Robeva and Terrell L. Hodge. Boston: Academic Press, pp. 143–178. ISBN: 978-0-12-415780-4. DOI: https://doi.org/10.1016/B978-0-12-415780-4.00005-3. URL: https://www.sciencedirect.com/science/article/pii/B9780124157804000053.

Hintze, Arend, P. Robin Hiesinger and Jory Schossau (2020). 'Developmental neuronal networks as models to study the evolution of biological intelligence'. In: *Artificial Life Conference, 2020 Workshop on Developmental Neural Networks*. URL: %5Curl%7Bhttps://www.irit.fr/devonn/2020/07/13/hintze.html%7D.

Miller, Julian Francis (2003). 'Evolving Developmental Programs for Adaptation, Morphogenesis, and Self-Repair'. In: *ECAL*.

Gerhart, John and Marc Kirschner (2007). 'The theory of facilitated variation'. In: *Proceedings of the National Academy of Sciences* 104.suppl 1, pp. 8582–8589. ISSN: 0027-8424. DOI: 10.1073/pnas.0701035104. eprint: https://www.pnas.org/content/104/suppl_1/8582.full.pdf. URL: https://www.pnas.org/content/104/suppl_1/8582.

Waskan, Jonathan (n.d.). *Connectionism*. https://iep.utm.edu/connect/.

Buckner, Cameron and James Garson (2019). 'Connectionism'. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Fall 2019. Metaphysics Research Lab, Stanford University.

Minsky, Marvin L. (June 1991). 'Logical Versus Analogical or Symbolic Versus Connectionist or Neat Versus Scruffy'. In: *AI Magazine* 12.2, p. 34. DOI: 10.1609/aimag.v12i2.894. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/894.

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang and Wojciech Zaremba (2016). 'Openai gym'. In: *arXiv preprint arXiv:1606.01540*.

'Neuronlike adaptive elements that can solve difficult learning control problems' (1983). In: *IEE Transactions on Systems, Man, and Cybernetics* SMC-13, pp. 834–846.

Dua, Dheeru and Casey Graff (2017). *UCI Machine Learning Repository*. URL: http://archive.ics.uci.edu/ml.

Fischer, R. A. (1936). 'The use of multiple measurements in taxonomic problems'. In: *Annals of Genetics*.

Willis, Mark, Hugo Hiden, P. Marenbach, Ben McKay and Gary Montague (Oct. 1997). 'Genetic programming: An introduction and survey of applications'. In: pp. 314–319. ISBN: 0-85296-693-8. DOI: 10.1049/cp:19971199.

Miller, Julian F. (2020). 'Carhtesian Genetic Programming: It's status and future'. In: *Genetic Programming and Evolvable Machines* 21, pp. 129–168.

Walker, James A. and Julian F. Miller (2004). 'Evolution and Acquisition of Modules in Cartesian Genetic Programming'. In: *Lecture Notes in Computer Science* 3003, pp. 187–197.

Kaufmann, Paul and Marco Platzner (2008). 'Advanced techniques for the creation and propagation of modules in Cartesian genetic programming'. In: *GECCO'08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation 2008*, pp. 1219–1226.

Lovinger, David M. (2008). 'Communication Networks in the Brain: Neurons, Receptors, Neurotransmitters, and Alcohol'. In: *Alochol Research and Health* 31, pp. 196–214.

Goldman, Brian W. and William F. Punch (2013). 'Reducing Wasted Evaluations in Cartesian Genetic Programming'. In: *Genetic Programming*. Ed. by Krzysztof Krawiec, Alberto Moraglio, Ting Hu, A. Şima Etaner-Uyar and Bin Hu. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 61–72. ISBN: 978-3-642-37207-0.

Wills, Peter and François G. Meyer (Feb. 2020). 'Metrics for graph comparison: A practitioner's guide'. In: *PLOS ONE* 15.2, pp. 1–54. DOI: `10.1371/journal.pone.0228728`. URL: `https://doi.org/10.1371/journal.pone.0228728`.

Själander, Magnus, Magnus Jahre, Gunnar Tufte and Nico Reissmann (2019). *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*. arXiv: `1912.05848 [cs.DC]`.

Russel, Stuart J., Peter Norvig and Ernest Davis (2010). *Artificial Intelligence: a Modern Approach*. 3rd edition. Upper Saddle River, NJ.: Prentice Hall.

Appendix A

Appendix A: NMS Search Space Proof

Proof for equation (2.1). There are $C(Z,2) = Z * (Z-1)/2$ different possible edges in undirected graphs without loops. Although CGP programs are acyclic directed graphs, a CGP node is not allowed to form a connection to a node at a lower depth. As such the directed graphs in CGP programs are equivalent to acyclic undirected graphs in the sense that directionality can be determined from edge depth. Therefore, as the CGP graph has a maximal arity of T, there are a maximum of $Z * T/2$ possible edges in a CGP graph. However, as CGP functions have different input slots/parameters, the order of connections to a node is not irrelevant. For a node with T inputs, there are T! different input orderings. As such the amount of different possible edges in a CGP graph is $Z * T!/2$ As such there exists $2^{\frac{Z*T!}{2}}$ possible CGP graphs with N nodes.

M of these Z nodes are chosen as output nodes. As the ordering of output nodes matter there are $\frac{Z!}{(Z-M)!}$ different permutations.

Each of the Z active nodes can have Q different CGP node functions, so that for each of the possible graph and output structures there are Z*Q different node function selections.

Finally, the Z active nodes may be connected to the N input nodes in several ways. In the worst case where every input to every node is connected to every input node there is for each node $\frac{N!}{(N-T)!}$ different permutations of connections to the input nodes, and these connections have T! different orderings as input to each node, giving $Z * T! * \frac{N!}{(N-T)!}$ different combinations.

Therefore, there is an upper bound of $2^{\frac{Z*T!}{2}} * \frac{Z!}{(Z-M)!} * Q * Z * Z * T! * \frac{N!}{(N-T)!}$ different genotypes in the search space.

It is assumed that each of the Q node functions has an arity equal to the maximal arity T, which does not need to be true in general, making (2.1) a loose upper bound.

Appendix B

Appendix B: CGP Node Functions

One set of CGP node functions is shown in **??**. These node functions are used when permitting randomness.

NMS-LOC also has an option to use the node functions described in Julian F. Miller (2021). These were chosen as an alternative as they had already been used successfully in that research application, which is similar to NMS-LOC in that it searches for neuron models. The only difference is that NMS-LOC can have values outside of the range [-1, 1], which can occur because NMS-LOC may write to internal state registers and accumulate larger values. To avoid overflow the range is still capped to [-100, 100].

Neuron function	*Arity*	*Description*
Sine	1	sin(in1)
Addition	2	in1 + in2
Subtraction	2	in1 - in2
Multiplication	2	in1 * in2, output capped to absolute value of 100
Division	2	in1/in2, output capped to absolute value of 100, in2 set to +/- 0.01 if smaller
Gaussian Sampling	2	One random sampling of Gaussian N(μ=in1, σ=in2)

Table B.1: One set of node functions used in CGP in NMS-LOC

Appendix C

Appendix C: Config files

The following section documents the config files used in the experiments. The config files use the following format (C.1):

Config field	*Description*
Mutation chance node	Initial chance of node type mutation and output index mutation
Mutation chance link	Initial chance of mutating CGP links in genotype
Non-crossover children	If true, on crossover also generate two children of parent 1 which are copies of parent 1 without crossover
Smooth gradient	Establish an additional small reward factor for having a certain amount of neurons, connectivity between hidden neurons and connections to input and output neurons. Intent is to provide a smoother gradient. See Section C.1
Genome count	Amount of genomes in population which can create offspring. Minimum 2.
Neuron internal state count	How many internal states a neuron should have.
Axon-dendrite internal state count	How many internal states an axon-dendrite should have.
Signal dimensionality	The dimension of the signals sent through the network.

Hox variant count	How many hox variants each CGP functino should have.
Iterations	How many child-generation + evaluation steps should be done.
Hox crossover chance	During crossover chance of copying over a homeobox function from other parent (If 0 the two children has all of parent functions after mutation, designates how often to mix). Chance check is repreated for each hex variant for each function.
Hox duplication chance	On crossover, chance of copying a homeobox variant function over to the next homeobox variant of the function, overwriting the previous. To support duplication and differentiation.
Fail mutation chance node multiplier	Number that node mutation chance of genome is multiplied with on fail to find equally good or better child of parent genome.
Fail mutation chance link multiplier	Number that link mutation chance of genome is multiplied with on fail to find equally good or better child of parent genome.
Neutral mutation chance node multiplier	Same as above, only for when neutral change is found but not improvement.
Neutral mutation chance link multiplier	Same as above, only for when neutral change is found but not improvement.
Max mutation chance node	Maximum chance of node mutation outside of hypermutation
Max mutation chance link	Maximum chance of link mutation outside of hypermutation
Hypermutation mutation chance	Chance of node and link mutation during hypermutation
CGP progam size	How many CGP each CGP function/program may contain.
Logger ignore messages	Designate list of detailed logging messages to ignore

Seek dendrite tries	When an axon-dendrite attempts to find a connection, it can evaluate this number of candidates.
CGP function constant numbers	Provide a list of constant numbers that CGP functions are given as input.
Grid count	Amount of grids that the neuron geometric space should contain
Grid size	How many points there should be in each 3D directions per grid
Actions max	How many CGP functions the neuron engine may call per training instance and per reward evaluation.
Instances per iteration	How many times the neuron engine can do an input-output cycle (i.e. evaluate training sample/environment interaction) per genome evaluation.
Advanced logging	Enable or disable advanced logging
engine-random	Whether or not the engine interprets numbers in the range [0.0, 1.0> as the probability of doing something. Alternative is binary check on >= 1.0.
use-miller-funcs	If true the CGP node functions will be based on Julian F. Miller (2021), otherwise as described in Appendix D.
miller-and-random	If true and if use-miller-funcs the Gauss node function as described in Appendix D will be used in addition to the node functions described in Julian F. Miller (2021).

Table C.1: Config file description

C.1 Smoothened Gradient

Russel et al. (2010) recommends that reward functions only reward the desired outcome, and not how the designer thinks the solution should work. The smoothened reward function takes this into consideration by giving a full score for modest criteria which are necessary for required behaviour.

The reward function works by including a punishment for some conditions which are necessary to have interesting networks:

Neuron penalty: If there are less than 3 hidden neurons, give a penalty equal to (3-hidden neuron count)/3.

Neuron connectivity penalty: If average neuron connectivity is less than 3, give a penalty equal to (3-average penalty)/3.

Input connectivity penalty: If there are no connection to any inputs, give a penalty of 1.

Output connectivity penalty: If there are no connections to any output, give a penalty of 1.

C.2 Config files set 1

The config files shown in Tables **??**, **??**, **??**, **??** are three settings of increasing NMS-LOC complexity. As such the more complex configs, i.e. config 3 and then config 4 as most complex, have larger search spaces. Config 4 also has a config variant using smoothened gradient, to see how smoothened gradient affects navigating the largest search space.

Config field	*Value*
Mutation chance node	0.1
Mutation chance link	0.1
Non-crossover children	True
Smooth gradient	False
Genome count	10
Neuron internal state count	1
Axon-dendrite internal state count	1
Signal dimensionality	1
Hox variant count	1
Iterations	400
Hox crossover chance	0.01
Hox duplication chance	0.01
Fail mutation chance node multiplier	0.8
Fail mutation chance link multiplier	0.8
Neutral mutation chance node multiplier	1.2
Neutral mutation chance link multiplier	1.2
Max mutation chance node	0.1
Max mutation chance link	0.2
Hypermutation mutation chance	0.3
CGP progam size	50
Logger ignore messages	n/a
Seek dendrite tries	4
CGP function constant numbers	None
Grid count	6
Grid size	20
Actions max	75
Instances per iteration	100
Advanced logging	False

Table C.2: Config file 2

Config field	*Value*
Mutation chance node	0.1
Mutation chance link	0.1
Non-crossover children	True
Smooth gradient	False
Genome count	10
Neuron internal state count	1
Axon-dendrite internal state count	1
Signal dimensionality	1
Hox variant count	3
Iterations	400
Hox crossover chance	0.01
Hox duplication chance	0.01
Fail mutation chance node multiplier	0.8
Fail mutation chance link multiplier	0.8
Neutral mutation chance node multiplier	1.2
Neutral mutation chance link multiplier	1.2
Max mutation chance node	0.1
Max mutation chance link	0.2
Hypermutation mutation chance	0.3
CGP progam size	50
Logger ignore messages	n/a
Seek dendrite tries	4
CGP function constant numbers	1,2,10
Grid count	6
Grid size	20
Actions max	75
Instances per iteration	100
Advanced logging	False

Table C.3: Config file 3

Config field	*Value*
Mutation chance node	0.1
Mutation chance link	0.1
Non-crossover children	True
Smooth gradient	False
Genome count	10
Neuron internal state count	4
Axon-dendrite internal state count	4
Signal dimensionality	4
Hox variant count	3
Iterations	400
Hox crossover chance	0.01
Hox duplication chance	0.01
Fail mutation chance node multiplier	0.8
Fail mutation chance link multiplier	0.8
Neutral mutation chance node multiplier	1.2
Neutral mutation chance link multiplier	1.2
Max mutation chance node	0.1
Max mutation chance link	0.2
Hypermutation mutation chance	0.3
CGP progam size	50
Logger ignore messages	n/a
Seek dendrite tries	4
CGP function constant numbers	1,2,10
Grid count	6
Grid size	20
Actions max	75
Instances per iteration	100
Advanced logging	False

Table C.4: Config file 4

Config field	*Value*
Mutation chance node	0.1
Mutation chance link	0.1
Non-crossover children	True
Smooth gradient	True
Genome count	10
Neuron internal state count	4
Axon-dendrite internal state count	4
Signal dimensionality	4
Hox variant count	3
Iterations	400
Hox crossover chance	0.01
Hox duplication chance	0.01
Fail mutation chance node multiplier	0.8
Fail mutation chance link multiplier	0.8
Neutral mutation chance node multiplier	1.2
Neutral mutation chance link multiplier	1.2
Max mutation chance node	0.1
Max mutation chance link	0.2
Hypermutation mutation chance	0.3
CGP progam size	50
Logger ignore messages	n/a
Seek dendrite tries	4
CGP function constant numbers	1,2,10
Grid count	6
Grid size	20
Actions max	75
Instances per iteration	100
Advanced logging	False

Table C.5: Config file 4 Smoothened Gradient

Appendix D

Appendix D: Neuron and Axon-Dendrite Functions

Function	Inputs	Outputs
Hex selection	Position, internal states, constant values	Float for each hex variant, highest is chosen
Axon-dendrite birth	Position, internal state, dendrite count, constant values	RB: Add dendrite, RB: Send signal, signal output, RB: Run action controller, internal state delta
Signal axon-dendrite	Signal input, position, internal states, constant values	RB: Send signal, signal output, internal state delta, RB: Run action controller
Receive signal	Signal input, global position, internal states, constant values	Signal output, RB: Run action controller, internal state delta
Receive reward	Position, internal states, reward, constant values	Internal state delta, RB: Run action controller
Move	Position, internal states, constant values	RBs for movement in x, y, z direction, +/-, RB: Send signal, signal output
Die	Global position, internal states, constant values	RB: Die, RB: Send signal, signal output
Neuron birth	Position, internal states, constant values	RB: Birth neuron, internal state delta
Action controller	Position, internal states, constant values	RBs for adding each of the preceding actions to engine queue

Table D.1: Neuron Functions: Shows each Neuron CGP-learnt function with defined inputs and outputs.

Function	Inputs	Outputs
Receive signal from neuron	Position, internal states, input signal, constant values	Signal output, RB: Run action controller, internal state delta
Receive signal from axon-dendrite	Position, internal states, input signal, constant values	Signal output, RB: Run action controller, internal state delta
Signal neuron	Position, internal states, input signal, constant values	RB: Send signal, internal state delta, RB: Run action controller
Signal axon-dendrite	Position, internal states, input signal, constant values	RB: Send signal, internal state delta, RB: Run action controller
Accept connection request from axon-dendrite, constant values	Own position, requesting axon-dendrite position, own internal states, requesting axon-dendrite internal states	RB: Accept connection, own internal state delta.
Break connection	Own position, requesting axon-dendrite position, own internal states, requesting axon-dendrite internal states, constant values	RB: Break connection
Receive reward	Position, internal states, reward, constant values	RB: Run action controller, internal state delta
Die	Position, internal states	RB: Die
Action controller	Position, internal states, constant values	RB: For running each of the other actions

Table D.2: Axon-Dendrite Functions: Shows each Axon-Dendrite CGP-learnt function with defined inputs and outputs.

Appendix E

Appendix E: Statistics Explained

The following section gives a description of statistics gathered by the NMS-LOC system. The NMS-LOC system does gather other statistics as well, but that is "behind the hood" and not used for analyzing the system, only as options for adding further analysis tools. For some degree of brevity only used statistics are presented here.

These stats can perform aggregates over a single run, in which case they perform average, max, min, std, and count aggregation per timestep. They may also be gathered over several separate runs, in which the average of the above aggregates on each run separately is used.

total-active-nodes-average: On average how many nodes were used in the entire CGP genome (not per chromosome)

max-module-depth-average: The maximum module depth. A module which is not in another module has depth 1. A model which is in a module which is not in another module has depth 2, and so on.

module-count-avg: How many modules there were on average in the entire CGP genome (not per chromosome)

recursive-module-count-avg: How many modules contains other modules

module-size-avg: How many nodes a module usually contained

unique-output-neuron-connections-avg: How many unique neurons are connected to an output neuron.

unique-input-node-connections-avg: How many unique neurons are connected to an input neuron.

neuron-connectivity-avg: How many axon-dendrite connections the average hidden neuron has.

input-neuron-connectivity-avg: How many axon-dendrite connections the average input neuron has

output-neuron-connectivity-avg: How many axon-dendrite connections the average output neuron has

hox-switch-count-average: How many times a switch between homeobox functions is done *max-link-mutation-chances:* The maximum link mutation value in the population

min-link-mutation-chances: The minimum link mutation value in the population

avg-link-mutation-chances: The average link mutation value in the population

max-node-mutation-chances: The maximum node mutation value in the population

min-node-mutation-chances: The minimum node mutation value in the population

avg-node-mutation-chances: The average node mutation value in the population

max-fitness: The maximum fitness in the population

min-fitness: The minimum fitness in the population

avg-fitness: The average fitness in the population

std-fitness: The standard deviation of fitness in the population

genome-takeover-counts: How many times a genome takes another genomes population slot using the replacement mechanism

eval-time-averages: The average amount of seconds used to evaluate every genome in a produce-child step (children + parents evaluated)

neuron-counts-avg: How many neurons there are in the phenotype at the end of evaluation

population-entropy-avg: The shannon entropy of genome ID's in the population

better-swap-avg: The percentage of the parent population which is replaced by a child with better performance

neutral-swap-avg: The percentage of the parent population which is replaced by a child with equal performance

any-swap-avg: The percentage of the parent population which is replaced by a child

better-child-percentage-avg: The percentage of child genotypes which are better than one of their parent genotypes

neutral-child-percentage-avg: The percentage of child genotypes which are equally good to one of their parent genotypes

any-change-avg: The percentage of child genotypes which are equally good or better than one of their parent genotypes

dendrite-internal-state-use-count-average: The average amount of CGP connections to a dendrite internal state input in CGP functions

constant-number-use-avg: The average amount of CGP connections to a constant number value input in CGP functions

neuron-engine-dim-use-avg: The average amount of CGP connections to a neurons coordinates in the neuron engine input in CGP functions

signal-dim-use-avg: The average amount of CGP connections to a signal input in CGP functions

neuron-internal-state-avg: The average amount of CGP connections to a neuron internal state input in CGP functions

CGP node type uses: For each type of CGP function node gather how many times it is used in each genome.

Further, statistics about the behavior of phenotypes gathered. The following are gathered, all of which are actions in the neuron engine, except for dendrite-seek-connection and dendrite-accept-connection which are done outside of the action control flow - note that some actions may sometimes be done outside of the action control flow and in that case are not counted. All are presented as average aggregates over the population at a given iteration.

axon-recieve-signal-dendrite: How many times an axon receives a signal from an axon-dendrite.

axon-recieve-signal-neuron: How many times an axon receives a signal from a neuron.

axon-signal-dendrite: How many times an axon runs the program to check if it should signals a dendrite

dendrite-accept-connection: ... runs the program to check if it should accepts an incoming connection

dendrite-action-controller: ... runs it's action controller program

dendrite-axon-death-connection-signal: ... axon or dendrite runs the program to check if it should send a signal to another axon-dendrite upon it's own death

dendrite-axon-death-neuron-signal: ... axon or dendrite runs the program to check if it should send a signal to a neuron upon it's own death

dendrite-break-connection: ... axon or dendrite runs the program to check if it should break it's connection

dendrite-die: ... axon or dendrite runs the program to check if it should die

dendrite-recieve-reward: ... axon or dendrite runs the receive reward program

dendrite-recieve-signal-axon: ... runs the receive signal from axon program[1]

dendrite-recieve-signal-dendrite: ... dendrite runs the receive signal from dendrite program

dendrite-recieve-signal-neuron: ... dendrite runs the receive neuron program

dendrite-seek-connection: ... axon or dendrite seeks connection to another axon or dendrite

dendrite-signal-axon: ... dendrite runs the program to check if it should signal an axon

dendrite-signal-dendrite: ... axon runs the program to check if it should signal an dendrite

dendrite-signal-neuron: ... axon or dendrite runs the program to check if it should signal to its parent neuron

neuron-action-controller: ... neuron runs its action controller

neuron-axon-birth: ... runs the program to check if it should create an axon

neuron-dendrite-birth: ... runs the program to check if it should create an dendrite

[1] Although these are spoken of as going in two directions/being separate programs, it is the same program for axons and dendrites, but they are counted differently depending on the context of the direction of the signal

neuron-die: ... runs the program to check if it should die

neuron-hox-variant-selection: ... runs the hox variant selection program

neuron-move: ... runs the move program

neuron-neuron-birth: ... runs the program to check if it should birth another neurno

neuron-recieve-axon-signal: ... receives a signal from an axon

neuron-recieve-reward: ... receives a reward signal from the neuron engine

neuron-signal-axon: ... runs the program to check if it should signal an axon

neuron-signal-dendrite: ... runs the program to check if it should signal a dendrite

skip-post-death: ... an action skipped because the object doing the action has died. Not counted towards the action limit.

Appendix F

Appendix F: Preliminary Thesis

The following appendix documents the preliminary thesis work done in the autumn of 2021 (Rambjør (2021)), which forms the basis for the master's thesis. Please note that between the preliminary work and the master's thesis the author has changed their legal name from Jon Oddvar Rambjør to Sara Rambjør, but both the preliminary work and the thesis are written by the same person.

Cartesian Genetic Programming for searching for novel Neuron Models

Jon Oddvar Rambjør

Abstract

This paper presents Neuron Model Search (NMS) using Carthesian Genetic Programming for defining functions of an abstract neuron model. These neuron models then interact with a problem domain to grow neuron structures. Most neuroevolutionary approaches focus on finding neuron networks for a specific neuron model, while this approach focuses on finding the neuron models which grow networks while interacting with a problem domain. The intent is for NMS to find neuron models with interesting or unexpected emergent properties and behaviour. The paper describes the implemented NMS algorithm, presents and discusses results on a test problem to show that the algorithm functions, and presents plans for and other possible further work.

1 Introduction

Evolutionary algorithms can lead to novel and unexpected results (Lehman et al., 2018); (Miikkulainen, 2021), as such using evolutionary search to search for models of neurons may result in novel designs. This approach differs from contemporary deep learning approaches, as deep learning approaches are in essence multilayer perceptrons trained using a variant of backpropagation, and as such use a defined and specific neuron model. It also differs from other neuroevolutionary approaches, as although neuroevolutionary methods differ greatly in how networks are evolved, they still typically produce multi-layer perceptron neural networks (ex. (Stanley and Miikkulainen, 2002), (Jackobi, 1995)), or networks consisting of another well-specified neuron model such as spiking neurons (ex. (Elbrecht and Schuman, 2020)).

The approach discussed in this article, dubbed Neuron Model Search (NMS) defines an abstract model of a neuron by defining what a neuron can do, and then uses a variant of Genetic Programming (GP) called Cartesian Genetic Programming (CGP) to search for programs which define when and how a neuron should take a specific action. This differs from contemporary deep learning and neuroevolution approaches which use a completely specified neuron model and focuses on adjusting the weights of connections or the connection topology. NMS contains a concept of connections between neurons, but how signals are processed through the connections, dubbed axon-dendrites, is evolved. Likewise, the neuron topology is produced by the evolved neuron model. NMS therefore has the potential to find unexpected models of neurons within the set of possible designs which satisfy the set constraints imposed by a abstract partially specified neuron model (see Section 3, 4 and the Appendix). These models could be interesting in terms of the emergence of intelligent behavior, as well as potentially being a suitable solution methodology for some problems.

NMS also differs from other neuroevolutionary approaches in that it makes no seperation between the development of the network and the runtime of the network. Further, many (but not all) neuroevolutionary approaches use training data only to evaluate the fitness of proposed solutions, while NMS uses training data during the developmental phase - as such a single NMS genotype can map to several different solutions and may be able to solve several problems. In principle a single NMS genome could be trained on several problems, and instead learn how to adapt and adjust to the problem domain it encounters, making it possible for a single neuron model to solve multiple problems, and possibly showing some degree of proficiency on problem domains it has not been trained on. Therefore, the intent behind NMS can be viewed as laying a groundwork for searching for learning

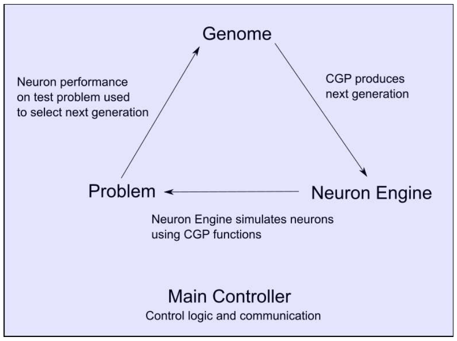

Genome
Neuron performance on test problem used to select next generation
CGP produces next generation
Problem
Neuron Engine
Neuron Engine simulates neurons using CGP functions
Main Controller
Control logic and communication

Figure 1: The overall logic of NMS. A CGP genome produces children, which are evaluated in a neuron simulation engine on a specific problem domain. By selecting children with a lower error as the next generation better models are evolved.

algorithms, rather than specific solutions to problems. This differs from deep learning and neuroevolution approaches which typically search for solutions to a specific problem, such as a minimizing the mean squared error of a model when trained on a specific training set.

Figure 1 shows the overall design of the NMS system, and the functionality of the system is discussed in more details in Sections 3 and 4. Additionally, the code for the version of the NMS algorithm used in this paper is available [1], and although this article presents the design of the NMS algorithm at a high level the code covers more technical details.

Ideally, NMS would be able to find neuron models capable of solving several problems. Because the method searches for models and not specific solutions and allows for interaction with the problem domain, NMS may be able to find neuron models implementing learning algorithms instead of specific solutions. This would be particularly interesting if a found model was able to solve a large class of problems using an unknown learning algorithm. Further, NMS searches specifically for functions determining when neurons should take specific actions, what signals they should send, and how they should change their internal state. This means that by tracing the neurons actions and state changes, and inspecting the found functions, it may be possible to explain how the found models work. As such, the planned experiments for the masters thesis will investigate the following questions:

1. Can NMS solve problems, specifically the n-pole balancing problem?

2. Can NMS find a genotype capable of solving several variants of the n-pole balancing problem?

3. Does starting the evolutionary search from an existing genotype improve search performance, i.e. training a genotype for the 2-pole balancing problem and then re-training it for the 3-pole balancing problem? Does this cause the genotype to forget how to solve the 2-pole balancing problem?

4. If a genotype is capable of solving several problems, can it also produce phenotypes capable of solving several problems at the same time? I.e., can one phenotype be found which solves several variants of the n-pole-balancing problem?

Due to the time constraints of the project, it is likely that not all of these questions can be investigated in detail or at all. This, and the novelty of the approach, indicates that focusing on one experiment at a time is the most suited research approach. By conducting the research on an experiment-by-experiment basis it is guaranteed that there is enough time to get some conclusive results, and it also allows flexibility in experimental design and selection. It is possible that the result of one experiment is useful for selecting further research directions. The master's thesis should nevertheless show whether NMS works and if it is suitable for further research, or if it does not work well, and if so, what could be done differently in similar approaches in the future. In this paper introductory experiments are done in a simple problem domain to show that the approach is capable of solving some problems to some degree.

2 Evolutionary Algorithms, Neuroevolution and CGP

This section gives a presentation of the core concepts of evolutionary computation and neuroevolution and provides examples of relevant approaches. Evolutionary algorithms work by searching over genotypes mapping them onto phenotypes whose fitness are evaluated and used

[1] https://github.com/jonoddram/CGP_Neuron_Masters/tree/semesteroppgave_2021

for selection. Core issues in evolutionary algorithms is the genotype representation, the genotype to phenotype mapping, the fitness function and how selection and reproduction should work. New genotypes are produced using the current best genotype(s) using mutation operators which makes some type of change in the genotype, or crossover operators which combine the genotype of two or more parents (Goldberg and Holland, 1988) (Eiben and Smith, 2015).

2.1 Introduction to Neuroevolution

Neuroevolution is a subfield of evolutionary algorithms focused on producing artificial neural networks. For a comprehensive overview see (Downing, 2015) or (Floreano et al., 2008). For a review of the state of the art of neuroevolution see (Stanley et al., 2019). Early work in neuroevolution often focused on evolving weight parameters in a fixed topology, such as in (Whitley, 1993). Later, work began on Topology and Weight Evolved Artificial Neural Networks (TWEANNs) in which both weights and topology is evolved. Earlier systems typically used direct genotype representations, where the topology and weights are directly encoded in the genotype (such as in (Stanley and Miikkulainen, 2002)), while later works moved on to using indirect genotype representation where a sophisticated mapping function describes how a phenotype can be produced from the genotype. The most well-known direct encoding approach may be NEAT (Stanley and Miikkulainen, 2002), and it is perhaps an endorsement of indirect approaches that NEAT too moved on to indirect encodings in the form of HyperNEAT (Stanley et al., 2009). NMS can be viewed as a form of indirect encoding, specifically a developmental approach.

Indirect genotypes allow for smaller genotypes which reduces the search space at the cost of introducing more human design in the mapping function. A smaller genotype is advantageous as it allows for evolving large networks without having very large genomes. Grammar-based approaches are an example of indirect encodings, which work by applying rules in a formal grammar to produce a neural network (Cangelosi et al., 1994). Some indirect encoding approaches use artificial chemical systems, such as Genomic Regulatory Networks (GRNs) (Jackobi, 1995) (Eggenberger, 1997). In GRNs chemical concentrations in and around cells define which genes should be active, which then defines which chemicals should be produced. The chemicals also define cell behavior, such as how they should migrate and connect, and how strong connection weights should be in the neural network phenotype produced by mapping from the chemical neuron simulation. Of these approaches GRNs are most similar to NMS, but there are two crucial distinctions. First, a distinction between training time and runtime is common in neuroevolution, where neuroevolutionary methods typically use a specific algorithm to produce an ANN, which then acts as a normal multilayer-perceptron model. However, in biological neural networks there is not such a strict distinction between learning time and runtime (ex. neurogenesis in adults (Zhao et al., 2008), or the simple facts that children can exhibit intelligent behaviour while their brains are developing, and that adults can still learn), therefore moving away from this separation could produce advances in lifetime learning and transfer learning. Secondarily, the aforementioned approaches only use training data to evaluate the fitness of genotypes. Biological neural structures grow by reacting and interacting with their surroundings, not through a one-to-one mapping from the genotype. For both reasons NMS has no separation between training/development time and runtime, using the same neuron model for both.

2.2 Developmental Approaches

NMS has a lot in common with other developmental approaches, who also produce a phenotype through interaction with training data. One difference is that developmental approaches sometimes map the developed structure onto a multilayer peceptron (ex. (**?**)). However, this is not always the case, and NMS has more in common with developmental approaches which do not map the solution to a multilayer perceptron. For example, Astor & Adami (Astor and Adami, 2000) outline an approach based on GRNs and chemical gradients where input neurons emit chemicals depending on their state, eventually producing a multi-cell phenotype from a GRN genotype. Despite the similarities to Astor & Adami's work, NMS attempts to move away from the use of chemical gradients as to not need to simulate chemicals to make computation more efficient, and to have a greater focus on CGP-program centric models, in the hopes that this would increase interpretability and explainability.

Several other authors have used CGP to evolve neural networks. The Cartesian Genetic Programming Artificial Neural Network (CGPANN) approach extends CGP-graphs to include weights on directed links, and then uses standard CGP-techniques to evolve networks (See (Turner and Miller, 2013), (Khan et al., 2010), (Khan et al., 2013)). The most relevant approach to this approach is however work is CGP Developmental Networks (CGPDN), which use CGP to evolve functions for use in a developmental process.

Julian, Wilson & Cussact-Blanc (Miller et al., 2019) presents a CGPDN consisting of two CGP programs: One simulating a neuron soma, and one simulating a dendrite. Using internal state variables and hyperparameter defined increments and action thresholds a one-dimensional network is grown. This continues by running their soma and dendrite programs a given number of times, or until fitness decreases in extracted ANNs. This makes Julian, Wilson & Cussact-Blanc approach an indirect encoding approach with a complicated genotype-phenotype mapping dependent on primarily the genotype and secondarily the fitness landscape of the problem domain for stopping the developmental process instead of a process which really interacts with the problem domain. Similarly to other neuroevolutionary methods it also makes a distinction between the development phase of the network and the runtime network.

In conclusion; NMS is a method inspired by several other neuroevolutionary algorithms but attempts to be more general. In a sense most neuroevolutionary approaches can be viewed as algorithms for finding a specific multi-layer perceptron for a specific problem, while NMS instead seeks for a controller for a neuron "robot", which interacts with other neuron robots to solve one or more problems requiring learning (or equivalently, NMS constitutes an agent-based approach (Laubenbacher et al., 2013) to neural network intelligence). By not extracting standard ANNs from the developed networks, by not separating the development and runtime phase and by not using a one-to-one genotype-phenotype mapping and by searching for "neuron robot controllers" (i.e., neuron models) NMS may be able to find novel solutions. This comes at the cost of potentially reduced computational efficiency, and the potential of neuron models being unstable and degenerating which cannot occur with standard ANNs extracted from developmental networks. NMS attempts to include less human design of the neuron models and learning models, and is instead a method for leveraging computation, which can be viewed as being in line with the design philosophy argued for in Sutton's Bitter Lesson (Sutton, 2019).

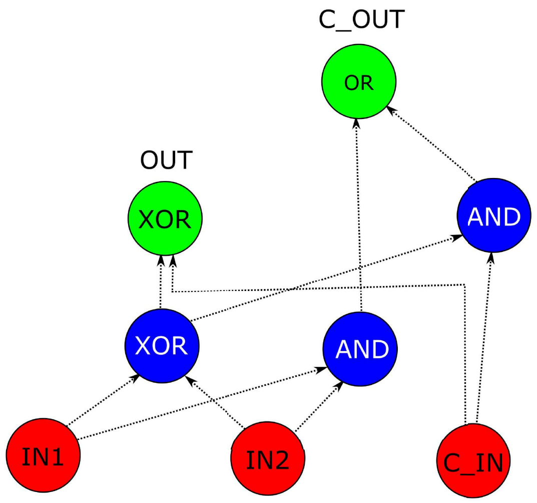


Figure 2: Example of a CGP function equivalent to a 1-bit full adder. Inputs are sent form bottom up, red nodes are inputs, green are outputs, blue neither.

3 CGP Modifications

In this section the NMS systems use of CGP is discussed, introducing CGP briefly as well as describing the modifications and choices made for this specific system.

3.1 Introduction to CGP

GP is a subfield of evolutionary algorithms which searches for computer programs by searching over syntax parse trees (Willis et al., 1997). CGP is a variant of GP where programs are represented as directed acyclic graphs. CGP genomes define nodes by defining which function they execute over their input(s), and which nodes they get input from. Additionally, the genome which nodes are output nodes. Input nodes are added as a part of decoding the genome (Miller, 2020). Many variants of CGP exists, of note for this work are modular CGP-functions, where other CGP-programs can be used as node functions (Walker and Miller,

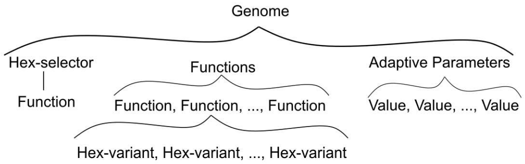


Figure 3: Shows the structure of the NMS genome.

2004). Consider figure 2, which shows an example of a CGP function computing the same function as a full adder. Modular CGP basically works by allowing a single node to be equivalent to a learned CGP-program, such as a full adder.

CGP does not use crossover operators by default. Default CGP uses mutation operators which can add nodes up to a predefined limit, change edge connections up to a maximum arity of the node or change node type. CGP genomes can contain inactive nodes which are either not fully connected to input or output nodes, and mutation over these nodes serves as neutral drift facilitating evolutionary exploration (Miller, 2020). The standard evolution strategy consists of maintaining one parent genotype and producing a given number of offspring and selecting the offspring with the highest fitness that is higher or equal to the parent to facilitate neutral drift through mutations in the inactive nodes. However, by maintaining only a single population member it may be difficult to explore a wide area of the design space. Therefore, the variant of CGP used in NMS introduces a crossover operator to facilitate the use of larger populations.

3.2 Details on CGP variant used in NMS

In this work the genome is spilt in three parts as illustrated in Figure 3. The first part defines a Homeobox selection program, which defines which variant of the other functions the neuron or axon-dendrite should use. The second part consists of all the other neuron and axon-dendrite functions along with their homeobox variants. The final part of the genome contains adaptive control parameters (Eiben and Smith, 2015). The genome can contain several variants of each Function by defining another CGP-program, the Hex selector, which uses coordinates and neuron state variables to determine which function variant to use. This is inspired by homeobox genes in vertebrate biology, where chemical markers in the body dictate which variant of an organism's genome should be used, ex. to determine whether to make an arm or a leg, or an hippocampus or frontal lobe (Downing, 2015). However, like Stanley & Miikkulainen (Stanley and Miikkulainen, 2003) point out computer programs can access cell coordinates directly, and as such there is no need to maintain chemical gradients for this purpose.

An alternative to having several functions for every neuron action would be to have a single master-control program controlling all possible input and output actions. The choice to have several programs is inspired by Walker et. al. (Walker and Miller, 2004) which successfully solved a circuit design problem by seperating the problem into n-subproblems where n is the desired amount of circuit outputs. Unlike Walker et. al. (Walker and Miller, 2004) the function programs have several outputs, and each genotype has a shared fitness instead of an individual fitness for each function as the fitness of each function is dependent on their relation to the other functions. The intent of using a design which has several functions (see the Appendix for a list) instead of one unfiied neuron-controller program is to simplify the search space, and to make the functionality of found programs easier to interpret. The design decision is also inspired by biology, specifically the theory of facilitated variation (Gerhart and Kirschner, 2007). Facilitated variation states that evolutionary variation in the post-pre-Cambrian era is primarily done through searching over combinations of gene blocks called core processes, which are gene blocks that are stable and robust to being combined with other blocks in many ways. The theory postulates that robust evolutionary variation can be achieved through evolving different ways for these core processes to interact and interregulate, rather than mutating the processes themselves. Each function can be viewed as a core process which has a specific functionality and is only weakly linked to the others through the neurons internal states and the effects each function has on the neurons behavior, which the implemented crossover operations attempt to take advantage of.

Crossover is implemented at two levels, one of which is also based on facilitated variation. Functions and their homeobox variants can be swapped with a given probability, and the order of the homeobox variants may also change with some probability, similarly adaptive hyperparameters can be swapped. The other level of

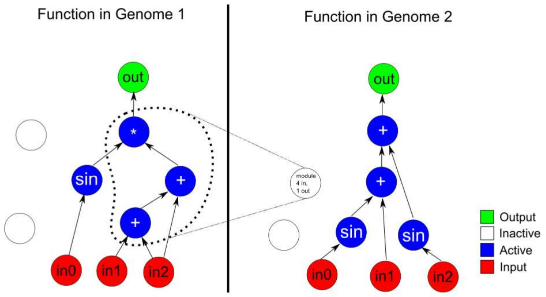


Figure 4: Illustrates how crossover works at the CGP-function level. A part of one parent genomes active nodes is made into a node function, and set as the node function executed by an inactive node in a copy of the other parent genome. This inactive node may then become active through mutation.

crossover is at the CGP level, where functions can be crossovered by extracting sub-graphs of one function and using it as a modular function in an inactive node in the other function, as shown in Figure 4. Modular CGP functions therefore act as core processes, potentially accelerating evolution and allowing greater complexity by finding good modules. Pairs of genomes are selected as crossover pairs randomly, more complex crossover selection could be implemented in the future, for example based on fitness or common ancestry.

After crossover mutation is applied. CGP mutation in this work uses the standard CGP-mutation operator of changing or adding edges or changing the type of a node, with one change: The existing modular CGP-functions in a genome is added to the set of types a node can mutate to.

Modular CGP-programs were introduced in (Walker and Miller, 2004), but unlike in Walker & Miller modules can contain other modules, and can not be expanded and changed after creation. Modules in this work are instead based on Kaufmann & Platzner (Kaufmann and Platzner, 2008), specifically modules are created from cones (i.e., beginning with a node and picking nodes connected to it such that there is always a path from any node to the first picked node). Kaufmann & Platzner also investigated using crossover, but found it tended to lead to increased computational cost - in case this applies to the NMS problem domain the code supports populations as low as 2, minimizing this overhead, and could be customized to single-population as in regular CGP relatively easily.

Each pair of parents create two or four children - four if using an optional setting which produces two extra children without using crossover, two otherwise. If a child has equal or higher fitness than its parent, then it replaces the parent. This is done instead of selecting the highest fitness genotypes overall for the next generation to reduce the speed at which dominant genotypes take over the population, in the hopes of preserving more diversity in the search. More sophisticated diversity-preserving mechanisms could be implemented in the future. A child of equal fitness replaces its parent to facilitate neutral drift, as this allows traversing a plateau in the fitness landscape.

Mutation rates can be configured using hyperparameters. By default, it is recommended to start with high mutation rates. On failure to perform neutral drift or positive improvements, that is, when stuck in the fitness landscape, the mutation rate is be decreased such that neutral drift is more likely to occur. If the mutation rate goes lower than a threshold hypermutation is be triggered, as a low enough mutation rate indicates that the genome is close to a local maximum, and needs to take larger steps to escape.

It should be noted that implementation of adaptive hyperparameters and homeobox function variants is not finished due to time constraints. These parts of the design will be implemented at the start of the next semester along with the n-pole balancing problems.

4 Neuron model & Engine

The neuron engine consists of a three-dimensional Cartesian grid, consisting of n-by-n-by-n discrete positions. Each position can contain several neurons which saves computational resources for checking and handling neuron collisions. Similarly, there is no concept of dendrite collisions. This is done to save computational resources on detecting and handling collisions and to take advantage of the fact that the artificial neurons are not actually constrained by physical space.

Axons and dendrite are unified in axon-dendrites, which each are connected to a neuron, and can be connected to another axon-dendrite or be a free axon-dendrite. The distinction is done based on whether signals are being sent forwards from dendrite to axon, or backwards through axon to the dendrite, and depending on the direction different programs are used for processing and transmitting signals. A unified axon-dendrite model was selected to reduce the search space. Neurons can perform the actions as defined in Appendix A. Input and Output neurons can perform no actions, but are always considered as having free dendrites, such that the genotype-controlled neurons can connect to them. When a neuron seeks a connection to an axon-dendrite it samples a power-law distribution to determine the target distance in order to favor shorter connections as in the brain (Downing, 2015), but this assumption could be reduced to make the model more general. This assumption is also made to avoid simulating axon-dendrite movement. To simplify distance calculations the overall grid is divided into sub-grids, and the sub-grid with free axon-dendrites with the closest grid-wise distance to the target distance.

When initializing the neuron grid for evaluating a genotype it is set to contain a single genotype-controlled neuron that is not connected to anything, as well as input and output nodes. For each problem instance the neuron engine allows up to a given amount of neuron functions to execute, and then stops the neuron engine to avoid infinite loops and select for quicker programs. When given the next problem instance the grown network is maintained, such that the growing neuron structure can learn.

The neuron simulation engine maintains an action queue, which determines which neuron function should be run next. Each action in the queue has a timestamp, and the lowest timestamp in the queue is always selected as the next action. When there are several actions with the same timestamp, they are selected in a first-in-first-out manner. The timestamp system ensures that actions are executed in a temporally sensible manner and gives each neuron equal access to computational resources.

Signals sent between neurons and axon-dendrites can be multi-dimensional, i.e., several floats can be sent in one signal. The number of floats per signal is a configurable hyperparameter. Likewise, the number of internal state variables is a configurable hyperparameter. These two design decisions are inspired by biological neurons, which maintain complex chemical states internally and in their local area (Lovinger, 2008), (Holland, 1998). Further, there is no reason to assume that one-dimensional signals are necessarily optimal.

A complete list of neuron and axon-dendrite functions is given in Table 1 and Table 2. Most functions are given internal state variables and neuron position as input, and some are given signals as input.

5 Results

During this semester software for NMS has been developed, along with analysis and logging tools to understand solutions. NMS was tested on a simple problem domain consisting of classifying binary numbers from 0 to 15 as either 1 or 0, where numbers less than 8 are 0. The system was able to do this with some proficiency, achieving at best mean squared error of 0.4 averaged over 50 samples when evolved with a signal arity of 1 and 1 internal state in neurons and dendrites. This is a good result, as the primary interest was to determine whether NMS could work at all, not achieving good performance on a test problem.

However, looking into the program logs reveals some interesting information about NMS. First, it is unable to evolve as much as try random genotypes and stick with the ones which work. This is evident by modular CGP-programs not being present in the program, and the inability of NMS to find better solutions from a good solution, effectively getting stuck in the state space. These problems are likely interrelated as the inability to use modules would make it more difficult to take larger steps in the state space, due to a limited genome size and requiring more mutations to re-evolve a module rather than starting to use it after crossover.

To alleviate these problems several changes will be made. First, cone-selection for module creation will be based on node age, as suggested by (Kaufmann and Platzner, 2008), as this could lead to more useful modules being evolved. Secondly, the probability of changing the function a neuron executes will be scaled with the number of inputs it requires: Modules can have more inputs than the standard unary and binary node functions, and are therefore more likely to mutate to another function before they evolve enough input connections. Thirdly, during mutation of node types it will be possible to mutate into a module that exists in the entire genome and not just the specific CGP function, as this will allow the functions in the genome to share information about useful modules. In

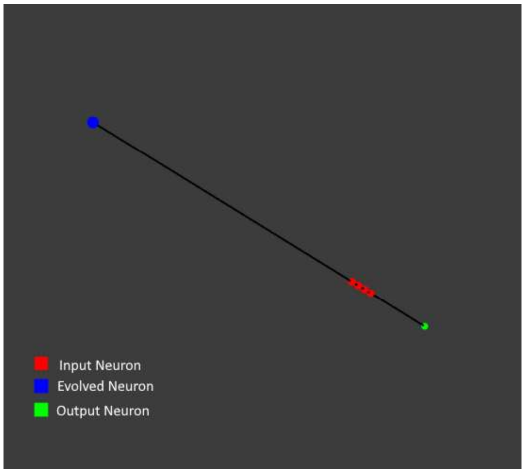


Figure 5: Neuron structure/phenotype produced by a genotype with MSE of 0.4. Although hard to tell from figure, looking at logs confirms that neuron is connected to all output and input neurons.

view of Kaufmann & Platzners (2008) findings that crossover operators using modular CGP increased computational cost the third improvement may be particularly important, as Kaufmann & Platzners results indicate that for crossover to be useful it may need to be coupled with other mechanisms.

Some other conclusions can be drawn from the genotypes which are able to solve the test problem to some degree. They only consist of a single evolved neuron, connected to the input nodes and to the output node, as shown in figure 5. Still, NMS is capable of creating structures with several neurons, as shown in figure 8, but as the figure also shows the investigated programs struggle to connect these neurons. To remedy this and make it easier to learn to develop multi-neuron structures the engine will be changed to include a connection between a parent and a child neuron per default. These can be contrasted with figure 6, which shows how a more desirable neuron structure might look like. The capability of producing multi-neuron structures is desirable, partially because this may provide insight into the evolution of cooperating neurons similar conceptually to biological neurons, and because a single neuron has a limited computational capacity due to the finite amount of memory and the finite size of its CGP-programs which means that some computations may only be possible with multiple neurons.

Looking into the engine execution logs of the best programs also reveals that they do not exhibit adaptive or reactive behaviour, but rather repeats a learned sequence of actions. In other words, the evolved genome does not react to the input signal in any meaningful way other than transmitting a signal to the output neuron. If genomes are trained using several versions of the problem or several problems this may improve, as it would make learning more complex behavior more impactful on performance. This result is somewhat analogous to the motivation for incremental learning discussed in (Gomez and Miikkulainen, 1996). It is also possible that more complex neuron models could be found if allowing for more computation per sample, and for longer evolutionary runs in general. Upping internal state count and signal arity will also allow for more complex behavior, but at the same time increase the size of the search space.

Looking into the population maintained by NMS evolution reveals that good solutions do not spread and are unable to take over the population at all. In one way this is positive, as it enforces diversity, but on the other hand it also makes the algorithm unable to exploit searching around good solutions, increasing the chance of finding better versions. An explicit spreading mechanism will be implemented, which will have a chance of replacing the worst population member with one of the better ones with likelihood proportional to their difference in fitness, given that the fitness difference is sufficiently large. This way good solutions will be able to spread in the population, while also ensuring that diversity is not discarded in favour of a slightly better solution.

As an example of a evolved CGP function consider figure 7. This figure uses software developed for this project to display the CGP program. Annotations in white and black are edited in for readability. Inputs are given to the red input nodes,and are transmitted upwards. Green nodes are output nodes. This program displays the logic a neuron uses when determining if it should transmit a signal or not. In essence this program always sends a signal of 0 and may increment the internal state variable with one. Because of this the single neuron in the solution can always transmit an answer to the output node. As an illustrative example also consider figure 9, which is a simplified illustration of a hypothethical CGP function that computes the sine of the input added with a bias term, and al-

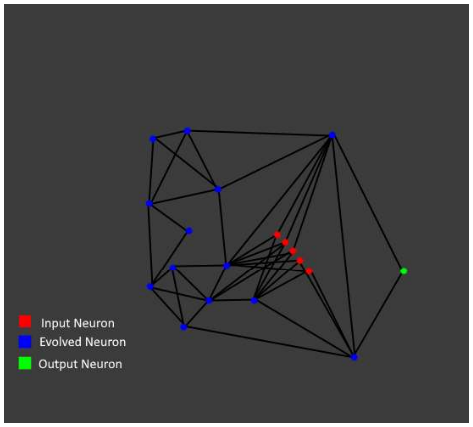


Figure 6: How a hypothetical idealized neural structure might look like. Any network connecting several neurons to solve a problem would be an improvement, but networks which can produce groups of connected neurons/cell assembly (Holland, 1998)/modularized neuron structures and recurrent would also be promising as their presence may indicate complexity and because these elements are present in the human brain (Downing, 2015).

ways transmits a signal.

Figure 9 is hypothethical, and also illustrates a considered addition to the CGP-framework, where some numbers, in this case one, is always given as input to each CGP-function. Giving CGP-functions access to some constant numbers would make it easier for CGP to evolve some types of functions, as it can rely on a input remaining constant, and because the function would otherwise require the evolution of a way to produce the number. The disadvantage is that by increasing the amount of CGP-inputs one also increases the amount of different functions that can be learned with a given number of nodes, which would increase the size of the search space per available node. Still, if the increased search space is increased because it contains more useful functions this would be beneficial.

Software used to create figure 5 and figure 8 was also developed for this project. During the masters more work will be done on improving the logging and analysis software, including producing more detailed and readable logs, and removing bugs. Further development of logging was not done this semester due to time constraints.

6 Planned Experiments

The following planned experiments are planned around the use of the n-pole balancing problem, but equivalent experiments can be conducted in any problem domain which contains classes of problem instances of varying complexity. This is advantageous as if the n-pole balancing domain is too complex for the available computational resources then a simpler problem domain can be used. The experiments also utilize the test problem described in Section 5, but any simple problem could substitute this.

6.1 Experiment 1

Compare the fitness over several epochs when searching for a neuron model to solve the 1-pole pole-balancing problem when beginning the search from a random genotype and from a high-performing genotype on the test problem. Average fitness per epoch over several runs using different random seeds.

Hypothesis: Using existing genotypes capable of solving some problem when starting genotype search in new problem domains leads to quicker convergence than using randomly initialized genotypes. This is expected because any genotype capable of solving problems needs to be able to connect axon-dendrites and transmit signals. Additionally, beginning evolutionary search from known solution is a well-known strategy within Evolutionary Algorithms (Eiben & Smith, 2015).

6.2 Experiment 2

Train a genotype for the 1-pole balancing problem and the 2-pole balancing problem. Compare the fitness convergence when utilizing each genotype on the opposite problem and with a baseline randomly initialized genotype.

Hypothesis: In both cases convergence will be quicker than from a randomly initialized genotype, both because of the assumption of similarity as discussed in Experiment 1 but also because the problem domains are similar. It is expected also that 2-pole to 1-pole transfer learning should be quicker than 1-pole to 2-pole because 2-pole balancing is the more complex problem.

6.3 Experiment 3

Train a genotype to solve both 1-pole balancing and 2-pole balancing, such that each time a new pole-balancing task is started the 1-pole or 2-pole

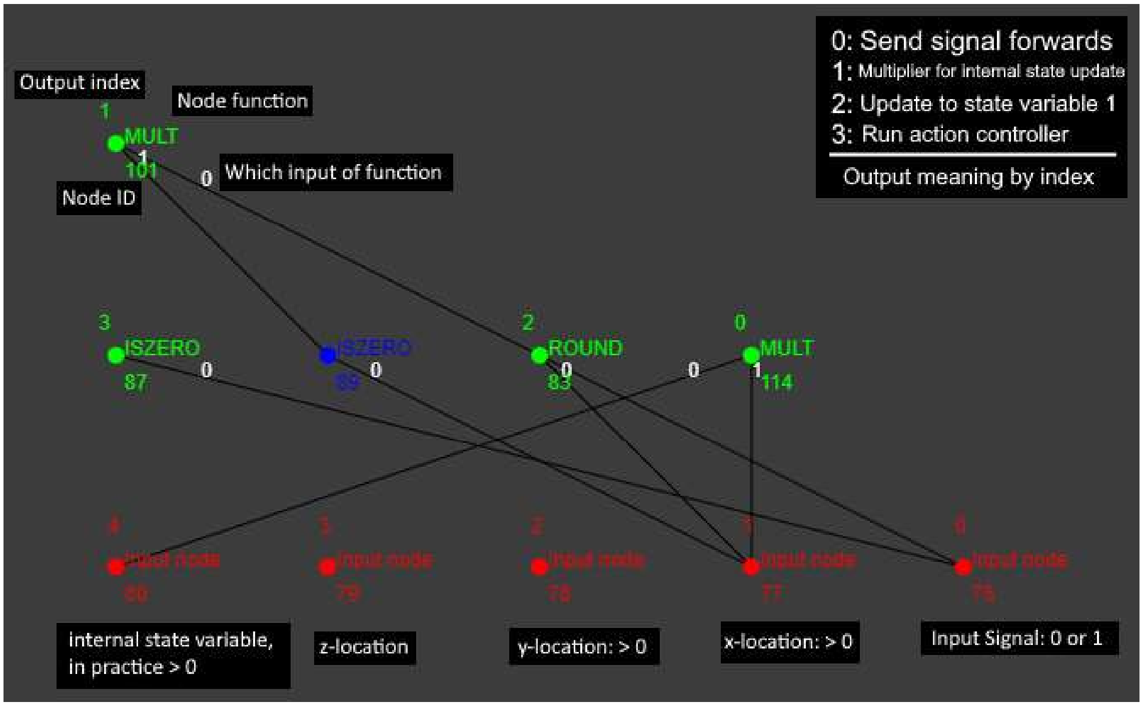


Figure 7: Shows the CGP program evolved for sending signals from neurons.

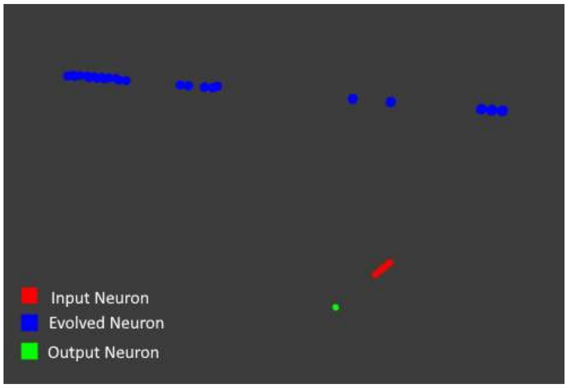


Figure 8: Phenotype/neuron structure which does not solve the test problem. Although it produced many neurons, it does not connect them with axon-dendrites.

is chosen randomly. Can a genotype grow a phenotype which is able to solve both tasks simultaneously?

Hypothesis: A genotype can grow a phenotype capable of solving several types of problems with some proficiency. By analyzing neuron firing patterns across the two problems, it can be determined whether or not the network uses the same neurons for each problem type, and if it does this indicates that NMS has found a model capable of learning what the problems have in common to some degree.

6.4 Finishing remarks on planned experiments

In addition to the experiments analysis will be conducted to determine how they work, as in Section 5. The experiments are designed to be relevant to the research goals presented in Section 1, while also taking into consideration the time constraints of the master's thesis. In addition to being limited in scope each experiment will be conducted as an individual package, to ensure that there is time to conduct the experiment, analyze results and write for at least some of the experiments. If there is sufficient time the experimental results will likely help guide what to investigate further, as such the experiment plan deems these three experiments

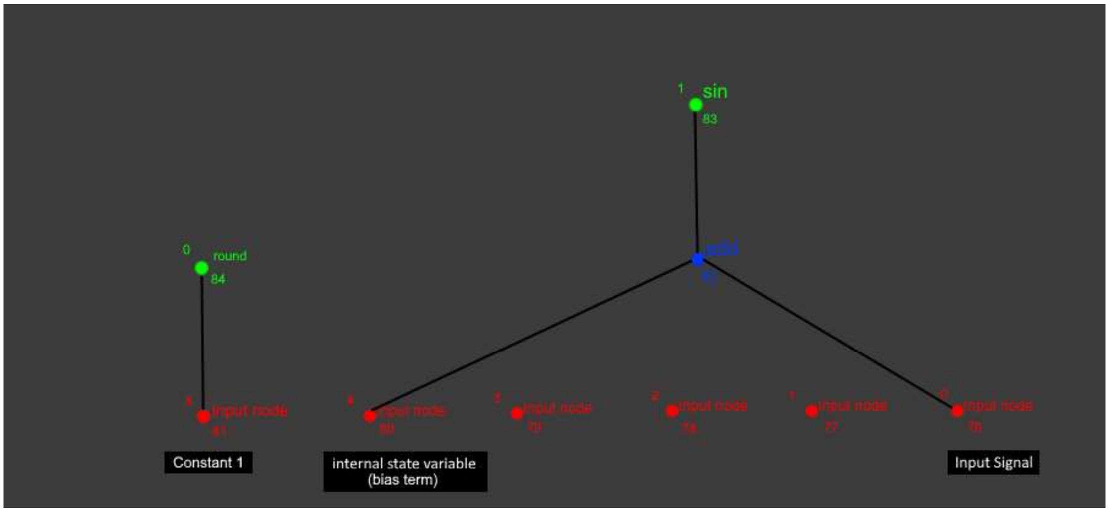


Figure 9: Illustrates a hypothetical evolved CGP-function equivalent to a normal sinusoidal perceptron with a single input

sufficient.

7 Conclusion

By searching for computational models of neurons it may be possible to detect novel models of neuron functionality, which can help guide further research into neuron models and the emergence of intelligence. The results in Section 6 show that the NMS approach is capable of learning to solve a simple problem to some degree. However, the results show that several weaknesses of the implemented NMS system, such as only finding solutions which execute a set sequence of actions, and which only utilize one evolved neuron. Further, it is an open question whether NMS can find neuron models capable of solving more general classes of problems, which will be investigated further in the thesis. However, the results do show that NMS is able to find a solution of some quality to the problem, which indicates that the concept of searching for neuron models has potential and is therefore worthy of further research. The software developed during this semester will developed further during the thesis semester and used to investigate NMS further.

8 Future Work

Suggested further research and algorithmic improvements are presented in the Future Work section. The suggestions made here include algorithmic improvements and further research that will likely be outside of the scope of the master's thesis, although the master's thesis may touch upon related topics. As such this section provides suggestions for parties interested in continued research or development of the NMS algorithm.

8.1 Further study

Of particular interest is investigating if a single genome can be evolved that is can solve a large class of problems. This is interesting because to accomplish this the genome needs to have evolved a capability for learning, which could inform further design of learning algorithms or be interesting in terms of the study of emergent intelligent behavior. To train such a genome it may be beneficial to first train the genome on a set of relatively easy problems, then as performance increases progressively add in more and more complex problems - the assumption being that the genomes capable of solving the easier and the harder problems is at least a partial subset of the genomes capable of solving the easier problems. This concept of incrementally introducing more difficult tasks has been discussed previously in neuroevolution (Gomez and Miikkulainen, 1996), but due to NMS searching for neuron models rather than specific solutions may also be applied to tasks from different problem domains. A simple version of this is present in the system, in the form of an optional smoothed gradient option, which gives a penalty for failing sub-tasks like having any connections at all, having any evolved neurons, producing some output; in practice sub-tasks which must be solved to solve a task in any real problem domain.

Although this work uses a specific algorithm for NMS, it is in principle possible to define other models in which to search for functions. The intent of this design is not necessarily to present an optimal algorithm for NMS, but rather to present an algorithm for NMS at all - as such further research can likely find other similar approaches which are more efficient or have other advantages. In the master's thesis a section will be included defining a conceptual framework for an abstract unified connectionist model, from which other connectionist models can be created on a conceptual level through the application of constraints to the abstract model, with the intent of providing a cognitive framework for comparison and design of connectionist models.

8.2 Algorithmic improvements

Each action with the same timestamp in the neuron engine could be executed in parallel, but implementation is non-trivial as this can introduce some resource conflicts. Several actions involving the same neuron or axon-dendrite introduces a conflict on the entities internal state, there can occur conflicts over free axon-dendrites, and the neuron engine would become non-deterministic due to a non-deterministic sequence of calls to a random function. With care these issues could potentially be alleviated or designed around. Parallel computation could also be done on a genome-phenotype mapping level.

Biological neural networks, such as those in the human brain, rely on neuromodulators deployed in brain regions to learn. The human basal ganglia seems to approximate TD-learning through releasing dopamine when experiencing suprise and other emotions (Gershman et al., 2014) (Downing, 2015). In the current design of the system neuromodulators can only be simulated through sending signals between neurons one by one. This may be difficult to learn, and because the neuron engine limits the amount of neuron functions that can be run the system may also be selecting against the emergence of approaches similar to regional neuromodulators. However, neurons are divided into sub-grids of the global neuron grid. As such an approximation of neuromodulators could be made by giving each sub-grid an internal state and allowing all neurons in the grid to access this state, in practice giving neurons in the same sub-grid a shared memory. If desirable chemical diffusion and chemicals could also be simulated at a subgrid level at a far smaller cost than simulating over each grid position.

It should be noted that if reward feedback is given to the NMS algorithm, which may or may not be done depending on the problem domain, then the NMS algorithm will select for neuron models which function well when given reward feedback. If these genotypes or trained phenotypes were applied to real-world problems which do not supply reward feedback performance may degenerate. A potential solution could be to include a phase where rewards are not given in training to select for genotypes producing stable phenotypes.

References

J.C. Astor and Chrstoph Adami. 2000. A developmental model for the evolution of artificial neural networks. *Artificial Life*, 6:189–218.

Angelo Cangelosi, Domenico Parisi, and Stefano Nolfi. 1994. Cell division and migration in a 'genotype' for neural networks (cell division and migration in neural networks). *Network Computation in Neural Systems*, 5.

Keith L. Downing. 2015. *Intelligence Emerging: Adaptivity and Search in Evolving Neural Systems*. The MIT Press.

Peter Eggenberger. 1997. Creation of neural networks based on developmental and evolutionary principles. In *Artificial Neural Networks - ICANN'97*, pages 337–342.

A.E. Eiben and James E. Smith. 2015. *Introduction to Evolutionary Computing*.

Daniel Elbrecht and Catherine Schuman. 2020. Neuroevolution of spiking neural networks using compositional pattern producing networks. In *International Conference on Neuromorphic Systems 2020*, pages 1–5.

Dario Floreano, Peter Dürr, and Claudio Mattiussi. 2008. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1:47–62.

John Gerhart and Marc Kirschner. 2007. The theory of facilitated variation. *Proceedings of the National Academy of Sciences*, 104(suppl 1):8582–8589.

Samuel Gershman, Ahmed Moustafa, and Elliot Ludvig. 2014. Time representation in reinforcement learning models of the basal ganglia. *Frontiers in Computational Neuroscience*, 7:194.

David E Goldberg and John Henry Holland. 1988. *Genetic algorithms and machine learning*. Kluwer Academic Publishers-Plenum Publishers; Kluwer Academic Publishers ....

Faustino Gomez and Risto Miikkulainen. 1996. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5, 10.

John H. Holland. 1998. *Emergence: From Chaos To Order*.

Nick Jackobi. 1995. Harnessing morphogenesis. In *International Conference on Informatino Processing in Cells and Tissues*, pages 29–41.

Paul Kaufmann and Marco Platzner. 2008. Advanced techniques for the creation and propagation of modules in cartesian genetic programming. In *GECCO'08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation 2008*, pages 1219–1226.

Maryam M. Khan, Gul M. Khan, and Julian F. Miller. 2010. Evolution of neural networks using cartesian genetic programming. In *IEEE Congress on Evolutionary Computation*.

Maryam M. Khan, Arbab M. Ahmad, GUl M. Khan, and Julian F. Miller. 2013. Fast learning neural networks using cartesian genetic programming. *Neurocomputing*, 121.

Reinhard Laubenbacher, Franziska Hinkelmann, and Matt Oremland. 2013. Chapter 5 - agent-based models and optimal control in biology: A discrete approach. In Raina Robeva and Terrell L. Hodge, editors, *Mathematical Concepts and Methods in Modern Biology*, pages 143–178. Academic Press, Boston.

Joel Lehman, Jeff Clune, Dusan Misevic, Christoph Adami, Julie Beaulieu, Peter Bentley, Samuel Bernard, Guillaume Beslon, David Bryson, Nick Cheney, Antoine Cully, Stephane Donciuex, Fred Dyer, Kai Olav Ellefsen, Robert Feldt, Stephan Fischer, Stephanie Forrest, Antoine Frénoy, Christian Gagneé, and Jason Yosinksi. 2018. The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *Artificial life*, 26:274–306, 03.

David M. Lovinger. 2008. Communication networks in the brain: Neurons, receptors, neurotransmitters, and alcohol. *Alochol Research and Health*, 31:196–214.

Risto Miikkulainen. 2021. Creative ai through evolutionary computation: Principles and examples. *SN Computer Science*, 2:163–170.

Julian F. Miller, Dennis G. Wilson, and Sylvain Cussat-Blanch. 2019. Evolving developmental programs that build neural networks for solving multiple problems. *Genetic Programming Theory and Practice*, pages 137–178.

Julian F. Miller. 2020. Carhtesian genetic programming: It's status and future. *Genetic Programming and Evolvable Machines*, 21:129–168.

Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127.

Kenneth O. Stanley and Risto Miikkulainen. 2003. A taxonomy for artificial embryogeny. *Artificial Life*, 9:93–130.

Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. 2009. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15:185–212.

Kenneth O. Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. 2019. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1:24–35.

Richard Sutton. 2019. The bitter lesson.

Andrew J. Turner and Julian F. Miller. 2013. Cartesian genetic programming encoded artificial neural networks: A comparison using three benchmarks. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, pages 1005–1012.

James A. Walker and Julian F. Miller. 2004. Evolution and acquisition of modules in cartesian genetic programming. *Lecture Notes in Computer Science*, 3003:187–197.

Dominic Stephen Das Rajarshi Anderson Charles W. Whitley, Darrell. 1993. Genetic reinforcement learning for neurocontorl problems. *Machine Learning*, 13.

Mark Willis, Hugo Hiden, P. Marenbach, Ben McKay, and Gary Montague. 1997. Genetic programming: An introduction and survey of applications. pages 314 – 319, 10.

Chunmei Zhao, Wei Deng, and Fred H. Gage. 2008. Mechanisms and functional implications of adult neurogenesis. *Cell*, 132(4):645–660.

9 Appendixes

Shows details of the functions learnt for neurons and axon-dendrites. Seperation between axons and dendrites is internal in neurons, to handle which way signals are being sent, but the same functions are used. RB is short for random bool, where output is 1 if sampling an uniform distribution between 0 and 1 yields a number lower than the program output, else 0. CGP-learnable Neuron functions are shown in Table 1, while CGP-learnable Axon functions are shown in Table 2.

Function	Inputs	Outputs
Hex selection	Position, internal states	Float for each hex variant, highest is chosen
Axon-dendrite birth	Position, internal state, dendrite count	RB: Add dendrite, RB: Send signal, signal output, RB: Run action controller, internal state delta
Signal axon-dendrite	Signal input, position, internal states	RB: Send signal, signal output, internal state delta, RB: Run action controller
Recieve signal	Signal input, global position, internal states	Signal output, RB: Run action controller, internal state delta
Recieve reward	Position, internal states, reward	Internal state delta, RB: Run action controller
Move	Position, internal states	RBs for movement in x, y, z direction, +/-, RB: Send signal, signal output
Die	Global position, internal state	RB: Die, RB: Send signal, signal output
Neuron birth	Position, internal states	RB: Birth neuron, internal state delta
Action controller	Position, internal states	RBs for adding each of the preceeding actions to engine queue

Table 1: Neuron Functions: Shows each Neuron CGP-learnt function with defined inputs and outputs.

Function	Inputs	Outputs
Recieve signal from neuron	Position, internal states, input signal	Signal output, RB: Run action controller, internal state delta
Recieve signal from axon-dendrite	Position, internal states, input signal	Signal output, RB: Run action controller, internal state delta
Signal neuron	Position, internal states, input signal	RB: Send signal, internal state delta, RB: Run action controller
Signal axon-dendrite	Position, internal states, input signal	RB: Send signal, internal state delta, RB: Run action controller
Accept connection request from axon-dendrite	Own position, requesting axon-dendrite position, own internal states, requesting axon-dendrite internal states	RB: Accept connection, own internal state delta.
Break connection	Own position, requesting axon-dendrite position, own internal states, requesting axon-dendrite internal states	RB: Break connection
Recieve reward Die	Position, internal states, reward Position, internal states	RB: Run action controller, internal state delta RB: Die
Action controller	Position, internal states	RB: For running each of the other actions

Table 2: Axon-Dendrite Functions: Shows each Axon-Dendrite CGP-learnt function with defined inputs and outputs.

CGP for evolving artificial neuron models

NTNU
Kunnskap for en bedre verden