

Amund Tenstad

# Deep Evolvable-Substrate HyperNEAT

Extending ES-HyperNEAT with Multiple Substrates  
in an Evolving Topology

Master's thesis in Computer Science

Supervisor: Pauline Haddow

July 2020



Amund Tenstad

# Deep Evolvable-Substrate HyperNEAT

Extending ES-HyperNEAT with Multiple Substrates in  
an Evolving Topology

Master's thesis in Computer Science  
Supervisor: Pauline Haddow  
July 2020

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





## Abstract

Neuroevolution is a technique that evolves artificial neural networks through evolutionary algorithms, inspired by the natural evolution of biological brains. HyperNEAT is one such method, evolving patterns to determine neural networks' weights based on their geometry within a substrate. Evolvable-Substrate HyperNEAT (ES-HyperNEAT) has extended the method to additionally evolve the network's geometry. Multi-Spatial Substrate (MSS) extends HyperNEAT in another direction, evolving separate patterns to determine the weights of a network constructed across multiple substrates.

This thesis introduces the new framework Deep Evolvable-Substrate HyperNEAT (DES-HyperNEAT), combining the characteristic features of Deep HyperNEAT, ES-HyperNEAT and MSS. The principal novelty of DES-HyperNEAT is the extension of ES-HyperNEAT, from network construction within a single substrate, to network construction across multiple substrates. The new approach essentially evolves deep neural networks by evolving and combining numerous substrates. DES-HyperNEAT separates complexity over multiple substrates and CPPNs, while also having the advantage of evolving node positions within each substrate. Additionally, it dynamically evolves deeper networks by inserting new substrates throughout evolution. The combination of these properties makes DES-HyperNEAT unique.

Three implementations of the framework are proposed. Through statistical analysis, the implementation Layered DES-HyperNEAT is selected, and its properties optimized. It evolves multiple unique CPPNs, one for each substrate and each connection between them. Its performance is evaluated with the datasets Iris, Wine, and Retina. It is shown that Layered DES-HyperNEAT consistently outperforms HyperNEAT and ES-HyperNEAT both when comparing fitness to runtime and number of generations.



## Sammendrag

Neuroevolusjon er en metode som utvikler kunstige neurale nettverk via evolusjonære algoritmer og er inspirert av den naturlige evolusjon av biologiske hjerner. HyperNEAT er en slik metode. Den utvikler mønstre til å bestemme neurale nettverks vekter basert på deres geometri i et substrat. Evolvable-Substrate HyperNEAT (ES-HyperNEAT) har utvidet metoden til å også utvikle nettverkens geometri. Multi-Spatial Substrate (MSS) utvider HyperNEAT i en annen retning, ved å utvikle forskjellige mønstre til å bestemme vektene i et nettverk som er konstruert over flere substrater.

Denne oppgaven introduserer rammeverket Deep Evolvable-Substrate HyperNEAT (DES-HyperNEAT), som kombinerer de karakteristiske egenskapene til Deep HyperNEAT, ES-HyperNEAT og MSS. Den viktigste innovasjonen med DES-HyperNEAT er utvidelsen av ES-HyperNEAT, fra nettverkskonstruksjon i et enkelt substrat til nettverkskonstruksjon over flere substrater. Den nye metoden utvikler dype nevrane nettverk ved å utvikle og kombinere en rekke substrater. DES-HyperNEAT skiller kompleksitet over flere substrater og CPPN-er, samtidig som den har fordelene med å utvikle node-posisjoner i hvert substrat. I tillegg utvikler den dynamisk dypere nettverk, ved å sette inn nye substrater under evolusjonen. Kombinasjonen av disse egenskapene gjør DES-HyperNEAT unik.

Tre implementasjoner av rammeverket foreslås. Basert på statistisk evaluering av disse blir implementasjonen kalt Layered DES-HyperNEAT valgt og optimalisert. I denne utvikles flere unike CPPN-er, ett for hvert substrat og hver forbindelse mellom dem. Metodens ytelse blir evaluert med datasettene Iris, Wine og Retina. Det vises at Layered DES-HyperNEAT konsekvent gir bedre resultater enn HyperNEAT og ES-HyperNEAT både når ytelse sammenliknes med kjøretid og antall generasjoner.





## Preface

The work herein constitutes a master's thesis in Computer Science at the Norwegian University of Science and Technology (NTNU). The initial literature review was conducted during a Specialization Project in the fall of 2019, where most of the underlying knowledge behind chapter 2 and 3 were accumulated. During the Master's Project in the spring of 2020, the DES-HyperNEAT framework and its implementations were designed and implemented, and all experiments were conducted.

I thank my supervisor Pauline Haddow for excellent guidance and feedback throughout the entire process.

Amund Tenstad  
Oslo, July 22, 2020



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Goals and Research Questions . . . . .	3
1.2	Research Method . . . . .	4
1.3	Structured Literature Review Protocol . . . . .	4
1.3.1	Search Terms and Sources . . . . .	4
1.3.2	Selection Criteria . . . . .	4
1.4	Preliminary Process . . . . .	5
1.5	Contributions . . . . .	6
1.6	Thesis Structure . . . . .	6
<b>2</b>	<b>Background Theory</b>	<b>7</b>
2.1	Machine Learning . . . . .	7
2.1.1	Supervised Learning . . . . .	7
2.1.2	Reinforcement Learning . . . . .	9
2.1.3	Generalization . . . . .	9
2.2	Artificial Neural Networks . . . . .	10
2.3	Evolutionary Algorithms . . . . .	11
2.3.1	Genetic Algorithm . . . . .	11
2.3.2	Development and Indirect Encoding . . . . .	13
2.3.3	Speciation . . . . .	13
2.4	Neuroevolution . . . . .	14
2.5	Neuroevolution of Augmenting Topologies . . . . .	14
2.6	Compositional Pattern-Producing Networks . . . . .	18
2.7	Hypercube-Based NEAT . . . . .	19
2.8	Deep Learning . . . . .	21
<b>3</b>	<b>State of the Art</b>	<b>23</b>
3.1	Deep Neural Networks . . . . .	23
3.2	Depth and Complexity . . . . .	24
3.3	Network Encoding . . . . .	25
3.4	Network Connectivity . . . . .	26

3.5	Multiple Substrates and CPPN Complexity . . . . .	27
3.6	ES-HyperNEAT Node Search . . . . .	30
<b>4</b>	<b>Model</b>	<b>33</b>
4.1	DES-HyperNEAT . . . . .	33
4.1.1	Introduction . . . . .	33
4.1.2	I/O Configuration . . . . .	36
4.1.3	Layouts . . . . .	36
4.1.4	CPPNs . . . . .	38
4.1.5	Assembling Networks . . . . .	39
4.2	Modified Node Search . . . . .	42
4.3	DES-HyperNEAT Implementations . . . . .	45
4.3.1	Layered DES-HyperNEAT . . . . .	46
4.3.2	Single CPPN DES-HyperNEAT . . . . .	48
4.3.3	Coevolutional DES-HyperNEAT . . . . .	49
4.3.4	Comparison . . . . .	51
4.4	Design Choices . . . . .	51
4.4.1	Framework . . . . .	52
4.4.2	Search and Depth Limits . . . . .	52
4.5	Implementation Details . . . . .	53
4.5.1	Custom Implementation . . . . .	53
4.5.2	Representation . . . . .	53
4.5.3	Activation Functions . . . . .	54
<b>5</b>	<b>Experiments and Results</b>	<b>55</b>
5.1	Introduction . . . . .	55
5.1.1	Experimental parameters . . . . .	55
5.1.2	Hyperparameters . . . . .	56
5.1.3	Results . . . . .	58
5.2	Preliminary Testing . . . . .	58
5.3	Experimental Plan . . . . .	60
5.4	Experimental Setup . . . . .	62
5.5	Phase 1: DES-HyperNEAT Implementations . . . . .	64
5.5.1	Experiment 1: DES-HyperNEAT Implementations . . . . .	64
5.6	Phase 2: Identity Mapping . . . . .	70
5.6.1	Experiment 2: Node Search Modifications . . . . .	71
5.6.2	Experiment 3: Identity Mapping . . . . .	73
5.7	Phase 3: Layered DES-HyperNEAT Tuning . . . . .	75
5.7.1	Experiment 4: I/O Substrate Depth . . . . .	75
5.7.2	Experiment 5: Hidden Substrate Depth . . . . .	77
5.7.3	Experiment 6: I/O Configuration . . . . .	82

5.7.4	Experiment 7: Layered DES-HyperNEAT Parameter Search	87
5.8	Phase 4: Related Methods Comparison . . . . .	87
5.8.1	Experiment 8: Related Methods Comparison . . . . .	88
<b>6</b>	<b>Conclusion</b>	<b>95</b>
6.1	Results and Discussion . . . . .	95
6.2	Comparison with state of the art . . . . .	98
6.3	Contributions . . . . .	99
6.4	Future Work . . . . .	100
	<b>Bibliography</b>	<b>101</b>
	<b>Appendix</b>	<b>105</b>



# List of Figures

2.1	Iris class segmentation . . . . .	9
2.2	Neural network example . . . . .	11
2.3	Genetic algorithm process . . . . .	12
2.4	NEAT crossover operation . . . . .	16
2.5	Compositional Pattern Producing Network . . . . .	18
2.6	HyperNEAT weight assignment . . . . .	20
3.1	Multi-Spatial Substrates . . . . .	28
3.2	ES-HyperNEAT point selection . . . . .	32
4.1	Terminology . . . . .	34
4.2	DES-HyperNEAT overview . . . . .	35
4.3	I/O configuration . . . . .	36
4.4	An evolved layout (a) and the network (b) assembled within it . . . . .	37
4.5	DES-HyperNEAT CPPN . . . . .	39
4.6	Multi-substrate iterative network completion part 1 . . . . .	40
4.7	Topologically sorted layout . . . . .	41
4.8	Multi-substrate iterative network completion part 2 . . . . .	42
4.9	Identity mapping between substrates . . . . .	43
4.10	Searched identity mapping pattern . . . . .	44
4.11	Layered DES-HyperNEAT . . . . .	46
4.12	Single CPPN DES-HyperNEAT . . . . .	48
4.13	Coevolutional DES-HyperNEAT . . . . .	49
5.1	Retina experiment . . . . .	63
5.2	Experiment 1: Performance results charts . . . . .	67
5.3	Experiment 5.2: Performance results charts . . . . .	81
5.4	Experiment 8: Performance results - Retina charts . . . . .	90
5.5	Experiment 8: Performance results - Iris and Wine charts . . . . .	92





# List of Tables

2.1	Subset of the Iris dataset . . . . .	8
4.1	Implementations comparison . . . . .	50
4.2	Implemented activation functions . . . . .	54
5.1	Example experiment 1: Experimental parameters . . . . .	56
5.2	Example experiment 2: Parameter search . . . . .	57
5.3	Example results . . . . .	57
5.4	Resulting hyperparamaters from preliminary testing . . . . .	60
5.5	Experimental plan . . . . .	61
5.6	Fitness functions . . . . .	62
5.7	Default node configurations . . . . .	63
5.8	Experiment 1: DES-HyperNEAT implementations . . . . .	64
5.9	Experiment 1: Performance results . . . . .	66
5.10	Experiment 1: Execution speed results . . . . .	68
5.11	Experiment 1: Network complexity results . . . . .	69
5.12	Experiment 2: Node search modifications . . . . .	71
5.13	Experiment 2: ES-HyperNEAT performance results . . . . .	72
5.14	Experiment 2: DES-HyperNEAT performance results . . . . .	73
5.15	Experiment 3: Identity mapping . . . . .	74
5.16	Experiment 3: Performance results . . . . .	74
5.17	Experiment 4: I/O substrate depth . . . . .	75
5.18	Experiment 4: Performance results . . . . .	76
5.19	Experiment 5.1: Hidden substrate depth . . . . .	77
5.20	Experiment 5.1: Performance and execution speed results . . . . .	79
5.21	Experiment 5.2: Hidden substrate depth - Part 2 . . . . .	81
5.22	Experiment 5.2: Performance and network complexity results . . . . .	82
5.23	Experiment 6: I/O configuration . . . . .	84
5.24	Experiment 6: Performance results . . . . .	85
5.25	Experiment 7: Layered DES-HyperNEAT parameter search . . . . .	87
5.26	Experiment 8: Related methods comparison . . . . .	89

5.27	Experiment 8: Performance results . . . . .	91
6.1	HyperNEAT related methods comparisons . . . . .	99
A1	Default hyperparameters . . . . .	106
A2	General parameter search grids . . . . .	107
A3	NEAT parameter search grids . . . . .	107
A4	CPPN parameter search grids . . . . .	108
A5	ES-HyperNEAT parameter search grids . . . . .	108
A6	Layered DES-HyperNEAT search grids . . . . .	109

# Introduction

The biological brain is an incredibly complex system, able to reason and solve diverse problems. It has inspired the creation of Artificial Neural Networks (ANNs), simulating the brain's neurons and synapses. Neuroevolution is one of the many approaches taken to create such artificial replicas. The field draws inspiration from natural evolution and generates ANNs through evolutionary algorithms. Neuroevolutionary methods have proven to generate solutions to problems of moderate sizes, and can even outperform humans at specific tasks [Hausknecht et al., 2014]. One such method is NeuroEvolution of Augmenting Topologies (NEAT) [Stanley and Miikkulainen, 2002]. It evolves the structure of neural networks and their weights, in contrast to other machine learning methods that optimize the weights of fixed topologies. Networks evolved with NEAT are initialized small, and their complexity increases during evolution. The evolutionary process is terminated when a solution of sufficient quality is discovered so that the solution is compact. It enables NEAT to adapt to problems of various sizes.

Although the sizes of networks produced by NEAT are adaptive in size, there is an upper bound for feasible network sizes. NEAT uses a direct encoding, meaning all nodes and connections are present in the genome. The direct representation, where the genome's contents directly map to the produced network, limits the sizes of networks evolved with NEAT. Since NEAT mutates genomes at a node and link level, the impact of a random mutation decreases as networks grow. Gillespie et al. [2017] found that evolution is unlikely to find good weights for large genomes evolved with NEAT because the search space is too large compared to the impact of isolated mutations.

An indirect representation is another way of encoding information about the solution in a genome. Instead of representing all the information directly, it

contains the information required to derive the solution. The indirect encoding is more volatile because a modification to the description may alter the result more than a modification to the result itself. The impact of isolated mutations can thus be much more significant with indirect encodings. Indirect representations could thus be able to evolve more complex networks than what is feasible with NEAT's direct encoding. Hypercube-based NEAT (HyperNEAT) [Stanley et al., 2009] is an extension of NEAT that uses such a representation.

HyperNEAT relies on a predefined fixed topology network, defined in a two-dimensional space called a substrate. An additional network is evolved with NEAT and is used to assign weights to the arbitrarily large fixed topology. The weight-assigning network is called a Compositional Pattern Producing Network (CPPN) [Stanley, 2007], and assigns weights to connections of a network based on the network's node locations. The positions of two nodes are fed into the CPPN. It then outputs the weight of the connection between the two nodes. A significant drawback with this approach is that one has to manually construct the fixed network, which can be an increasingly difficult task for higher complexity problems. If networks are constructed with poor geometry, high-quality solutions will be more difficult to discover by evolution [Pugh and Stanley, 2013], leading to solutions of poor quality.

Evolvable-Substrate HyperNEAT (ES-HyperNEAT) [Risi and Stanley, 2012] was designed to omit a manually defined topology requirement. The topology is instead derived through analysis of the weight patterns output by the CPPN. The topology of the network is thus evolved along with its weights, as in NEAT, but with an indirect encoding. Without the limitation of a direct encoding, the networks evolved by ES-HyperNEAT could potentially grow larger and solve more complex problems. However, when the networks are more extensive, the CPPN network must logically also be more complex, to support the increased number of weights to assign. As the CPPN is evolved with NEAT, with a direct encoding, it may become challenging to evolve sufficiently complex CPPNs.

Pugh and Stanley [2013] found that a single CPPN can struggle to generate good weights for a network in a single substrate. They propose that the weights of the HyperNEAT network are placed between multiple substrates, instead of in a single one. Each pair of substrates can then have a dedicated CPPN output node generate weights for the connections between nodes in the two substrates. By distributing the work of assigning weights to multiple CPPN output nodes, their complexity can be reduced. Instead of having one CPPN output capturing the entire weight-space, multiple ones can optimize their unique network parts.

Reduction of the required CPPN complexity will likely also benefit ES-HyperNEAT because of its similarities with HyperNEAT. The topic of this thesis is to investigate it by applying multiple substrates to ES-HyperNEAT. Unlike the multi-substrate extension of HyperNEAT, the topology of substrates will not be predefined and static. It will be evolved with NEAT, so no manual construction is required, and the method can dynamically adapt the network size to the problem's complexity.

## 1.1 Goals and Research Questions

The goal of the thesis is as follows:

**Goal** *Investigate how ES-HyperNEAT can be extended with multiple substrates in an evolving topology, to reduce the required complexity encapsulated by a single CPPN, and increase adaptation to problems through gradual complexification.*

It is desired to distribute the evolved network over multiple substrates and use one CPPN output node per substrate. Additional CPPN outputs per pair of substrates will also be used to generate weights for the connections between substrates. The distribution will reduce the number of weights determined by a single CPPN output node. It is hypothesized that it is easier to evolve good weights when a single CPPN node no longer has to encapsulate the entire network's weights.

It is additionally desired to evolve the topology of substrates to make the method dynamically adapt to problems of different complexities. The method could initially use a single or a few substrates and gradually add new substrates during evolution. As in NEAT, it should produce a compact solution while also alleviating manual topology construction.

It is believed that achieving this goal will allow the method to adapt to problems of different sizes and evolve good weights faster, even when the solution requires high complexity.

The following research questions are explored to accomplish the goal:

**Research question 1** *How can the topology in a multi-substrate layout be evolved in parallel with each of its individual substrates?*

**Research question 2** *How can nodes in different substrates be connected in a way that allows the layout's topology to be evolved?*

**Research question 3** *How should the inputs and outputs of the problem be organized in substrates so that the method can produce the best results?*

## 1.2 Research Method

The chosen research method comprises three phases. First, an analytical process, analyzing research within the fields of neuroevolution and HyperNEAT. HyperNEAT related methods and extensions are analyzed, and their traits evaluated. With the knowledge gathered in the first phase, the second phase comprises the design and implementation of algorithmic models based on aspects of existing work and new ideas. In the final phase, experiments are conducted to evaluate critical aspects of the proposals through simulation. Statistical analysis is used to evaluate them and for comparison with prior work.

## 1.3 Structured Literature Review Protocol

This section describes the protocol used to search for and gather relevant literature to answer the research questions and accomplish the goal described in section 1.1. The searched terms and sources are first presented in subsection 1.3.1, followed by the selection criteria in subsection 1.3.2.

### 1.3.1 Search Terms and Sources

A literature review is initially conducted to gain knowledge relevant to accomplishing the goal and answering the research questions. The results of the initial search mainly define the scope of this work. All the results of the search are reviewed to make sure that no information is lost. The term *hyperneat OR tweann OR (neuroevolution AND deep) OR (“indirect encoding” AND network) OR “architecture search” AND evolution* is used to search for relevant literature, with the search engines *IEEE Xplore* and *The ACM Guide to Computing Literature*. Relevant literature, either cited in or citing material from the search, is also evaluated and included when appropriate.

After the initial literature search is completed, additional literature is gathered by searching *Google Scholar*. It is chosen because the search engine indexes many others, so a single relevance ordered list of results can be reviewed. In contrast to the initial review, not all results from the additional searches are reviewed. The initial review is extensive and discovers the methods and knowledge defining the scope, while later searches refine it.

### 1.3.2 Selection Criteria

The following inclusion criteria (IC) and quality criteria (QC) are used to select relevant literature from the search process described in subsection 1.3.1. Extra

focus is given to methods with indirect encodings or increased depth, although these features are not required.

**IC1** *The study's primary concern is the evolution of both topology and weights of neural networks.*

**IC2** *The study's focus is a method, not an application.*

**IC3** *The study's method does not employ CNN, is not specifically recurrent, and does not require gradient descent.*

**IC4** *The environments in which the study's method is evaluated is static.*

**IC5** *The study presents empirical results.*

**QC1** *The study presents a precise aim.*

**QC2** *The study is put into the context of other studies.*

**QC3** *The study reflects on design decisions and their implications.*

## 1.4 Preliminary Process

A research goal was developed by investigating how neuroevolutionary methods can produce deep neural networks. Initial research focused on neuroevolution with direct encodings. The focus then shifted to methods utilizing indirect encodings, as these are able to produce deeper neural networks. Increased depth makes the networks more complex, which likely enables them to solve more complex problems. Research into evolving both the topology and weights with an indirect encoding lead to the discovery of HyperNEAT. The limitation that networks must be manually constructed was identified, and research into how it has been mitigated was conducted. It led to the discovery of Evolvable Substrate HyperNEAT (ES-HyperNEAT) [Risi and Stanley, 2012]. The method introduces the concept of an evolving substrate, which alleviates manual network construction. However, the method only utilizes a single substrate, in contrast to Multi-Spatial Substrate HyperNEAT, which was researched alongside HyperNEAT. Therefore, it was decided to devote this work to remove limitations that restricted the size of networks crated with ES-HyperNEAT, specifically extend it to utilize multiple substrates. It enables the evolution of deeper and more complex networks than in ES-HyperNEAT. Therefore, it may solve more complex problems.

## 1.5 Contributions

The main contribution within this thesis is the introduction of the framework Deep Evolvable Substrate HyperNEAT (DES-HyperNEAT). It is the combination of the dynamic node placement from ES-HyperNEAT [Risi and Stanley, 2012] and an evolving substrate topology that is novel. The framework and implementation are described in chapter 4.

A contribution is also made to the node search algorithm proposed by Risi and Stanley [2012], specifically to allow for an identity function between substrates. These modifications are described in section 4.2.

The final contribution is an open-source implementation of the DES-HyperNEAT framework. It is available from a public Git repository at <https://github.com/tenstad/des-hyperneat>.

## 1.6 Thesis Structure

The background theory is introduced in chapter 2, and the state of the art is described in chapter 3. Chapter 4 presents the proposed framework, DES-HyperNEAT, along with three implementations of it. The conducted experiments are presented in chapter 5, and the findings are concluded in chapter 6.



# Background Theory

The following chapter presents the background theory for this work. The field of machine learning is presented in section 2.1 and artificial neural networks in section 2.2. The background leading up to HyperNEAT is then presented in the following order: evolutionary algorithms in section 2.3; the field of neuroevolution in section 2.4; the neuroevolution of augmenting topologies algorithm in section 2.5; compositional pattern-producing networks in section 2.6; HyperNEAT in section 2.7. Finally, a brief comparison between deep learning and neuroevolution is presented in section 2.8.

## 2.1 Machine Learning

A machine learning *model* is like a mathematical function. When it is given one or more values  $x$ , it returns one or more values  $y$ , just like the function  $y = f(x)$ . Through experience in an environment, the model automatically updates its internal representation to improve its performance, thereby learning.

This section presents two types of machine learning, supervised learning in subsection 2.1.1, and reinforcement learning in subsection 2.1.2. The concept of generalization is then presented in subsection 2.1.3.

### 2.1.1 Supervised Learning

In supervised learning, input-output pairs,  $(x, y)$ , are used to train the model. It is trained to yield the correct output  $y$  for each input  $x$  by learning the relationship between the inputs and outputs. These pairs of data are often divided into three sets: *training data*, *validation data*, and *test data*. The model uses the training

Sepal length	Sepal width	Petal length	Petal width	Name
5.1	3.5	1.4	0.2	Setosa
5.4	3.9	1.7	0.4	Setosa
7.0	3.2	4.7	1.4	Versicolor
5.5	2.5	4.0	1.3	Versicolor
6.3	3.3	6.0	2.5	Virginica
5.8	2.7	5.1	1.9	Virginica

**Table 2.1: Subset of the Iris dataset.** Data from Dua and Graff [2017].

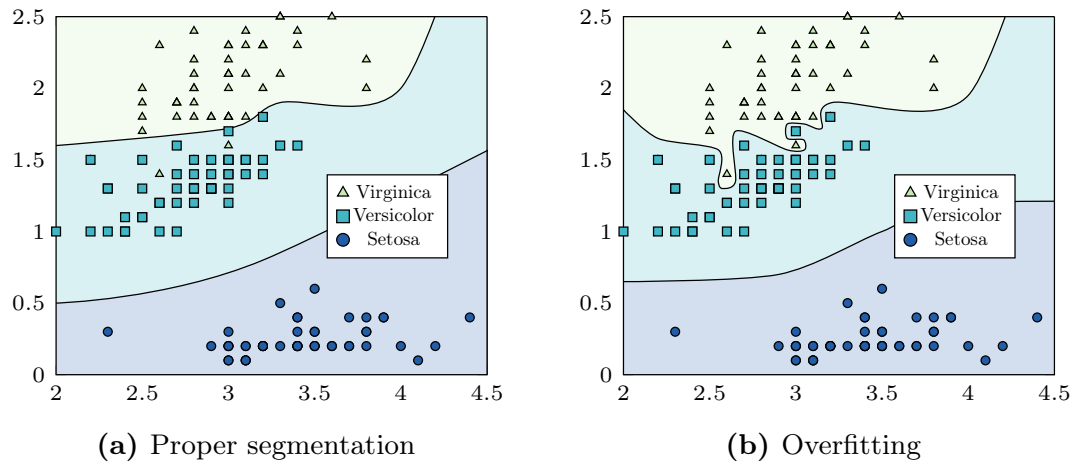
data to learn from and the validation data to evaluate its performance while learning. The model’s performance is then finally evaluated with the test set, containing data not used during training. The learning process and the three datasets are further elaborated upon in subsection 2.1.3.

Classification is a type of supervised learning. The goal is to categorize instances based on their attributes, often referred to as *features*. The set of classes is finite, and each instance belongs to a single class. One approach is for the model to predict a separate number for each class. The correctness of a prediction can then be calculated by the *mean squared error*:  $\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$ , where  $y$  is the correct answer,  $\hat{y}$  the prediction, and  $n$  the number of examples in the dataset. It is the square of the distance between the predicted number and the correct answer, averaged over all predictions.

Instead of predicting a single class id, a probability can be predicted for each class. When doing so, the output *one-hot encoded*.  $[0.1, 0.7, 0.2]$  is an example prediction in classification with three classes. The array contains one probability for each class: 0.1 for the first, 0.7 for the second, and 0.2 for the third. Thus, the model believes the features it received most strongly corresponds with the second class, although it is not entirely sure. When the prediction is one-hot encoded, the *cross-entropy loss* is commonly used to evaluate the incorrectness of a prediction. The formula from cross-entropy is  $\sum_{i=1}^m -y_i \cdot \ln(\hat{y}_i)$ , where  $m$  is the number of classes in the array.

The Iris dataset [Dua and Graff, 2017] is a classification problem, shown in Table 2.1. It contains the three flowers Iris Setosa, Iris Versicolour, and Iris Virginica. Each row in the table is an *example* from the dataset, a data point. They contain four measurements of a specific flower and its name. The goal is to train a model to segment the four-dimensional feature space so that each segment corresponds with a specific flower. Figure 2.1a illustrates such a segmentation, although only two of the four dimensions are visualized. Each color represents one of the tree species, and one can see that most examples from the dataset fall within their respective segments. When such a class segmentation is learned, the

model can predict the classes of previously unknown examples. It predicts a new flower’s class based on the segment surrounding its position.



**Figure 2.1: Iris class segmentation.** Instances are plotted with sepal width along the  $x$ -axis and petal width along the  $y$ -axis. Data from Dua and Graff [2017].

## 2.1.2 Reinforcement Learning

There are no pairs of inputs and outputs in Reinforcement Learning (RL). Instead, the *agent* is either rewarded or punished after several actions in an environment. In RL, the learning entity is often referred to as an agent, rather than a model. The agent receives some input signals each timestep and performs an action that leads to the next timestep. At some point, it is informed that it did either good or bad. It does not know what caused the feedback, so it must learn without knowing the correct action in every situation. Reinforcement learning can thus be more complicated than supervised learning. The correct actions for each input are unknown, so the agent cannot directly learn the relationship between inputs and outputs.

## 2.1.3 Generalization

Generalization is an essential aspect of machine learning. A model can easily remember what it has seen. What is essential is its ability to perform well in new scenarios. When predicting the output of a never previously experienced combination of inputs, it has to generalize based on what it has learned previously and provide a reasonable output.

When a machine learning model is incapable of reproducing outputs it has already seen, it is generally not complex enough to capture all the information. However, when it is too complex or trained too long, it often learns every little detail about the training data. It is called overfitting and significantly reduces the model's ability to generalize beyond the training data.

Figure 2.1 illustrates an algorithms class segmentation in two different scenarios. In Figure 2.1a, the model has correctly learned where the general class boundaries lie. Although it might miss some outliers, it captures the general concept and can generalize to new unseen data. Figure 2.1b has divided the instances into classes such that every instance is correctly classified. It has been overfitting, resulting in a complex boundary that is likely not correct. Because it overfits on a few outliers, it lost the bigger picture and is unable to generalize. Therefore, it likely performs worse than Figure 2.1a on the test data, even though its performance is higher on the training data.

To combat overfitting, one can use a validation set to continuously validate that further training on the training set is beneficial. When the model starts to overfit, the validation error will increase, and one knows to stop training. A non-biased performance score can be calculated by evaluating the model on a separate test set after completing the training. It has to be another piece of unseen data because the trained model is strongly affected by the training set, and also the validation set to a certain degree. When evaluating the performance on the test set, one can see whether the model can generalize.

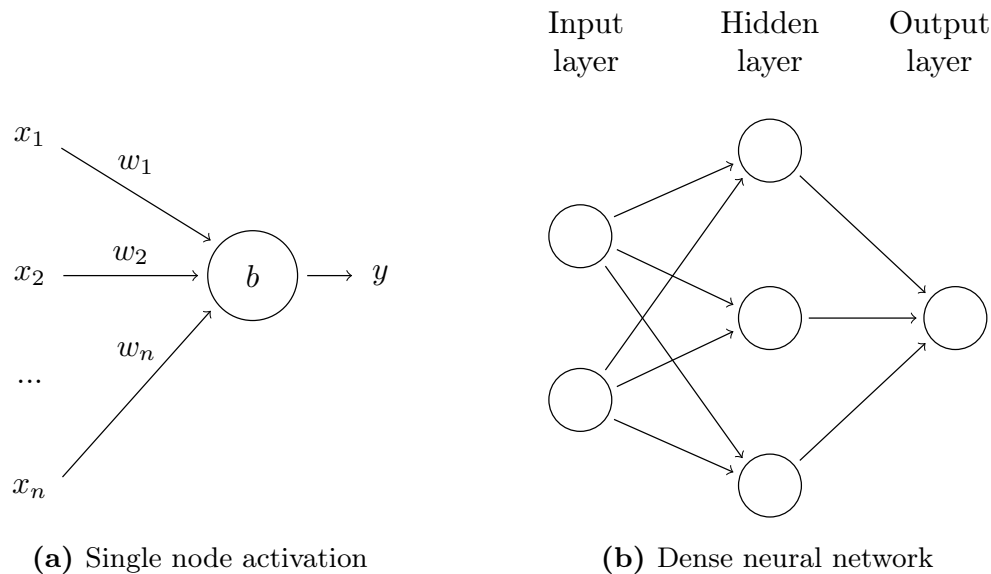
## 2.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are simplifications of the neural circuits found in biological brains, where synapses connect neurons. A neural network is a directed graph, where each connection between two nodes has an associated weight, and each node a bias [Goodfellow et al., 2016]. Values are propagated from input nodes, through the network, to one or more output nodes. The output value of a neuron,  $y$ , shown in Equation 2.1 and Figure 2.2a, is the sum of each incoming input,  $x_i$ , multiplied by the corresponding connection weight,  $w_i$ , plus the node's bias,  $b$ . The value is finally evaluated by an activation function,  $f$ .

$$y = f\left(\sum_{i=1}^n w_i x_i + b\right) \quad (2.1)$$

A neural network can be structured in various ways, although the most common are a feed-forward networks. Information flows in a single direction within a feed-forward network, from input to output. They are acyclic and do not possess any form of memory. Networks are often constructed in layers, as seen in Figure 2.2b.

Layered approaches are conventional because they can be executed in parallel, and the reduced topology space is easier to exploit when manually designing networks. Many nodes, as the one in Figure 2.2a, have been connected into an ANN in Figure 2.2b. It consists of an input layer, a hidden layer, and an output layer. Each node is connected to every node in its neighboring layers, making it a *dense* network. ANNs can have hundreds of layers or only a few, and there exist many different types of layers [Goodfellow et al., 2016].



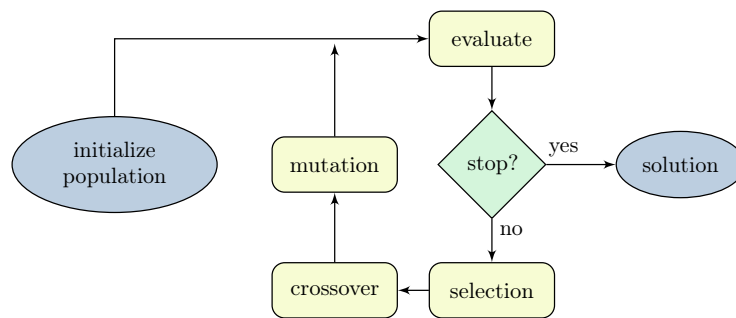
**Figure 2.2: Neural network example.** A dense neural network (b) comprising many individual nodes (a).

## 2.3 Evolutionary Algorithms

This section presents the genetic algorithm in subsection 2.3.1 and two variations of it, with indirect encodings in subsection 2.3.2 and speciation in subsection 2.3.3.

### 2.3.1 Genetic Algorithm

Genetic algorithms draw inspiration from evolution in nature, where the individuals best fit to the environment are more likely to reproduce and generate offspring. When two individuals mate, the produced offspring inherits some combination of the parents' genetic makeup. The process is not perfect, so the child is not always an exact combination of its parents.



**Figure 2.3: Genetic algorithm process.**

The genetic algorithm [Yang, 2014] simulates these processes from natural selection in a controlled environment. The environment is represented by a problem, where the corresponding population represents solutions to the problem. Figure 2.3 illustrates the different steps of the algorithm. An initial population is created in the first step. Each individual in the population has some variables associated with it. These are called *genes*, and together they form a *chromosome*. An example chromosome is the array  $[0, 1, 1, 0]$ , consisting of four binary genes. Another individual in the same population could be  $[0, 1, 0, 1]$ . The population of chromosomes is often initialized randomly, but can also be initialized in a controlled manner.

The next step is to evaluate the population. All individuals are evaluated by their ability to solve the problem. They are assigned a value that reflects their performance, called *fitness*. Based on the individual's fitness, some are selected and allowed to mate. This step is called *selection*, which can be performed in various ways. Selection methods generally simulate nature by providing the chromosomes with high fitness a higher probability to mate.

Selected individuals from the previous step are allowed to produce offspring. The process is called *crossover* in the genetic algorithm. The parents' genes are combined into new sets of genes, forming children. The crossover operation often resembles the biological mating process, where children's genes are some random combination of their parents'. Iteratively copying a gene from either parent is one of many crossover approaches. With the example chromosomes described earlier, a crossover between  $[0, 1, 1, 0]$  and  $[0, 1, 0, 1]$  could with the described method produce  $[0, 1, 1, 1]$ . Both parents share the first two genes. The child's third gene was selected from the first parent and the last one from the second parent.

The genetic algorithm has a *mutation* step, replicating the imperfections of nature, where children are imperfect combinations of their parents. The offsprings' genes are modified randomly during mutation. With the binary string chromosome, a mutation operation may be to flip one of the bits, making  $[0, 1, 1, 1]$  into  $[1, 1, 1, 1]$ . When the children are mutated, they replace the parents

and form the next generation. It is additionally common to assure that several best-fit individuals are not replaced. The concept is called *elitism*, where some of the best-fit parents are directly copied into the next generation.

The entire process is then repeated, as shown in Figure 2.3, starting with the evaluation of the new generation. The evolutionary process is continued until reaching a stopping criterion, upon which the current population's best individual represents the solution found by the algorithm.

### 2.3.2 Development and Indirect Encoding

When each chromosome directly represents a solution, it is a *direct encoding* scheme. The information in the chromosome, the *genotype*, directly maps to the solution. Although commonly used in bio-inspired methods, direct encodings are not grounded in nature. Biological DNA contains a relatively small piece of information compared to the complexity of the creatures it encodes. Complex *phenotypes* are created from smaller genotypes through embryogeny, the growth process in which the phenotype is *developed*.

With the inclusion of a process either mapping or developing genotypes into phenotypes, it is the phenotypes that represent the solutions. Opposed to a direct encoding, solutions can now be *indirectly encoded*. The genotypes can be seen as a description of a solution rather than the solution itself. Use on indirect encodings introduces an additional step before evaluation, where genotypes are developed into phenotypes.

Bentley and Kumar [1999] have found that evolutionary computation can significantly benefit indirect encodings. The indirect encoding enables the genotype to be magnitudes smaller than the solution it produces, thereby reducing the search space. Changes in the genotype may be reflected at multiple locations in the phenotype. Mutations can consequently be more significant than with a comparable change in a direct encoding. It enables faster traversal in the search space, but might also become unstable. Features may also be reused at multiple locations in the phenotype, instead of being discovered at multiple locations independently.

### 2.3.3 Speciation

Creatures in nature are divided into *species*, each optimized to their specific niche. Individuals within a species share some common traits, making them distinct from other species. In the fight for reproduction, an individual only competes with other individuals within the same species, not every individual on Earth. The same concept can be introduced in the genetic algorithm, where solutions only compete with other solutions in their near vicinity.

First, some distance measure has to be determined, either between genotypes, phenotypes, or between the behavior in an environment. Individuals are then divided into species based on the distance measure so that the distances between individuals of the same species are below a certain threshold.

The use of speciation in the genetic algorithm enables certain features. Species can be optimized separately, almost like separate populations, or they can fight for resources. The number of offsprings within a species can be determined based on the species fitness, its age, or its ability to improve. Innovation could also be rewarded, where new species are allocated more offsprings than old species. Although concrete implementations vary, the general concept is to group the population into species based on their similarities.

## 2.4 Neuroevolution

Neuroevolution draws inspiration from nature, in which complex brains with billions of neurons and trillions of connections are the product of biological evolution. Similarly, the field of neuroevolution attempts to evolve Artificial Neural Networks (ANNs), described in section 2.2, with evolutionary algorithms [Stanley et al., 2019]. They are though orders of magnitudes smaller than the human brain. The genetic algorithm presented in subsection 2.3.1 is one of the many evolutionary algorithms used to evolve ANNs.

Both the direct and indirect encodings, described in subsection 2.3.2, are used in the field of neuroevolution. The genotype either directly contains information about all the nodes and connection, or in some way indirectly specify how the ANN should be constructed. When the genomes are mapped into ANNs, these can be assigned fitness based on their performance in a task. When using GA, this process follows the diagram in Figure 2.3. The best ANNs are selected for reproduction. These are the parents of the next generation, and crossover is performed on their genomes. The resulting children are then mutated, forming the next generation.

Methods within this field can be divided into two groups [Xin Yao, 1999]. The first group constitutes methods that evolve the weights of fixed topology networks. The topology itself is additionally evolved in the second group.

## 2.5 Neuroevolution of Augmenting Topologies

NeuroEvolution of Augmenting Topologies (NEAT) [Stanley and Miikkulainen, 2002] is a method that uses GA with speciation and a direct encoding to evolve both the topology and weights of neural networks. Networks are initialized small, and increase in complexity during evolution. Mutations either connect two existing



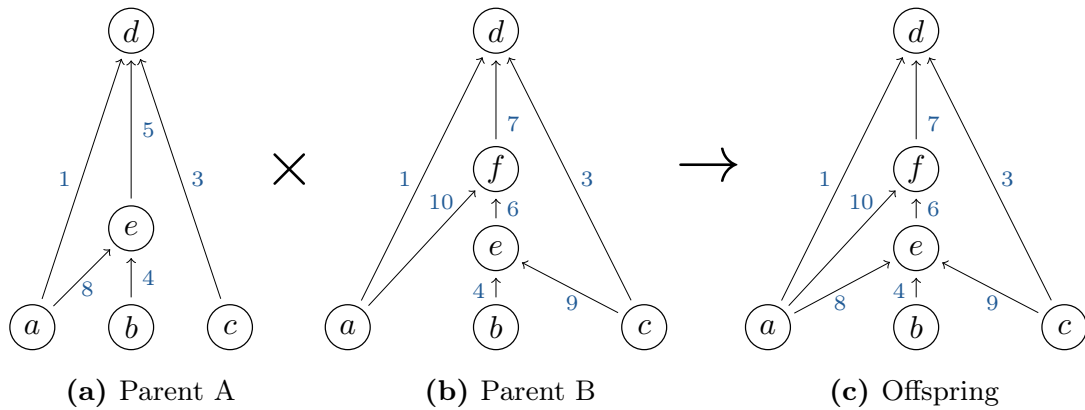
nodes or insert a new node into an existing connection, splitting it in two. Both operations increase the network complexity.

NEAT is built upon three main principles. First, targeting compact solutions through incremental growth from minimal initialization. Second, while the network topologies grow, the method keeps track of the innovations that mutate the topology. It does so to enable the crossover operation to combine the correct parts in two entirely different network topologies. Finally, NEAT utilizes speciation so that individuals only compete with similar networks. When new topologies arise, these therefore have time to optimize their weights before competing with the rest of the population.

By incrementally expanding the topology while optimizing its weights, the network complexity is adaptive. If the complexity required to solve a problem is low, the algorithm can terminate early. If the problem requires a complex solution, the GA runs for more iterations. Either way, NEAT produces compact solutions, where the resulting ANN is not more complex than required for a specific fitness. It may take many generations to reach the required complexity when starting small, compared to methods that are initialized with some pre-defined topology. However, early iterations are faster because of the smaller network sizes.

Stanley and Miikkulainen [2002] introduce the concept of *global innovation numbers*. Any modification to the network topology is defined as an innovation. Innovations receive a number that is global for the entire population. The same innovation in two different individuals will receive the same innovation number. Nodes have global ids, so *the same* innovation means an innovation involving nodes with the same ids. Each network is represented by a list of link genes, as illustrated in Figure 2.4d. These genes all have a *historical marking*, an id denoting which innovation created it. It is termed a marking when an innovation number is assigned to a gene. The markings in the genes are used to keep track of the networks' elements so that crossover can be performed. The markings indicate how two different topologies should be combined, even though parts of the networks have evolved differently. The first time an innovation occurs in a population, it is given a new innovation number from the global counter. Whenever the same innovation occurs in another network, the existing innovation number is assigned to the created gene. Therefore, it is later known that these two innovations are the same, even though the two topologies may seemingly not have anything in common. As seen in Figure 2.4d, all genes that connect the same pair of nodes have the same marking.

The default NEAT implementation has two topological mutations, either connect nodes or insert new. It can be extended to remove nodes and links as well. The connect mutation connects to nodes that are not previously connected. It can be seen in Figure 2.4b, where a mutation connected node *c* and *e* in parent B. If these two nodes were not previously connected in any network, a new historical



Parent A	1	(2)	3	4	5	8				
	$a \rightarrow d$	$b \rightarrow d$	$c \rightarrow d$	$b \rightarrow e$	$e \rightarrow d$	$a \rightarrow e$				
Parent B	1	(2)	3	4	(5)	6	7	9	10	
	$a \rightarrow d$	$b \rightarrow d$	$c \rightarrow d$	$b \rightarrow e$	$e \rightarrow d$	$e \rightarrow f$	$f \rightarrow d$	$c \rightarrow e$	$a \rightarrow f$	
Offspring	1	(2)	3	4	(5)	6	7	8	9	10
	$a \rightarrow d$	$b \rightarrow d$	$c \rightarrow d$	$b \rightarrow e$	$e \rightarrow d$	$e \rightarrow f$	$f \rightarrow d$	$a \rightarrow e$	$c \rightarrow e$	$a \rightarrow f$

(d) Genes

**Figure 2.4: NEAT crossover operation.** Offspring (c) is a result of crossover between parent A and B. Their genes (d) consist of markings and source-target pairs. Those in parenthesis are disabled. Adapted from Stanley and Miikkulainen [2002].

marking, 9, was created by incrementing the global innovation number. The gene [9,  $c \rightarrow e$ ] was then appended to the list of genes. If the same mutation were to happen in parent A, the global innovation 9 would already exist. Therefore, the global counter is not incremented. The gene [9,  $c \rightarrow e$ ] is directly appended to the gene list.

Node insertions are the second topological mutation. Whenever a node is added to a network, it is inserted into an existing link. In Figure 2.4b, the node  $f$  has been inserted into parent B's  $e \rightarrow d$  link. Two new historical markings, 6 and 7, were created because the two links did not already exist in the population. The two link genes [6,  $e \rightarrow f$ ] and [7,  $f \rightarrow d$ ] was then appended to parent B's genome. At the same time, the old link gene, [5,  $e \rightarrow d$ ], was disabled because  $e \rightarrow f \rightarrow d$  now replace  $e \rightarrow d$ . Link  $e \rightarrow f$  received a weight of 1.0, and the weight from  $e \rightarrow d$  was copied to  $f \rightarrow d$ . Because the first weight is 1.0, it is an identity mapping, meaning the value is propagated through the network without modification. Thus, the functionality of the network remains the same even though a new node was

inserted. Even though the network's output is initially the same, future weight mutations will later evolve these weights, and new links may be connected to or from node  $f$ . The node insertion mutation can therefore be seen as a preparation for future change, even though it does not itself affect the network functionality.

The historical markings simplify the crossover process as there is no need to search through, compare, and combine the networks based on their topology alone. When crossover is performed, the genes are lined up based on their marking, as illustrated in Figure 2.4d. Genes with markings present in both parents are called *matching*. These are randomly chosen from either parent and passed on to the offspring. In contrast, markings only present in one of the parents are passed from the more fit parent to the child. However, if parents are equally fit, the genes with unique markings are passed on from both parents. In this case, gene 1 to 5 are matching, and each these are copied from a random parent onto the offspring. Gene 5 is disabled in parent B, so it is also disabled in the offspring. The rest of the genes do not match. All are passed on to the offspring because this example assumes that both parents have equal fitness. Otherwise, only the fitter parent's genes are passed on. It would only be gene 8 if A was more fit and genes 6, 7, 9, and 10 otherwise.

When new topologies arise within the population, they can initially be fragile because they need time to optimize before yielding good results. Stanley and Miikkulainen [2002] argue that when competing with the entire population, they will not survive because new topologies often perform worse initially, even though their innovation might be a long-term step in the right direction. The NEAT algorithm protects innovations through speciation, where the individuals mainly compete within their species. Species are formed based on compatibility distance, so new innovations are likely to form a new species. This distance measure is calculated from the number of disjoint genes and the internal difference between overlapping genes. The population contains a list of species. New individuals are inserted into the first species where the distance measure between it and the species' first individual is below a specific threshold value.

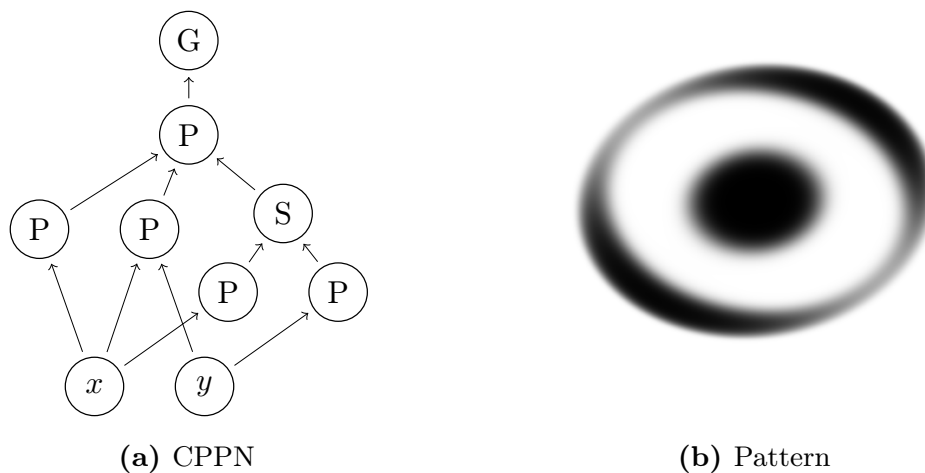
The number of offsprings each species produces is determined by its performance relative to the other species. Also, young species are given more offsprings, and stagnant species are given less. Individuals mainly mate within their species, but there is a small probability for inter-species reproduction.

## 2.6 Compositional Pattern-Producing Networks

Compositional Pattern-Producing Networks (CPPN) [Stanley, 2007] are neural networks. They consist of nodes and connections, with activation functions and weights. What makes them different from traditional neural networks is the plethora of unique activation functions used within the same network. These activation functions are used to enable the CPPN to generate intricate output patterns. Gaussian, Sigmoid, Sine, Step, and Tanh are activation functions often used in CPPNs, where Gaussian and Sine are quite uncommon in traditional ANNs. The two have unique properties that are likely beneficial for a pattern production. Sine is used for repetition, a feature that is difficult to accomplish without a repeating activation function. The Gaussian function is mirrored around the  $y$ -axis, and can therefore generate symmetric patterns.

CPPNs can be evolved with the NEAT algorithm, making them grow more complex during evolution. NEAT has to be modified to additionally evolve and randomly mutate an activation function for each of the network's nodes. No other modification is required before using NEAT to evolve CPPNs. Because of the incremental complexification, the produced patterns are initially simple and become more intricate throughout evolution.

A CPPN can be viewed as a function of  $n$  input parameters, yielding one or more output values. The concept of a pattern arises when the CPPN is queried for a continuous  $n$ -dimensional space. In two dimensions, this can be regarded as



**Figure 2.5: Compositional Pattern Producing Network.** The CPPN (a), with Gaussian (G), sine (S), and squared (P) as activation functions, generates the pattern (b). Adapted from Stanley [2007].

a height-map, where each  $x, y$  location has a specific height  $h$ . Let  $G$  be a grid with a resolution  $r$ , in the area defined by  $-1 \leq x \leq 1$  and  $-1 \leq y \leq 1$ . When all the points within  $G$  are input to the CPPN, it will provide a pattern within the output values.

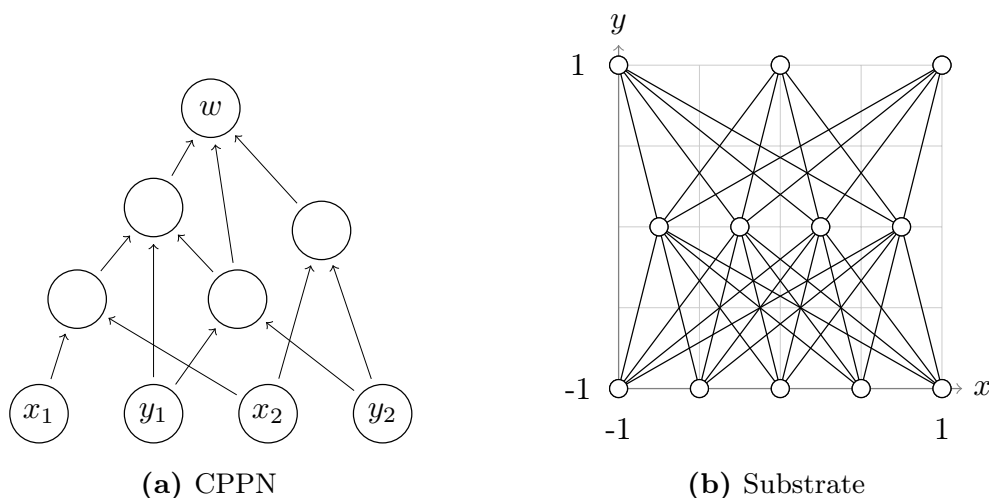
An example pattern is provided in Figure 2.5b. A two-dimensional grid between  $-1$  and  $1$  is used. The resolution is  $512$ , meaning the patterns consist of  $512 \cdot 512 = 262144$  values. To create the illustration, each of these was collected with a separate execution of the CPPN in Figure 2.5a. The output values were normalized to produce the image. Within the grid, black illustrates the lowest values, and white the highest. The CPPN was artificially created but could be evolved with NEAT.

One of the features within this pattern is that it is circular. It arises from the sum of  $x^2$  and  $y^2$ , which creates gradually increasing values out from the center,  $x = 0, y = 0$ . The dot in the middle and the ring around it is the works of the sine function. When  $x^2 + y^2$  are fed into it, the result is a wave that starts at zero and moves outwards. The reason why a second ring is not visible is that the left part of the CPPN,  $(x^2 + y^2)^2$ , overshadows the negative output of the sine function outside a radius of one. The slight variation along  $x = -y$  comes from the fact that both  $x$  and  $y$  feed into the same node  $P$  in the center of the CPPN. It is only a simple example of a pattern that may arise from a CPPN. Much more intricate patterns may evolve through evolution.

## 2.7 Hypercube-Based NEAT

Hypercube-based NEAT (HyperNEAT) [Stanley et al., 2009] is a neuroevolutionary method that uses CPPNs and an indirect encoding. A static neural network topology is manually defined, and CPPNs are evolved to assign weight values to it. For the CPPN to assign weights to the ANN's connections, the network is geometrically defined within a two-dimensional grid called a *substrate*. All the network's nodes are manually positioned within the substrate. They are placed in the range  $-1$  to  $1$  on the  $x$  and  $y$ -axis. Figure 2.6b illustrates a network's geometry within a substrate. All 12 nodes have a unique  $(x, y)$  position within the substrate. These positions are used to assign weights to the network's connections.

As described in section 2.6, a CPPN defines an  $n$ -dimensional pattern. The pattern is used to assign weights to ANNs in HyperNEAT. Specifically, a 4-dimensional pattern is used, meaning the CPPN has four inputs. Such a CPPN is illustrated in Figure 2.6a, with the four inputs  $x_1, y_1, x_2, y_2$ , and the output  $w$ . To assign a weight to a connection in the network in Figure 2.6b, the positions of the two nodes forming the connection,  $(x_1, y_1)$  and  $(x_2, y_2)$ , are used as inputs to the CPPN. The output weight,  $w$ , is then assigned to the connection between



**Figure 2.6: HyperNEAT weight assignment.** The CPPN (a) produces a weight  $w$  for the connection between each pair of nodes  $(x_1, y_1)$ ,  $(x_2, y_2)$  in the example network with five inputs and three outputs (b). Adapted from Pugh and Stanley [2013].

the two nodes in the substrate. Figure 2.6b has 32 connections, so 32 CPPN executions are required to assign weights to the network. For each pair of nodes with a connection between them, the CPPN in Figure 2.6a is queried for a weight value. If the absolute value of the weight is higher than a specific threshold, the connection is assigned the weight. However, if the magnitude is lower than the threshold, the connection is deactivated.

Unlike NEAT, HyperNEAT does not explicitly denote all the network’s nodes and weights in its genotype. The genotype in HyperNEAT encodes the CPPN, which indirectly determines all the connection weights in the pre-determined static network. The static networks may contain millions of connections, while the CPPN only a hundred. Since only the CPPN is evolved, rather than the full-scale network itself, HyperNEAT is searching in a compressed space. It may enable the method to evolve the weights of networks that are larger than what is viable with a direct encoding.

The conventional approach is to organize nodes based on the problem. An example is shown in Figure 2.6, depicting the network used in a hypothetical navigation problem. A robot has five distance sensors, left to right, and three actions: turn left, turn right, and move forward. Based on the problems geometry, the input nodes are placed evenly along  $y = -1$ , corresponding to their real-world placement. Several hidden nodes are placed at  $y = 0$  and the movement actions left, forward, right at  $(-1, 1)$ ,  $(0, 1)$ , and  $(1, 1)$ , respectively. The network’s

geometry within the substrate is constructed based on the problem so that the CPPN may exploit spatial relations among the nodes when assigning weights their connections. When constructed this way, the pattern may evolve to be symmetric. When the pattern is symmetric along  $y = 0$ , the left and right section of the network is mirrored. Such a network should work well in the navigation domain, and will not be as easily to discover without the spatial relationship between the network and the problem.

The bias of an ANN node is added to the sum of the incoming signals, as described in section 2.2. Bias can be implemented in two ways in HyperNEAT. The first approach is to place an extra input node within the substrate and connect it to all other nodes. A constant value is input to the extra node whenever the network is executed. Therefore, Connection weights between it and all other nodes control the bias of the other nodes. Alternatively, the CPPN can have two output nodes, one for determining weights within the network and another for determining the bias of all nodes. Although the connections within the substrate need all four CPPN inputs to be defined, nodes only need two. The bias is therefore found by executing the CPPN with 0, 0,  $x$ , and  $y$  as the input. The two obsolete dimensions are set to 0. The value from the CPPN bias output is then assigned as the bias of the node at position  $x, y$ .

## 2.8 Deep Learning

Deep Learning (DL) is a type of machine learning that optimizes Artificial Neural Networks (ANNs) to solve a variety of tasks. The concept *deep* comes from the use of hidden layers in the networks, extending them from only an input and output layer. While there is no concrete definition of what qualifies as deep networks, they have gradually been constructed deeper and deeper. Today's use of the word generally suggests more than a few hidden layers. Although NeuroEvolution (NE) can generate deep neural networks, the field of deep learning generally revolves around the use of *backpropagation* and *gradient descent*.

Deep learning methods typically use backpropagation and gradient descent [Goodfellow et al., 2016] to improve the ANN's weights gradually. Backpropagation is a method that calculates the gradient of the *loss* function. Gradient descent is the optimization method that updates the ANN's weights through incremental steps in the opposite direction of the gradient. The loss function, also known as *error*, is a function that calculates how incorrect the model is. It is a function of the model's parameters. By computing the gradient of it with backpropagation, and updating the parameters with gradient descent, the model is strategically updated. It incrementally improves through small steps in the direction that reduces error. However, it does require the error function to be differentiable.

Although both the field of deep learning and neuroevolution are centered around the construction of neural networks, networks are constructed and optimized differently in the two fields. One fundamental difference between DL and NE is how the weights of ANNs are optimized. Neuroevolution takes an entirely different approach and uses evolutionary algorithms to evolve the weights. Each update is not as targeted as with gradient descent, but there is no need for a differentiable error function. It makes neuroevolution well suited for Reinforcement Learning (RL), where there is no differentiable error function. Because a population of solutions is maintained throughout evolution, NE is also much more exploratory than the single search used in gradient descent.

Another difference between traditional DL and some NE methods is how the network topology is determined. While some NE methods evolve the topology, it is common to manually construct networks in layered architectures within the field of deep learning. Multiple hand-crafted layers are linked together to form entire networks. Finding the best architectures is a tedious and time-consuming trial and error process that requires expert knowledge.



# State of the Art

In contrast to the general introduction to neuroevolution and deep learning in chapter 2, the following chapter contains details about selected topics and aspects of state of the art methods. The topics and methods are mainly a result of the literature review process. Deep neural networks are presented in section 3.1, the relationship between depth and complexity in section 3.2, and the network's encoding in section 3.3. The following sections then focus on HyperNEAT and various modifications and extensions of it. Network connectivity is presented in section 3.4. Multiple substrates and CPPN complexity is presented in section 3.5, followed by a description of the node search used in ES-HyperNEAT in section 3.6.

## 3.1 Deep Neural Networks

An Artificial Neural Network (ANN) is in itself only a model representation, independent from its construction and optimization. Deep ANNs (DNNs) have generally been very successful, able to surpass human performance in domains such as board games [Silver et al., 2016] and strategic planning [Vinyals et al., 2019]. In the field of Deep Learning (DL), gradient descent has successfully optimized complex networks with millions of weights [He et al., 2016] but requires the error function to be differentiable. The network topologies are also typically manually constructed, requiring a trial and error search process. Methods within the field of NeuroEvolution (NE) have reached superhuman performance on specific tasks as well [Hausknecht et al., 2014], but the networks are generally not as deep and complex as in DL [Such et al., 2017]. However, NE has some desirable traits generally not found within DL. Many neuroevolutionary methods do not require the user to design the network topology, as it is done by evolution. NEAT

[Stanley and Miikkulainen, 2002] and EANT [Kassahun and Sommer, 2005] are two methods that evolve the weights and structure of neural networks without the need for human expertise. By maintaining a population of solutions, the search is also broader than the single targeted search in gradient descent.

Utilizing the benefits within the field of both DL and NE, hybrids have been proposed. Sun et al. [2018] use the Genetic Algorithm (GA), commonly used in NE, to generate DNN topologies and gradient descent to optimize their weights. The method, called GA-CNN, is able to generate architectures that outperform the compared architectures constructed by humans. It highlights the power of search within evolutionary algorithms, where a population of solutions in GA can traverse the architectural search space better than humans.

With the advantages of an automatically evolved topology, an exploratory search, and without the need for a differentiable error function, neuroevolution seems like the superior choice compared to gradient descent. However, NE often struggles with complex problems and large search spaces [Verbancsics and Harguess, 2015; Such et al., 2017]. Some deep networks have been evolved with evolutionary algorithms [Such et al., 2017], but they are generally not able to succeed in the same complexities and depths found in the state of the art within deep learning.

## 3.2 Depth and Complexity

NEAT struggle with the large search space of deeper networks [Miikkulainen et al., 2019]. It is a significant drawback, as deeper networks are advantageous in their ability to represent more complex functions and solve more complex problems. Recent advances by He et al. [2016] indicate that deeper networks can yield higher accuracy, evident by their 152-layered network yielding higher performance than their shallower networks and other state of the art architectures.

Even though deep networks are beneficial, there is an upper bound where further depth increase does not increase the performance. In addition to the 152-layer deep network, He et al. [2016] also created a 1202-layered network, which did not perform as well. They argue it is caused by the deeper network overfitting, as the data it is trained on do not require such a complex network. The 152-layer network was sufficiently deep for the given task and further depth increase reduced performance. Therefore, the challenge is to find the point of optimal complexity, where the network can learn the domain, but not so complex that it is likely to overfit. Methods that employ gradual complexification automatically adapts complexity without the need to test multiple complexities manually. CoDeepNEAT [Miikkulainen et al., 2019] is one such method. It uses the gradual complexification from the NEAT algorithm to evolve layer-based architectures. These are then

trained with backpropagation like any other handcrafted deep learning methods. Their results indicate that the approach is comparable to handcrafted architectures, while its automation is advantageous.

Execution time is also worth considering when determining the complexities of ANNs. More complex networks can capture larger domains, but will also use more time when predicting an output. In some situations, one would like the best possible result, no matter how long it will take to produce it. However, sometimes the execution speed of the network is crucial, and the goal is to achieve the best possible result in limited execution time. Neuroevolution has properties that should make it succeed in this area. When execution speed is important, the ANN topology is essential. One cannot use state of the art DNNs, as these have too many connections and are computationally heavy to execute. Lan et al. [2019] show that NEAT is able to evolve a compact object-recognizing network with few connections, making it efficient enough to be executed on low-performance hardware. However, they argue that the network could benefit the use of backpropagation and gradient descent to optimize the final network, as NEAT might not be able to tune the weights sufficiently.

### 3.3 Network Encoding

Neuroevolutionary methods based on *direct encodings*, such as NEAT, have issues optimizing large networks. Gillespie et al. [2017] argue that it may be caused by the search space being too large compared to the impact of isolated mutations. When each weight in the network is directly encoded in the genome, a mutation to a single weight will not substantially impact the output. Therefore, it is highly unlikely that directly encoded networks will be able to successfully evolve the weights of networks with the complexities seen in deep learning.

The backpropagation method uses a direct representation. All weights in the network are represented and updated in each iteration of the algorithm. It manages to search the large weight spaces that NEAT cannot. It is likely due to how the updates are determined. The targeted updates in backpropagation, towards the area that reduces error the most, are able to advance the search. It is therefore logical that the reason why NEAT cannot traverse the same space is due to the combination of a direct encoding and evolutionary search. Evolution can seemingly not handle search spaces at the scale used in deep learning with a direct encoding.

An alternative to direct encoding is an *indirect encoding*. With an indirect encoding, the genome no longer contains all the information about the neural network, only an indirect description. Indirect encodings enable evolution of more complex networks with smaller search spaces than direct encodings. They might be better suited for evolving deep and complex networks with evolutionary

methods because they can be searched in a compressed space [Koutnik et al., 2010]. HyperNEAT [Stanley et al., 2009] is an extension of NEAT that uses an indirect encoding. It is described in section 2.7. The main concept is to evolve a weight-generating network called a Compositional Pattern Producing Network (CPPN), described in section 2.6, with the NEAT algorithm described in section 2.5. The CPPN is used to assign weights to another network with a static topology. There is no limit to the size of the static network, Therefore, a small CPPN genome can define the weights of a much larger network. The indirect encoding enables the evolution of networks that are much larger than when evolved with NEAT.

### 3.4 Network Connectivity

With the manually constructed network topology in HyperNEAT, one can create thousands of connections. However, not all of them are beneficial for the network. To allow evolution to control which connections to use, HyperNEAT employs a threshold technique. If the absolute value of the weight output by the CPPN is below a certain threshold, the connection is disabled. It allows the CPPN to determine which connections to include in addition to their weight.

When the same weight output value determines both whether to include the connection and the weight of it, there is a strong relationship between the two. Verbancsics and Stanley [2011] propose to decouple a connections presence from its weight. They argue that evolution has more freedom when two separate CPPN outputs determine presence and weight. The topology can then evolve independently from the weight assignments. They accomplish it by extending the CPPN to include an additional output node. The output provides what is called a link *expression*. Connections are only included if their expression value is above 0. The weight value from the other CPPN output node is still used to determine the weight. The extension is called Link Expression Output (LEO). They found that evolving the connectivity pattern separate from the weights increase the performance of HyperNEAT.

Another approach to limiting connectivity is associating a cost with each connection, proportional to the square of its length. It punishes networks that have connections between nodes that are far apart. Huizinga et al. [2014] employed the technique in HyperNEAT, terming it HyperNEAT Connection Cost Technique (HyperNEAT-CCT). The inclusion of connection cost results in sparser connectivity within the substrate, where connections are more prone to exist between nodes that are close to each other. The technique can be combined with either the weight threshold in the original HyperNEAT or with LEO.

Evolvable Substrate HyperNEAT (ES-HyperNEAT) is an extension of HyperNEAT that automatically constructs a network within the substrate. It is an

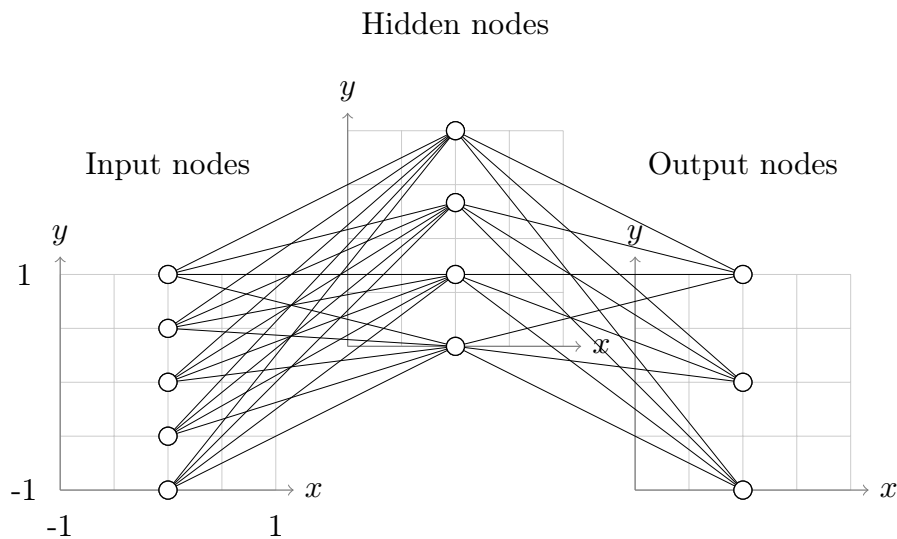
entirely different approach than LEO and CCT, as the method constructs a network instead of limiting the connections within an existing network. It is the other way around. Unlike traditional HyperNEAT, no network is manually constructed in ES-HyperNEAT. Only the input and output nodes are positioned within the substrate. ES-HyperNEAT then determines the hidden node's positions and the connections between nodes based on the weight pattern produced by the CPPN. Networks are constructed based on the variance within the pattern. Connections are formed so that the variance between their weights are high. It ensures that connections do not have uniform weights, while also alleviating manual network construction. The method is described in detail in section 3.6.

### 3.5 Multiple Substrates and CPPN Complexity

When the output of a single CPPN is used to assign weights to an entire network in HyperNEAT, as described in section 2.7, it must be able to produce complex patterns. Each connection within the substrate is defined within a four-dimensional space between  $-1$  and  $1$  on each axis, by the positions of the two nodes it is between. If a network with thousands of connections is defined within a single substrate, the CPPN must produce adequate weights for all of them. It may become an issue when constructing networks with many nodes tightly placed within the substrate. All weights are determined by the four-dimensional CPPN output pattern, called the *hypercube*. Connections that are spatially near each other in the network are assigned weights that are near each other in the hypercube. If these nearby connections should ideally have very different weights, the CPPN must produce a hypercube pattern with distinct and complex transitions. The required complexity of such a function makes it difficult to learn, and it might not even be reachable by evolution.

The method Multi-Spatial Substrate (MSS) [Pugh and Stanley, 2013] extends HyperNEAT from a single substrate to network construction across multiple substrates. Pugh and Stanley [2013] proposed that inputs and outputs that are not correlated should be placed in separate substrates. It would simplify the manual construction of networks in HyperNEAT, which becomes difficult with many input, output, and hidden nodes. They also argue that by grouping related inputs and outputs, evolution can more easily discover good weights.

Nodes are still manually placed, but now distributed among multiple spatially independent substrates. Figure 3.1 illustrates the same network as in Figure 2.6b, only divided among three separate substrates. Connections are between substrates instead of within substrates. Each pair of substrates are allocated a unique CPPN output node. The CPPN still has four inputs, but the number of weight output nodes is increased. The weight-producing CPPN used to determine the weights of the network in Figure 3.1 will have two outputs. One determines the weights



**Figure 3.1: Multi-Spatial Substrates.** Adapted from Pugh and Stanley [2013].

between input and hidden nodes and another the weights between hidden and output nodes. These two weight producing CPPN output nodes create separate hypercubes, independent from each other. The nodes can share substructure within the CPPN, they can connect to the same groups of hidden nodes, but also be entirely independent.

When multiple hypercube patterns assign weights to different parts of the constructed network, they do not need to encapsulate the weights of the entire network. Their complexity can thus be reduced because they only need to optimize their separate are of the network. Pugh and Stanley [2013] designed an experiment with a robot with multiple sensors and actuators. Some of the sensors were of different types, and their corresponding input nodes were therefore put in separate substrates. The CPPN could then optimize the use of different sensors independently. They found that distributing the network among multiple substrates, and thus the complexity among multiple CPPN outputs, improved the performance of HyperNEAT significantly. Solutions to the problem were found in over 80% of cases with MSS, compared to only 50% in the best single-substrate network.

The use of multiple substrates is not only limited to the robotic experiment performed by Pugh and Stanley [2013]. Any HyperNEAT network can be divided among multiple substrates, regardless of the topology and input types. More generally, when reducing the number of weights determined by each CPPN, their complexity can be reduced. By distributing the network among multiple substrates, the complexity of the hypercube pattern can be divided over multiple

hypercubes. Each optimizes their separate part of the network, and together optimize the entire network. Such a separation may be beneficial in any network containing many connections, not only when there are multiple correlated groups of inputs and outputs.

Deep HyperNEAT [Sosa and Stanley, 2018] is another method that uses multiple substrates and CPPN output nodes. Although similar to MSS, two aspects separate the two methods. Only input and output nodes are manually placed in substrates in Deep HyperNEAT, and the number of substrates increases during evolution. No hidden nodes are specified, as only substrates containing input and output nodes are determined a priori. Deep HyperNEAT starts with direct connections between these. During evolution, new substrates are added in-between existing connections between substrates. The inserted substrates contain nodes at pre-determined positions, which are connected to the nodes in the two substrates they are inserted between. As in NEAT, an identity mapping is created so that the addition of a new substrate does not affect the functionality of the network. Details about such identity mappings are described in section 2.5. Another mutation duplicates a substrate and all the connections from and to nodes within it. The outgoing connections from the two substrates are halved so that the network retains the same functionality, but can later evolve the two substrates separately.

The CPPN in Deep HyperNEAT has one output for each pair of connected substrates. As new substrates are added to the network, new output nodes are added to the CPPN. These output nodes are connected in a way that enables new substrates to be added without disrupting the network output, as described earlier. It is accomplished by copying existing parts of the CPPN when duplicating substrates. New output nodes are also manually connected when a substrate is added, so that it produces an identity mapping output pattern.

Although Deep HyperNEAT alleviates manual positioning of nodes and connections, their positions are still predetermined and not adapted during evolution. The network topology does change when a new substrate is added, but the positions of nodes within the new substrates are the same each time. As HyperNEAT determines the weights based on the node's positions, these must be placed in a way that enables the CPPN to learn good weights. If nodes are placed inadequately, the CPPN might not be able to produce a pattern that intersects the correct hypercube positions with the correct weights.

## 3.6 ES-HyperNEAT Node Search

Evolvable-Substrate HyperNEAT (ES-HyperNEAT) [Risi and Stanley, 2012] is the topic of this section. Unlike the previous, the section presents a detailed description of a method, not a topic. ES-HyperNEAT is an extension of HyperNEAT that determines the network's topology based on the pattern produced by the Composition Pattern Producing Network (CPPN). The original HyperNEAT method is described in section 2.7. It evolves a CPPN to assign weights of a network that is manually constructed in a substrate. However, ES-HyperNEAT only requires that the input and output nodes are placed within the substrate, as the hidden nodes and all connections are determined by implicit information within the evolved CPPN's output pattern.

Risi and Stanley [2012] present the idea that because the CPPN determines all the weights in HyperNEAT, implicit information within its output weight pattern could also provide insight into the nodes' placements. Each point in the four-dimensional pattern represents a connection, and thereby two nodes. The pattern is analyzed to find points of interest. Selected points correspond with the included connections in the network. If points in uniform areas are selected as connections in the network, the connection's weights will be similar. When all outgoing connections from a node have the same weight, the propagated value will be the same for all of them. The authors argue that multiple uniform connections are redundant and that points within areas of high variance should instead be selected. They therefore propose a search algorithm that finds points in areas of high variance in the hypercube. They construct a network with the substrate nodes and connections that correlate with the selected hypercube points.

*Iterative network completion* is the name of the algorithm used to construct networks in ES-HyperNEAT. A detailed description, along with pseudo-code, is available in their publication [Risi and Stanley, 2012]. The algorithm essentially constructs a neural network between the predefined input and output nodes, based on an evolved CPPNs' weight pattern. As the name suggests, it is an iterative algorithm. The input nodes are the starting point, and these are used in the first iteration. From each node, the pattern is searched for potential connections. If any are found, the target nodes are added to the substrate, and the connections are created. The new nodes are then searched in the next iteration, further increasing the depth of the constructed network for each iteration. However, no cycles are allowed, so they may not connect to nodes from earlier iterations. If no new nodes are discovered in an iteration, or the iteration limit is reached, no more iterations are performed. A final search within the pattern is then performed to determine the hidden nodes that should be connected to the output nodes.

The node search is the main element within the algorithm. It takes the position of a node as input and yields all the nodes it should be connected to as output,

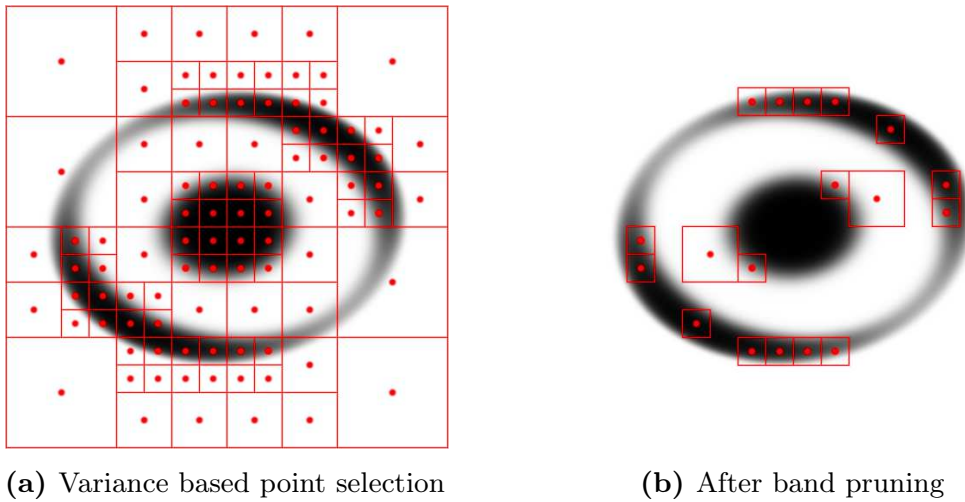


along with the connection weights. The four-dimensional CPPN weight pattern is the basis for the search, where each point  $(x_1, y_1, x_2, y_2)$  corresponds with a connection between a node at position  $(x_1, y_1)$  and another at  $(x_2, y_2)$ . Two of the four dimensions are specified by the  $x$  and  $y$  position of the searched node. The searched pattern is therefore two-dimensional, where the two axes correspond with the  $x$  and  $y$  value of the potential target nodes. Points within areas of high variance in this pattern are chosen, and the point's positions and values are returned. The positions of the selected points within the pattern becomes the positions of nodes added to the substrate. The weight at the selected positions in the pattern becomes the weights of the connections between the search-input node and the newly added nodes.

The values of the weight patterns are not initially known. The CPPN has to be executed to find the value of each point within it. To avoid that the CPPN is executed an unnecessary amount of times; a quad-tree search is used instead of a fine grid. The two-dimensional pattern is divided into four quadrants, and the value of each center is queried from the CPPN. When a section is divided into four, the node is said to be *expanded*. These sections are then each further divided and also their center values queried from the CPPN. Each section is a node in the quad-tree, with the four sections it is divided into as children nodes. The root of the tree is at the center of the entire pattern, and its children the four segments the entire pattern is divided into. Figure 3.2a illustrates the segments after fully expanding the tree, where the red dots are leaf nodes in the quad-tree. Regardless of the variance, nodes are always expanded until a specific quad-tree depth, *initial resolution*, is reached. Only the nodes with high variance are expanded further, and no nodes are expanded past the *max resolution* depth limit. The section sizes therefore vary in the illustration, where certain nodes are expanded deeper than others. The sections with low variance are larger because they have not been expanded as much.

When the quad-tree expansion phase is finished, the collection phase begins. Starting at the root node, nodes are extracted if the variance between all leaf nodes below it are below a specific threshold. If not, the same test is performed for all of its four children. The result is that nodes in areas of high variance are extracted at a deeper level than those in uniform areas. More nodes are therefore included from areas of higher variance.

The final phase is the *band* pruning phase, where nodes that are not in a band are discarded. Bands are areas that differ from their surroundings, like a valley through an otherwise flat surface. The white and black rings in Figure 3.2b are bands. The function  $\beta = \max(\min(d_{top}, d_{bottom}), \min(d_{left}, d_{right}))$  determine the band value, where the  $d$  values are the difference between the value at the current position and its neighboring sections. All positions with  $\beta$  below a certain threshold value are discarded. Figure 3.2b shows the result of band pruning.



**Figure 3.2: ES-HyperNEAT point selection.** The points (a) are distributed based on variance within the pattern, with higher density of points in areas of high variance. Points outside bands are then pruned (b). Adapted from Risi and Stanley [2012].

The remaining positions, the red dots, are inside the pattern’s bands. These are the nodes that should be connected to the node at the position the search was performed from.

As mentioned earlier, the search is used in every iteration of the iterative network completion. A different weight pattern, as the one in Figure 3.2b, will be searched for each node in each iteration. The same CPPN generates all patterns. However, the patterns are different for each node because of the node’s different positions. Each search then potentially yield new nodes that are added to the network and searched in the next iteration. Finally, when no more nodes are discovered, or the iteration limit reached, the output nodes are connected. It is also a search, but different from all the others. Instead of searching for outgoing connections, it is a search for incoming connections. The positions of the output nodes are therefore searched, and if any nodes are discovered, they are connected in reverse. Because they are searched in reverse, the CPPN is provided with  $x, y, x_n, y_n$ , instead of  $x_n, y_n, x, y$ , where  $n$  is the searched position. It is so that the direction of the searched connections are correct when the search is reversed. When the output nodes are connected, the network is fully constructed. Finally, any node not on a path between input and output nodes are excluded from the network, as they do not affect its functionality.

# Model

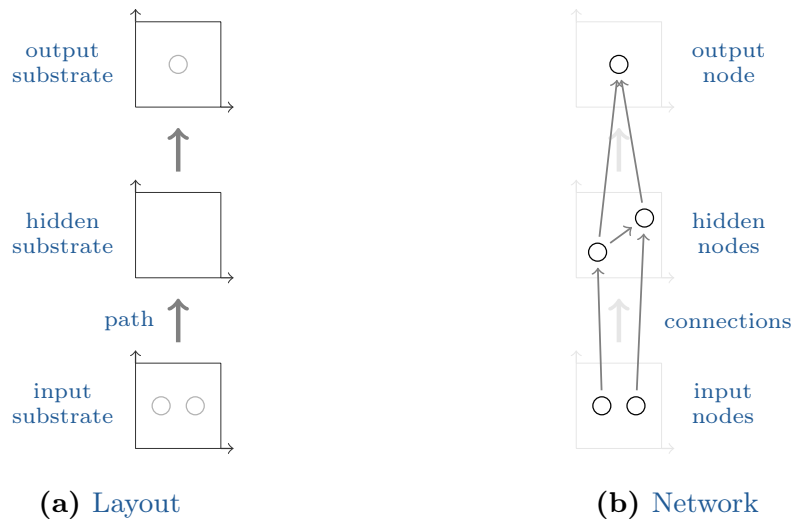
The following chapter introduces the proposed model, Deep Evolvable Substrate HyperNEAT (DES-HyperNEAT). A description of the DES-HyperNEAT framework is given in section 4.1, followed by the modifications made to the node search from ES-HyperNEAT in section 4.2. Three different implementations of the framework are presented and compared in section 4.3: Layered DES-HyperNEAT (LaDES), Single-CPPN DES-HyperNEAT (SiDES) and Coevolutional DES-HyperNEAT (CoDES). Design choices are discussed in section 4.4, and implementation details are given in section 4.5.

## 4.1 DES-HyperNEAT

This section describes the DES-HyperNEAT framework proposed in this work. It is itself not a concrete method, but a general framework that can be implemented with various approaches. Subsection 4.1.1 introduces the framework. The configuration of inputs and outputs is presented in subsection 4.1.2. Details about the evolving layout and CPPNs are given in subsection 4.1.3 and subsection 4.1.4. The assembling process is described in subsection 4.1.5.

### 4.1.1 Introduction

The novelty of DES-HyperNEAT is the extension of ES-HyperNEAT [Risi and Stanley, 2012], from network construction within a single *substrate*, to network construction across multiple substrates. The new approach essentially evolves deep neural networks by evolving and combining multiple ES-HyperNEAT-networks, in an evolved topology.



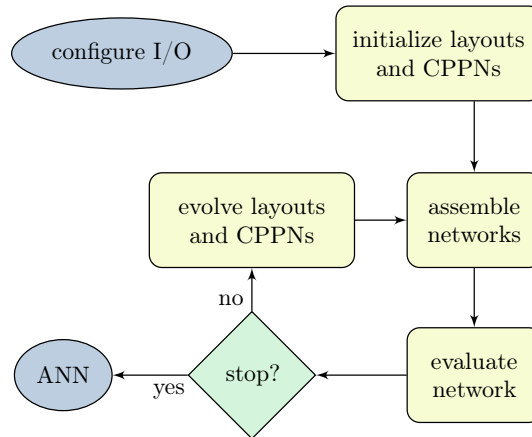
**Figure 4.1: Terminology.** A network (b) is *assembled* within a layout (a).

In the context of ES-HyperNEAT, a substrate is a two-dimensional space that contains neural network nodes. It provides a common spatial reference point for the nodes and acts as a work surface on which networks are constructed. Input and output nodes are manually positioned at specific positions within the substrate, and ES-HyperNEAT essentially constructs a neural network between the placed input and output nodes.

DES-HyperNEAT expands upon ES-HyperNEAT by using multiple substrates, which are spatially independent of each other. Figure 4.1a contains three such substrates. They are organized as a graph, with *paths* connecting them. A configuration of substrates and paths is termed a *layout*. A network, as shown in Figure 4.1b, is *assembled* within each layout in each generation. When doing so, multiple hidden nodes are created and connected in each hidden substrate. The nodes within different substrates are also connected together. As seen in Figure 4.1, two connections are created for each of the two paths in the layout. In this case, a connection cannot be created between the input nodes and the output node directly because the layout does not contain a path between the input substrate and the output substrate. To separate the two graph types in Figure 4.1, the terminology *substrate*, *path*, and *layout* is used to describe the topology of substrates. In contrast, *node*, *connection*, and *network* are used for the ANNs that are evolved during this process.

Both input nodes and output nodes can be placed in multiple substrates in DES-HyperNEAT, as in Multi-Spatial Substrate HyperNEAT. The layout in Figure 4.1a has two input nodes within one substrate and an output node within another. Substrates containing input nodes are called *input substrates*. Those

containing outputs are *output substrates*, and substrates with no input or output nodes are called *hidden substrates*. When layouts are initialized, they only contain input and output substrates. They are then evolved with NEAT, described in section 2.5, which inserts new substrates and connects paths between them. As in ES-HyperNEAT, CPPNs are used to assemble networks. In DES-HyperNEAT, one CPPN is assigned to each substrate and each path, and is used to assemble that part of the layout in each generation.



**Figure 4.2: DES-HyperNEAT overview.**

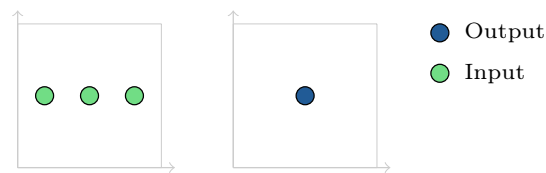
DES-HyperNEAT's process is illustrated in Figure 4.2. The first step is to position input and output nodes in substrates, represented by *configure I/O* in the figure. The population is then initialized with layouts containing these input and output substrates, along with a CPPN for each substrate. After initialization, the process then enters the main loop within the figure. A network is assembled in each layout, using its assigned CPPNs. They are assembled similarly to in ES-HyperNEAT, by determining nodes and connections based on the weight output pattern of the CPPNs, described in section 3.6. The assembled networks are then evaluated in the next step, and their fitness contributed back to the individuals that assembled them. If the stopping criterion, the diamond within the figure, is not reached, a new generation is created. The fittest individuals are selected for reproduction, and crossover performed on their layouts and CPPNs. Both the children's layouts and CPPNs are then mutated. The mutations insert substrates and paths into the layouts and also add nodes and connections to the CPPNs. The process then continues the same loop over and over again, until the stopping criterion is reached. When doing so, the process ends, and the population's fittest ANN is returned.

### 4.1.2 I/O Configuration

Similar to ES-HyperNEAT, the input and output (I/O) nodes are manually positioned within substrates in DES-HyperNEAT. As in MSS-HyperNEAT [Pugh and Stanley, 2013], the nodes can be positioned in multiple substrates. The number of substrates used, and the nodes' positions within these, is referred to as the *I/O configuration*.

The I/O configuration requires that all input and output nodes are placed in substrates. All nodes must be present, and a node cannot be placed more than once. Substrates can not contain both input and output nodes. Nodes may be placed alone in a substrate or grouped together. The I/O configuration is manually predefined and remains static for all generations.

An example is given in Figure 4.3, with three inputs and a single output. It represents a domain with three inputs and one output, and such a network will be evolved with the configuration. Inputs are configured along  $y = 0$  in a single substrate, and the output is placed at  $(0, 0)$  in another. The configuration consists of one input substrate and one output substrate, which is the starting point for evolving the layout.

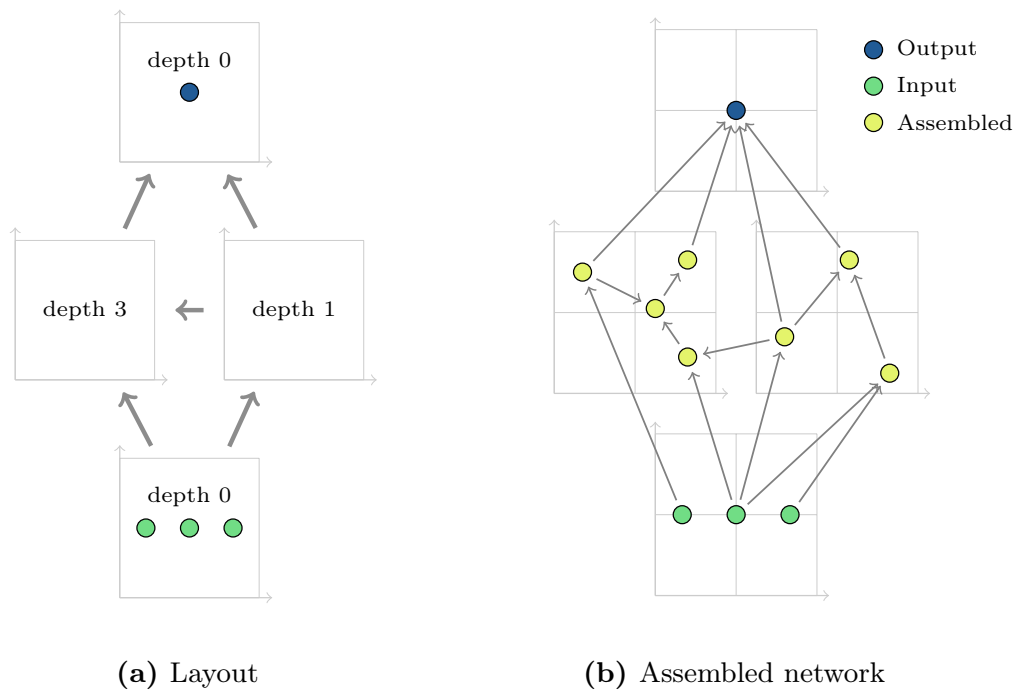


**Figure 4.3: I/O configuration.**

### 4.1.3 Layouts

Input and output substrates from the I/O configuration are used to initialize each layout. All layouts contain the I/O configuration. As stated, the layouts are then evolved using NEAT, inserting new substrates and connecting the substrates with paths. Substrates and paths are called *layout elements*. A result of evolving a layout with the I/O configuration of Figure 4.3 is presented in Figure 4.4a, where seven elements have been added. Two hidden substrates have been inserted and five paths connect the four substrates.

Layouts are the basis for assembling neural networks. They determine the substrates that should be included and how these should be connected. They also determine which CPPNs should be used to assemble each element. In Figure 4.4b, the layout in Figure 4.4a has been assembled into a neural network. Networks have been assembled within each hidden substrate. Connections have also been created within each path, connecting everything together. As seen in the figure,



**Figure 4.4: An evolved layout (a) and the network (b) assembled within it.**

connections between substrates may only be created within paths, and in the direction of the path. Additionally, all connection weights are multiplied by the weight of the path they are created within.

In addition to evolving the substrates and paths, the NEAT algorithm evolves a depth-limit value within each substrate. The value is displayed within each of the four substrates in Figure 4.4a. The value is used when assembling networks, limiting the depth of the network assembled within substrates. These are randomly mutated and included in the crossover operation between layouts. The depth-limit is evolved between 0 and the *max substrate depth* hyperparameter. However, it is fixed at 0 for all input and output substrates. The layout in Figure 4.4a has depth-limits one and three in its hidden substrates. These depths are reflected in the corresponding sub-networks in Figure 4.4b, where the depth of the networks within each substrate is less or equal to the depth limit.

Similarly to ES-HyperNEAT, DES-HyperNEAT uses CPPNs to assemble networks within substrates. DES-HyperNEAT also uses CPPNs to determine the connections between substrates. Each substrate and path must therefore have some CPPN assigned to it. How each layout element is assigned a CPPN is not strictly defined in DES-HyperNEAT. The method is defined as a framework, where this aspect differs between implementations. However, DES-HyperNEAT

requires that a CPPN output node must be assigned to each layout element, and the CPPNs are evolved in parallel with the layout. This is further elaborated upon in subsection 4.1.4.

#### 4.1.4 CPPNs

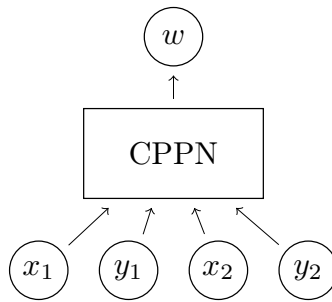
Compositional Pattern Producing Networks (CPPNs) [Stanley, 2007] are neural networks that produce four-dimensional patterns. In HyperNEAT, the produced patterns are used to determine the weights of connections, as described in section 2.6. As Figure 4.5 illustrates, they have four inputs,  $x_1$ ,  $y_1$ ,  $x_2$ , and  $y_2$ , and one output,  $w$ . The inputs are the positions of two potential nodes, relative to the substrates they each are within. The output,  $w$ , outputs the four-dimensional pattern over the four inputs. The pattern value at the coordinate corresponding to the four inputs determine the weight of the connection between the two nodes  $(x_1, y_1)$  and  $(x_2, y_2)$ . The box labeled “CPPN” is an abstraction for all the hidden nodes and connections within the network.

DES-HyperNEAT uses CPPNs, which have two use-cases. The pattern produced by a CPPN output node assigned to a substrate is used to assemble the networks within that substrate, as in ES-HyperNEAT. Likewise, the pattern from an output node assigned to a path is used to determine connections between nodes in the two substrates connected by that path. In Figure 4.4, the CPPN output assigned to the left-most hidden substrate is used to assemble the network within the left-most substrate in Figure 4.4b. The CPPN output assigned to the top left path is used to connect nodes within the mentioned substrate to the output node, where two connections are created in the path.

As mentioned, the DES-HyperNEAT framework does not define how the CPPNs are configured and evolved, only that each layout element must be assigned a CPPN output node. These can be unique to each element or reused. They can all be part of the same large CPPN, or from multiple CPPNs. The CPPNs can be part of the same genome as the layouts, or external. Three different implementations are described in section 4.3. The implementations all differ in the mentioned aspects. The layered approach embeds a unique CPPN in each substrate and path. The single CPPN approach maintains one large CPPN alongside the layout in each genome, with a unique output node assigned each layout element. The coevolutionary approach uses an entirely separate population of CPPNs, where each layout element references a species in the CPPN population. However different, they all share that all substrates and paths in some way have been assigned a CPPN output node, and that the CPPNs are evolved alongside the layouts.

In DES-HyperNEAT, the weight pattern from CPPNs are analyzed for the connections to create, rather than the querying the CPPN for the weights of pre-





**Figure 4.5: DES-HyperNEAT CPPN.** Adapted from Pugh and Stanley [2013].

determined connections. Similarly to ES-HyperNEAT [Risi and Stanley, 2012], the weight pattern is searched to determine the positions of nodes and their connections within a substrate. The process is described in detail in section 3.6, and used to assemble networks. It is an algorithm that iteratively creates new nodes within a substrate. Each iteration searches for new nodes within the substrate by analyzing the CPPN produced weight pattern. An existing node’s position is input to the search algorithm. It then outputs is the position of all nodes that should be connected to the searched node. The search additionally returns the weights of all the connections. Unique to DES-HyperNEAT, the patterns are also searched for connections between substrates. Subsection 4.1.5 describes how networks are assembled across all substrates in a layout.

### 4.1.5 Assembling Networks

Layouts are the basis for assembling networks. They must be assembled in a topologically sorted order, meaning a substrate cannot be assembled before all paths pointing to it are assembled. Paths can also not be assembled until its source substrate has been assembled. A topologically sorted order of the substrates and paths composing the layout in Figure 4.7 is denoted by the # beside each element. Since the layout is acyclic, there is always at least one valid ordering. An ordering must be found before assembling each layout in each generation. The exact ordering does not matter as long as it is a valid topological sorting.

Figure 4.6 and Figure 4.8 show the steps taken to assemble a network inside the layout in Figure 4.7. The topological order of substrates and paths directly relates to the steps shown in Figure 4.6 and 4.8, where one layout element is assembled in each step.

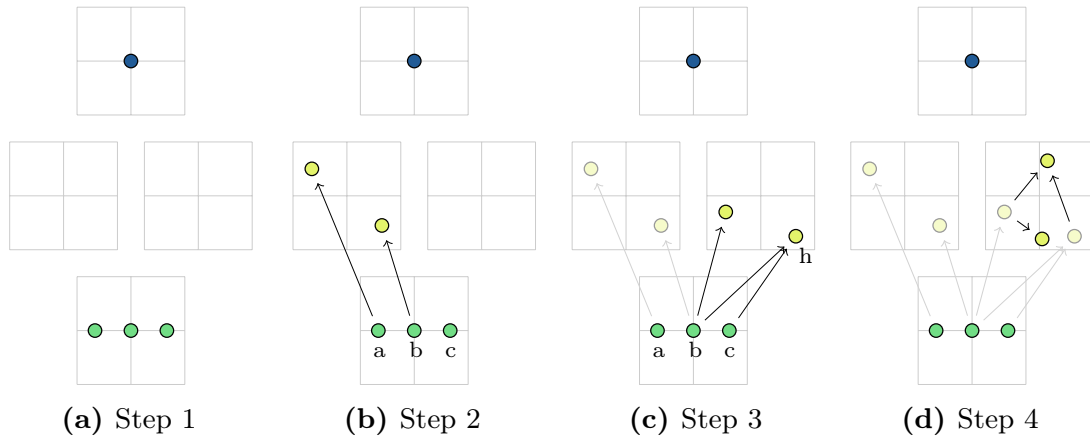
The first element to be assembled is substrate #1 in Figure 4.7. Substrates are assembled with the iterative network completion algorithm from ES-HyperNEAT, which is described in section 3.6. When no more nodes are discovered, or the search has reached the max depth determined by the substrate, that substrate

is completely assembled. Since the depth value in substrate #1 is zero, zero iterations are performed, and nothing happens in Step 1.

Step 2 and 3 in Figure 4.6 assemble paths #2 and #3 in Figure 4.7. Paths are also assembled with the iterative network completion algorithm, though only with a single iteration. The search for new nodes is performed from all nodes in the source substrate. If the search returns any positions, nodes are added to the target substrate at these positions. They are then connected to the node in the source substrate from which they were *discovered*. A node at position  $p$  is said to be discovered when a node search yields the position  $p$ , and it is discovered *from* the node at the position used as input to the search.

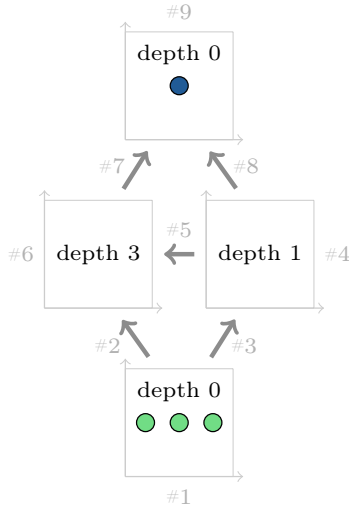
Step 2 consists of a single iteration with three searches for nodes in the target substrate, one search from each of the nodes in the source substrate. Figure 4.6b shows the result of the three searches in this step. The search from node  $a$  returns a single position. A node is added to that position in the target substrate. It is then connected to  $a$ , the node it was discovered from. Similarly, a single node is also discovered from, and connected to,  $b$ . The third and final search does not return anything, so no more nodes are added to the target substrate. Therefore, node  $c$  has no outgoing connections after Step 2.

Only one of the two paths to substrate #6 has been assembled at this point. Path #5 in Figure 4.7 has yet to be assembled. Therefore, substrate #6 cannot be assembled at this point. The process returns to the input substrate and assembles the other outgoing path from the input substrate, path #3, in Step 3. This step is very similar to Step 2, except that one node,  $h$ , is discovered from two different nodes,  $b$  and  $c$ . These can be seen in Figure 4.6c, where both node  $b$  and  $c$  is



**Figure 4.6: Multi-substrate iterative network completion part 1.** The first four steps where layout in Figure 4.4a is iteratively developed into the assembled network in Figure 4.4b.

connected to  $h$ .  $h$  is originally discovered from  $b$ , then later re-discovered from  $c$ . A connection is created between  $c$  and  $h$  even though  $h$  already exists.



**Figure 4.7: Topologically sorted layout.** Order denoted by #.

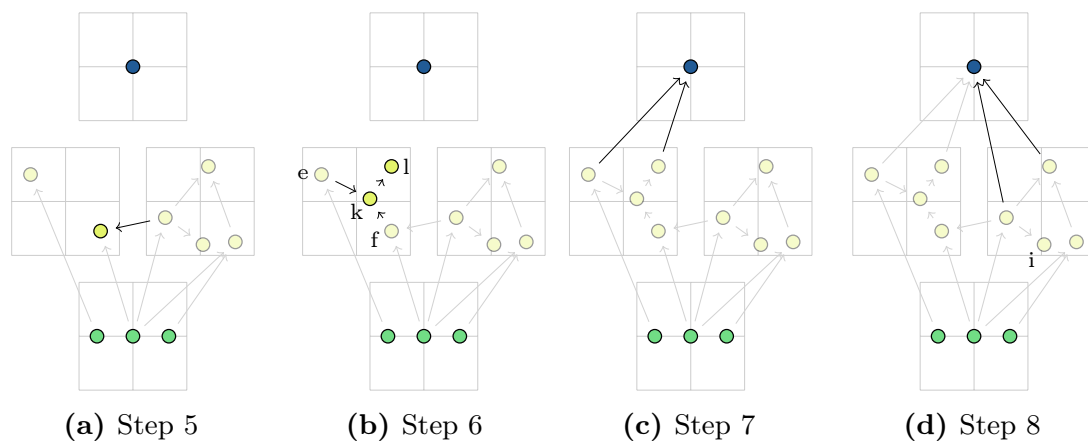
When a node is re-discovered, a connection is always created if a path is being assembled. However, if a substrate is assembled, it is only connected if the node was initially discovered the same iteration that it was re-discovered. This is to avoid cycles within the substrates. It is not an issue when assembling of paths, because the layout itself is acyclic. No cycles can therefore be formed when connecting nodes in different substrates in accordance with the path directions.

The substrate that was connected to the input substrate in Step 3 is then assembled in Step 4. All its inbound paths have been assembled at this point. Two additional nodes are discovered in the first iteration in this substrate. Because the depth of substrate #4 is one, only a single iteration is performed. Step 5 in Figure 4.8 searches for nodes in the left-most substrate, from all four nodes in the right-most substrate. A single existing node is discovered and then connected.

Step 6 is the only step with three iterations, as substrate #6 specifies a depth of three. The first iteration searches from  $e$  and  $f$ , both discovering  $k$  and connecting to it. The second iteration only searches from  $k$ , because  $e$  and  $f$  have already been searched.  $l$  is discovered, connected, and then searched in the third and final iteration. The only discovery when searching from  $l$  is a re-discovery of  $f$ . Because  $f$  was initially discovered in another iteration, connecting it will cause a cycle. It is therefore not connected.

Step 7 and 8 in Figure 4.8 are special because the assembled paths, #7 and #8 in Figure 4.7, connect to an output substrate. The node search that discovers nodes only search locations formed by a quadtree, meaning all potential node discoveries follow a distinct position pattern. As there is no guarantee that the output nodes are manually placed in locations discoverable from the node search, these paths are also assembled in reverse. The reverse search enables output nodes to connect to nodes in other substrates, regardless of the output node positions. The concept is further described in section 3.6 and is a central part of the iterative network completion from ES-HyperNEAT [Risi and Stanley, 2012]. Paths with output substrates as target substrates are therefore searched both ways, and any overlapping connections are merged.

Path #7 and #8 both point to an output substrate. Therefore, they are



**Figure 4.8: Multi-substrate iterative network completion part 2.** The final four steps.

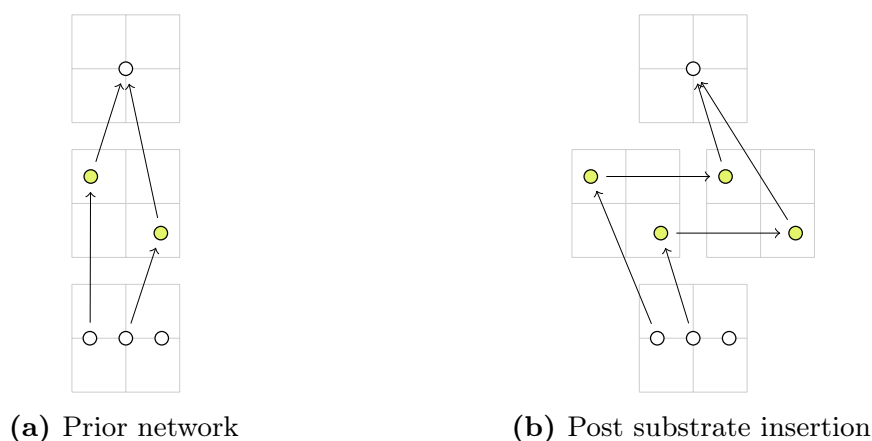
searched both directions; from all nodes in both the source substrate and the output nodes. The assembling of path #7 and #8 is shown in Figure 4.8c and Figure 4.8d, where two connections are created in each search. In Step 8, either the output node is discovered from the nodes in the source substrate; the nodes in the source substrate are discovered from the output node; or both. The final step is not shown, as substrate #9 has depth zero, and zero iterations are therefore performed.

When all the layout elements are assembled, any nodes not on a path between input and output are removed because they will not contribute to the network. The node labeled  $i$  in Step 8 is therefore removed. After pruning, the network is fully assembled. The network assembled in this process is illustrated in Figure 4.4b.

## 4.2 Modified Node Search

This section describes the modification made to how node positions are discovered, based on implicit information in the produced CPPN weight patterns. The original search from ES-HyperNEAT [Risi and Stanley, 2012], described in section 3.6, has been modified with the primary goal of enabling the possibility of an identity mapping. Such a mapping will for any search at position  $(x_1, y_1)$  yield exactly one node at position  $(x_2, y_2)$ , such that  $x_1 = x_2$  and  $y_1 = y_2$ . It is the same concept as the identity mapping with a weight of 1.0 in the NEAT algorithm, described in section 2.5, although between substrates instead of nodes.

Support for an identity mapping is created so that a substrate can be placed in-between two existing substrates in the substrate topology, similar to how nodes are inserted in the NEAT algorithm [Stanley and Miikkulainen, 2002]. When

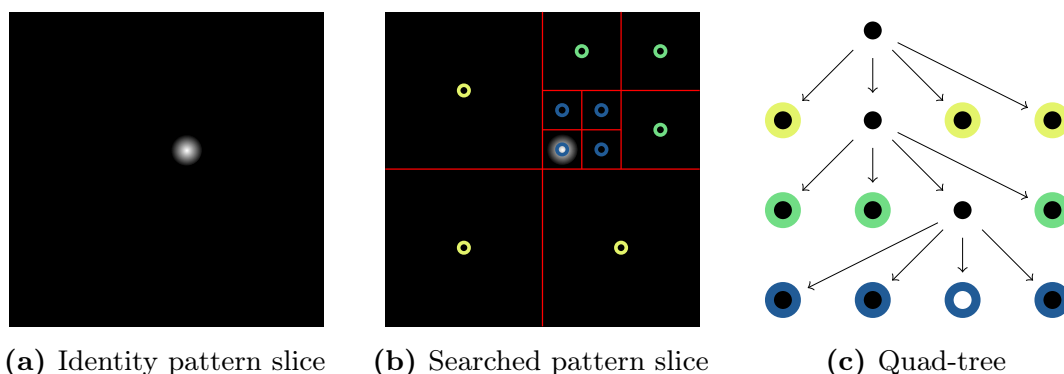


**Figure 4.9: Identity mapping between substrates.** A substrate is inserted into the layout (a), resulting in (b). An identity mapping is created between the two hidden substrates, so that the functionality of the network remains unchanged.

nodes are inserted in NEAT, they split an existing connection, creating two new. One of them keeps the old weight value. The other weight is set to 1. The connection with weight 1 is an identity mapping that ensures minimal disruption to the network output.

DES-HyperNEAT’s substrate topology is evolved with the NEAT algorithm. To avoid that insertion of a new substrate disrupts the network output, an identity mapping between substrates is created. Since the connections between substrates are determined by the node search from ES-HyperNEAT, the search must support an identity mapping. The search is therefore modified such that this mapping is possible to construct. Figure 4.9 visualizes this identity mapping. When a substrate is inserted into the layout in Figure 4.9a, the CPPN assigned to the new path is modified such that it produces an identity mapping pattern. The result is displayed in Figure 4.9b, where two horizontal connections with weights 1.0 connect the two nodes in the existing hidden substrate to the same exact locations in the newly inserted substrate. Since the CPPNs assigned to the other two paths remain the same, the connections from and to the input and output nodes remain unchanged. The functionality of both networks in Figure 4.9 is thus the same, even though one has an additional substrate.

The formula  $Gaussian(7.5 \cdot (7.5x_1 - 7.5x_2)^2 + 7.5 \cdot (7.5y_1 - 7.5y_2)^2)$  is used to create the identity mapping pattern. The function is created in the CPPN whenever an identity mapping is wanted between two substrates. When sliced in two dimensions, at  $x_1 = a$  and  $y_1 = b$ , it creates a small point at  $(a, b)$ . Figure 4.10a illustrates such a two-dimensional slice, where  $x_1 = 0.125$  and  $y_1 = 0.125$ . It has a white point at  $(0.125, 0.125)$ . The point has a weight value of



**Figure 4.10: Searched identity mapping pattern.** An identity pattern slice is searched (b) with a quad tree (c). The four-dimensional pattern is searched at  $x_1 = 0.125$  and  $y_1 = 0.125$ , resulting in a two-dimensional slice (a) with a spot with positive values at that position. Inspired by Risi and Stanley [2012].

1.0, in an otherwise uniform weight space represented by the black color. The node search algorithm is modified so that the point in the weight space is discovered as a node, and no other nodes are discovered.

When determining if a node or its children in the quadtree should be extracted, the original algorithm, used in ES-HyperNEAT, checks if the variance among the leaf nodes,  $\frac{1}{k} \sum_{i=1}^k (\bar{w} - w_i)^2$ , is above the variance threshold.  $w_i$  are the leaf nodes' weights, and  $\bar{w}$  is the mean weight value. If a quadtree was expanded to a depth of 4 within the pattern in Figure 4.10a, there would be 63 nodes with weight 0.0 and a single node with weight 1.0. Figure 4.10 illustrates this, where the nodes within the quadtree are visualized both within the pattern (b) and as a separate tree structure (c). In Figure 4.10c, nodes with borders surrounding them are leaf nodes. The borders are colored based on the node's depth within the quadtree. Although not all nodes in this quadtree are expanded to the depth 4, there is only a single node with weight 1.0 among the 10 leaf nodes. The node with weight 1.0 is the one positioned inside the white spot in Figure 4.10b and the third node at the bottom row in Figure 4.10c. The original ES-HyperNEAT search would determine that the variance within the quadtree's root node is low. Therefore, the node with weight 1.0 would not be extracted. Only the root's four children, the layer at depth 1 in Figure 4.10c, would be extracted with the original method.

The search is modified to use the formula  $\max_{i=1}^k (\bar{w} - w_i)^2$  instead of the variance among leaf nodes. It does no longer matter that only a single leaf node is different from the rest, because the max difference from the mean value is used. The node within the white spot in Figure 4.10b is therefore extracted even though it is outnumbered by a large number leaf nodes with uniform values.

Another modification to the algorithm is that all tree nodes are used in the previously mention formula, not only leaf nodes. The point of interest, with weight 1.0, can be at any depth-level in the quadtree. If the identity pattern were searched at  $x_1 = 0.5$ ,  $y_2 = 0.5$  instead, the node with weight 1.0 would be one of the root's children in Figure 4.10c. The formula must therefore use all nodes within the quadtree and not only leaf nodes for the node with weight 1.0 to be discovered.

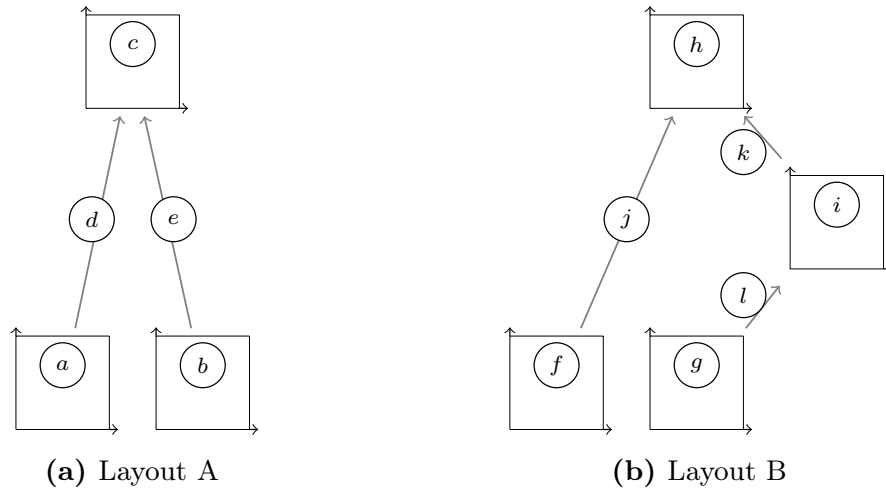
The final modification is that weight values are normalized before the variation metric is calculated. It is accomplished by dividing the variation by the max absolute difference between the discovered weight values. The formula becomes  $\max_{i=1}^k \left( \frac{\bar{w} - w_i}{w_{max} - w_{min}} \right)^2$ , where  $w_{max}$  and  $w_{min}$  are the max and min weight value discovered in the quadtree. The modification emphasizes actual variation in the weight pattern regardless of the size of the variation. When normalized, the identity mapping CPPN output may work even when the activation is changed from Gaussian. Since the pattern is normalized, it is the difference between minimum and maximum value within the pattern that matters. The pattern does not have to be between 0.0 and 1.0, as with Gaussian.

To summarize, three modifications are made. The max difference is used instead of the mean difference; weight values from all tree nodes are used instead of only leaf nodes; and the weight space is normalized during the search. These three modifications can all be toggled on or off, resulting in 8 combinations.

Finally, a minor modification to NEAT was created, so that identity-mappings only are created between hidden nodes. The modification is made because the ES-HyperNEAT identity mapping does not work for any position, only those in the quadtree. As there is no guarantee that nodes in input and output substrates are at these locations, the mapping is only guaranteed between hidden substrates. Because nodes in these substrates are discovered with the node search, they will always be at locations where the identity-mapping works. When inserting a node  $C$  between  $A$  and  $B$  in the original NEAT method [Stanley and Miikkulainen, 2002], the weight between  $A$  and  $C$  is set to 1.0. The weight between  $C$  and  $B$  remains the old value between  $A$  and  $B$ . These weight assignments are slightly modified so that the identity function is between  $C$  and  $B$  instead of  $A$  and  $C$  if  $A$  is an input node.

### 4.3 DES-HyperNEAT Implementations

This section describes the three implementations of DES-HyperNEAT in detail. As stated, they all use NEAT to evolve the layouts, but differ in their implementations of CPPNs. They are named Layered DES-HyperNEAT (LaDES), Single-CPPN DES-HyperNEAT (SiDES), and Coevolutional DES-HyperNEAT (CoDES).



**Figure 4.11: Layered DES-HyperNEAT.** Each circle labeled  $a$  to  $l$  represents a unique CPPN instance.

### 4.3.1 Layered DES-HyperNEAT

Layered DES-HyperNEAT (LaDES) embeds a unique CPPN in each of the layout's substrates and paths. Figure 4.11 illustrates two layouts, where each substrate and path has a circle representing a CPPN. The individual CPPN's are labeled  $a$  to  $l$ . The implementation is called layered because each element in the layout contains a CPPNs. Thus, there are two layers of graphs, where each node and edge at the upper level contains an entire graph at the lower level. Thus, the layouts constitute the upper level, and the CPPNs the lower level. The CPPN in each layout element is structured as the one in Figure 4.5, with four inputs and a single output. The produced weight output is used to assemble the layout element containing the CPPN. In Figure 4.11a, CPPN  $a$  and  $b$  are used to assemble the two input substrates. CPPN  $d$  and  $e$  are used to assemble the two paths, and CPPN  $c$  is used to assemble the output substrate.

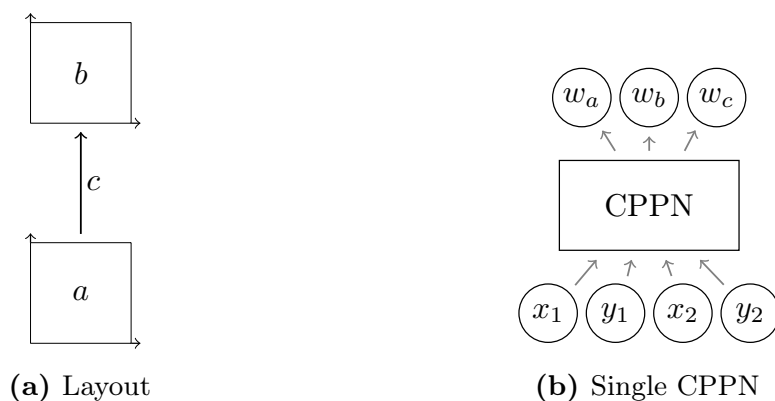
A single speciated population is evolved with the NEAT algorithm. A genome consists of a layout, with a CPPN genome in each element, as shown in Figure 4.11a. Whenever the layout is mutated, all its CPPNs are also mutated. Layout A and B in Figure 4.11 are similar, although an additional substrate has been inserted into layout B. As with the regular NEAT algorithm, described in section 2.5, each element has an id that enables the correct elements to be matched during crossover. Whenever crossover is performed between the two layouts, their substrates and paths are matched by id. For each matching element, crossover is performed on their CPPNs. When performing crossover on layout A and B in Figure 4.11, the CPPN in the child's output substrate will thus be the result of crossing over



CPPN  $c$  and  $h$ . Likewise, CPPN  $d$  and  $j$  will be crossed over, and the result inserted into the child layout's left path. Crossing over two individuals thus combines both layouts and the CPPNs. The pattern produced by a child's CPPN is a combination of the patterns produced by its parents. The distance measure used for speciation is also a combination of both layouts and CPPNs, where the distance between each matching substrate and path also encompasses the difference between their CPPNs.

As elements in the layout are matched based on their id, crossover is only performed on CPPNs within two substrates, or paths, with the same id. In Figure 4.11, crossover may be performed on CPPN  $c$  and  $h$ , as the substrates they are within have the same id. In this case, they are both output substrates. CPPN  $c$  will never be crossed over with CPPN  $f$  or  $j$ . Thus, it can be regarded as multiple CPPN groups exist within the population of Layered DES-HyperNEAT individuals. There is one such grouping of CPPNs for each substrate id and path id. CPPN  $c$  and  $h$  is part of the same group, as are  $d$  and  $j$ . The CPPNs in each group share historical markings, but are totally independent of the other CPPN groups. In the population, there is one *state* of innovations and marking ids for the layouts themselves. Additionally, a state is maintained for each id in the layout. It enables the CPPNs evolved to assemble the output substrate to be completely independent of CPPNs used to assemble any other substrates. Each area of the assembled network can thus optimize individually.

Whenever innovations are evolved in a layout, the new elements receive new historical markings from the NEAT algorithm. A new CPPN is initialized and put inside each of the new elements. A new state for CPPNs is also created for each new historical marking in the layout. When the substrate containing  $i$  was inserted into Figure 4.11b, the CPPNs  $i$  and  $k$  were initialized. As it is the first time this exact innovation occurs within the population, two new states are created alongside CPPN  $i$  and  $k$ . The next time a substrate is inserted between an individual's output substrate and right-most input substrate, the innovation already exists within the layouts' innovation log. It has already occurred within the population. As in NEAT, the new substrate thus receives the same id as the one inserted into Figure 4.11b. Additionally, the two CPPN states are not created, as these also exist from the first innovation. Instead, the CPPN in the inserted substrate becomes part of the same group as  $i$ . Likewise, the CPPN inserted into the new path is part of the same group as  $k$ .



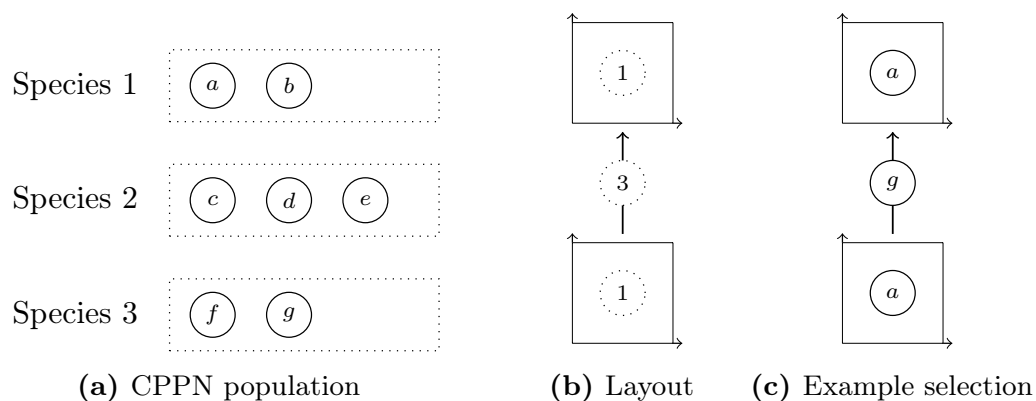
**Figure 4.12: Single CPPN DES-HyperNEAT.** The elements in the layout (a) are each assigned a unique output node in the CPPN (b).

### 4.3.2 Single CPPN DES-HyperNEAT

Single CPPN DES-HyperNEAT (SiDES) is inspired by MSS HyperNEAT [Pugh and Stanley, 2013], where a single combined CPPN is used to assemble all layout elements. The single CPPN is illustrated in Figure 4.12b. It has the usual four inputs, but has three outputs instead of one. It has three outputs because the layout it is used to assemble, in Figure 4.12a, has three elements. Each substrate and path is assigned a unique CPPN output node. In Figure 4.12, the bottom substrate is assembled with the pattern produced by output node  $w_a$ . The top substrate is assembled with  $w_b$ , and the path is assembled with  $w_c$ .

The genome in this implementation contains both the topology and CPPN. Whenever a new substrate or path is created in the topology, new output nodes are added to the CPPN. The number of output nodes in an individual's CPPN therefore increases linearly with the number of elements in its layout. NEAT is used to evolve both graphs at the same time. Two states of innovations and markings are maintained, one for layouts and another for CPPNs. The mutation and crossover operations are performed on the layout and CPPN separately. The distance between genomes, used for speciation, is the sum of the distance between two individual's layouts and CPPNs.

A downside with a single CPPN is that the entire CPPN has to be executed when only the value of a single output node is queried. This problem is omitted by extracting the different output nodes into separate CPPNs before networks are assembled. In this process, each output node is copied out from the CPPN, and all the nodes it depends upon are cloned with it. The output nodes can then be executed separately, although their functionality remains the same. Separating the CPPNs is also time consuming, but enables rapid execution of each output value. The extracted clones are discarded once the network is assembled.



**Figure 4.13: Coevolutional DES-HyperNEAT.** Each element in a layout (a) references a species in the CPPN population (b). A random individual from the referenced species is selected in each element.

### 4.3.3 Coevolutional DES-HyperNEAT

Coevolutional DES-HyperNEAT (CoDES) evolves two populations with NEAT simultaneously, layouts and CPPNs [Stanley, 2007]. It is an application of coevolutionary concepts from CoDeepNEAT [Miikkulainen et al., 2019] in DES-HyperNEAT. Since NEAT employs evolution through speciation, each population is divided into multiple species. As illustrated in Figure 4.13b, each substrate and path in the layout reference one of the species in the CPPN population, illustrated in Figure 4.13a. A randomly selected CPPN individual from each of the referenced species is selected to assemble the corresponding part of the layout in each iteration. The same CPPN can be used to assemble multiple elements, if its species is referenced multiple times. An example selection is shown in Figure 4.13c. The bottom substrate in the layout references species 1, so either  $a$  or  $b$  is selected at random. In this case,  $a$  is selected for both the input and output substrates. The path references species 3, and the CPPN individual  $g$  is randomly selected from it.

The main concept is that new CPPNs are selected each time the layout is assembled. Therefore, the elements are not assembled by the same CPPN every generation. The same CPPN individual can also be reused, to assemble multiple elements in the same or different layouts. Assigning fitness is a challenge because multiple CPPN individuals contribute to assemble a single network. When the assembled network is evaluated, the fitness is then returned to both the layouts and the CPPNs that are used to assemble it. Fitness is assigned as in CoDeepNEAT, where the fitness of a layout or CPPN is the average fitness of all assembled networks to which they contribute. Each layout is developed once, so the fitness of a layout is the fitness of the network assembled inside it. The fitness of a CPPN

is the average fitness of all networks it was used to assemble, regardless if it was assigned to assemble a single or multiple elements within the networks.

The coevolutional approach initializes all the CPPNs at the beginning, and no new are created during evolution. In contrast to the other two implementations, newly created layout elements do not receive new CPPNs. Instead, CoDES assigns an existing species within the CPPN population, and an existing CPPN is used when assembling the new layout element. It also makes the manually constructed identity function discussed in section 4.2 useless, as there is no way it can be assigned to the correct path. The identity mapping is therefore not used in this implementation. CoDES therefore disrupts the network functionality whenever a new substrate is inserted.

	<b>LaDES</b>	<b>SiDES</b>	<b>CoDES</b>
Genome	CPPN genomes embedded in each element in the layout genome.	Layout genome and CPPN genome concatenated.	Separate populations with layout genomes and CPPN genomes.
Fitness assignment	Accurate	Accurate	Less accurate
Number of outputs in each CPPN	1	Equal to number of elements in layout.	1
CPPN reuse	No	Somewhat	Yes
CPPN initialization	A new CPPN is initialized when a layout element is created.	A new output node is added to the CPPN when a layout element is created.	CPPNs are only initialized the when algorithm starts.
Identity mapping	Yes	Yes	No
Implementation	Two-layered NEAT. Each substrate and path at contains an entire CPPN.	NEAT with two networks in each individual.	Two populations evolve with NEAT, in parallel.

**Table 4.1: Implementations comparison.**

### 4.3.4 Comparison

The three implementations are compared in Table 4.1. Layout genomes and CPPN genomes exist within the same individual in LaDES and SiDES. CoDES uses two separate populations, where each layout element references a species in the CPPN population. Fitness can therefore be accurately assigned in LaDES and SiDES, because a single individual determines how the network is assembled. However, in CoDES, multiple individuals contribute, and the exact fitness contribution from each CPPN individual is unknown. Averaging fitness over all the assembled networks an individual contributes to introduces some randomness, so the fitness assignment is less accurate in CoDES.

LaDES and CoDES both use CPPNs with a single output node, while SiDES use one large CPPN with multiple outputs. A combined CPPN, with multiple outputs, enables output nodes to share common substructure within the CPPN. The same CPPN can therefore somewhat be reused to assemble multiple layout elements. However, it is not the same explicit reuse as in CoDES, where multiple layout elements can reference the same CPPN species. LaDES has no reuse, meaning the CPPNs used to assemble different layout elements are entirely independent. Nothing can be reused, but has to be evolved separately for separate layout elements. If good CPPN structures are evolved in SiDES, or a good CPPN is evolved in CoDES, these can be utilized to assemble multiple layout elements. However, it may also be a disadvantage. Different CPPN outputs in SiDES might conflict and not be able to optimize at the same time when they share hidden nodes. Also, multiple uses of the same CPPN in CoDES might create redundancies within the network, adding complexity without improving performance.

A unique aspect of CoDES is that CPPNs are never created during evolution, meaning new substrates use existing CPPNs instead of initializing new. LaDES initialize an entirely new CPPN for each layout element, while CoDES adds a new output node to the existing CPPN. Since a CPPN is not initialized when inserting a substrate in CoDES, it does not support the identity mapping. There is no way to guarantee that CPPN with the identity mapping function is assigned the newly created path.

LaDES require that NEAT is extended to two layers, which increases the implementation complexity. CoDES is also complex, because the two populations must evolve in parallel. However, SiDES only require minor modification to NEAT for it to evolve two networks in each individual.

## 4.4 Design Choices

Important choices made while developing the DES-HyperNEAT framework are discussed in this section. The reason why DES-HyperNEAT is proposed as a

framework instead of a method, is discussed in subsection 4.4.1. The depth and complexity of networks within substrates are discussed in subsection 4.4.2.

### 4.4.1 Framework

DES-HyperNEAT is designed as a framework to enable comparisons between multiple approaches. All the properties in Table 4.1 factor into the performance and implementation complexity. The three proposed implementations all have some properties that are thought to be beneficial, and others that might be disadvantageous. The three implementations presented in section 4.3 are chosen because they all have promising features, and each represents unique aspects. By evaluating and comparing all three, insight into which features benefit DES-HyperNEAT can be gained. An additional benefit is that when comparing methods that represent multiple aspects of each property, it might also be apparent how the features impact DES-HyperNEAT.

### 4.4.2 Search and Depth Limits

While developing the implementations, it was discovered that nodes sometimes connected to almost every other possible node in the substrate. The increased amount of nodes and connections slowed down evolution. Both the node search algorithm's and network execution's runtime increase with the number of connections. It was observed that this property affected the reproducibility of results, especially when the performance was measured after a specific time-period. Two countermeasures were employed to avoid that some abnormally complex networks emerged and slowed down evolution. The number of nodes expanded in the node search quadtree was limited to 256, and the number of nodes discovered when searching a single node was limited to 32. These numbers are chosen as they seemingly can limit too many connections without impacting performance. The quadtree search explores top-down and is stopped when reaching the node expansion limit. Only the 32 outgoing connections with the highest absolute weight are selected when limiting discovered nodes.

Depth limits within substrates are employed when assembling the networks within substrates, for two reasons. First of all, more computation is required to evolve deeper networks. If the complexity of a deeper network is not required, it should not be created. The second reason is that substrate can limit the depth even though the CPPN used to assemble it produces a weight space, that when assembled with the iterative network completion algorithm, yields a deep network. Instead of the algorithm stopping at the iteration where no more nodes are discovered, it is stopped at the depth limit determined by the substrate. Different substrates might require different depths, so this value is evolved separately in each one.

It is decided to lock the depth limit of input substrates at zero. It is to directly distribute the network's input signals among multiple substrates. If a complex network is formed within the input substrates, the input signals will be processed before reaching any hidden substrates. Critical information might be lost if the networks in input substrates perform poorly. When the depth is zero, no new nodes are discovered in the input substrates, and the input signals are thus directly distributed to the hidden substrates.

## 4.5 Implementation Details

This section briefly elaborates upon the implementation details of some of the methods and the simulator. The custom implementation of both methods and the simulator is discussed in subsection 4.5.1, the representation implemented for the NEAT algorithm is presented in subsection 4.5.2, and the activation functions used in CPPNs are presented in subsection 4.5.3.

### 4.5.1 Custom Implementation

The proposed framework DES-HyperNEAT, and the three implementations, are complex and require extensive modification to aspects of NEAT. The layered DES-HyperNEAT implementation requires that the NEAT algorithm has two layers, where each node and link on the upper level contains an entire network that is also evolved. The single CPPN approach requires that two NEAT networks are evolved simultaneously and that a mutation in the layout updates the CPPN. The coevolutionary approach requires two separate populations and customized fitness assignments. In addition to the three implementations, the search for nodes in ES-HyperNEAT is also modified.

Because of the extensive modifications required to existing algorithms, it was decided to create the entire implementation from scratch, instead of extending existing code implementations. It was to gain full control over the simulator and all the implemented methods. With the opportunity to design everything, the parts were created with a modular design. It enabled that the NEAT algorithm could be easily extended to support the three custom implementations. Therefore, it simplified the DES-HyperNEAT extension as a whole, compared to extending an existing ES-HyperNEAT implementation.

### 4.5.2 Representation

Stanley and Miikkulainen [2002] describe the NEAT algorithm with a list of connection genes. It has been implemented as a HashMap, enabling fast lookup for any connection based on the two nodes it connects. An additional data

structure is used to lookup outgoing connections from any node. It is used when assembling the genome into a neural network. Based on the HashMap of outgoing connections, a topologically sorted order of nodes and edges is created. It is transformed into a separate neural network representation, which can then be executed element for element in linear time. Although the genome representation may differ from common implementations, the linear execution time is not novel.

### 4.5.3 Activation Functions

The activation functions presented in Table 4.2 were implemented for the CPPNs. These are adapted from earlier work [Green, 2006; McIntyre et al., 2017]. Exp is limited to  $x$  values below 1.0, to avoid large values.

None	$x$
Linear	$\begin{cases} 1.0, & \text{if } x \geq 1.0 \\ -1.0, & \text{if } x \leq -1.0 \\ x, & \text{otherwise} \end{cases}$
Step	$\begin{cases} 1.0, & \text{if } x > 0.0 \\ 0.0, & \text{otherwise} \end{cases}$
ReLU	$\begin{cases} x, & \text{if } x > 0.0 \\ 0.0, & \text{otherwise} \end{cases}$
Exp	$\begin{cases} e^x, & \text{if } x < 1.0 \\ e, & \text{otherwise} \end{cases}$
Sigmoid	$\frac{1}{1+e^{-x}}$
Tanh	$\tanh(x)$
Gaussian	$e^{-(2.5x)^2}$
OffsetGaussian	$2.0 \cdot e^{-(2.5x)^2} - 1.0$
Sine	$\sin(2x)$
Square	$x^2$
Abs	$ x $

**Table 4.2: Implemented activation functions.** Formulas adapted from Green [2006] and McIntyre et al. [2017].



# Experiments and Results

The experiments conducted to gain knowledge and provide answers to the research questions are the topic of this chapter. An introduction to the types of experiments and how they are presented and conducted is given in section 5.1. The preliminary testing is then presented in section 5.2, the experimental plan in section 5.3, and the experimental setup in section 5.4. Finally, results from the four experimental phases are presented and discussed in section 5.5 - 5.8.

## 5.1 Introduction

The following section describes how experiments are presented and conducted, and how results are collected and presented. All experiments are defined by *experimental parameters*, outlining the experiment. They may additionally have some *hyperparameters* controlling the methods.

### 5.1.1 Experimental parameters

An example experiment comparing NEAT, HyperNEAT, and ES-HyperNEAT is presented in Table 5.1. Its experiment number is presented in the top right corner and its title in the text below the table. All four key-value pairs are experimental parameters. They define the experiment. The three methods are evaluated in their ability to learn the Iris, Wine, and Retina datasets. The comma-separated list notation illustrates the grid of parameters that is executed. All combinations of values within the lists are executed, while the parameters with single values are static in all executions. Table 5.1 defines that three methods, three datasets,

## Experiment X.1

method	[NEAT, HyperNEAT, ES-HyperNEAT]
dataset	[Iris, Wine, Retina]
stop criterion	200 generations
repeats	50

**Table 5.1: Example experiment 1: Experimental parameters.**

and a single stopping criterion is tested, which are nine combinations in total. All of these each run for 200 generations and are repeated 50 times.

### 5.1.2 Hyperparameters

In addition to the experimental parameters, experiments may have some hyperparameters defined. These can either be static or part of a parameter search. A large number of hyperparameters make a complete grid-search infeasible, due to the enormous search space. Therefore, when determining parameters, it is decided to adopt some from earlier work and only test those expected to impact performance significantly. Parameters are also tested in batches to reduce the search space further. Those that are thought to be correlated are tested together. The hyperparameters with the highest expected impact are determined first. Tested parameter grids are initially rough and refined through multiple iterations of testing. Initial values are selected based on earlier work, to reduce the number of iterations required to discover optimal values.

An example parameter search experiment is presented in Table 5.2. The top part of the table contains experimental parameters described in subsection 5.1.1. The middle determines that the activation function Sigmoid is overwriting the default activation function in this experiment. The table additionally contains batches with hyperparameter grids. Each batch is performed by running all combinations of the experimental parameters, static hyperparameters, and batch hyperparameters. Batch 1.1 thus comprises 1200 individual runs: all combinations of methods, datasets, population sizes, and species targets, each running for 120 seconds, and each repeated 50 times.

When all runs in a batch iteration are complete, the average performance of each combination is calculated. The top three or more combinations of batch parameters are then reviewed for each combination of experimental and static parameters. With two methods and two datasets, this results in reviewing 12 combinations when the top three in each are compared. The batch parameters within these top 12 combinations are then evaluated based on the number of times

## Experiment X.2

method	[NEAT, HyperNEAT]
dataset	[Iris, Wine]
stop criterion	120 seconds
repeats	50
activation function	Sigmoid
<b>Batch 1.1</b>	
population size	[100, 400]
species target	[None, 10, 20]
<b>Batch 1.2</b>	
population size	[75, 200]
species target	[8, 12]
<b>Batch 2</b>	
add node prob.	[0.05, 0.15, 0.25]
add edge prob.	[0.05, 0.15, 0.25]

Table 5.2: Example experiment 2: Parameter search.

	NEAT	HyperNEAT
<b>Iris</b>	0.988 {p.size: 100, s.target: 10}	0.960 {p.size: 100, s.target: 10}
	0.986 {p.size: 400, s.target: 10}	0.959 {p.size: 100, s.target: None}
	0.986 {p.size: 100, s.target: 20}	0.956 {p.size: 100, s.target: 20}
<b>Wine</b>	0.662 {p.size: 100, s.target: 20}	0.510 {p.size: 100, s.target: 10}
	0.661 {p.size: 100, s.target: 10}	0.498 {p.size: 400, s.target: 10}
	0.601 {p.size: 400, s.target: None}	0.495 {p.size: 100, s.target: 20}

Table 5.3: Example results. From batch 1.1 in Table 5.2

they appear and the performance of the combinations in which they appear.

Table 5.3 provides example results, where the 12 combinations are compared. The top three combinations of batch parameters are displayed for each combination of method and dataset. In this case, the population size 100 is part of 9 of the 12 combinations in batch 1.1, implying it generally performs better than 400. The value 100 scores well and is part of all four top combinations of methods and datasets. The area surrounding 100 is therefore further tested in the batch's next iteration, batch 1.2. However, if there were no significant fitness differences between the combinations containing 100 and 400, one of them would be selected. The described process is conducted for each parameter in each batch iteration. The search is complete when a single value is selected for each hyperparameter.

### 5.1.3 Results

Results presented within tables are from the final populations, after reaching the stopping criteria. Values are gathered from the individual with the highest validation fitness within a population. The presented mean values are averages of all repeated runs. Graphs also present mean values for all runs. The number of times an experiment is repeated is presented in the table defining the experiment. The standard deviations (SD) within the samples are presented alongside the mean values.

Where a  $p$ -value determining significance is presented,  $p < 0.05$  is regarded as significant. The  $p$ -value is the result of a Welch's t-tests when the means of two samples are compared, and Two-Factor ANOVA with replication where the effect of a change is measured over more than one variable [Devore and Berk, 2007]. Thus, to see if model  $A$  performs significantly better than model  $B$  on a single dataset, the t-test is used. However, if the models are compared in multiple datasets simultaneously, the Two-Factor ANOVA test is used.

## 5.2 Preliminary Testing

This section describes the preliminary tests conducted to find good hyperparameters. The goal is not to optimize the parameters to a specific method or problem. It is instead to find a single set of parameters that achieve good results across all the following methods: NEAT, CPPN, HyperNEAT and ES-HyperNEAT. Such parameters are thought to be a good baseline for comparing the three implementations of DES-HyperNEAT.

All three implementations of DES-HyperNEAT use the same set of parameters. They are so similar that a parameter set optimal for one of them is likely also the optimal set for the others. However, a deliberate choice is made not to include any of the three proposed implementations in this search. As the goal of the two first phases of experiments is to investigate how the unique properties of each one impact the framework, a single set of parameters is used. If each one were individually optimized, it would be difficult to distinguish the difference in implementation from the difference in parameters. If a set of parameters were optimized to work well with all three, it would be difficult not to bias the value choices towards favoring some of them. Therefore, to give them all a fair evaluation, the parameters used are the ones that work well for the methods they are built upon. It will not result in them performing their absolute best, but will enable analysis of their properties. After evaluating the properties of DES-HyperNEAT, a new parameter search will be performed to optimize the chosen implementation and configurations of it.

The NEAT algorithm and algorithms building on NEAT are affected by a large

number of hyperparameters. These can be divided into two categories: *general* and *method-specific* parameters. The general parameters affect the population, its division into species, and how the species interact and reproduce. These are not limited to a single algorithm, but general to the evolution of any population divided into species. The method-specific parameters regard a specific method, such as NEAT and ES-HyperNEAT.

DES-HyperNEAT uses aspects of ES-HyperNEAT, which in turn uses CPPNs, using the NEAT algorithm. To find the hyperparameter set for DES-HyperNEAT, the parameters will be determined from the bottom up. Meaning the general parameters will be decided first, then the ones introduced by NEAT, CPPN, and ES-HyperNEAT. In each of the four phases of the parameter search, the methods used for evaluation will be the ones using the currently optimized parameters. When determining the general and NEAT parameters, they will be evaluated using all three methods. When the CPPN parameters are determined, only CPPN and ES-HyperNEAT will be used. Finally, only ES-HyperNEAT will be evaluated when determining the parameters introduced by ES-HyperNEAT. The parameters selected in each of the four parameter search phases are presented in Table 5.4. DES-HyperNEAT uses all these parameters, though they are categorized by the methods that introduce them.

It is decided to use time as stopping criteria, instead of the number of generations, because the different methods differ a lot in time per generation. The parameter searches are run for 30 seconds. This duration is chosen because many combinations of methods and datasets increase fitness the most during this time. How they converge after longer training is not regarded when comparing parameters in the preliminary testing. Because they mostly do not overfit during this training time, they are scored based on training fitness for the entire dataset. The preliminary testing is the only experiment performed without a validation set for determining fitness.

Three datasets are chosen: Iris, Wine, and Retina. Iris and Wine [Dua and Graff, 2017] are two class prediction datasets with 4 and 13 attributes, respectively. Both data sets contain three classes. They are chosen because they are commonly used in machine learning, and they differ in difficulty and their number of attributes. Additionally, the Iris dataset has inputs of similar types, while the input types differ in Wine. The inputs are all length measurements in Iris, and unique attributes with different units of measurement in Wine. The Retina experiment [Kashtan and Alon, 2005] used by Risi and Stanley [2012] is also chosen because it has been used in previous work and has distinct relationships between inputs.

The parameter search is performed as described in subsection 5.1.2. Selected values are the result of iteratively testing various parameter grids. All the tested combinations throughout the parameter search are presented in Table A2

<b>General</b>	
population size	100
species target	8
survival ratio	0.2
initial mutations	100
asexual reprod. prob.	0.25
<b>NEAT</b>	
add node prob.	0.03
add link prob.	0.2
remove node prob.	0.006
remove link prob.	0.08
initial link size	0.5
link mutation size	0.5
link mutation prob.	0.9
<b>CPPN</b>	
mutate bias prob.	0.8
mutate bias size	0.03
mutate activation prob.	0.1
activations	[Tanh OffsetGauss Gaussian Sine Sigmoid]
<b>ES-HyperNEAT</b>	
division threshold	0.03
variance threshold	0.03
band threshold	0.3

**Table 5.4: Resulting hyperparameters from preliminary testing.** All parameters are used by DES-HyperNEAT, although they are categorized by the methods that introduce them.

- A5. Table 5.4 presents the resulting set of values. These parameters are chosen for analyzing the DES-HyperNEAT framework, by comparing the three implementations. After one of them is selected and properties of the framework configured, a new parameter search will be performed to optimize it, based on the parameters selected in this search.

### 5.3 Experimental Plan

The conducted experiments are divided into four phases. A description of each phase, and its purpose, is presented in Table 5.5. Hypotheses are formed in each phase to define their purpose further and contribute to the discussion of results. These hypotheses are presented in their respective sections and numbered separately in each phase. Experiments are globally numbered.

---

### Phase 1: DES-HyperNEAT implementations

**Experiment 1:** Compare and analyze the three model implementations

Gain knowledge about the simultaneous evolution of substrate topologies and networks within each substrate. Specifically, which of the three implementations best evolves CPPNs and layouts, and why.

---

### Phase 2: Identity mapping

**Experiment 2:** Evaluate the three node search modifications

**Experiment 3:** Determine if an identity mapping is required

Determine whether any of the modifications to the node search are beneficial, and if an identity function is required to successfully evolve a network across multiple substrates.

---

### Phase 3: DES-HyperNEAT tuning

**Experiment 4:** Determine depths in I/O substrates

**Experiment 5:** Determine depths in hidden substrates

**Experiment 6:** Evaluate I/O configurations

**Experiment 7:** DES-HyperNEAT parameter search

Investigate the impact of different substrate depths, both in I/O substrates and hidden substrates. Additionally, determine how to create well-performing I/O configurations. Finally, perform a parameter search to optimize parameters to the implementation selected in phase 1 and the findings of phase 3.

---

### Phase 4: Related methods comparison

**Experiment 8:** Compare NEAT and HyperNEAT related methods

Compare DES-HyperNEAT to NEAT, HyperNEAT, and ES-HyperNEAT, to evaluate DES-HyperNEAT in the context of similar methods within the field. Determine whether the incremental method complexity in HyperNEAT, ES-HyperNEAT, and DES-HyperNEAT is beneficial compared to the simpler methods they build upon.

---

**Table 5.5: Experimental plan.**

Datasets	Based on	Activation	Fitness function
Retina	Mean square error	Tanh	$1 - \frac{1}{n} \sum_{i=1}^n \frac{1}{m} \sum_{j=1}^m (y_{ij} - \hat{y}_{ij})^2$
Iris and Wine	Cross-entropy loss	Softmax	$e^{\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m y_{ij} \cdot \ln(\hat{y}_{ij})}$

Table 5.6: Fitness functions.

The experiments will be executed with the same datasets as in the preliminary testing: Iris, Wine, and Retina. Validation fitness and accuracy, network complexity, and execution speed will be collected in each experiment. This data is analyzed to gain insight into how different combinations of methods and configurations work, beyond what can be concluded from the fitness alone. Each experiment is repeated at least 50 times to gain reliable results and standard deviations.

## 5.4 Experimental Setup

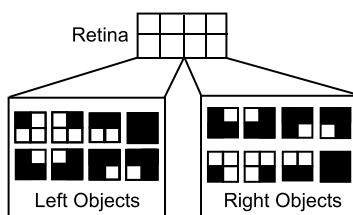
All experiments are performed on a single Intel Xeon E5-2630 v2 core. The default hyperparameters used in all experiments are presented in Table A1. If any hyperparameters are defined within an experiment, these are used instead of the default values.

The fitness functions used for different datasets, and the output activation functions of the networks, are presented in Table 5.6. The functions are based on mean square error and cross-entropy loss.  $n$  is the number of examples within the dataset and  $m$  the number of values in each target.  $y$  is the correct target, and  $\hat{y}$  is the network prediction. Iris and Wine are one-hot encoded so that each of the three prediction classes has a separate output node.

The datasets are divided into training and validation sets, with a validation fraction of 0.2. They are randomly shuffled before doing so, and the random selection is the same for all experiments. The training set is used to train the methods, while they are continuously evaluated with the validation set. Test sets are not used because the validation set never affect the training. Instead of stopping when the validation performance decreases, a separate stopping criterion is used, limiting either time or number of generations.

All 256 combinations of the Retina dataset [Kashtan and Alon, 2005], shown in Figure 5.1, are used. It is a classification experiment, where the left and right part of the retina are independently classified. For each side, the pattern is either part of the objects listed or not. The prediction is thus two booleans for each





**Figure 5.1: Retina experiment.** The patterns within the left and right part of the retina are classified as either part of the shown objects or not. Image from Risi and Stanley [2012].

Method \ Dataset	Iris	Wine	Retina
HyperNEAT			
ES-HyperNEAT			
DES-HyperNEAT			

● Output    ● Input    ● Hidden

**Table 5.7: Default node configurations.** HyperNEAT and ES-HyperNEAT use a single substrate. ES-HyperNEAT uses two substrates, one for inputs and another for output.

input of eight booleans. These booleans are encoded as  $-1$  and  $1$  in the dataset. The left and right objects shown in Figure 5.1 will be used.

The default configurations of input, hidden, and output nodes is presented in Table 5.7. Inputs and outputs are in the order they appear in the Iris and Wine datasets. In Retina, the left and right inputs are placed in the left and right half of the substrate, with the respective outputs in the top left and right corner. Inputs and outputs are configured in different substrates in DES-HyperNEAT. Networks in HyperNEAT are densely connected between the four layers.

## Experiment 1

method	[LaDES, SiDES, CoDES]
dataset	[Iris, Wine, Retina]
stop criterion	[1200 seconds, 600 generations]
repeats	50

**Table 5.8: Experiment 1: DES-HyperNEAT implementations.**

## 5.5 Phase 1: DES-HyperNEAT Implementations

DES-HyperNEAT was developed as a framework to enable comparisons of multiple implementations. By constructing multiple implementations, where some properties are shared and other unique, the impacts of properties can be analyzed. The three implementations, Layered DES-HyperNEAT (LaDES), Single CPPN DES-HyperNEAT (SiDES), and Coevolutional DES-HyperNEAT (CoDES), are compared in subsection 4.3.4, with a summary of their properties presented in Table 4.1. They are the subject of the first experiment.

### 5.5.1 Experiment 1: DES-HyperNEAT Implementations

An experiment is conducted to analyze how the unique properties of different implementations affect the DES-HyperNEAT framework, and the neural networks evolved with it. Details about the experiment are presented in Table 5.8. The hyperparameters from preliminary testing, presented in Table 5.4, are used for all three implementations. Using equal parameters for all three implementations enables a better comparison of their properties. If different parameters were used, optimized for each implementation, it would be difficult to distinguish the difference in implementation from the difference in parameters.

The datasets Iris, Wine, and Retina are used to compare the implementations, with both 1200 seconds and 600 generations defined as stopping criteria. Generations are used to analyze the absolute impact of different implementation properties, such as how CPPN initialization affect network complexities. Time is used to compare properties relative to their execution speed. One implementation might create more complex networks than an other in the same number of generations. However, if it is slower, they might create an equally complex network when run for the same amount of time.

CoDES uses two separate populations, one containing layouts and another containing CPPNs. When a substrate or path in a layout is assembled, a random

CPPN individual within the referenced species is selected. Such random selection is, in itself, not seen as an issue. However, since a random CPPN is selected from each species every time the layout is assembled, the combination of CPPNs varies between generations. It will likely make the algorithm more unstable than when the same combination of CPPNs is used each time.

LaDES and SiDES have a layout and its required CPPNs within a single individual. The fitness of an individual is the fitness of the network assembled with its layout and CPPNs. CoDES however use multiple CPPN individuals when assembling a single network. The fitnesses of the CPPN individuals are the average fitness of networks they assemble. Therefore, a CPPN's fitness depends on the random selection of CPPNs it is grouped with when assembling a network, causing its fitness assignment to be less accurate. The exact contribution of each CPPN individual is unknown.

**Hypothesis 1** CoDES will perform worst of the three proposed methods because of the varying selection of CPPN individuals developing the network within a layout and less accurate fitness assignment. The two properties are not present in LaDES and SiDES, because they use individuals with both a layout and the required CPPNs.

**Hypothesis 2** SiDES will use more time per generation than LaDES, because of the time used in the additional step needed to extract individual CPPNs from the single large CPPN. LaDES use a separate individual for each layout element, so they are already separated.

**Hypothesis 3** SiDES will produce the most complex networks because it uses a single CPPN. The CPPN output nodes share hidden nodes, which may cause more complex weight patterns than the individual CPPNs. LaDES and CoDES will produce equally complex networks because they both use individual CPPNs, and the average pattern complexity should therefore be similar.

**Hypothesis 4** SiDES will perform worse than LaDES when comparing fitness to generations because the correlation between CPPN outputs will negatively affect their optimization.

The performance of the three implementations is presented in Table 5.9. It contains both validation fitness and validation accuracy, measured with both time and generations as stopping criteria. Fitness is a more stable and reliable performance measure than accuracy, as fitness is derived from loss functions. The fitness functions are shown in Table 5.6. They measure how correct each prediction is, by accumulating the distance between the predictions and each correct answer. It thus measures how correct each prediction is. In contrast,

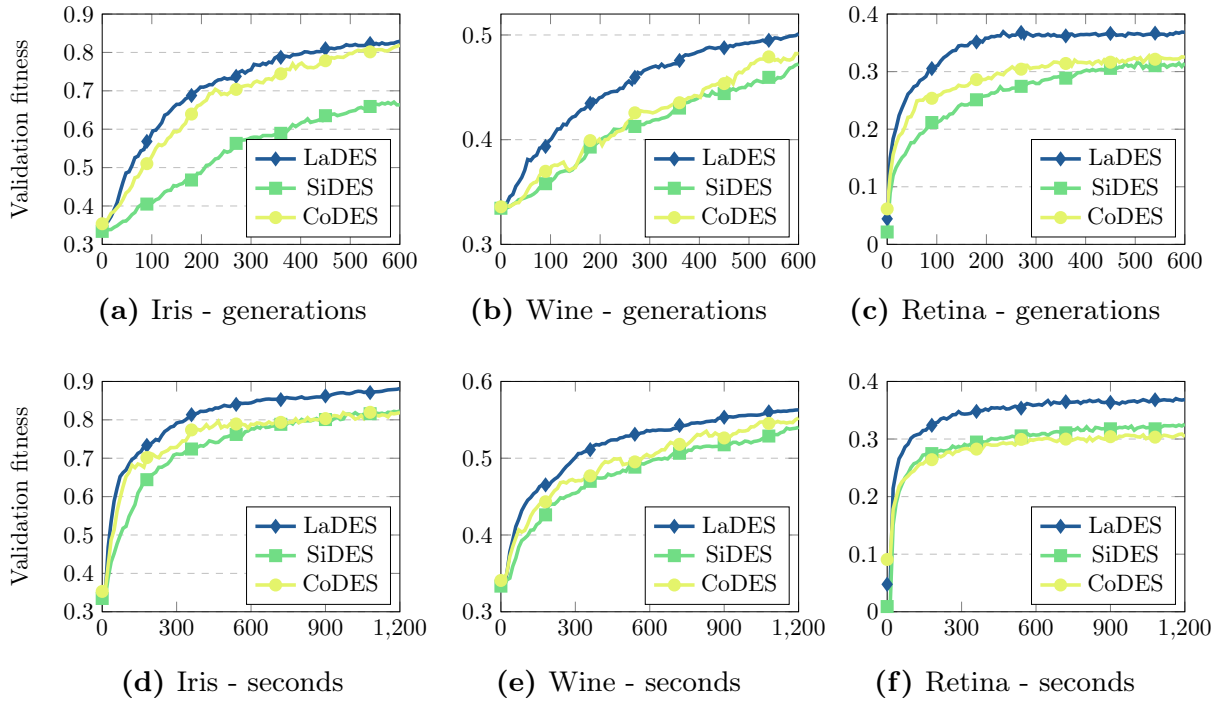
Dataset	Method	Fitness				Accuracy			
		Generations		Time		Generations		Time	
		Mean	SD	Mean	SD	Mean	SD	Mean	SD
Iris	LaDES	<b>0.829</b>	0.089	<b>0.882</b>	0.035	<b>0.911</b>	0.099	<b>0.940</b>	0.020
	SiDES	0.663	0.243	0.823	0.162	0.691	0.287	0.875	0.205
	CoDES	0.819	0.170	0.819	0.136	0.889	0.164	0.886	0.159
Wine	LaDES	<b>0.501</b>	0.092	<b>0.563</b>	0.097	<b>0.627</b>	0.136	<b>0.698</b>	0.112
	SiDES	0.472	0.150	0.540	0.127	0.546	0.233	0.686	0.181
	CoDES	0.482	0.145	0.552	0.136	0.567	0.194	0.697	0.176
Retina	LaDES	<b>0.368</b>	0.027	<b>0.369</b>	0.026	<b>0.702</b>	0.038	<b>0.701</b>	0.034
	SiDES	0.315	0.090	0.326	0.062	0.675	0.060	0.696	0.036
	CoDES	0.325	0.037	0.304	0.051	0.696	0.037	0.694	0.033

Table 5.9: Experiment 1: Performance results.

accuracy does not measure how correct a classification is, only that the class with the highest predicted probability is the correct class. Therefore, fitness will be used for all comparisons.

Table 5.9 show that CoDES does not generally perform worse than the other two methods. Its fitness is often better than or similar to SiDES. There is no significant ( $p > 0.5$ ) difference between the two in Wine, and with time as stopping criteria in Iris. Additionally, CoDES perform significantly ( $p < 0.05$ ) better than SiDES when comparing generations in Iris, with fitness 0.819 compared to 0.663. It indicates that hypothesis 1 is incorrect. The graphs in Figure 5.2 confirm that CoDES and SiDES perform similarly throughout evolution as well, and not only the end result is similar. Their lines even intersect in Figure 5.2b and Figure 5.2d.

CoDES and LaDES both use a unique CPPN for each layout element, opposed to the combined CPPN in SiDES. Comparing the results of these two methods can therefore give insight into whether a separate population of reusable CPPNs or unique CPPNs should be used. A major difference between CoDES and LaDES is that the entire CPPN population is initialized at the beginning in CoDES, and new individuals are never created. However, in LaDES, a new individual is created each time a new substrate or path element emerges. This difference might enable LaDES to improve continuously by creating and optimizing new CPPNs. Unlike CoDES, which may only utilize the CPPNs it has already evolved and is unable to adapt these further. If this were true, they should have comparable fitness improvement in early generations, but LaDES pull ahead while CoDES converges earlier. This is in fact the case in Figure 5.2c, 5.2d, and 5.2f. They initially improve at a comparable rate, but CoDES then converges earlier. It is not able to



**Figure 5.2: Experiment 1: Performance results charts.**

reach the performance of LaDES, evident by the significant ( $p < 0.01$ ) difference between their respective final fitnesses in these three cases. It is reasonable to believe that LaDES is better able to improve continuously due to the newly instantiated CPPNs throughout evolution. Although, it might also be because it is difficult to fine-tune networks when the combination of CPPNs differs between generations in CoDES. Each species will internally have some variation. It will cause some variation between networks assembled in two subsequent generations even though the layout is unchanged. Regardless, it is concluded that a separate population, enabling reuse of CPPNs and the ability to use existing CPPNs in new elements, is not beneficial compared to uniquely allocated CPPNs in each layout element.

Figure 5.2 show that SiDES performs worse than LaDES, as hypothesis 2 suggests. Its fitness is below LaDES in all six cases. It is however comparably worse over 600 generations (a) than 1200 seconds (d) at the Iris dataset. It suggests that its runtime is not entirely the issue, as it can compete when limiting runtime. Hypothesis 2's conclusion is seemingly correct, although its argument is incorrect. The number of generations each algorithm can perform per second, presented in Table 5.10, further disproves hypothesis 2. These results do show that the SiDES algorithm is slower than LaDES on all three datasets. The difference is significant

Dataset	Method	Generations per second	
		Mean	SD
Iris	LaDES	1.538	0.681
	SiDES	1.437	1.828
	CoDES	0.692	0.371
Wine	LaDES	2.454	1.461
	SiDES	1.949	2.041
	CoDES	0.916	0.478
Retina	LaDES	0.713	0.318
	SiDES	0.325	0.134
	CoDES	0.335	0.130
MEAN	LaDES	1.568	0.820
	SiDES	1.237	1.334
	CoDES	0.648	0.326

**Table 5.10: Experiment 1: Execution speed results.** Measured over 1200 seconds.

( $p < 0.05$ ) in both Wine and Retina, although not Iris ( $p = 0.606$ ). However, the high standard deviation for SiDES on Iris and Wine suggests that the CPPN extraction is not the issue. The CPPN extraction operation uses a similar amount of time each time, so it should not cause such high deviations. It might rather be the node search within patterns that causes SiDES to be slower than LaDES. If the CPPN weight outputs in SiDES were to produce more complex patterns, these would require more time to be searched. Additionally, more complex patterns lead to more complex networks, which are more computational heavy to execute.

The complexities of the assembled networks are presented in Table 5.11. Derived from the table, the average number of nodes per hidden substrate is 21.48 in LaDES, 33.12 in SiDES, and 22.75 in CoDES. These results somewhat correlate with hypothesis 3, but not entirely. The number of nodes in each hidden substrate is similar between LaDES and CoDES, indicating that the complexity of patterns produced in both methods is similar. Although this similarity correlates with hypothesis 3, the two methods differ in the number of substrates. LaDES has an average of 2.23 hidden substrate, while CoDES has 4.98. Even though the networks within each substrate are similar in complexity, the increased number of substrates in CoDES results in its network as a whole being more complex than in LaDES'. As hypothesis 3 states, the CPPNs in CoDES and LaDES produce patterns of equal complexity, and of lower complexity than the combined CPPN

Dataset	Method	Nodes		Edges		Hidden Substrates	
		Mean	SD	Mean	SD	Mean	SD
Iris	LaDES	53.40	54.70	234.1	410.6	2.280	1.096
	SiDES	38.18	52.47	116.1	181.5	1.320	1.287
	CoDES	112.4	75.74	687.2	626.9	5.100	2.476
Wine	LaDES	51.00	43.03	196.0	235.1	2.040	1.019
	SiDES	47.54	73.72	186.1	425.6	1.400	1.442
	CoDES	67.22	72.31	397.0	735.9	3.180	2.463
Retina	LaDES	39.50	47.64	150.4	383.5	2.380	1.294
	SiDES	92.46	117.7	322.0	489.4	2.660	1.728
	CoDES	160.2	114.9	1235	1404	6.660	3.011
MEAN	LaDES	47.97	48.46	193.5	343.1	2.233	1.137
	SiDES	59.39	81.29	208.0	365.5	1.793	1.486
	CoDES	113.3	87.65	773.1	922.3	4.980	2.650

**Table 5.11: Experiment 1: Network complexity results.** Measured after 600 generations.

in SiDES. However, because CoDES use more substrates, the network complexity is higher, and hypothesis 3 therefore incorrect.

CoDES might use more hidden substrates because the CPPNs can be reused. When a substrate is assigned CPPN  $a$ , and the path connecting it to another substrate assigned substrate  $b$ , these CPPNs can be reused. CPPN  $b$  might be used in multiple outgoing paths from the same substrate, causing many identical connections. The same CPPN,  $a$ , might also be used to assemble networks in some of these target substrates. It can potentially cause chains with multiple substrates and paths, where substrates are developed by the same CPPN  $a$  and paths by the same CPPN  $b$ . It is unknown whether many identical outgoing paths cause the increased number of substrates or if these chains emerge. It might also be caused by something else entirely. What is certain is that CoDES produce more complex networks than the other two methods, shown by the number of edges in Table 5.11. CoDES produce significantly ( $p < 0.05$ ) more edges than the other two on all three datasets. The networks produced by CoDES are therefore more computationally expensive to execute.

When comparing the performance during 600 generations in Figure 5.2, SiDES performs worse than the other two. Its fitness lies below the other two in all generations with the Iris (a) and Retina (c) dataset, and most of the generations in Wine (b). SiDES also reach a final fitness below the other two in all three datasets.

The difference is significant ( $p < 0.01$ ) on Iris, though not on Wine ( $p = 0.632$ ) and Retina ( $p = 0.306$ ). This comparison is with equally many generations and not dependent on time, meaning the number of generations performed per second does not matter in this comparison. It seems that the single CPPN limits its performance, as proposed in hypothesis 4. Figure 5.2a and Figure 5.2c show that SiDES improves slower and converges at a lower fitness than the other two. The difference between it and LaDES is that SiDES uses a combined CPPN, they are otherwise identical. It is therefore concluded that hypothesis 4 is correct, and that individual CPPNs are better than one large combined CPPN.

### Conclusion

It is concluded that a separate population with CPPNs is not beneficial, layouts and CPPNs should instead be part of the same individual. The evolved networks are then less complex, and the fitness converges to a higher value. Additionally, multiple CPPNs are superior to a single one, so a unique CPPN should be created for each layout element. The fitness achieved with a single CPPN increases slower than with multiple CPPNs, and it also converges to a lower fitness result. Based on these conclusions, Layered DES-HyperNEAT is selected as the implementation moving forward. The other two implementations, SiDES and CoDES, are not considered further.

## 5.6 Phase 2: Identity Mapping

In the NEAT algorithm, described in section 2.5, an identity function is created whenever a node is inserted into a network. Nodes are inserted into existing links, in-between two existing nodes. The link it is inserted into is disabled, and two new are created. The weight of one of the new links is set to the weight value of the one that was disabled, while the other is set to 1. The link with weight 1 functions as an identity mapping, sending out the same value it receives when the network is executed. Thus, the functionality of the network is not changed even though a new node is inserted. Likewise, an identity mapping is created between substrates in DES-HyperNEAT, where nodes in one substrate is connected to nodes at the same position in another. This is described in section 4.2. Three modifications to the node search from ES-HyperNEAT were made to make the identity mapping between substrates possible. These three modifications are evaluated in this phase.



## Experiment 2

method	[ES-HyperNEAT, DES-HyperNEAT]
dataset	[Iris, Wine, Retina]
stop criterion	300 generations
repeats	50
variance threshold	[0.03, 0.15, 0.3]
division threshold	[0.03, 0.15, 0.3]
max difference	[false, true]
normalized search	[false, true]
all nodes	[false, true]

Table 5.12: Experiment 2: Node search modifications.

### 5.6.1 Experiment 2: Node Search Modifications

The modifications are tested for both ES-HyperNEAT and the selected DES-HyperNEAT implementation. Details for the experiment is presented in Table 5.12. In addition to the threshold value 0.03, used in ES-HyperNEAT [Risi and Stanley, 2012], larger division and variance threshold values are tested. It is because the different modifications might benefit other threshold values.

Three modifications are proposed in section 4.2: Normalizing the searched pattern, using the max weight difference instead of variation, and calculating variation between all nodes and not only leaf nodes. They are named *norm.*, max difference and *all nodes* within this experiment. When all are enabled, the addition of a new substrate no longer disrupts the existing network output. It allows for an identity function to be inserted when increasing depth.

**Hypothesis 1** The modification that normalizes the searched pattern will improve the performance of ES-HyperNEAT. It is because the CPPN outputs are normalized when searching the pattern. Emphasis is therefore put on the patter variation itself rather than how substantial the variations are.

**Hypothesis 2** The three modifications to the node search will improve DES-HyperNEAT, as they together enable the identity mapping between substrates.

Table 5.13 presents the validation fitness for ES-HyperNEAT, with each combination of the three modifications. The line above indicates that a feature is disabled. The original method uses the combination [norm., max difference, all nodes]. Each combination of the modifications has been evaluated with all nine combinations of threshold values. In the table, only the best fitness is displayed.

Dataset	max difference				max difference				
	all nodes		all nodes		all nodes		all nodes		
	<b>Fitness</b>		<b>Fitness</b>		<b>Fitness</b>		<b>Fitness</b>		
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	
norm.	Iris	<b>0.854</b>	0.049	0.840	0.049	0.842	0.049	0.834	0.047
	Wine	0.450	0.032	0.463	0.047	<b>0.468</b>	0.051	0.460	0.044
	Retina	<b>0.308</b>	0.068	0.304	0.043	<b>0.308</b>	0.056	0.307	0.053
	MEAN	0.537	0.045	0.536	0.039	<b>0.539</b>	0.042	0.533	0.040
norm.	Iris	0.850	0.056	0.837	0.050	0.821	0.060	0.830	0.054
	Wine	0.462	0.047	<b>0.468</b>	0.053	0.467	0.036	0.462	0.039
	Retina	0.290	0.045	0.292	0.057	0.301	0.054	0.301	0.052
	MEAN	0.534	0.046	0.532	0.046	0.530	0.049	0.531	0.042

**Table 5.13: Experiment 2: ES-HyperNEAT performance results.**

The fitness of each combination of the modification is thus presented with its optimal threshold values. The results within each cell in Table 5.13 thus share division and variance thresholds, but the threshold values may differ between cells.

The results in Table 5.13 indicate that some of the modifications can be beneficial. The combination of norm. and all nodes scores well, with mean fitness of 0.537 and 0.539 (with and without max difference). The combination where all modifications are enabled (top left in the table) has the highest fitness on both Iris and Retina, 0.854 and 0.308. It is significantly ( $p < 0.01$ ) better than with no modifications (bottom right) on Iris, but also significantly ( $p < 0.05$ ) worse on Wine. The combination [norm., max fitness, all nodes] achieves the highest fitness on both Wine and Retina, and has the best mean fitness on all datasets, 0.539. The difference between mean fitness in the best and worst combination is though only 0.009. There is seemingly no combination that is significantly better in all three datasets. It might be that the CPPNs adapt their output to the search method and produce patterns that work well for all the modifications. However slightly, hypothesis 1 seems to be correct. The mean fitnesses of the four combinations with normalized search enabled are respectively higher than the corresponding combinations without the normalized search. The standard deviation is though high compared to the mean difference, so there is some uncertainty.

Table 5.14 presents the same information as Table 5.13, for the method DES-HyperNEAT. The combination that enables the identity mapping, where all are enabled, has a mean validation fitness of 0.581 across the three datasets. It is not the best, but also not the worst. The original search method, with

Dataset	max difference				max difference				
	all nodes		all nodes		all nodes		all nodes		
	<b>Fitness</b>		<b>Fitness</b>		<b>Fitness</b>		<b>Fitness</b>		
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	
norm.	Iris	0.897	0.026	0.893	0.032	0.896	0.025	0.897	0.023
	Wine	<b>0.497</b>	0.060	0.471	0.059	0.491	0.028	0.493	0.027
	Retina	0.350	0.029	<b>0.367</b>	0.034	0.356	0.028	0.347	0.025
	MEAN	0.581	0.039	0.577	0.031	0.581	0.029	0.579	0.025
norm.	Iris	0.889	0.028	0.896	0.024	0.897	0.025	<b>0.902</b>	0.024
	Wine	0.491	0.067	0.490	0.021	0.493	0.049	0.491	0.052
	Retina	0.355	0.033	0.354	0.032	0.347	0.032	0.356	0.036
	MEAN	0.578	0.034	0.580	0.026	0.579	0.035	<b>0.583</b>	0.034

Table 5.14: Experiment 2: DES-HyperNEAT performance results.

none of the modifications enabled, yields the highest mean fitness, 0.583. The difference between with and without the three modifications is however insignificant ( $p = 0.670$ ) across all three datasets. Hypothesis 2 is therefore likely incorrect. The modifications made to enable an identity mapping does not improve the DES-HyperNEAT method.

## Conclusion

The results indicate that the modifications made to enable an identity mapping do not improve the performance. DES-HyperNEAT can evolve the substrate topology equally well without them. It does so even though the addition of a new substrate may disrupt the network output. It might be that speciation within the NEAT algorithm protects the new innovations. When a substrate is inserted into an existing layout, it might no longer be part of the same species. If so, the identity mapping would not be as essential. Regardless, none of the modifications to the node search outlined in section 4.2 are used in DES-HyperNEAT.

### 5.6.2 Experiment 3: Identity Mapping

Since the modifications made so that an identity function can be inserted are disabled, the manually constructed CPPN output  $Gaussian(7.5 \cdot (7.5x_1 - 7.5x_2)^2 + 7.5 \cdot (7.5y_1 - 7.5y_2)^2)$  does not create an identity function. This CPPN output was originally manually constructed and assigned to the path of a newly inserted substrate. Together with the modifications tested in the previous experiments, the function produces an identity mapping between substrates. Since the modifications

## Experiment 3

method	DES-HyperNEAT
dataset	[Iris, Wine, Retina]
stop criterion	800 seconds
repeats	100
identity function	[True, False]

Table 5.15: Experiment 3: Identity mapping.

Dataset	identity function		no initialization	
	Fitness		Fitness	
	Mean	SD	Mean	SD
Iris	0.906	0.021	<b>0.912</b>	0.014
Wine	0.549	0.087	<b>0.574</b>	0.105
Retina	<b>0.361</b>	0.027	0.358	0.022
MEAN	0.605	0.045	<b>0.615</b>	0.048

Table 5.16: Experiment 3: Performance results.

are disabled, the Gaussian function should no longer be needed as the network output is disrupted anyway. It is tested to initialize the CPPN output without a pre-determined structure instead. The experiment outlined in Table 5.15.

The results of the experiment are presented in Table 5.16. When the CPPN assigned to the link that is supposed to be an identity mapping is uninitialized, DES-HyperNEATs performance is improved on both Iris and Wine. The increase from 0.906 to 0.912 on Iris is significant ( $p = 0.019$ ), though the increase from 0.549 to 0.574 on Wine ( $p = 0.068$ ) is not. The decrease from 0.361 to 0.358 in Retina is also insignificant ( $p = 0.390$ ). In fact, the performance with no initialization is significantly ( $p < 0.05$ ) higher when comparing the two on all three datasets. When inserting a substrate into the layout, the CPPN associated with the identity link should therefore not be initialized in any way. As with any other new CPPN, no connections should be manually created. This is likely because evolution has more freedom when it is not restricted by pre-determined nodes and connections in the CPPN.

## Conclusion

Results show that DES-HyperNEAT does not need the identity pattern produced by the manually constructed CPPN, when all the modifications to the node search are disabled. DES-HyperNEAT is modified based on the results within this phase. Special measures are no longer taken to ensure an identity mapping is created when inserting new substrates. An uninitialized CPPN, without any connections, is created for both the new inbound and outgoing path when a substrate is inserted.

## 5.7 Phase 3: Layered DES-HyperNEAT Tuning

The following section presents experiments conducted to tune the DES-HyperNEAT framework. Specifically how substrate depth should be controlled in I/O and hidden substrates, and how to determine I/O configurations. The entire phase is essentially a parameter search, as the configurations in a way are parameters to the framework. They are though more impactful to the execution of the framework than other hyperparameters. Hyperparameters found in the preliminary testing, in section 5.2, are used until they are updated in a new parameter search at the end of the phase.

### Experiment 4

method	DES-HyperNEAT
dataset	[Iris, Wine, Retina]
stop criterion	300 seconds
repeats	100
input substrate depth	[0, evolved]
output substrate depth	[0, evolved]
max input substrate depth	5
max output substrate depth	5

**Table 5.17: Experiment 4: I/O substrate depth.**

### 5.7.1 Experiment 4: I/O Substrate Depth

As described in subsection 4.1.1, I/O substrates are those manually defined in the I/O configuration, while hidden substrates are added by evolution. Experiment 4 tests whether the depth in I/O substrates should be forced to zero or if the depth should evolve as in hidden substrates. As a single connection has depth one, while two connections in a chain has depth two. Therefore, no connections may be

		Dataset	input substr. depth			
			0		evolved	
			<b>Fitness</b>		<b>Fitness</b>	
output substr. depth	0	Mean	SD	Mean	SD	
		Iris	<b>0.819</b>	0.106	0.765	0.109
		Wine	<b>0.484</b>	0.079	0.457	0.089
		Retina	0.350	0.044	0.341	0.051
	MEAN	<b>0.551</b>	0.076	0.521	0.083	
	evolved	Iris	0.785	0.108	0.748	0.106
		Wine	0.481	0.078	0.475	0.076
		Retina	<b>0.357</b>	0.041	0.340	0.050
MEAN		0.541	0.076	0.521	0.077	

Table 5.18: Experiment 4: Performance results.

created when the depth is zero. Thus, when a substrate has zero depth, the nodes within it may only be connected to nodes in other substrates. No connections are assembled between nodes within it.

When the depth in I/O substrates are nonzero, networks may be assembled in them. The network within an input substrate may then perform pre-processing of the input data before it is distributed to hidden substrates. The same logic applies for output substrates and post-processing of the values from the hidden substrates. Details about experiment 4 are presented in Table 5.17. The default I/O configurations for DES-HyperNEAT are used in this experiment, illustrated in Table 5.7. These configurations have available space within the input and output substrates, for potential networks to be assembled within them. They also only use a single input substrate, meaning a potential pre-processing can involve all inputs and greater affect the evolved network.

**Hypothesis 1** It is beneficial to force the depth to be zero in input substrates, so that the potentially bad pre-processing within input substrates is avoided. Instead, the input values are directly passed on to hidden substrates.

The results of experiment 4 are presented in Table 5.18. The fitness achieved by all four combinations of zero and evolved depth are presented for each dataset. The best mean performance across all datasets is 0.551, with zero depth in both input and output substrates. In the Iris dataset, the same configuration performs the best, yielding a fitness of 0.819. In Iris, it is significantly ( $p < 0.05$ ) higher than any of the three other combinations, with fitness 0.765, 0.785, and 0.748. The configuration also yields the best results in Wine, with fitness 0.484. Though,

it is not significantly ( $p = 0.787$ ) higher than the next best configuration. The next best is with zero input depth and evolved output depth, achieving fitness 0.481. The Retina dataset favors use of evolved output substrate depth, instead of setting it to zero. The validation fitness is then 0.357. It is though not significantly ( $p = 0.246$ ) higher than the combination where both input and output substrates have zero depth.

## Conclusion

The configuration with depth zero in both input and output substrates performs the best in two of the three datasets, Iris and Wine. It is significantly better than the other depth combinations in Iris and not significantly worse in Retina. It is therefore concluded that hypothesis 1 is correct, and the depth in both input and output substrates should be zero. It enables hidden substrates to receive the input directly, without it being pre-processed in input substrates.

### 5.7.2 Experiment 5: Hidden Substrate Depth

The weight patterns produced by the CPPNs determine the depth of the network assembled in a substrate. The iterative algorithm that assembles networks extend the network's depth until no more nodes are discovered when searching the pattern. However, to avoid overly complex networks, a depth limit is used. It limits the maximum number of iterations performed when assembling networks. ES-HyperNEAT uses a static depth limit. Since DES-HyperNEAT use multiple substrates it is proposed to evolve such a depth-limit value in each substrate, limiting them separately. Experiment 5.1 is conducted to investigate if the depth limit in substrates should be evolved, of it it should be pre-determined and static. Details about the experiment are presented in Table 5.19. In addition to evolving the depths separately in each substrate, it tests a static depth in all of them.

Experiment 5.1	
method	DES-HyperNEAT
dataset	[Iris, Wine, Retina]
stop criterion	[300 seconds, 200 generations]
repeats	100
depth	[evolved, 0, 1, 2, 3, 4, 5]
max substrate depth	5

**Table 5.19: Experiment 5.1: Hidden substrate depth.**

When the depth-limit is evolved in each substrate, it is evolved in the range 0-5. Therefore, static depths in the range 0-5 are tested.

**Hypothesis 2** The difference in performance will be minor between nonzero static depth and evolved substrate depth. The evolved depth is however though to be beneficial when using time as stopping criteria. It allows the algorithm to adapt the of networks within the substrates dynamically.

**Hypothesis 3** When the depth is zero, the number of generation performed per second will be significantly higher, as it alleviates network assembling within substrates. However, depth zero will not perform better when using time as stopping criteria, as networks are no longer assembled in substrates.

### Nonzero depth

Table 5.20 presents the results of experiment 5.1. Evolved and nonzero static depths will first be compared. Zero depth will then be elaborated upon. As seen in Table 5.20a, when using number of generations as stopping criteria, the depth 4 in Iris and Retina, and 5 in Wine yield the highest validation fitness. These fitness values are 0.772 in Iris, 0.434 in Wine, and 0.363 in Retina. Since a high depth value produce the highest fitness in all three datasets, there could be a correlation between the depth and fitness. The difference between the depth one and four in Iris is significant ( $p < 0.05$ ). Although, there is no significant difference in Wine and Retina ( $p = 0.870$ ,  $p = 0.468$ , respectively), when comparing the best depth in the generation column to depth one. In the generations column, there is also no significant ( $p = 0.124$ ,  $p = 0.371$ ,  $p = 0.108$ , respectively) difference between the best performing static depths and the evolved depth in the three datasets. Therefore, when using number of generations as stopping criteria, it cannot certainly be determined whether deeper networks are beneficial or that evolved or static depths are best. Hypothesis 2 is correct in that the difference between nonzero static and dynamic depth is minor.

When the depth is evolved, both fitness in the generation column in Table 5.20a and execution speed in Table 5.20b are similar those with static depth either 1 or 2. It indicates that the evolved depths might often be in the lower part of the 1 to 5 range. However, the results in the time column in indicate that evolved depth might be better that a nonzero static depth. In the time column, the evolved depth yield the highest fitness of the nonzero depths in both Wine and Retina. However, there is no significant difference between evolved and the best nonzero static depths in any of the three datasets ( $p = 483$ ,  $p = 0.449$ ,  $= 0.225$ , respectively). The difference is minor between static nonzero and evolved depth, as hypothesis 2 suggests. However, no significant difference can support its statement that evolved depth is beneficial when using time as stopping criteria.



D.set	Depth	Fitness				Generations per second	
		Generations		Time		Mean	SD
		Mean	SD	Mean	SD		
Iris	evolved	0.742	0.154	0.805	0.115	1.601	1.834
	0	0.727	0.185	<b>0.876</b>	0.060	2.354	1.502
	1	0.767	0.139	0.816	0.106	1.984	2.155
	2	0.758	0.151	0.793	0.112	1.227	0.995
	3	0.746	0.166	0.772	0.129	1.072	1.366
	4	<b>0.772</b>	0.118	0.763	0.136	0.970	0.896
	5	0.738	0.156	0.781	0.133	1.070	0.984
Wine	evolved	0.423	0.092	0.500	0.104	2.838	2.602
	0	0.432	0.091	<b>0.513</b>	0.089	3.805	3.016
	1	0.431	0.091	0.490	0.081	2.195	1.896
	2	0.425	0.078	0.476	0.094	2.131	2.221
	3	0.424	0.084	0.464	0.088	2.074	2.325
	4	0.417	0.079	0.483	0.093	2.031	2.460
	5	<b>0.434</b>	0.081	0.447	0.077	1.881	2.519
Retina	evolved	0.355	0.037	0.348	0.047	0.678	0.561
	0	0.359	0.044	<b>0.362</b>	0.034	1.111	0.607
	1	0.356	0.034	0.339	0.057	0.657	0.392
	2	0.353	0.044	0.330	0.063	0.429	0.260
	3	0.357	0.038	0.337	0.061	0.492	0.323
	4	<b>0.363</b>	0.033	0.343	0.058	0.439	0.358
	5	0.356	0.035	0.328	0.057	0.471	0.326

(a) Performance

(b) Execution speed

**Table 5.20: Experiment 5.1: Performance and execution speed results.** Execution speed (b) measured over 1200 seconds.

When comparing the execution speed achieved with static depths in the range 1 - 5 in Table 5.20b, the number of generations per second is generally higher for lower depths. There seems to be an inverse relationship between the two. Iris runs 1.984 generations per second with depth 1, compared to 1.070 with depth 5. It is a significant difference ( $p < 0.01$ ) between them. The difference between depth 1 and 5 is also significant ( $p < 0.01$ ) in Retina, although not in Wine ( $p = 0.321$ ). In addition to the difference between the endpoints, most of the results with depths 1-5 are strictly decreasing in Table 5.20b. When the depth in substrates is increased, more time is used to assemble them, resulting in fewer generations per second. The relationship between the two are therefore logical.

There also seems to be an inverse relationship between depth and fitness in the time column in Table 5.20a. The results indicate that higher depths run fewer generations per second and achieves lower fitness. When comparing depth 1

to 5 across all datasets, there is a significant ( $p < 0.05$ ) difference between the fitnesses. Even though higher depths achieve higher fitness in the generation column, the extra time used when the depth is higher results in them performing worse when compared to the time they use. Lower depths are thus more efficient, which supports hypothesis 2. It should be beneficial to evolve the depth so that they limit unnecessary complexity dynamically.

### Zero depth

The zero-depth configuration is unique. All nonzero depth values results in assembling networks within substrates. However, when the depth is zero, no networks are created within substrates. The substrates are therefore not assembled, and connections may only form between substrates. Zero depth greatly highlights the trade-off between complexity and run-time in Table 5.20. It saves much time per generation, evident by the higher execution speed in Table 5.20b.

The results clearly show that the algorithm is able to run faster when substrates are not assembled. It runs 2.354 generations per second in Iris, 3.805 in Wine, and 1.111 in Retina. When comparing depth zero to one, on all three datasets, depth zero results in significantly ( $p < 0.05$ ) more generations per second. The question is then if it is worth it to run more generations when it results in only creating connections between substrates and not assembling networks within them. When comparing against the same number of generations in Table 5.20b, higher depths were advantageous. However, depth zero performs the best when using time as stopping criteria. The time column shows zero depth achieves fitness 0.876 in Iris, 0.513 in Wine, and 0.362 in Retina. All datasets combined, the depth zero is significantly ( $p < 0.05$ ) better than any other depth configuration in the time column. Hypothesis 3 is therefore incorrect. Although networks are no longer created inside each substrate, zero depth is superior to both other static depths and evolved depth.

The experiment comparing the depth values was only run for 200 generations and 300 seconds, separately. Zero depth was superior to any other when using 200 seconds as stopping criteria. However, it might converge to a lower fitness than the others, as connections are only created between and not within substrates. A second experiment is designed to investigate the convergence of zero and nonzero depth. It is described in Table 5.21. Evolved and zero depth is compared over 800 seconds, to see if the fitness achieved with the two depth configurations converge differently.

## Experiment 5.2

method	DES-HyperNEAT
dataset	[Iris, Wine, Retina]
stop criterion	800 seconds
repeats	100
depth	[evolved, 0]

Table 5.21: Experiment 5.2: Hidden substrate depth - Part 2.

Figure 5.3 presents the fitness achieved with zero and nonzero depth. Although zero is higher, the two converge somewhat similarly in Iris (a) and Retina (c). However in Wine (b), zero depth maintains a stable fitness increase longer than nonzero depth. On Wine, the resulting fitness is 0.624 compared to 0.560, presented in Table 5.22a. The difference between final fitnesses values is significant ( $p < 0.01$ ) in all three datasets. It is therefore concluded that zero depth is superior to nonzero depth when comparing time.

As networks are not assembled in substrates, the total complexity of networks is also lower with zero depth. In Table 5.22b, the number of nodes and edges is significantly ( $p < 0.05$ ) lower with zero depth in Iris. Zero depth results in an average of 63.47 nodes and 435.9 edges, while nonzero depth results in 90.36 nodes and 902.9 edges. The mean values show the number of nodes and edges are also lower with zero depth in Wine and Retina. However, the difference in those datasets are not as significant (nodes:  $p = 0.747$  and  $p = 0.313$ , edges:  $p = 0.826$  and  $p = 0.971$ , respectively). Thus, compared to nonzero depth, the fitness is generally higher and the network complexity lower or equal with zero depth.

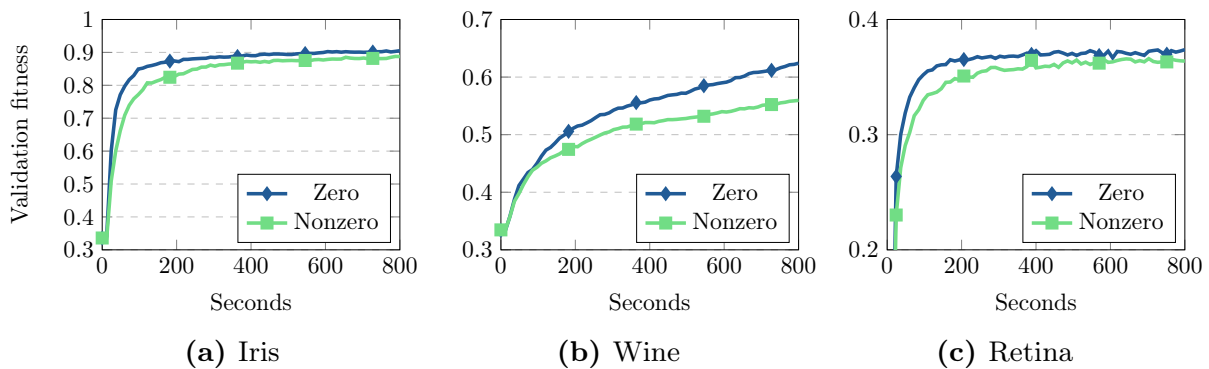


Figure 5.3: Experiment 5.2: Performance results charts.

Dataset	Depth	Fitness		Nodes		Edges	
		Mean	SD	Mean	SD	Mean	SD
Iris	Nonzero	0.887	0.043	90.36	109.6	902.9	2046
	Zero	<b>0.905</b>	0.035	63.47	68.62	435.9	776.8
Wine	Nonzero	0.560	0.100	70.09	70.22	437.6	621.1
	Zero	<b>0.624</b>	0.122	66.91	69.21	458.6	724.5
Retina	Nonzero	0.364	0.023	42.39	37.26	182.2	206.7
	Zero	<b>0.374</b>	0.021	36.23	48.23	184.2	512.4

(a) Performance

(b) Network complexity

**Table 5.22: Experiment 5.2: Performance and network complexity results.**

## Conclusion

Results show that substrates in DES-HyperNEAT should have zero depth, and thus not be assembled. Compared to nonzero depth, zero depth results in more generations per second. Additionally, the fitness improvement per time unit is greater with zero depth, and it also converges to a higher fitness than evolved depth. Therefore, it is suggested to be the default depth configuration in DES-HyperNEAT, and will be used in the next experiments. By setting the depth to zero, DES-HyperNEAT is no longer a collection of ES-HyperNEAT instances. Nodes are created in substrates when assembling paths, but the nodes do not connect to other nodes in the same substrate. As networks are no longer assembled in substrates, substrates do not need an assigned CPPN. It makes DES-HyperNEAT similar to Deep HyperNEAT [Sosa and Stanley, 2018], with the main difference being that nodes in hidden substrates have static positions in Deep HyperNEAT while they are dynamically positioned using the node search in DES-HyperNEAT. Additionally, Deep HyperNEAT uses a single CPPN with multiple outputs, like Single CPPN DES-HyperNEAT, while the selected DES-HyperNEAT implementation, Layered, uses a separate CPPN for each path in the layout.

### 5.7.3 Experiment 6: I/O Configuration

Experiment 6 is conducted to determine how inputs and outputs should be configured into one or more substrates. As there are two to three outputs in all datasets, output configurations are not tested. The three datasets vary in their number of inputs and the relationship between them. Inputs thus allow for more extensive evaluation. The knowledge learned from inputs can likely be applied to outputs as well.

Tested I/O configurations are presented in Table 5.23. The same configuration scheme is tested in multiple datasets. *Line* is the most basic, with all nodes positioned at in a line at  $y = 0$ , in a single substrate. *Lines*, *grids* and *rotated grids* distribute the inputs among multiple substrates. Rotated versions of the grids are included to see if the relationships between positions are important. When rotated, the nodes do not share  $x$  or  $y$  position, and are additionally not in the same diagonal. All datasets are also tested with inputs distributed among unique substrates, in the *individual* configuration.

In an I/O configuration, all inputs can potentially be crammed into a single substrate, or separated so that each input is in a unique substrate. If related inputs are placed together, these will likely be connected to the same parts of the assembled network. The network can thus learn the relationship between related inputs. Inputs that are not related can be placed in different substrates, as they then can be connected to the appropriate parts of the assembled network. Too many inputs in a single substrate is likely not be beneficial because of the increased CPPN complexity required to distinguish them. Contrary, too many substrates may also make the layout unnecessarily complex.

**Hypothesis 4** Related inputs should be placed together in a substrate, while independent inputs should be placed in different substrates.

The Iris dataset has four inputs: sepal length, sepal width, petal length, and petal width. All are measured in centimeters. The sepal and petal are two distinct parts of a flower. The Iris dataset contains three iris species, which need to be distinguished based on their sepal and petal measurements. The first three input configurations in Table 5.23 position these four measurements in a single substrate, in a *line*, *grid* or *rotated grid*. The configuration *split* separates the sepal and petal measurements in two substrates, and the *individual* configuration places each measurement in a unique substrate.

Table 5.24 contains the validation fitnesses achieved with the different input configurations. In Iris, the *grid* configuration yields the best validation fitness, 0.903. The other two configurations with a single substrate are close behind, *rotated grid* with 0.900, and *line* with 0.893. Configurations with multiple substrates, *split* and *individual*, perform the worst, both with fitness 0.851. There is a significant ( $p < 0.01$ ) difference between the worst performing single substrate configuration, *line*, and either of the configurations with multiple substrates, *split* and *individual*. Thus, the inputs should be placed in a single substrate. These results correlate with hypothesis 4. All measurements in the Iris dataset are of equal type, they are distances in centimeters. Even though the sepal and petal measurements are from different parts of the flower, two of them are widths of two are lengths. Therefore, they are all in a way related, and should according to the results be placed together.

Experiment 6




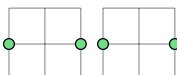
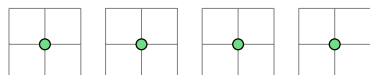


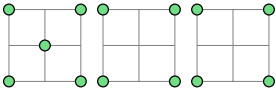
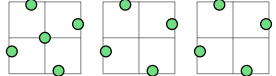
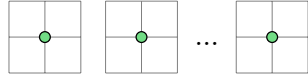


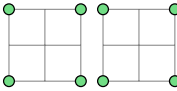
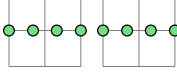
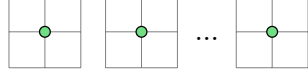
method	DES-HyperNEAT	
dataset	[Iris, Wine, Retina]	
stop criterion	300 seconds	
repeats	100	
substrate depth	0	
<b>Iris</b>	line	
	grid	
input config	rotated grid	
	split	
	individual	
<b>Wine</b>	line	
	single	
input config	grids	
	rotated grids	
	individual	
<b>Retina</b>	line	
	single	
input config	grids	
	lines	
	individual	

Table 5.23: Experiment 6: I/O configuration.

In contrast to the Iris dataset, the attributes in Wine are not related. It contains attributes such as alcohol, acid, ach, magnisium, color intensity, and hue. The inputs differ in what they represent and in their units of measurement. The results in Table 5.24 show that the more substrates these inputs are separated in, the higher the fitness. It seemingly does not matter how the nodes are placed, only the number of substrates used. The difference between *line* at 0.532 and *single* at 0.543 is insignificant ( $p = 0.476$ ). Likewise, the difference between *grids* and *rotated grids*, at 0.687 and 0.694, is also insignificant ( $p = 0.552$ ).

The number of substrates are what differentiates the configurations in Wine. There is a significant difference ( $p < 0.01$ ) between the best performing configuration with a single substrate and the worst performing configuration with multiple substrates. Multiple substrates are thus beneficial. The *individual* configuration separate inputs in unique substrates, resulting in the best fitness, 0.697. However, it is not significantly ( $p = 0.782$ ) higher then with *rotated grids* at 0.694. The conclusion is thus that the number of substrates significantly impact the performance when the inputs are unrelated. Also, inputs' positions within substrates does not significantly impact performance in Wine.

In Iris and Wine, the relationship between fitness and number of substrates is opposite. Inputs grouped in a single substrate performs better than separated

Dataset	Input conf.	Fitness	
		Mean	SD
Iris	line	0.893	0.043
	grid	<b>0.903</b>	0.062
	rotated grid	0.900	0.048
	split	0.851	0.085
	individual	0.851	0.048
Wine	line	0.532	0.112
	single	0.543	0.106
	grids	0.687	0.087
	rotated grids	0.694	0.079
	individual	<b>0.697</b>	0.074
Retina	line	0.366	0.028
	single	0.360	0.032
	grids	<b>0.377</b>	0.015
	lines	0.365	0.024
	individual	0.329	0.051

**Table 5.24: Experiment 6: Performance results.**

inputs in Iris. However, the fitness on the Wine dataset increases as the inputs are further separated. Hypothesis 4 matches with the results in both datasets. Inputs in Iris should be placed together because the inputs are related, and apart in Wine because they are unrelated. The two datasets however share that there is no significant difference between different node placements when the number of substrates are equal. The performance is similar when nodes share  $x$  and  $y$  values compared to being placed rotated.

The Retina dataset is unique in its relationship between inputs, there are two unrelated inputs groups. As seen in Figure 5.1, the retina is split into a left and right part. The left and right pattern is classified separately, making them totally independent. The tested input configuration in Table 5.23 utilize the spatial relationship between inputs, where *single* places the inputs as shown in the retina figure, *grids* splits the left and right part in two substrates, and *lines* additionally flattens them in a line. The *individual* configuration additional test total separation of inputs, where the network itself has to learn to distinguish the two input groups.

The results in Table 5.24 show that the *grids* configuration yields the best fitness in Retina, with 0.377 as the mean validation fitness. This is significantly higher ( $p < 0.01$ ) than each of the other configurations for the Retina dataset. The *individual* configuration has a mean validation fitness of 0.329 for the Retina dataset, which is significantly ( $p < 0.01$ ) lower than any other configuration. There is thus an advantage to grouping inputs, they should not be entirely separated. Comparing *single* to *grids*, there is a significant ( $p < 0.01$ ) benefit to separating the left and right part of the retina in two different substrate. However, there is no significant difference ( $p = 0.864$ ) between *line* and *lines*, so not all configuration benefit the separation.

## Conclusion

The results from Iris and Wine, and some of the results in Retina, support hypothesis 4. The inputs should be placed in the same substrate in Iris, as the inputs are all related. In Wine, the inputs all have different types, and should therefore be separated in multiple substrates. The best results are achieved when each input is placed in a unique substrate. The results from the Retina dataset are not as conclusive. Even though the highest fitness is achieved when the left and right part is separated, the separation does not increase fitness for all configurations. The conclusion is therefore that hypothesis 4 is correct, but it is possible to configure I/O such that grouping related inputs do not improve fitness.



### 5.7.4 Experiment 7: Layered DES-HyperNEAT Parameter Search

The experiments conducted up to this point have selected one of the three implementations of the DES-HyperNEAT framework, evaluated the proposed identity mapping between substrates, and determined the best substrate depths and I/O configurations. Layered DES-Hyperneat is selected as the superior implementation, no identity mapping is used and it has been determined to only evolve connections between substrates and not within them. Now that these choices have been made, the hyperparameters will be optimized.

Parameters are optimized as described in subsection 5.1.2. Multiple iterations will optimize the parameters thought to impact performance the most. The best I/O configuration from experiment 6 is used. All the iterations are presented in Table A6, and the resulting values in Table 5.25. These parameters will be used in the next phase.

Experiment 7	
method	Layered DES-HyperNEAT
dataset	[Iris, Wine, Retina]
stop criterion	60 seconds
repeats	50
<b>Final values</b>	
variance threshold	0.5
division threshold	0.05
band pruning threshold	0.0
add substrate prob.	0.025
remove substrate prob.	0.001
add path prob.	0.4
remove path prob.	0.05
add CPPN node prob.	0.025
add CPPN link prob.	0.4
CPPN activation functions	[None, Linear, Step, ReLU, Sigmoid, Exp, Sigmoid, Tanh, Gaussian, OffsetGaussian, Sine, Square, Abs]

**Table 5.25: Experiment 7: Layered DES-HyperNEAT parameter search.** Presents the Layered DES-HyperNEAT parameter search and its final parameter values.

## 5.8 Phase 4: Related Methods Comparison

Multiple neuroevolutionary methods have been created based on the NEAT algorithm. HyperNEAT use a CPPN evolved with NEAT to assign weights to a

separate manually constructed network. ES-HyperNEAT further extends HyperNEAT with evolvable substrates, where the nodes' positions and the connections between them are determined based on the CPPN weight pattern. Additionally, MSS have extended HyperNEAT with multiple substrates, and Deep HyperNEAT further evolves the topology of substrates.

The extensions are often more complex than the methods they build upon. ES-HyperNEAT adds an additional node search step to HyperNEAT. DES-HyperNEAT use the same node search, and additionally evolves a topology of substrates. Experiment 8 investigates whether these more complex methods increase performance, or if the simpler methods work just as well.

### 5.8.1 Experiment 8: Related Methods Comparison

The methods tested in experiment 8, and the datasets they are tested on, are presented in Table 5.26. NEAT, HyperNEAT, ES-HyperNEAT, and DES-HyperNEAT are compared. Both 1200 seconds and 600 iterations will be used as stopping criteria. The results of these tests will thus provide information about the method's performance compared to their execution speed, something that is lacking in the work that introduced HyperNEAT [Stanley et al., 2009] and ES-HyperNEAT [Risi et al., 2010]. Stanley et al. [2009] found that HyperNEAT performs better than NEAT, and Risi and Stanley [2012] that ES-HyperNEAT better than HyperNEAT. The presented results are however only based on specific number of generations, not runtime.

**Hypothesis 1** Each method will perform better than the one it is extending, when comparing performance to generations. The order should thus be NEAT, HyperNEAT, ES-HyperNEAT, and DES-HyperNEAT, with increasing performance.

**Hypothesis 2** The node search in ES-HyperNEAT and DES-HyperNEAT make them so complex that they, compared to NEAT and HyperNEAT, will use too much time in simple problems where solutions can be easily found. When using time as stopping criteria, they will therefore perform worse than NEAT and HyperNEAT in the Iris dataset, but better than them in Wine.

**Hypothesis 3** DES-HyperNEAT will perform better than HyperNEAT and ES-HyperNEAT, as it is a similar method, but able to use multiple substrates and separate CPPNs.

## Experiment 8

method	[NEAT, HyperNEAT, ES-HyperNEAT, DES-HyperNEAT]
dataset	[Iris, Wine, Retina]
stop criterion	[1200 seconds, 600 iterations]
repeats	100

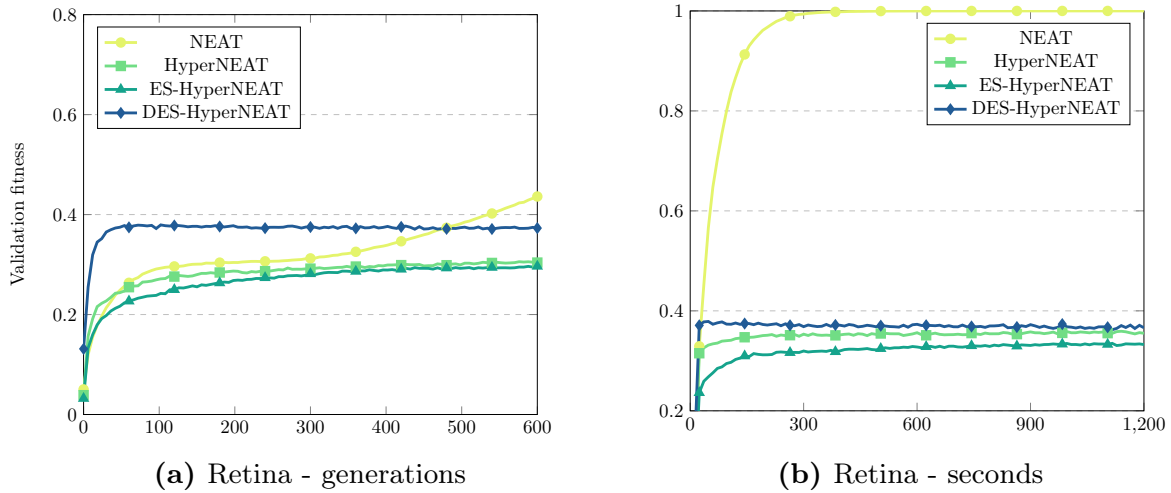
Table 5.26: Experiment 8: Related methods comparison.

**Retina**

The validation fitnesses achieved in Retina are presented in Figure 5.4. Comparing the four methods, it is clear that HyperNEAT, ES-HyperNEAT, and DES-HyperNEAT converge to specific limits, and struggle to learn the domain. NEAT, however, is able to completely distinguish the retina objects, and consistently reaches maximal fitness when given about 300 seconds. Therefore, hypothesis 1 is not correct for all problems. Methods do not perform better than the ones they extend in Retina. It might be that it is difficult for CPPNs to assign the appropriate weights in Retina, as all the methods using CPPNs for weight assignment struggle. Even though the fitness is low, the accuracy in Table 5.27 show that the Hyper-methods reach an accuracy of about 0.7. Clune et al. [2010] got similar results with their standard retina setup. HyperNEAT ended up with accuracy between 0.7 and 0.75. Risi and Stanley [2012] also found that ES-HyperNEAT variants struggle in Retina. They found that ES-HyperNEAT successfully learned the dataset in 30% of cases when run for 2000 generations.

**Iris and Wine**

Figure 5.5a show performance on the Iris dataset, along with 600 generations as the stopping criteria. Between generation 0 and generation 100, the graphs show the fitness increase per generation is lowest in NEAT. HyperNEAT and ES-HyperNEAT increase fitness similarly, though ES-HyperNEAT somewhat slower than HyperNEAT. DES-HyperNEAT increases fitness the fastest of all four, reaching 0.9 in under 50 generations. Although it is not certain how NEAT will continue, they seemingly converge in the same order. In this case, hypothesis 1 is partially correct. Except for HyperNEAT and ES-HyperNEAT being in the opposite order, the methods perform better than the ones they are based upon. The fitness in Table 5.27 confirm the same order, where there is a significant ( $p < 0.01$ ) difference between each pair in the increasing fitness order: NEAT, ES-HyperNEAT, HyperNEAT, and DES-HyperNEAT.



**Figure 5.4: Experiment 8: Performance results - Retina charts.**

In Figure 5.5c, generation is also used as stopping criteria, but with the Wine dataset. Wine is a more difficult dataset to classify than Iris, evident by the methods lower performance. As in Iris, HyperNEAT and ES-HyperNEAT perform similarly, although ES-HyperNEAT is marginally better than HyperNEAT after 100 generations on the Wine dataset. The fitnesses in Table 5.27 show that ES-HyperNEAT ends with a significant ( $p < 0.05$ ) higher fitness at 600 generations. Also, DES-HyperNEAT performs significantly ( $p < 0.01$ ) better than the others, with an even greater margin than in Iris. The most noticeable difference between Iris (a) and Wine (c) when comparing generations is that HyperNEAT and ES-HyperNEAT have dropped so low that they perform worse than NEAT in Wine. It might be caused by HyperNEAT and ES-HyperNEAT being designed to exploit spatial relations between inputs, which there are none of in Wine. However, so is DES-HyperNEAT, and it is able to learn the Wine dataset. HyperNEAT and DES-HyperNEAT are seemingly limited by their single substrate, compared to the multiple substrates used by DES-HyperNEAT.

When using time as stopping criteria, the methods performance is measured against its runtime. The results in Figure 5.5b and Figure 5.5d are similar to the corresponding with 600 generations, though there are some differences. DES-HyperNEAT starts overfitting in Figure 5.5b, evident by the validation accuracy decreasing after the initial peak. Unlike HyperNEAT, with a static network, DES-HyperNEAT continues to increase the network complexity and marginally reduces its ability to generalize. It is able to learn to distinguish the classes within seconds and would normally be stopped when the fitness starts to decrease.

Compared to ES-HyperNEAT and DES-HyperNEAT, NEAT and HyperNEAT increase fitness faster in Figure 5.5b than Figure 5.5a. Therefore, HyperNEAT and

Dataset	Method	Fitness				Accuracy			
		Generations		Time		Generations		Time	
		Mean	SD	Mean	SD	Mean	SD	Mean	SD
Iris	NEAT	0.833	0.014	0.888	0.017	0.940	0.013	0.954	0.016
	HyperN.	0.905	0.023	<b>0.928</b>	0.010	0.948	0.017	0.950	0.017
	ES-HN	0.861	0.073	0.903	0.023	0.930	0.052	0.943	0.016
	DES-HN	<b>0.929</b>	0.006	0.926	0.007	<b>0.958</b>	0.015	<b>0.960</b>	0.013
Wine	NEAT	0.659	0.013	0.875	0.018	0.876	0.047	0.931	0.028
	HyperN.	0.481	0.039	0.594	0.102	0.638	0.051	0.742	0.111
	ES-HN	0.498	0.068	0.593	0.094	0.693	0.110	0.780	0.121
	DES-HN	<b>0.890</b>	0.032	<b>0.911</b>	0.026	<b>0.952</b>	0.024	<b>0.963</b>	0.023
Retina	NEAT	<b>0.436</b>	0.054	<b>1.000</b>	0.000	<b>0.761</b>	0.046	<b>1.000</b>	0.000
	HyperN.	0.304	0.047	0.355	0.035	0.727	0.041	0.718	0.052
	ES-HN	0.296	0.063	0.333	0.042	0.691	0.043	0.687	0.039
	DES-HN	0.373	0.021	0.366	0.021	0.702	0.032	0.701	0.032

**Table 5.27: Experiment 8: Performance results.** Presents validation fitness after reaching the stopping criteria. Therefore not representative when methods overfit.

NEAT is running more generations per second than the other two. It results in HyperNEAT being comparable to DES-HyperNEAT and NEAT to ES-HyperNEAT. Even though NEAT runs more generations per second, it is not better than HyperNEAT and DES-HyperNEAT, so hypothesis 2 is incorrect. The methods using node searches do not perform worse when using time as stopping criteria on Iris. Also, DES-HyperNEAT is able to outperform the others when comparing both time and generations, further disproving hypothesis 2 for the Iris dataset. Figure 5.5d also shows that hypothesis 2 is incorrect in its prediction for the Wine dataset. It is correct that DES-HyperNEAT performs better than NEAT and HyperNEAT, but ES-HyperNEAT does not. Thus, methods using node searches are not necessarily better at solving Wine.

### DES-HyperNEAT

Results show that DES-HyperNEAT is unable to learn the Retina dataset. It is likely because it built upon principles from HyperNEAT and ES-HyperNEAT. These methods do not succeed on that specific task, and it is therefore unlikely that DES-HyperNEAT will. Even though DES-HyperNEAT is unable to compete with NEAT in the Retina dataset, it performs better than NEAT in both Iris and

Wine, when comparing both generations and time. Even though DES-HyperNEAT is overfitting in Iris, the time column in Table 5.27 show that DES-HyperNEAT achieved a significant ( $p < 0.01$ ) higher fitness than NEAT: 0.926 compared to 0.888. Likewise there is a significant ( $p < 0.01$ ) difference between it and NEAT in Wine, where DES-Hyperneat achieves a fitness of 0.911 and NEAT 0.875. DES-HyperNEAT is thus better than NEAT in some problems, but NEAT show to be more versatile in these three experiments. Although NEAT is unable to compete with DES-HyperNEAT in Iris and Wine, it has the advantage that it can provide reasonable solutions in all three datasets. It would be beneficial to investigate why DES-HyperNEAT is unable to solve Retina. As mentioned, it might be that it has the properties of HyperNEAT and ES-HyperNEAT, and that

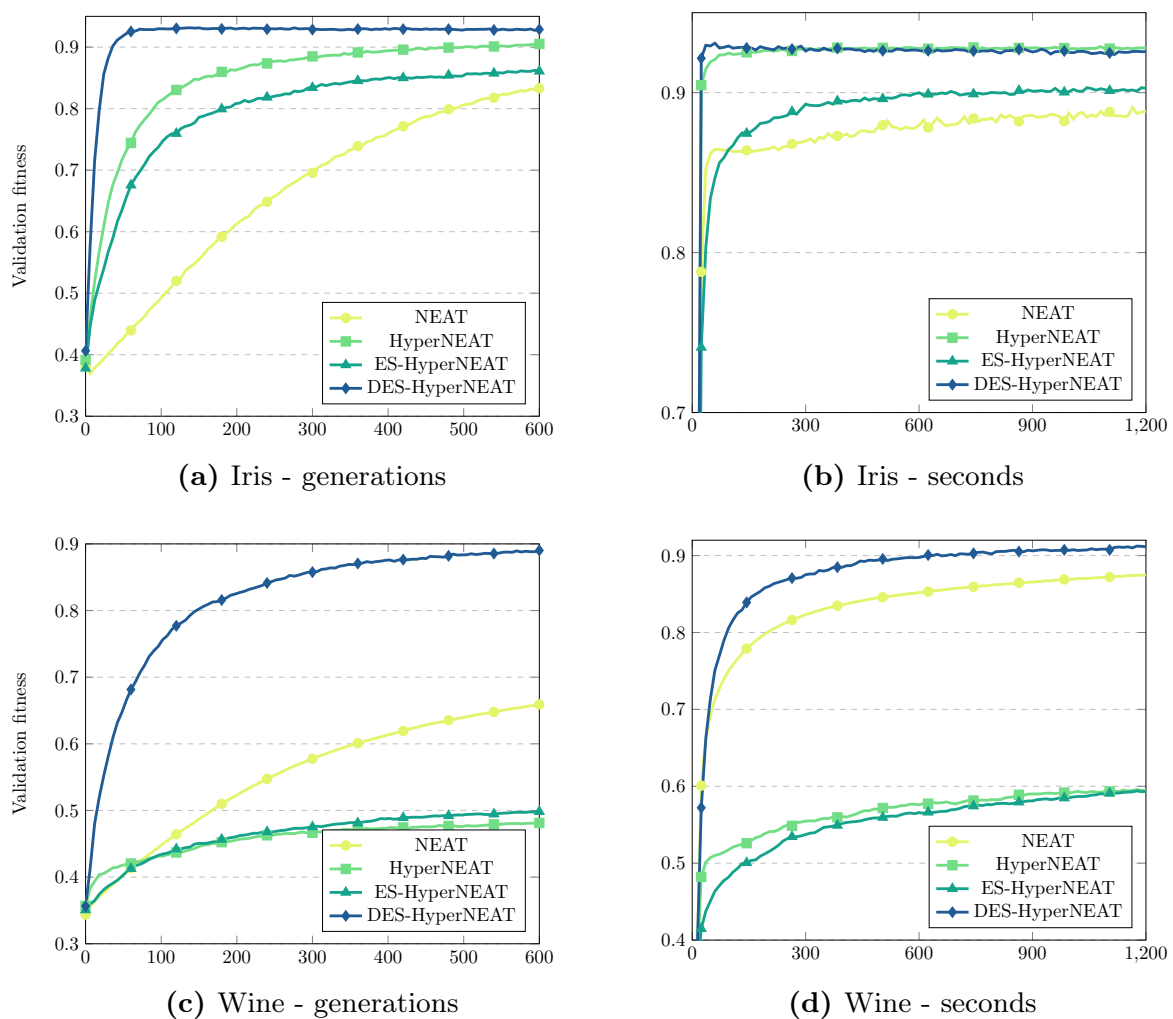


Figure 5.5: Experiment 8: Performance results - Iris and Wine charts.

these properties are not suited for that dataset.

In both Figure 5.4 and Figure 5.5, DES-HyperNEAT reaches a higher fitness, and does it faster than HyperNEAT and ES-HyperNEAT. Although it starts to overfit in Figure 5.2d, it could be stopped earlier and been better or equal to HyperNEAT. In Wine, when the problem is more complex and the inputs not as related, DES-HyperNEAT reaches significantly higher fitness than HyperNEAT and ES-HyperNEAT. Table 5.27 show that DES-HyperNEAT achieves a fitness of 0.911 when using time as topping criteria in Wine. HyperNEAT and ES-HyperNEAT only achieve 0.594 and 0.593, which is significantly ( $p < 0.01$ ) lower.

### Conclusion

The conclusion is that DES-HyperNEAT performs better than HyperNEAT and ES-HyperNEAT in all the three datasets used in this experiment. DES-HyperNEAT also performs better than NEAT in some datasets, though it is not able to learn the Retina dataset. However, it remains to be seen how DES-HyperNEAT performs compared to HyperNEAT and ES-HyperNEAT in reinforcement learning domains where these methods are commonly employed. It is believed that DES-HyperNEAT will perform well there as well, as it is built upon the same principles as ES-HyperNEAT and Multi-Spatial Substrates. The fact that it reaches a similar fitness to HyperNEAT and ES-HyperNEAT in Retina indicates that it is in many ways equal to them. Since it has the same properties as them and is able to outperform them in Iris and Wine, it should also be promising in the domains where HyperNEAT related methods perform well.





# Conclusion

The research goal, research questions, and results are evaluated and discussed in section 6.1. The proposed framework is then compared to state of the art methods in section 6.2. The contributions of the thesis are revisited in section 6.3. Finally, future work is proposed in section 6.4.

## 6.1 Results and Discussion

This work's goal, and the research questions explored to accomplish the goal, are evaluated in this section. The goal will be evaluated after the following research questions:

**Research question 1** *How can the topology in a multi-substrate layout be evolved in parallel with each of its individual substrates?*

The DES-HyperNEAT framework was created to evaluate multiple answers to research question 1. DES-HyperNEAT uses the NEAT algorithm to evolve a substrate topology, termed a *layout*. It also evolves CPPNs that are used to assemble a network in each substrate. Additional CPPNs are evolved and used to connect nodes in different substrates, to form one combined deep network. *Paths* are elements in the layout that determine how the substrates should be connected. Each substrate and path is assigned a CPPN. The assigned CPPNs produces patterns that are used to assemble networks within and connections between substrates. Thus, the question becomes how to simultaneously evolve a layout's topology and the CPPNs that are used to assemble a network in the layout.

Three proposals have been designed, implemented, and evaluated to answer the question. They are called Layered DES-HyperNEAT (LaDES), Single-CPPN DES-HyperNEAT (SiDES), and Coevolutional DES-HyperNEAT (CoDES), described in section 4.3. They are designed to answer two aspects of the questions. The first is whether multiple separate CPPNs, or a single combined CPPN with multiple output nodes, should be used to assemble the different parts of the layout. The second is whether a single genome should contain both a layout and the CPPNs required to assemble it, or if layouts and CPPNs should be separate individuals, in two co-evolved populations.

LaDES uses separate CPPNs for each layout element, while SiDES combines them into a single one, with a separate output per layout element. Evaluation of the results from phase 1 of experiments found that separate CPPNs are superior to a single combined. SiDES produced networks with higher complexity and lower fitness than LaDES, and could not perform as many generations per second. The difference between them was likely caused by the CPPN outputs in SiDES sharing hidden nodes, and they were therefore unable to optimize individually.

LaDES and CoDES both use a separate CPPN for each layout element. However, a layout individual contains its CPPNs in LaDES, while the CPPNs are a separate population in CoDES. Phase 1 of experiments concluded that a combined individual, containing both the layout and all CPPNs, is superior to coevolution. Although the two implementations performed comparably in early generations, CoDES converged earlier and could not reach the fitness of LaDES. It could be caused by new CPPNs not being initialized during evolution in CoDES, or the CPPNs not being able to optimize to an individual layout when used to assemble multiple of them.

The most promising answer to the question is thus to evolve a layout containing a separate CPPN in each of its elements, and using the CPPNs to assemble the element they are within. When crossover is performed between two substrates in the layout, it is also performed on the CPPNs within them. Therefore, both the layout's topology and every substrate within it is evolved in parallel.

**Research question 2** *How can nodes in different substrates be connected in a way that allows the layout's topology to be evolved?*

In DES-HyperNEAT, nodes are connected across different substrates similarly to how networks are constructed within substrates. Paths in the layout determine which substrates are connected, and each path is assigned a CPPN. The hidden substrates in the layout do not possess any nodes. However, when a path between two substrates in the layout is assembled, nodes are created in the path's target substrate. The process is almost identical to when networks are assembled within substrates. All nodes in the source substrate are searched for outgoing connections, though the newly connected nodes are placed in the target substrate instead of

the source substrate. It thus allows for the connections between two substrates to evolve, rather than being static.

In addition to the connections between substrates evolving, the whole layout is evolving. New substrates are inserted in-between existing substrates. When a substrate is inserted, the network output may only remain the same if an identity function is created with it. The feature was investigated in DES-HyperNEAT, as it could be beneficial to avoid disrupting the output when inserting substrates. Three modifications were created so that it was possible to manually construct a CPPN that produced an *identity mapping pattern*. By assigning such a CPPN to a path, the nodes in the path's target substrate would be placed at the exact same locations as in the path's source substrate. As the weights of the connections between them were 1, nodes in the two substrates would have the same value when the network was executed.

The conducted experiments in phase 2 found that the identity mapping is not needed. DES-HyperNEAT is able to evolve networks even though the insertion of a new substrate may initially disrupt the assembled network's output. Therefore, an answer to the research question is that connecting substrates with the same method that crate networks within substrates work well. No further modifications are required. By utilizing the node search from ES-HyperNEAT to create connections between substrates, the node's positions are evolved. As the variance-based node search is used, connections between substrates will also not have uniform weights.

In fact, the results from phase 3 strongly indicate that by utilizing the technique to create connections between substrates, there is no need to create a network in each one. Connections should instead only be between substrates, not within them. The conclusion is therefore that the node search from ES-HyperNEAT is well suited to connect nodes in different substrates. It even alleviates the need to assemble networks within each one.

**Research question 3** *How should the inputs and outputs of the problem be organized in substrates so that the method can produce the best results?*

Since DES-HyperNEAT utilizes multiple substrates, inputs and outputs can be configured in multiple substrates. Related inputs can be placed together, and inputs that are not can be separated. Results from experiments show that it is a beneficial feature. DES-HyperNEAT is even able to evolve networks to solve problems where the inputs are not related. If each input is placed in a unique substrate, DES-HyperNEAT will evolve a layout that connects them appropriately. However, if one has some prior information about the inputs, it is beneficial to group those related in the same substrate and separate those that are not.

**Goal** *Investigate how ES-HyperNEAT can be extended with multiple substrates in an evolving topology, to reduce the required complexity encapsulated by a single CPPN, and increase adaptation to problems through gradual complexification.*

The overall goal has been accomplished. Multiple solutions to research question 1 have been evaluated, and one is selected. Layered DES-HyperNEAT is an extension of ES-HyperNEAT, where an evolving substrate topology is utilized. It does reduce the required complexity encapsulated by each CPPN, as separate CPPN are used to assemble different parts of the network. As the layout and CPPNs are evolved with NEAT, it enables gradual complexification by both adding new substrates and increasing the complexity in each one.

In addition to achieving the goal, it is also found that the natural interpretation of the goal is not the optimal use of properties from ES-HyperNEAT. Results show that the new method should not be a direct extension of ES-HyperNEAT, with multiple instances of ES-HyperNEAT evolved substrates. It should rather be an application of ES-HyperNEAT's node search in a topologically evolved version of Multi-Spatial Substrates. The DES-HyperNEAT framework becomes such a method when limiting the depth in all substrates to zero.

## 6.2 Comparison with state of the art

HyperNEAT, ES-HyperNEAT, MSS, Deep HyperNEAT, and DES-HyperNEAT are presented in Table 6.1, along with some of their properties. HyperNEAT and MSS have predefined static topologies, while the others evolve both network topologies and weights. HyperNEAT and ES-HyperNEAT use a single substrate, while others use multiple. As HyperNEAT and ES-HyperNEAT only have a single substrate, the network connections are within the substrate. In contrast, MSS and Deep HyperNEAT use multiple substrates. Therefore, they create connections between substrates, instead of within them. DES-HyperNEAT differs from the others, as it enables connections to be created both within and between substrates. DES-HyperNEAT also differs in its use of multiple CPPN, where the others use a single one.

As seen in Table 6.1, DES-HyperNEAT has many properties found in other methods. It borrows from ES-HyperNEAT and MSS to enable connections both within and between substrates. It also evolves both the network topology and weights. As in ES-HyperNEAT, node positions within substrates are evolving instead of static. Multiple substrates are also used, as in MSS. Similar to Deep HyperNEAT, DES-HyperNEAT additionally increases the number of substrates during evolution. However, it does so by using NEAT to evolve the substrate topology and not the layered approach in Deep HyperNEAT.

	<b>HyperNEAT</b>	<b>ES-HN</b>	<b>MSS</b>	<b>Deep HN</b>	<b>DES-HN</b>
Connections	Within substrate	Within substrate	Between substrates	Between substrates	Within and between
Network topology	Static	Evolving	Static	Evolving	Evolving
Node positions in substrate	Static	Evolving	Static	Static	Evolving
No. substrates	1	1	Multiple	Evolving	Evolving
No. CPPNs	1	1	1	1	Evolving
No. weight outputs per CPPN	1	1	Multiple	Evolving	1

**Table 6.1: HyperNEAT related methods comparisons.** A comparison of properties among methods related to HyperNEAT. The presented DES-HyperNEAT (DES-HN) implementation is Layered DES-HyperNEAT. HyperNEAT is abbreviated HN in ES-HyperNEAT and Deep HyperNEAT as well.

DES-HyperNEAT thus utilizes the properties thought to be most beneficial in the other four methods. Many of which have been shown to be advantageous. Being build upon the same properties as the others, it should succeed in the same environments as them. It should also be more versatile than the others, as it is a combination of beneficial properties. It possesses MSS’s ability to separate the complexity over multiple CPPNs while also having ES-HyperNEAT’s advantage of evolving node positions. Additionally, like Deep HyperNEAT, it is also able to evolve deeper networks dynamically. It is the combination of all these beneficial properties that make DES-HyperNEAT unique. These properties are what enabled it to outperform HyperNEAT and ES-HyperNEAT in all the conducted experiments.

### 6.3 Contributions

The most significant contribution within this work is the framework Deep Evolvable Substrate HyperNEAT (DES-HyperNEAT). The novelty is the extension of ES-HyperNEAT, from network construction within a single substrate, to network construction across an evolving substrate topology. In addition to the framework, the implementation Layered DES-HyperNEAT is also a contribution. It was selected as the best implementation based on analysis of it and to two other implementations, Single-CPPN DES-HyperNEAT and Coevolutional DES-

HyperNEAT. Multiple configurations of it were then analyzed to make Layered DES-HyperNEAT perform optimally. It outperformed both HyperNEAT and ES-HyperNEAT in the conducted comparisons.

A contribution is also made to the node search algorithm proposed by Risi and Stanley [2012], within CPPN patterns. Three modifications were proposed and evaluated, specifically to allow for an identity function between substrates. However, these are not pursued, as the framework did not benefit an identity mapping.

The final contribution is an open-source implementation of the DES-HyperNEAT framework and the three framework implementations: Layered DES-HyperNEAT, Single-CPPN DES-HyperNEAT, and Coevolutional DES-HyperNEAT. It is available in a public Git repository at <https://github.com/tenstad/des-hyperneat>.

## 6.4 Future Work

Three datasets are used for evaluation and comparison in this work. Retina was chosen because it has been used to evaluate methods related to HyperNEAT previously. Iris and Wine datasets were selected as they are commonly used in the field of machine learning. However, Iris and Wine are not commonly used to evaluate HyperNEAT related methods. Therefore, it would be beneficial to investigate DES-HyperNEAT's performance in other environments as well. As DES-HyperNEAT enables the use of multiple input substrates and output substrates, the maze experiment used by Pugh and Stanley [2013], to evaluate MSS, would be fitting. Likewise, the dual task and maze navigation Risi and Stanley [2012] used to evaluate ES-HyperNEAT would enable a more extensive evaluation. As DES-HyperNEAT is built upon the same principles as MSS and ES-HyperNEAT, discussed in section 6.2, it likely also performs well in the two proposed environments.

DES-HyperNEAT did not succeed in the Retina experiment. HyperNEAT and ES-HyperNEAT struggled as well. It is not known why they reach a fitness limit and are seeming not able to continue learning. An investigation into why DES-HyperNEAT cannot learn the Retina dataset could potentially lead to such limitations being removed, making DES-HyperNEAT a more versatile method.

Bias within network nodes and Link Expression Output [Verbancsics and Stanley, 2011] are two features that each use an additional CPPN output node. Thus, the CPPN can provide each node in the network with a bias, which is commonly used in neural networks. LEO is used to separate a connection's weight from its presence in the network. The pattern determining weights is separate from the one that determines whether to include a connection in the assembled network. Both the use of bias and LEO are thought to be beneficial in DES-HyperNEAT, and should thus be investigated.

# Bibliography

- Bentley, P. and Kumar, S. (1999). Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, GECCO'99, pages 35–43, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.
- Clune, J., Beckmann, B., Mckinley, P., and Ofria, C. (2010). Investigating whether hyperneat produces modular neural networks. pages 635–642.
- Devore, J. L. and Berk, K. N. (2007). *Modern Mathematical Statistics with Applications*. Springer, 2 edition.
- Dua, D. and Graff, C. (2017). UCI machine learning repository.
- Gillespie, L. E., Gonzalez, G. R., and Schrum, J. (2017). Comparing direct and indirect encodings using both raw and hand-designed features in tetris. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 179–186, New York, NY, USA. Association for Computing Machinery.
- Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. <http://www.deeplearningbook.org>.
- Green, C. (2006). Sharpneat. <http://sharpneat.sourceforge.net>.
- Hausknecht, M., Lehman, J., Miikkulainen, R., and Stone, P. (2014). A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366.
- He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.
- Huizinga, J., Clune, J., and Mouret, J.-B. (2014). Evolving neural networks that are both modular and regular: Hyperneat plus the connection cost technique. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary*

- Computation*, GECCO '14, page 697–704, New York, NY, USA. Association for Computing Machinery.
- Kashtan, N. and Alon, U. (2005). Spontaneous evolution of modularity and network motifs. *Proceedings of the National Academy of Sciences of the United States of America*, 102:13773–8.
- Kassahun, Y. and Sommer, G. (2005). Efficient reinforcement learning through evolutionary acquisition of neural topologies. pages 259–266.
- Koutnik, J., Gomez, F., and Schmidhuber, J. (2010). Evolving neural networks in compressed weight space. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, page 619–626, New York, NY, USA. Association for Computing Machinery.
- Lan, G., de Vries, L., and Wang, S. (2019). Evolving efficient deep neural networks for real-time object recognition. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 2571–2578.
- McIntyre, A., Kallada, M., Miguel, C. G., and da Silva, C. F. (2017). neat-python. <https://github.com/CodeReclaimers/neat-python>.
- Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., and Hodjat, B. (2019). Chapter 15 - evolving deep neural networks. In Kozma, R., Alippi, C., Choe, Y., and Morabito, F. C., editors, *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293 – 312. Academic Press.
- Pugh, J. K. and Stanley, K. O. (2013). Evolving multimodal controllers with hyperneat. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 735–742, New York, NY, USA. ACM.
- Risi, S., Lehman, J., and Stanley, K. O. (2010). Evolving the placement and density of neurons in the hyperneat substrate. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 563–570, New York, NY, USA. ACM.
- Risi, S. and Stanley, K. O. (2012). An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons. *Artificial Life*, 18(4):331–363. PMID: 22938563.
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach,



- M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.
- Sosa, F. A. and Stanley, K. O. (2018). Deep hyperneat: Evolving the size and depth of the substrate.
- Stanley, K. O. (2007). Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162.
- Stanley, K. O., Clune, J., Lehman, J., and Miikkulainen, R. (2019). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35.
- Stanley, K. O., D’Ambrosio, D. B., and Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212. PMID: 19199382.
- Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127.
- Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*.
- Sun, Y., Xue, B., Zhang, M., and Yen, G. (2018). Automatically designing cnn architectures using genetic algorithm for image classification.
- Verbancsics, P. and Harguess, J. (2015). Image classification using generative neuro evolution for deep learning. In *2015 IEEE Winter Conference on Applications of Computer Vision*, pages 488–493.
- Verbancsics, P. and Stanley, K. O. (2011). Constraining connectivity to encourage modularity in hyperneat. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO ’11*, pages 1483–1490, New York, NY, USA. ACM.
- Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D. (2019).

Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

Xin Yao (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.

Yang, X.-S. (2014). Chapter 5 - genetic algorithms. In Yang, X.-S., editor, *Nature-Inspired Optimization Algorithms*, pages 77 – 87. Elsevier, Oxford.

# Appendix

---

<b>General</b>	
population size	100
species target	8
speciation threshold	0.8
speciation threshold delta	0.05
asexual reprod. prob.	0.25
interspecies reprod. prob.	0.001
tournament size	2
dropoff age	20
young species multipleier	1.01
young age limit	20
stagnant species multipleier	0.2
survival ratio	0.2
initial mutations	100
elites	1

---

<b>NEAT</b>	
add node prob.	0.03
add link prob.	0.2
remove node prob.	0.006
remove link prob.	0.08
link mutation prob.	0.9
initial weight size	0.5
weight mutation size	0.5

---

<b>CPPN</b>	
mutate bias prob.	0.8
mutate bias size	0.03
mutate activation prob.	0.1
activations	[None, Linear, Step, ReLU, Sigmoid, Exp, Sigmoid, Tanh, Gaussian, OffsetGaussian, Sine, Square, Abs]

---

<b>HyperNEAT</b>	
weight threshold	0.1
hidden activation	None

---

<b>ES-HyperNEAT</b>	
variance threshold	0.03
division threshold	0.03
band threshold	0.3
initial resolution	4
max resolution	5
iteration level	3

---

<b>DES-HyperNEAT</b>	
min substrate depth	0
max substrate depth	5
max input substrate depth	0
max output substrate depth	0
mutate depth prob	0.1

---

**Table A1: Default hyperparameters.** Where methods extend others, they use the parameters listed in the methods they extend.

method	[NEAT, CPPN, HyperNEAT, ES-HyperNEAT]
dataset	[Iris, Wine, Retina]
stop criterion	30 seconds
repeats	50
population size	[100, 400]
species target	[None, 10, 20]
survival ratio	[0.2, 0.5]
initial mutations	[100, 250]
asexual reprod. prob.	[0.15, 0.5]
population size	[75, 200]
species target	[8, 12]
survival ratio	[0.15, 0.3]
initial mutations	[80, 150]
population size	[50, 75, 125, 150]
species target	[4, 6, 8, 10]

**Table A2: General parameter search grids.**

method	[NEAT, CPPN, HyperNEAT, ES-HyperNEAT]
dataset	[Iris, Wine, Retina]
stop criterion	30 seconds
repeats	50
add node prob.	[0.03, 0.05]
add link prob.	[0.05, 0.1]
remove node prob.	[0.006, 0.01]
remove link prob.	[0.01, 0.02]
add link prob.	[0.1, 0.15, 0.2]
remove link prob.	[0.01, 0.05, 0.08]
initial link size	[0.2, 0.8, 2.0]
link mutation size	[0.2, 0.8, 2.0]
link mutation prob.	[0.5, 0.9]
initial link size	[0.5, 0.8]
link mutation size	[0.5, 0.8]
link mutation prob.	[0.7, 0.8, 0.9]

**Table A3: NEAT parameter search grids.**

method	[CPPN, HyperNEAT, ES-HyperNEAT]
dataset	[Iris, Wine, Retina]
stop criterion	30 seconds
repeats	50
mutate bias prob.	[0.7, 0.8, 0.9]
mutate bias size	[0.1, 0.5, 1.0]
mutate bias size	[0.01, 0.03, 0.05, 0.08, 0.1, 0.15]
mutate activation prob.	[0.01, 0.05, 0.1, 0.15]
activation functions.	[[None Linear Tanh OffsetGauss Sin], [None Linear Tanh Gauss Sin], [None Linear Step Sigmoid Tanh OffsetGauss Sin], [None Linear Step Sigmoid Tanh Gauss Sin], [None Linear Step ReLU Sigmoid Tanh Gauss OffsetGauss Sin], [None Linear Step ReLU Sigmoid Tanh Gauss OffsetGauss Sin Cos Square Abs Exp]]
activation functions	[[None], [Tanh Sigmoid], [None Linear Tanh OffsetGauss Gauss Sin], [None Tanh OffsetGauss Gauss Sin], [Linear Tanh OffsetGauss Gauss Sin], [Linear OffsetGauss Gauss Sin], [Tanh OffsetGauss Gauss Sin], [Tanh OffsetGauss Gauss Sin Sigmoid], [Tanh OffsetGauss Gauss Sin Step], [Tanh OffsetGauss Gauss Sin ReLU]]

**Table A4: CPPN parameter search grids.**

method	ES-HyperNEAT
dataset	[Iris, Wine, Retina]
stop criterion	30 seconds
repeats	50
division threshold	[0.01, 0.03, 0.1]
variance threshold	[0.01, 0.03, 0.1]
band threshold	[0.15, 0.3, 0.5]

**Table A5: ES-HyperNEAT parameter search grids.**

method	Layered DES-HyperNEAT
dataset	[Iris, Wine, Retina]
stop criterion	60 seconds
repeats	50
activation functions	[[Tanh, OffsetGaussian, Gaussian, Sine, Sigmoid], [None, Linear, Step, ReLU, Sigmoid, Sigmoid, Tanh, Gaussian, OffsetGaussian, Sine, Square], [None, Linear, Step, ReLU, Sigmoid, Exp, Sigmoid, Tanh, Gaussian, OffsetGaussian, Sine, Square, Abs]]
variance threshold	[0.03, 0.15, 0.5]
division threshold	[0.03, 0.15, 0.5]
variance threshold	[0.4, 0.5, 0.6]
division threshold	[0.01, 0.03, 0.05]
add substrate prob.	[0.02, 0.04]
add path prob.	[0.1, 0.2]
add CPPN node prob.	[0.02, 0.04]
add CPPN link prob.	[0.1, 0.2]
add substrate prob.	[0.02, 0.04]
add path prob.	[0.25, 0.35]
add CPPN node prob.	[0.02, 0.04]
add CPPN link prob.	[0.25, 0.35]
add substrate prob.	[0.015, 0.025]
add path prob.	[0.35, 0.4]
add CPPN node prob.	[0.015, 0.025]
add CPPN link prob.	[0.35, 0.4]
add substrate prob.	[0.025, 0.03]
add path prob.	[0.4, 0.45]
add CPPN node prob.	[0.025, 0.03]
add CPPN link prob.	[0.4, 0.45]
remove substrate prob.	[0.0025, 0.005]
remove path prob.	[0.05, 0.1]
remove substrate prob.	[0.001, 0.0025]
remove path prob.	[0.02, 0.05]
band pruning threshold	[0.0, 0.1, 0.2, 0.3]

**Table A6: Layered DES-HyperNEAT search grids.**

